Chapter 1
Array Processors and Mapping

High Performance Signal Processing Via Systolic Array Architectures and Algorithms

IN the last fifty years, the development of array processors for high performance signal processing has been closely interweaved with the development of high-performance digital computers. The earliest and most well known computer design was based on von Neumann architecture. It has a single processor executing single sequence of instructions (SI) on a single stream of data (SD) and is called a SISD machine [19].

This conceptual simplicity and associated technology led to the immense success of the digital computer revolution. In order to achieve higher computational throughput speed, the utilization of newer device technology with a faster switching speed has been an obvious approach. Often, this approach based only on device-technology improvement has not been sufficient. However, various alternative basic computer processing concepts and architectures have also been proposed.

In one approach, various forms of concurrent operations are used. **Concurrency** denotes the ability of a computational system to perform more than one operation at a given time and can be achieved through either parallelism or pipelining, or both. **Parallelism** utilizes concurrency by replicating some processing elements (PE). High throughput rate is achieved by having simultaneous operations of these functions on different parts of the program. Many parallel-processing, vector-data processor arrays operate in the single instruction (SI) and multiple data (MD) SIMD mode.

On the other hand, **pipelining** addresses concurrency by breaking some demanding part of the task into many smaller, simpler pieces, using many corresponding PEs, so that the processing can be performed consecutively. This digital pipe is arranged so that it is capable of processing the instructions and data independent of the number of PEs in the pipe. The high throughput rate in the pipe can be achieved by having fast PEs. A computer utilizing the pipelining feature is considered as a MISD machine since different parts of the single data (SD) stream are being processed simultaneously by multiple instructions (MI). Finally, multiple-instruction and multiple-data stream computers have many PEs with independent computational capabilities.

For many years, these general "parallel" processing computer architectures have been used only on specialized mainframe and minicomputer array processing computers for some very computational intensive off-line scientific, economic, meteorologic, and military applications. These "parallel" processing computers were almost always extremely expensive and were generally not available for dedicated real-time signal processing applications. Only with the advent of VLSI semiconductor fabrication technology have these parallel processing computer architectures been significant to high-performance modern signal processing. One of the most basic and important links between parallel computation/architecture and high throughput modern signal processing is the concept of systolic arrays.

The systolic array concept and term were coined by H. T. Kung and C. E. Leiserson [32] to denote a simple class of concurrent processor, in which processed data move in a regular and periodic manner similar to the systolic pumping action of blood by the heart. The most basic idea of a systolic array is that once data are available, they are used effectively inside many PEs in the array to yield a higher throughput rate. Thus, a systolic array may exploit both the parallelism and pipelining capabilities of various algorithms. In the motivational and tutorial reprinted paper by H. T. Kung, a systolic array is required to satisfy the properties when there are only a few types of PEs in the array, with each type performing the same specific operation. All the operations are performed in a synchronous manner independent of the processed data. The only control data broadcast to the PEs are the synchronous clock signals, and the PEs communicate only with their nearest neighbors. These regular structure and local communication properties of a systolic array are consistent with efficient modern VLSI designs. A series of systolic designs for correlation/convolution was obtained intuitively. These examples demonstrate the basic property that a given signal processing algorithm can have many different systolic implementations with different hardware requirements and implications.

Various extensions of the earlier systolic array assumptions have since been made. These arrays may have the properties of wavefront array processing which allow PEs to start/end/control their own processing tasks dependent on the data. Some of the PEs can perform a limited number of different functions depending on the presence of some control data. Some PEs can communicate with a few nearby neighbors while the PEs at the edge of the array are allowed to have wrap-around communications. The reprinted paper by S. Y. Kung introduces the concept of asynchronous communication among PEs in wavefront array processing. While the original systolic array concept was applied only at the cellular PE level, the reprinted paper by H. T.

Kung and Lam considers systolic array methodology to the PEs resulting in a two-level pipelined systolic array with a more efficient design.

The earlier work on mapping a signal processing algorithm onto a systolic array was performed intuitively. In 1982, a systematic approach based on space-time transformation was proposed for the systolic design of convolution [7]. Since then, various formal systematic procedures for systolic designs of many classes of algorithms, often called dependence graph mapping techniques, have been proposed. One of the earliest works in this direction is the reprinted paper by Moldovan which considers the transformation of algorithms with loops into parallel forms suitable for systolic implementations. Examples with Fortran loop structure and LU decompositions are given. Another early systematic systolic design approach of Quinton [62] introduces the concept of shift-invariance of the dependence graph, modeling the index variables of the algorithm. Uniform Recurrence Equations (URE) require only one computational element, while the Generalized URE (GURE) can use multiple computational elements. The reprinted paper by Quinton and Van Dongen extends Quinton's earlier techniques by using general systems of linear recurrences to define the class of algorithms suitable for systolic implementation. The reprinted paper by Rao and Kailath first critically reviews previously known techniques of URE and GURE and then introduces a general class of algorithms for systematic systolic designs, denoted as regular iterative algorithms (RIA). The concepts of PE scheduling and allocating in the systolic design were defined precisely, and the linear programming procedure for their computations was proposed [29]. Variations and generalizations of these approaches have appeared in other works [25, 50, 67, 69, 84].

Another recent systematic approach [12] for efficient systolic design is to formally interpret the dependence graph of an algorithm as a lattice in a multidimensional integral space [57]. The whole procedure of mapping an algorithm onto a systolic-type processor consists of two different but interdependent operations using space and time transformations. The former projects the dependence graph onto a lower dimensional structure which can then be mapped onto the physical array, while the latter yields the execution of each computation in the array. Recent Ph.D. thesis work on general systolic mapping techniques include [4, 41, 75].

As stated earlier, the systolic concept normally applied at the PE word level of a processor array can also be applied at the lower bit level [43]. Various generalizations of this bit-level design have been demonstrated for correlation, convolution, Winograd matrix-vector multiplication, and IIR [44, 80]. In the reprinted paper by McCanny, McWhirter, and S. Y. Kung, the dependence graph technique based on the cut-set method [34] was proposed to perform general bit-level systolic array designs. On the other hand, it is well known that many modern signal processing tasks are matrix-computational problems [24, 72]. These complex higher-level problems are also well suited for parallel/systolic processing. The reprinted paper by Moreno and Lang considers matrix computations on application-specific systolic-type array architectures. They proposed a multimesh graph method for matching the fine granularity of the architecture to that of the matrix algorithm for an efficient design.

In recent years, there have been many proposed schemes to perform systematic and efficient systolic array designs. One basic approach uses partitioning to split a large dimensional systolic array problem so that it can be implemented on a physically smaller fixed array, as proposed in [51]. Other partitioning papers include [26, 27, 42, 55, 56, 76]. On the other hand, in many practical complex DSP problems, such as in systolic Kalman filtering [39, 66] and systolic eigen/singular value decompositions [1, 40], the computational algorithms have several distinct stages which need to be processed sequentially. Dedicated systolic array hardware for implementing each stage yields a highly inefficient system. In the reprinted paper by Hwang and Hu, a multistage systolic-design technique has been proposed to map different phases of the algorithm expressed in several nested loops onto a single systolic array hardware. The critical issue of interfacing data flow and distribution between stages is addressed.

As the systolic design field becomes more mature, it is natural to consider optimized systolic designs. The reprinted paper by Wong and Delosme considers the derivation of time-optimal linear schedules of systolic designs for RIA algorithms. The time performance is measured as the product of the number of systolic cycles needed to perform some computation. The maximum time duration of a cycle as well as other optimizations are based on integer linear programming techniques. Many other generalizations and optimization criteria for systolic designs have been proposed in [10, 17, 30, 33, 37, 38, 70, 71, 81, 84].

From the beginning of systolic designs, there is continual interest in constructing CAD tools for automated VLSI synthesis of the processing array architecture. The goal is to start with some high level DSP algorithm specification (such as in C, Matlab, and others) and end up with a reasonably efficient systolic processor design, taking into consideration user-imposed constraints on throughput, chip area, I/O, power consumption, and so on. While much interesting work has been done on this problem [13, 15, 56, 61, 65], the design of a practical CAD program for the automated synthesis of systolic arrays at the multiprocessor level so far remains unsolved. Some recent work at the automated design of systolic array at the chip level include [45, 77].

In the last fifteen years, hundreds of technical papers on array processor architectures and mapping techniques have appeared. A detailed treatment of these topics is given in the book by S. Y. Kung [34] in 1988. Many other books [16, 18, 23, 48, 52, 59, 60, 63, 68, 73, 78] and edited chapters and tutorial articles [20, 28, 31, 46, 49, 74, 83] dealing with these topics have appeared. New research results have appeared regularly in the proceedings of the VLSI Signal Processing Workshop [2, 5, 6, 14, 35, 54, 58, 64, 82] and Applications Specific Array Processors [3, 8, 9, 11, 21, 22, 36, 47, 53, 79].

References

[1] R. P. Brent and F. T. Luk, "The solution of singular value and symmetric eigenvalue problems on multiprocessor," SIAM J. Sci. Stat. Comp., Vol. 6, pp. 69–84, 1985.

[2] R. W. Brodersen and H. S. Moscovitz, ed., *Signal Processing, III,* IEEE Press, 1988.

[3] K. Bromley, S. Y. Kung, and E. E. Swartzlander, ed., *Proc. Inter. Conf. on Systolic Arrays,* IEEE Computer Soc. Press, 1988.

[4] J. Bu, "Systematic design of regular VLSI processor arrays," Ph.D. thesis, Delft Univ. of Tech., Delft, The Netherlands, 1990.

[5] W. Burleson, K. Konstantinides, and T. Meng, ed., *Signal Processing, IX,* IEEE Press, 1996.

[6] P. R. Cappello, et al., ed., *Signal Processing, I,* IEEE Press, 1984.

[7] P. R. Cappello and K. Steiglitz, "Unifying VLSI array designs with geometric transformations," Proc. Inter. Conf. Parallel Processing, pp. 448–457, 1983.

[8] P. R. Cappello, R. M. Owens, E. E. Swartzlander, and B. W. Wah, ed., *Proc. Inter. Conf. on Application Specific Array Processors,* IEEE Computer Soc. Press, 1994.

[9] P. R. Cappello, et al., ed., *Proc. Inter. Conf. on Application Specific Array Processors,* IEEE Computer Soc. Press, 1995.

[10] P. Clauss, C. Mongenet, and G. -R. Perrin, "Calculus of Space-Optimal Mappings of Systolic Algorithms on Processor Arrays," Journal of VLSI Signal Processing, Vol. 4, No. 1, pp. 27, Feb. 1992.

[11] L. Dadda and B. W. Wah, ed., *Proc. Inter. Conf. on Application Specific Array Processors,* IEEE Computer Soc. Press, 1993.

[12] A. Darte and J. M. Delosme, "Partitioning for array processors," Technical Report LIP-IMAG 90–23, Laboratoire de l'Informatique du Parallélisme, Ecole Supérieure de Lyon, Oct. 1990.

[13] E. Deprettere, P. Held, and P. Wielage, "Model and methods for regular array design," Int. Journal of High Speed Electronics, Special Issue on Parallel Computing-II, Vol. 4, No. 2, pp. 133–201, 1993.

[14] L. D. J. Eggermont, P. Dewilde, E. Deprettere, and J. van Meerbergen, ed., *Signal Processing, VI,* IEEE Press, 1993.

[15] B. R. Engstrom and P. R. Cappello, "The SDEF programming system," Jour. of Parallel and Dist. Computing, pp. 201–203, Oct. 1989.

[16] D. J. Evans, *Systolic Algorithms,* Gordon & Breach, 1991.

[17] A. L. Fisher and H. T. Kung, "Synchronizing large VLSI arrays," IEEE Trans. Computers, pp. 734–740, Aug. 1985.

[18] T. Fountain, *Processor Arrays,* Academic Press, 1987.

[19] M. J. Flynn, "Very high-speed computing systems," Proc. of the IEEE, pp. 1901–1909, 1966.

[20] J. Fortes and B. W. Wah, "Systolic Array," IEEE Computer, Vol. 20, No. 7, July 1987.

[21] J. Fortes, E. Lee, and T. Meng, ed., *Proc. Inter. Conf. on Application Specific Array Processors,* IEEE Computer Soc. Press, 1992.

[22] J. Fortes, C. Mongenet, K. Parhi, and V. E. Taylor, ed., *Proc. Inter. Conf. on Application Specific Array Processors,* IEEE Computer Soc. Press, 1996.

[23] M. A. Frumkin, *Systolic Computations,* Kluwer, 1992.

[24] G. H. Golub and C. F. Van Loan, *Matrix Computations,* 2nd ed., Johns Hopkins Univ. Press, Baltimore, MD, 1989.

[25] C. Guerra and R. Melhem, "Synthesis of systolic algorithm design," Parallel Computing, pp. 195–207, Nov. 1989.

[26] D. Heller, "Partitioning big matrices for small systolic arrays," *VLSI and modern signal processing,* ed. S. Y. Kung, H. J. Whitehouse, and T. Kailath, Prentice-Hall, pp. 185–199, 1985.

[27] K. Hwang and Y. H. Cheng, "Partitioned matrix algorithms for VLSI arithmetic systems," IEEE Trans. on Computers, Vol. 31, pp. 1215–1224, 1982.

[28] K. T. Johnson and A. R. Huron, "General-Purpose Systolic Arrays," IEEE Computer, Vol. 26, No. 11, pp. 20–31, Nov. 1993.

[29] R. M. Karp, R. E. Miller, and S. Winograd, "The organization of computations for uniform recurrence equations," J. ACM, Vol. 14, pp. 563–590, 1967.

[30] V. K. Prasana Kumar and Y. C. Tsai, "On synthesizing optimal family of linear systolic arrays for matrix multiplication," IEEE Trans. Computers, pp. 770–774, June 1991.

[31] H. T. Kung, B. Sproull, and G. Steele, *VLSI Systems and Computations,* Computer Science Press, 1981.

[32] H. T. Kung, and C. E. Leiserson, "Systolic Arrays for VLSI," *Introduction to VLSI Systems,* C. A. Meads and L. A. Conway, Addison-Wesley, 1980.

[33] S. Y. Kung, K. S. Arun, R. J. Gal-Ezer, and D. V. B. Rao, "Wavefront array processor: language, architecture, and applications," IEEE Trans. Computers, C-31, 11, pp. 1054–1066, Nov. 1982.

[34] S. Y. Kung, *VLSI Array Processors,* Prentice-Hall, 1988.

[35] S. Y. Kung, R. E. Owen, and J. G. Nash, ed., "Signal Processing II," IEEE Press, 1986.

[36] S. Y. Kung, E. E. Swartzlander, J. A. B. Fortes, and K. W. Przytula, ed., *Proc. Inter. Conf. on Application Specific Array Processors,* IEEE Computer Soc. Press, 1990.

[37] H. Leverge, C. Mauras, and P. Quinton, "A language-oriented approach to the design of systolic chips," Proc. Int. Workshop on Algorithms and Parallel VLSI Architectures, pp. 309–327, June 1990.

[38] G. J. Li and B. W. Wah, "The design of optimal systolic arrays," IEEE Trans. Computers, Vol. 34, No. 1, pp. 66–77, Jan. 1985.

[39] R. A. Lincoln and K. Yao, "Efficient systolic Kalman filter design by dependence graph mapping," *VLSI Signal Processing, III,* ed. R. W. Brodersen and H. S. Moscovitz, IEEE Press, pp. 396–410, 1990.

[40] K. J. R. Liu and K. Yao, "Multi-phase systolic algorithms for spectral decomposition," IEEE Trans. on Signal Proc., Vol. 40, No. 1, pp. 190–201, Jan. 1992.

[41] F. Lorenzelli, "Systolic mapping with partitioning and computationally intensive algorithms for signal processing," Ph.D. thesis, University of California, Los Angeles, 1993.

[42] F. Lorenzelli and K. Yao, "An integral matrix-based technique for systematic systolic design," Integration, the VLSI Journal, Vol. 20, No. 3, pp. 269–285, July 1996.

[43] J. V. McCanny and J. G. McWhirter, "On the implementation of signal processing functions using one bit systolic array," Electronic Letters, pp. 242–243, 1982.

[44] J. V. McCanny and J. G. McWhirter, "Some aspects of systolic array research in the U.K.," IEEE Computer, pp. 51–63, July 1987.

[45] J. V. McCanny, Y. Hu, and M. Yan, "Automated design of DSP array processor chips," *Proc. Inter. Conf. on Applications Specific Array Processors,* ed. E. E. Swartzlander and P. R. Cappello, IEEE Computer Press, pp. 33–44, 1994.

[46] J. G. McWhirter and J. V. McCanny, "Systolic and Wavefront Arrays," *VLSI Technology and Design,* ed. J. V. McCanny and J. C. White, Academic Press, pp. 253–300, 1987.

[47] J. V. McCanny, J. McWhirter, and E. E. Swartzlander, ed., *Proc. Inter. Conf. on Systolic Arrays,* Prentice-Hall, 1989.

[48] G. M. Megson, *An Introduction to Systolic Algorithm Design,* Oxford Press, 1992.

[49] G. E. Megson, ed., *Transformational Approaches to Systolic Design,* Chapman and Hall, 1994.

[50] D. I. Moldovan, "On the analysis and synthesis of VLSI algorithms," IEEE Trans. on Computers, Vol. 31, No. 11, pp. 1121–1126, Nov. 1982.

[51] D. I. Moldovan and J. A. B. Fortes, "Partitioning and mapping of algorithms into fixed size systolic arrays," IEEE Trans. Computers, Vol. 35, No. 1, pp. 1–12, Jan. 1986.

[52] D. I. Moldovan, *Parallel Processing from Applications to Systems,* Morgan Kaufman, 1993.

[53] W. Moore, A. McCabe, and R. Uraqhart, ed., *Systolic Arrays,* Adam Hilger, 1987.

[54] H. S. Moscovitz, K. Yao, and R. Jain, ed., *Signal Processing, IV,* IEEE Press, 1990.

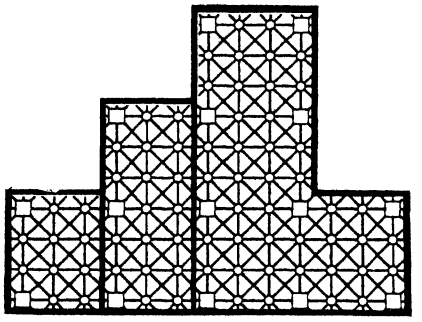
[55] J. J. Navarro, J. M. Llaberia, and M. Valero, "Partitioning: an essential step in mapping algorithms into systolic arrays," IEEE Computer, Vol. 20, No. 7, pp. 77–89, July 1987.

[56] H. W. Nelis and E. F. Deprettere, "Automatic Design and Partitioning of Systolic/Wavefront Arrays for VLSI," Circuits, Systems, Signal Processing, Vol. 7, No. 2, pp. 235–252, 1988.

[57] M. Newman, *Integral Matrices,* Academic Press, New York, 1972.

[58] T. Nishitani and K. Parhi, ed., *Signal Processing, VIII,* IEEE Press, 1995.
[59] N. Petkov, ed., *Systolic Parallel Processing,* North Holland, 1993.
[60] S. U. Pillai, *Array Signal Processing,* Springer-Verlag, 1989.
[61] P. Quinton, "Automatic synthesis of systolic arrays from uniform recurrent equations," Proc. IEEE 11-th Int. Sym. on Computer Arch., pp. 208–214, 1984.
[62] P. Quinton, "The systematic design of systolic arrays," IRISA Technical Report 193, Apr. 1983.
[63] P. Quinton and Y. Robert, *Systolic Algorithms and Architectures,* Prentice-Hall, 1991.
[64] J. Rabaey, P. M. Chau, and J. Eldon, ed., *Signal Processing, VII,* IEEE Press, 1994.
[65] S. V. Rajopadhye and R. M. Fujimoto, "Automating the design of systolic arrays," Integration, the VLSI Journal, Vol. 9, No. 3, pp. 225–242, July 1990.
[66] P. Rao and M. A. Bayoumi, "An algorithm-specific VLSI parallel architecture for Kalman filter," *VLSI Signal Processing—IV,* ed. H. S. Moscovitz, K. Yao, and R. Jain, IEEE Press, pp. 264–273, 1990.
[67] S. Rao, "Regular iterative algorithms and their implementations on processor arrays," Ph.D. thesis, Stanford University, Stanford, CA, 1985.
[68] Y. Robert, *The Impact of Vector and Parallel Architectures on the Guassian Elimination Algorithm,* Oxford Press, 1990.
[69] V. P. Roychowdhury and T. Kailath, "Subspace scheduling and parallel implementation of non-systolic regular iterative algorithms," Journal of VLSI Signal Processing, Vol. 1, No. 2, pp. 127–142, Oct. 1987.
[70] C. Scheiman and P. R. Cappello, "A period-processor-time-minimal schedule for cubical mesh algorithms," IEEE Trans. on Parall. and Dist. Sys., Vol. 5, No. 3, pp. 274–280, Mar. 1994.
[71] C. Seitz, "Concurrent VLSI architectures," IEEE Trans. Computes, Vol. 33, 1247–1265, Dec. 1984.
[72] J. M. Speiser and H. Whitehouse, "A review of signal processing with systolic arrays," SPIE Real-Time Signal Processing, pp. 2–6, Aug. 1983.
[73] E. E. Swartzlander, *Systolic Signal Processing Systems,* Marcel Dekker, 1987.
[74] E. E. Swartzlander, "VLSI Signal Processing" *Signal Processing Handbook,* ed. C. H. Chen, pp. 221–256, Marcel Dekker, 1988.
[75] J. Teich, "A compiler for application-specific processor array," Ph.D. thesis, University of Saarland, Germany, 1993.
[76] J. Teich and L. Thiele, "Partitioning of processor arrays: a piece wise regular approach," Integration, the VLSI Journal, Vol. 14, pp. 297–332, Feb. 1993.
[77] D. W. Trainor, R. F. Woods, and J. V. McCanny, "Architectural Synthesis of an Image Processing Algorithm using IRIS," *VLSI Signal Processing, VIII,* ed., T. Nishitani and K. Parhi, IEEE Press, pp. 167–176, 1995.
[78] P. S. Tseng, *A Systolic Array Parallelizing Compiler,* Kluwer, 1990.
[79] M. Valero, S. Y. Kung, T. Lang, and J. Fortes, ed., *Proc. Inter. Conf. on Application Specific Array Processors,* IEEE Computer Soc. Press, 1991.
[80] R. F. Woods, S. C. Knowles, J. V. McCanny, and J. G. McWhirter, "Systolic IIR filtering with bit level pipelining," Proc. IEEE Inter. Conf. on Acous., Speech, and Sig. Proc., pp. 2072–2075, Apr. 1988.
[81] Y. Yaacoby and P. R. Cappello, "Scheduling a system of nonsingular affine recurrence equations onto a processor array," Journal of VLSI Sig. Proc., Vol. 1, No. 2, pp. 115–125, Oct. 1989.
[82] K. Yao, R. Jain, and J. Rabaey, ed., *Signal Processing, V,* IEEE Press, 1992.
[83] K. Yao and F. Lorenzelli, "Systolic Arrays," *The Circuit and Filter Handbook,* ed. W. K. Chen, pp. 2034–2071, CRC Press and IEEE Press, 1995.
[84] X. Zhong, S. Rajopadhye, and I. Wong, "Systematic generation of linear allocation functions in systolic array design," Jour. of VLSI Signal Proc., Vol. 4, No. 4, pp. 279–293, Nov. 1992.

Systolic architectures, which permit multiple computations for each memory access, can speed execution of compute-bound problems without increasing I/O requirements.



Why Systolic Architectures?

H. T. Kung
Carnegie-Mellon University

High-performance, special-purpose computer systems are typically used to meet specific application requirements or to off-load computations that are especially taxing to general-purpose computers. As hardware cost and size continue to drop and processing requirements become well-understood in areas such as signal and image processing, more special-purpose systems are being constructed. However, since most of these systems are built on an ad hoc basis for specific tasks, methodological work in this area is rare. Because the knowledge gained from individual experiences is neither accumulated nor properly organized, the same errors are repeated. I/O and computation imbalance is a notable example—often, the fact that I/O interfaces cannot keep up with device speed is discovered only after constructing a high-speed, special-purpose device.

We intend to help correct this ad hoc approach by providing a general guideline—specifically, the concept of systolic architecture, a general methodology for mapping high-level computations into hardware structures. In a systolic system, data flows from the computer memory in a rhythmic fashion, passing through many processing elements before it returns to memory, much as blood circulates to and from the heart. The system works like an automobile assembly line where different people work on the same car at different times and many cars are assembled simultaneously. An assembly line is always linear, however, and systolic systems are sometimes two-dimensional. They can be rectangular, triangular, or hexagonal to make use of higher degrees of parallelism. Moreover, to implement a variety of computations, data flow in a systolic system may be at multiple speeds in multiple directions—both inputs and (partial) results flow, whereas only results flow in classical pipelined systems. Generally speaking, a systolic system is easy to implement because of its regularity and easy to reconfigure (to meet various outside constraints) because of its modularity.

The systolic architectural concept was developed at Carnegie-Mellon University,[1-7] and versions of systolic processors are being designed and built by several industrial and governmental organizations.[8-10] This article reviews the basic principle of systolic architectures and explains why they should result in cost-effective, high-performance special-purpose systems for a wide range of problems.

Key architectural issues in designing special-purpose systems

Roughly, the cycle for developing a special-purpose system can be divided into three phases—task definition, design, and implementation. During task definition, some system performance bottleneck is identified, and a decision on whether or not to resolve it with special-purpose hardware is made. The evaluation required for task definition is most fundamental, but since it is often application-dependent, we will concentrate only on architectural issues related to the design phase and will assume routine implementation.

Simple and regular design. Cost-effectiveness has always been a chief concern in designing special-purpose systems; their cost must be low enough to justify their limited applicability. Costs can be classified as nonrecurring (design) and recurring (parts) costs. Part costs are dropping rapidly due to advances in integrated-circuit technology, but this advantage applies equally to both special-purpose and general-purpose systems. Furthermore, since special-purpose systems are seldom produced in large quantities, part costs are less important than design costs. Hence, the design cost of a special-purpose system must be relatively small for it to be more attractive than a general-purpose approach.

Fortunately, special-purpose design costs can be reduced by the use of appropriate architectures. If a structure can truly be decomposed into a few types of simple substructures or building blocks, which are used repetitively with simple interfaces, great savings can be achieved. This is especially true for VLSI designs where a single chip comprises hundreds of thousands of components. To cope with that complexity, simple and regular designs, similar

Reprinted from *IEEE Computer*, pp. 37-46, January 1982.

to some of the techniques used in constructing large software systems, are essential.[11] In addition, special-purpose systems based on simple, regular designs are likely to be modular and therefore adjustable to various performance goals—that is, system cost can be made proportional to the performance required. This suggests that meeting the architectural challenge for simple, regular designs yields cost-effective special-purpose systems.

Concurrency and communication. There are essentially two ways to build a fast computer system. One is to use fast components, and the other is to use concurrency. The last decade has seen an order of magnitude decrease in the cost and size of computer components but only an incremental increase in component speed.[12] With current technology, tens of thousands of gates can be put in a single chip, but no gate is much faster than its TTL counterpart of 10 years ago. Since the technological trend clearly indicates a diminishing growth rate for component speed, any major improvement in computation speed must come from the concurrent use of many processing elements. The degree of concurrency in a special-purpose system is largely determined by the underlying algorithm. Massive parallelism can be achieved if the algorithm is designed to introduce high degrees of pipelining and multiprocessing. When a large number of processing elements work simultaneously, coordination and communication become significant—especially with VLSI technology where routing costs dominate the power, time, and area required to implement a computation.[13] The issue here is to design algorithms that support high degrees of concurrency, and in the meantime to employ only simple, regular communication and control to enable efficient implementation.

Balancing computation with I/O. Since a special-purpose system typically receives data and outputs results through an attached host, I/O considerations influence overall performance. (The host in this context can mean a computer, a memory, a real-time device, etc. In practice, the special-purpose system may actually input from one "physical" host and output to another.) The ultimate performance goal of a special-purpose system is—and should be no more than—a computation rate that balances the available I/O bandwidth with the host. Since an accurate a priori estimate of available I/O bandwidth in a complex system is usually impossible, the design of a special-purpose system should be modular so that its structure can be easily adjusted to match a variety of I/O bandwidths.

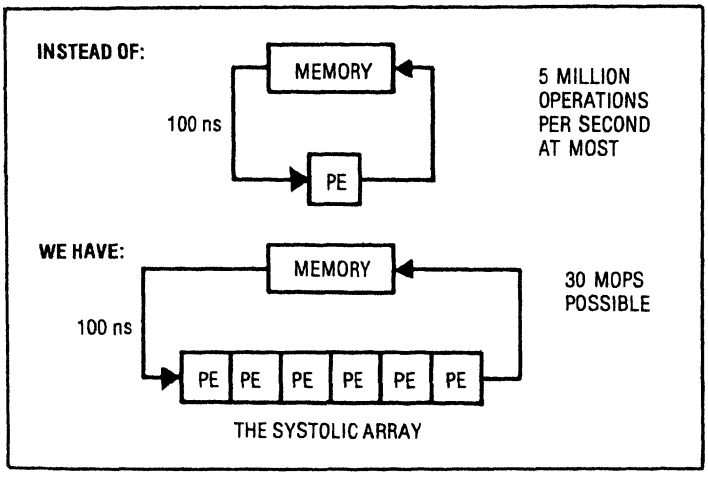


Figure 1. Basic principle of a systolic system.

Suppose that the I/O bandwidth between the host and a special-purpose system is 10 million bytes per second, a rather high bandwidth for present technology. Assuming that at least two bytes are read from or written to the host for each operation, the maximum rate will be only 5 million operations per second, no matter how fast the special-purpose system can operate (see Figure 1). Orders of magnitude improvements on this throughput are possible only if multiple computations are performed per I/O access. However, the repetitive use of a data item requires it to be stored inside the system for a sufficient length of time. Thus, the I/O problem is related not only to the available I/O bandwidth, but also to the available memory internal to the system. The question then is how to arrange a computation together with an appropriate memory structure so that computation time is balanced with I/O time.

The I/O problem becomes especially severe when a large computation is performed on a small special-purpose system. In this case, the computation must be decomposed. Executing subcomputations one at a time may require a substantial amount of I/O to store or retrieve intermediate results. Consider, for example, performing the n-point fast Fourier transform using an S-point device when n is large and S is small. Figure 2 depicts the n-point FFT computation and a decomposition scheme for $n = 16$ and $S = 4$. Note that each subcomputation block is sufficiently small so that it can be handled by the 4-point device. During execution, results of a block must be temporarily sent to the host and later retrieved to be combined with results of other blocks as they become available. With the decomposition scheme shown in Figure 2b, the total number of I/O operations is $O(n \log n/\log S)$. In fact, it has been shown that, to perform the n-point FFT with a device of $O(S)$ memory, at least this many I/O operations are needed for any decomposition scheme.[14] Thus, for the n-point FFT problem, an S-point device cannot achieve more than an $O(\log S)$ speed-up ratio over the conventional $O(n \log n)$ software implementation time, and since it is a consequence of the I/O consideration, this upper bound holds independently of device speed. Similar upper bounds have been established for speed-up ratios achievable by devices for other computations such as sorting and matrix multiplication.[14,15] Knowing the I/O-imposed performance limit helps prevent overkill in the design of a special-purpose device.

In practice, problems are typically "larger" than special-purpose devices. Therefore, questions such as how a computation can be decomposed to minimize I/O, how the I/O requirement is related to the size of a special-purpose system and its memory, and how the I/O bandwidth limits the speed-up ratio achievable by a special-purpose system present another set of challenges to the system architect.

Systolic architectures: the basic principle

As a solution to the above challenges, we introduce systolic architectures, an architectural concept originally proposed for VLSI implementation of some matrix operations.[5] Examples of systolic architectures follow in the next section, which contains a walk-through of a family of designs for the convolution computation.

A systolic system consists of a set of interconnected cells, each capable of performing some simple operation. Because simple, regular communication and control structures have substantial advantages over complicated ones in design and implementation, cells in a systolic system are typically interconnected to form a systolic array or a systolic tree. Information in a systolic system flows between cells in a pipelined fashion, and communication with the outside world occurs only at the "boundary cells." For example, in a systolic array, only those cells on the array boundaries may be I/O ports for the system.

Computational tasks can be conceptually classified into two families—compute-bound computations and I/O-bound computations. In a computation, if the total number of operations is larger than the total number of input and output elements, then the computation is compute-bound, otherwise it is I/O-bound. For example, the ordinary matrix-matrix multiplication algorithm represents a compute-bound task, since every entry in a matrix is multiplied by all entries in some row or column of the other matrix. Adding two matrices, on the other hand, is I/O-bound, since the total number of adds is not larger than the total number of entries in the two matrices. It should be clear that any attempt to speed up an I/O-bound computation must rely on an increase in memory bandwidth. Memory bandwidth can be increased by the use of either fast components (which could be expensive) or interleaved memories (which could create complicated memory management problems). Speeding up a compute-bound computation, however, may often be accomplished in a relatively simple and inexpensive manner, that is, by the systolic approach.

The basic principle of a systolic architecture, a systolic array in particular, is illustrated in Figure 1. By replacing a single processing element with an array of PEs, or cells in the terminology of this article, a higher computation throughput can be achieved without increasing memory bandwidth. The function of the memory in the diagram is analogous to that of the heart; it "pulses" data (instead of blood) through the array of cells. The crux of this approach is to ensure that once a data item is brought out from the memory it can be used effectively at each cell it passes while being "pumped" from cell to cell along the array. This is possible for a wide class of compute-bound computations where multiple operations are performed on each data item in a repetitive manner.

Being able to use each input data item a number of times (and thus achieving high computation throughput with only modest memory bandwidth) is just one of the many advantages of the systolic approach. Other advantages, such as modular expansibility, simple and regular data and control flows, use of simple and uniform cells, elimination of global broadcasting, and fan-in and (possibly) fast response time, will be illustrated in various systolic designs in the next section.

A family of systolic designs for the convolution computation

To provide concrete examples of various systolic structures, this section presents a family of systolic designs for the convolution problem, which is defined as follows:

Given the sequence of weights $\{w_1, w_2, \ldots, w_k\}$ and the input sequence $\{x_1, x_2, \ldots, x_n\}$.

compute the result sequence $\{y_1, y_2, \ldots, y_{n+1-k}\}$ defined by

$$y_i = w_1 x_i + w_2 x_{i+1} + \ldots + w_k x_{i+k-1}$$

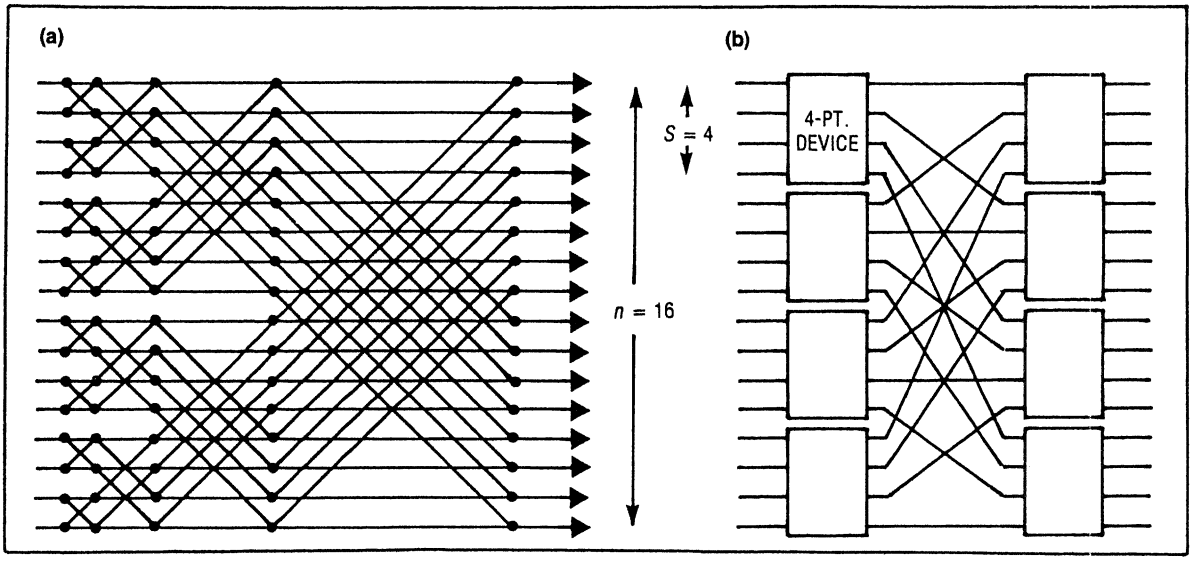


Figure 2. (a) 16-point fast-Fourier-transform graph; (b) decomposing the FFT computation with $n = 16$ and $S = 4$.

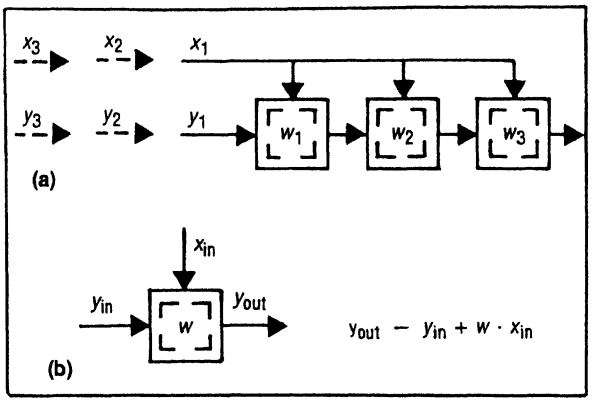


Figure 3. Design B1: systolic convolution array (a) and cell (b) where x_i's are broadcast, w_i's stay, and y_i's move systolically.

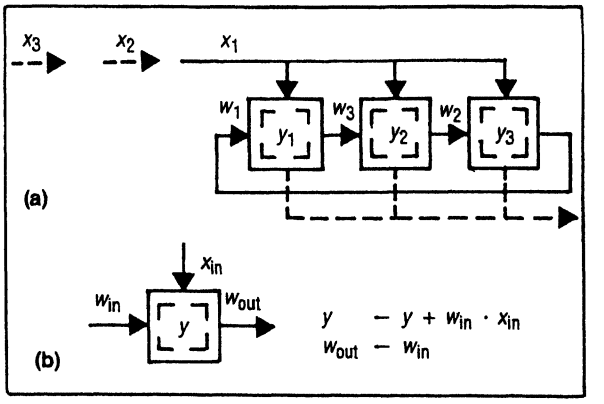


Figure 4. Design B2: systolic convolution array (a) and cell (b) where x_i's are broadcast, y_i's stay, and w_i's move systolically.

We consider the convolution problem because it is a simple problem with a variety of enlightening systolic solutions, because it is an important problem in its own right, and more importantly, because it is representative of a wide class of computations suited to systolic designs. The convolution problem can be viewed as a problem of combining two data streams, w_i's amd x_i's, in a certain manner (for example, as in the above equation) to form a resultant data stream of y_i's. This type of computation is common to a number of computation routines, such as filtering, pattern matching, correlation, interpolation, polynomial evaluation (including discrete Fourier transforms), and polynomial multiplication and division. For example, if multiplication and addition are interpreted as comparison and boolean AND, respectively, then the convolution problem becomes the pattern matching problem.[1] Architectural concepts for the convolution problem can thus be applied to these other problems as well.

The convolution problem is compute-bound, since each input x_i is to be multiplied by each of the k weights. If the x_i is input separately from memory for each multiplication, then when k is large, memory bandwidth becomes a bottleneck, precluding a high-performance solution. As indicated earlier, a systolic architecture resolves this I/O bottleneck by making multiple use of each x_i fetched from the memory. Based on this principle, several systolic designs for solving the convolution problem are described below. For simplicity, all illustrations assume that $k = 3$.

(Semi-) systolic convolution arrays with global data communication. If an x_i, once brought out from the memory, is broadcast to a number of cells, then the same x_i can be used by all the cells. This broadcasting technique is probably one of the most obvious ways to make multiple use of each input element. The opposite of broadcasting is fan-in, through which data items from a number of cells can be collected. The fan-in technique can also be used in a straightforward manner to resolve the I/O bottleneck problem. In the following, we describe systolic designs that utilize broadcasting and fan-in.

Design B1—broadcast inputs, move results, weights stay. The systolic array and its cell definition are depicted in Figure 3. Weights are preloaded to the cells, one at each cell, and stay at the cells throughout the computation. Partial results y_i move systolically from cell to cell in the left-to-right direction, that is, each of them moves over the cell to its right during each cycle. At the beginning of a cycle, one x_i is broadcast to all the cells and one y_i, initialized as zero, enters the left-most cell. During cycle one, $w_1 x_1$ is accumulated to y_1 at the left-most cell, and during cycle two, $w_1 x_2$ and $w_2 x_2$ are accumulated to y_2 and y_1 at the left-most and middle cells, respectively. Starting from cycle three, the final (and correct) values of $y_1, y_2, \ldots$ are output from the right-most cell at the rate of one y_i per cycle. The basic principle of this design was previously proposed for circuits to implement a pattern matching processor[16] and for circuits to implement polynomial multiplication.[17-20]

Design B2—broadcast inputs, move weights, results stay. In design B2 (see Figure 4), each y_i stays at a cell to accumulate its terms, allowing efficient use of available multiplier-accumulator hardware. (Indeed, this design is described in an application booklet for the TRW multiplier-accumulator chips.[21] The weights circulate around the array of cells, and the first weight w_1 is associated with a tag bit that signals the accumulator to output and resets its contents.* In design B1 (Figure 3), the systolic path for moving y_i's may be considerably wider than that for moving w_i's in design B2 because for numerical accuracy y_i's typically carry more bits than w_i's. The use of multiplier-accumulators in design B2 may also help increase precision of the results, since extra bits can be kept in these accumulators with modest cost. Design B1, however, does have the advantage of not requiring a separate bus (or other global network), denoted by a dashed line in Figure 4, for collecting outputs from individual cells.

Design F—fan-in results, move inputs, weights stay. If we consider the vector of weights $(w_k, w_{k-1}, \ldots, w_1)$ as being fixed in space and input vector $(x_n, x_{n-1}, \ldots, x_1)$ as sliding over the weights in the left-to-right direction, then the convolution problem is one that computes the inner product of the weight vector and the section of input vector it overlaps. This view suggests the systolic array

*To avoid complicated pictures, control structures such as the use of tag bits to gate outputs from cells are omitted from the diagrams of this article.

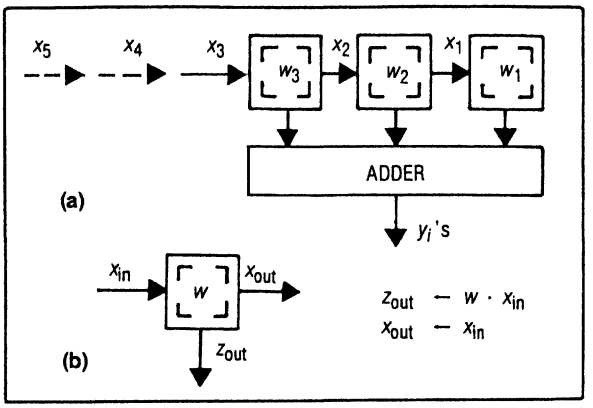


Figure 5. Design F: systolic convolution array (a) and cell (b) where w_i's stay, x_i's move systolically, and y_i's are formed through the fan-in of results from all the cells.

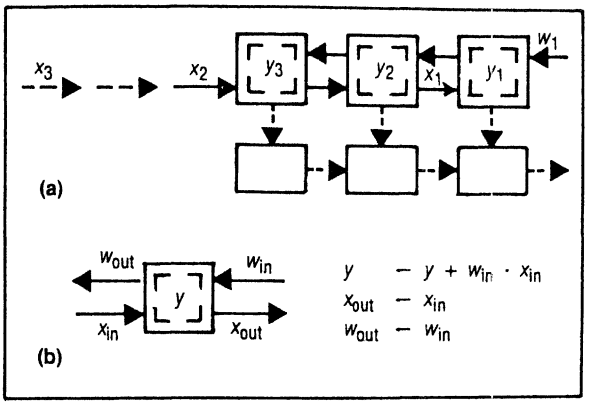


Figure 6. Design R1: systolic convolution array (a) and cell (b) where y_i's stay and x_i's and y_i's move in opposite directions systolically.

shown in Figure 5. Weights are preloaded to the cells and stay there throughout the computation. During a cycle, all x_i's move one cell to the right, multiplications are performed at all cells simultaneously, and their results are fanned-in and summed using an adder to form a new y_i. When the number of cells, k, is large, the adder can be implemented as a pipelined adder tree to avoid large delays in each cycle. Designs of this type using unbounded fan-in have been known for quite a long time, for example, in the context of signal processing[33] and in the context of pattern matching.[43]

(Pure-) systolic convolution arrays without global data communication. Although global broadcasting or fan-in solves the I/O bottleneck problem, implementing it in a modular, expandable way presents another problem. Providing (or collecting) a data item to (or from) all the cells of a systolic array, during each cycle, requires the use of a bus or some sort of tree-like network. As the number of cells increases, wires become long for either a bus or tree structure; expanding these non-local communication paths to meet the increasing load is difficult without slowing down the system clock. This engineering difficulty of extending global networks is significant at chip, board, and higher levels of a computer system. Fortunately, as will be demonstrated below, systolic convolution arrays without global data communication do exist. Potentially, these arrays can be extended to include an arbitrarily large number of cells without encountering engineering difficulties (the problem of synchronizing a large systolic array is discussed later).

Design R1—results stay, inputs and weights move in opposite directions. In design R1 (see Figure 6) each partial result y_i stays at a cell to accumulate its terms. The x_i's and w_i's move systolically in opposite directions such that when an x meets a w at a cell, they are multiplied and the resulting product is accumulated to the y staying at that cell. To ensure that each x_i is able to meet every w_i, consecutive x_i's on the x data stream are separated by two cycle times and so are the w_i's on the w data stream.

Like design B2, design R1 can make efficient use of available multiplier-accumulator hardware; it can also use a tag bit associated with the first weight, w_1, to trigger the output and reset the accumulator contents of a cell.

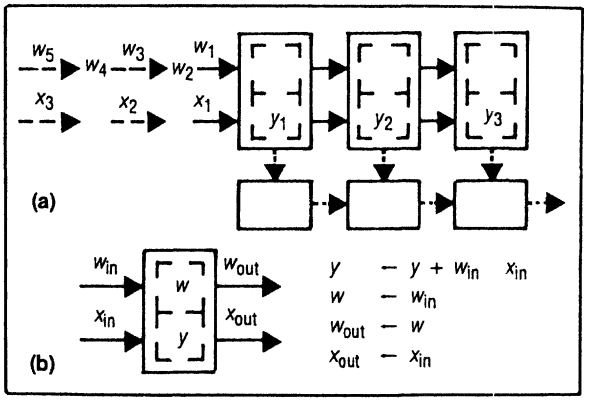


Figure 7. Design R2: systolic convolution array (a) and cell (b) where y_i's stay and x_i's and w_i's both move in the same direction but at different speeds.

Design R1 has the advantage that it does not require a bus, or any other global network, for collecting output from cells; a systolic output path (indicated by broken arrows in Figure 6) is sufficient. Because consecutive w_i's are well separated by two cycle times, a potential conflict—that more than one y_i may reach a single latch on the systolic output path simultaneously—cannot occur. It can also be easily checked that the y_i's will output from the systolic output path in the natural ordering $y_1, y_2, \ldots$. The basic idea of this design, including that of the systolic output path, has been used to implement a pattern matching chip.[1]

Notice that in Figure 6 only about one-half the cells are doing useful work at any time. To fully utilize the potential throughput, two independent convolution computations can be interleaved in the same systolic array, but cells in the array would have to be modified slightly to support the interleaved computation. For example, an additional accumulator would be required at each cell to hold a temporary result for the other convolution computation.

Design R2—results stay, inputs and weights move in the same direction but at different speeds. One version of design R2 is illustrated in Figure 7. In this case both the x and w data streams move from left to right systolically, but the x_i's move twice as fast as the w_i's. More precisely,

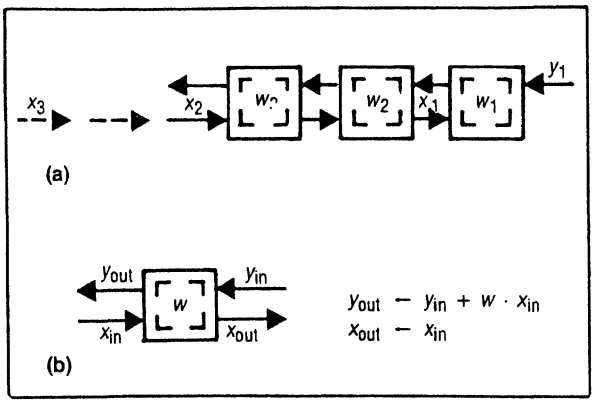


Figure 8. Design W1: systolic convolution array (a) and cell (b) where w_i's stay and x_i's and y_i's move systolically in opposite directions.

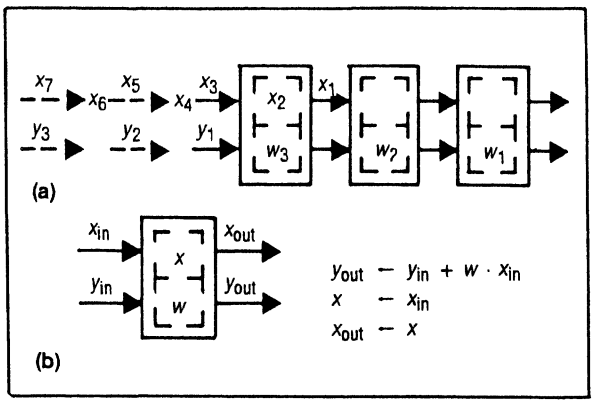


Figure 9. Design W2: systolic convolution array (a) and cell (b) where w_i's stay and x_i's and y_i's both move systolically in the same direction but at different speeds.

each w_i stays inside every cell it passes for one extra cycle, thus taking twice as long to move through the array as any x_i. In this design, multiplier-accumulator hardware can be used effectively and so can the tag bit method to signal the output of the accumulator contents at each cell. Compared to design R1, this design has the advantage that all cells work all the time when performing a single convolution, but it requires an additional register in each cell to hold a w value. This algorithm has been used for implementing a pipeline multiplier.[22]

There is a dual version of design R2; we can have the w_i's move twice as fast as the x_i's. To create delays for the x data stream, this dual design requires a register in each cell for storing an x rather than a w value. For circumstances where the w_i's carry more bits than the x_i's, the dual design becomes attractive.

Design W1—weights stay, inputs and results move in opposite directions. In design W1 (and design W2, below), weights stay, one at each cell, but results and inputs move systolically. These designs are not geared to the most effective use of available multiplier-accumulator hardware, but for some other circumstances they are potentially more efficient than the other designs. Because the same set of weights is used for computing all the y_i's and different sets of the x_i's are used for computing different y_i's, it is natural to have the w_i's preloaded to the cells and stay there, and let the x_i's and the y_i's move along the array. We will see some advantages of this arrangement in the systolic array depicted in Figure 8, which is a special case of a proposed systolic filtering array.[3] This design is fundamental in the sense that it can be naturally extended to perform recursive filtering[2,3] and polynomial division.[23]

In design W1, the w_i's stay and the x_i's and y_i's move systolically in opposite directions. Similar to design R1, consecutive x_i's and y_i's are separated by two cycle times. Note that because the systolic path for moving the y_i's already exists, there is no need for another systolic output path as in designs R1 and R2. Furthermore, for each i, y_i outputs from the left-most cell during the same cycle as its last input, x_{i+k-1} (or x_{i+2} for $k = 3$), enters that cell. Thus, this systolic array is capable of outputting a y_i every two cycle times with constant response time. Design W1, however, suffers from the same drawback as design R1, namely, only approximately one-half the cells work at any given time unless two independent convolution computations are interleaved in the same array. The next design, like design R2, overcomes this shortcoming by having both the x_i's and y_i's move in the same direction but at different speeds.

Design W2—weights stay, inputs and results move in the same direction but at different speeds. With design W2 (Figure 9) all the cells work all the time, but it loses one advantage of design W1, the constant response time. The output of y_i now takes place k cycles after the last of its inputs starts entering the left-most cell of the systolic array. This design has been extended to implement 2-D convolutions,[6,24] where high throughputs rather than fast responses are of concern. Similar to design R1, design W2 has a dual version for which the x_i's move twice as fast as the y_i's.

Remarks. The designs presented above by no means exhaust all the possible systolic designs for the convolution problem. For example, it is possible to have systolic designs where results, weights, and inputs all move during each cycle. It could also be advantageous to include inside each cell a "cell memory" capable of storing a set of weights. With this feature, using a systolic control (or address) path, weights can be selected on-the-fly to implement interpolation or adaptive filtering.[24] Moreover, the flexibility introduced by the cell memories and systolic control can make the same systolic array implement different functions. Indeed, the ESL systolic processor[8,10] utilizes cell memories to implement multiple functions including convolution and matrix multiplication.

Once one systolic design is obtained for a problem, it is likely that a set of other systolic designs can be derived similarly. The challenge is to understand precisely the strengths and drawbacks of each design so that an appropriate design can be selected for a given environment. For example, if there are more weights than cells, it's useful to know that a scheme where partial results stay generally requires less I/O than one where partial results move, since the latter scheme requires partial results to be input and output many times. A single multiplier-accumulator hardware component often represents a cost-effective implementation of the multiplier and adder

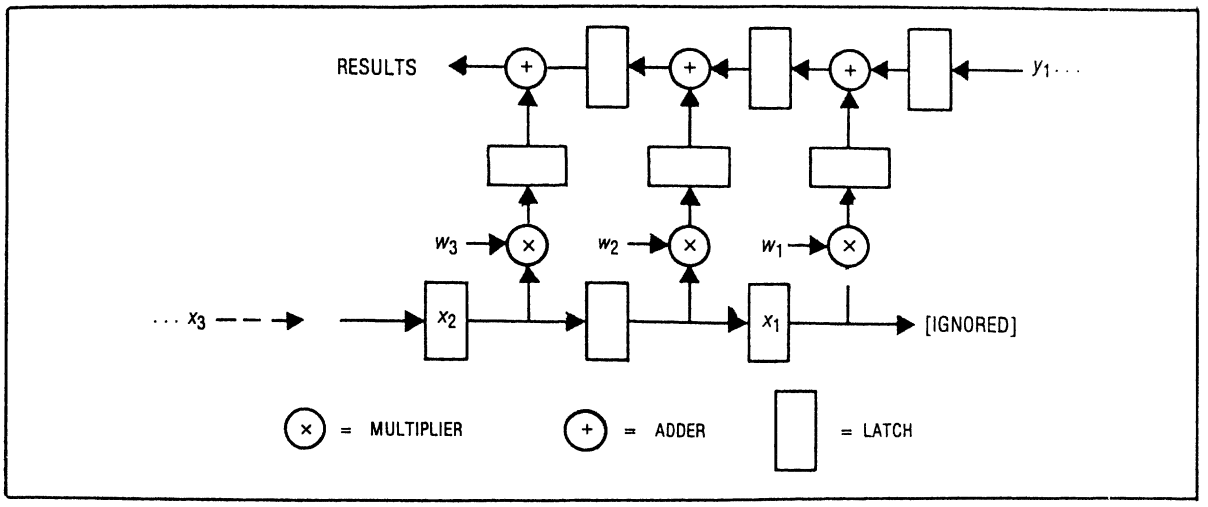


Figure 10. Overlapping the executions of multiply and add in design W1.

needed by each cell of a systolic convolution array. However, for improving throughput, sometimes it may be worthwhile to implement multiplier and adder separately to allow overlapping of their executions. Figure 10 depicts such a modification to design W1. Similar modifications can be made to other systolic convolution arrays. Another interesting scenario is the following one. Suppose that one or several cells are to be implemented directly with a single chip and the chip pin bandwidth is the implementation bottleneck. Then, since the basic cell of some semi-systolic convolution arrays such as designs B1 and F require only three I/O ports, while that of a pure-systolic convolution array always requires four, a semi-systolic array may be preferable for saving pins, despite the fact that it requires global communication.

Criteria and advantages

Having described a family of systolic convolution arrays, we can now be more precise in suggesting and evaluating criteria for the design of systolic structures.

(1) *The design makes multiple use of each input data item.* Because of this property, systolic systems can achieve high throughputs with modest I/O bandwidths for outside communication. To meet this criterion, one can either use global data communications, such as broadcast and unbounded fan-in, or have each input travel through an array of cells so that it is used at each cell. For modular expansibility of the resulting system, the second approach is preferable.

(2) *The design uses extensive concurrency.* The processing power of a systolic architecture comes from concurrent use of many simple cells rather than sequential use of a few powerful processors as in many conventional architectures. Concurrency can be obtained by pipelining the stages involved in the computation of each single result (for example, design B1), by multiprocessing many results in parallel (designs R1 and R2), or by both. For some designs, such as W1, it is possible to completely overlap I/O and computation times to further increase concurrency and provide constant-time responses.

To a given problem there could be both one- and two-dimensional systolic array solutions. For example, two-dimensional convolution can be performed by a one-dimensional systolic array[24,25] or a two-dimensional systolic array.[6] When the memory speed is more than cell speed, two-dimensional systolic arrays such as those depicted in Figure 11 should be used. At each cell cycle, all the I/O ports on the array boundaries can input or output data items to or from the memory; as a result, the available memory bandwidth can be fully utilized. Thus, the choice of a one- or two-dimensional scheme is very dependent on how cells and memories will be implemented.

As in one-dimensional systolic arrays, data in two-dimensional arrays may flow in multiple directions and at multiple speeds. For examples of two-dimensional systolic arrays, see Guibas et al.[26] and Kung and Lehman[4] (type R), Kung and Leiserson[5] and Weiser and Davis[27] (type H), and Bojanczyk et al.[28] and Gentleman and Kung[29] (type T). In practice, systolic arrays can be chained together to form powerful systems such as the one depicted in Figure 12, which is capable of producing on-the-fly the least-squares fit to all the data that have arrived up to any given moment.[29]

For the systolic structures discussed in the preceding section, computations are pipelined over an array of cells. To permit even higher concurrency, it is sometimes possible to introduce another level of pipelining by allowing the operations inside the cells themselves to be pipelined. (Note that pipelined arithmetic units will become increasingly common as VLSI makes the extra circuits needed for

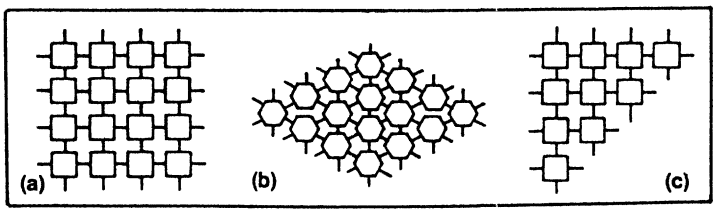


Figure 11. Two-dimensional systolic arrays: (a) type R, (b) type H, and (c) type T.

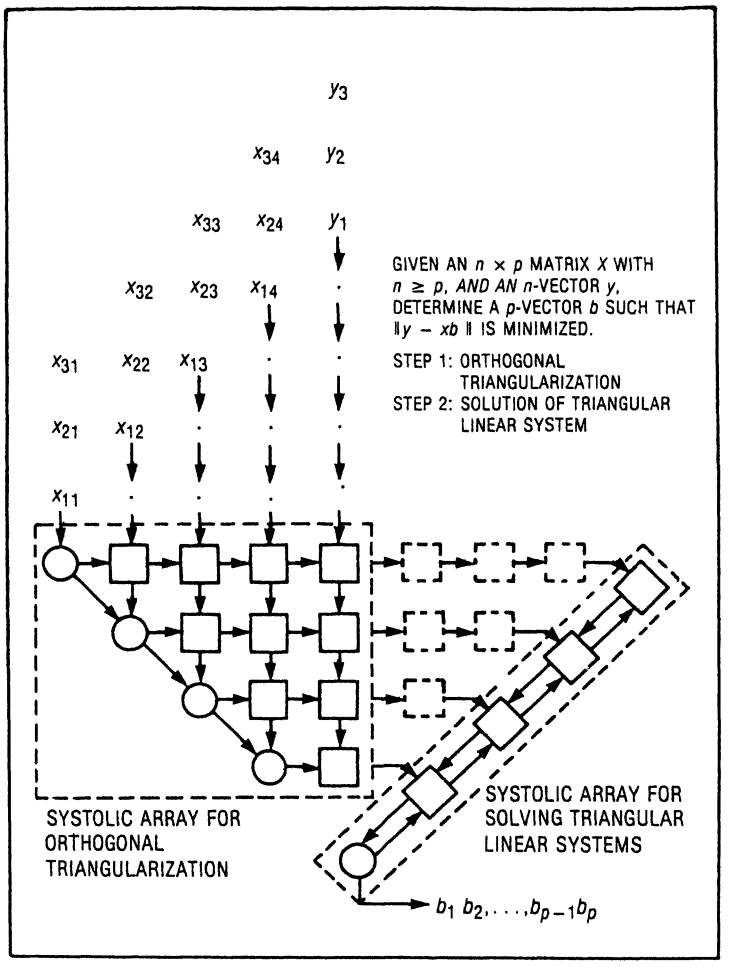


Figure 12. On-the-fly least-squares solutions using one- and two-dimensional systolic arrays, with $p = 4$.

staging affordable.) Both designs W1 and W2 support two-level pipelining.[25] Since system cycle time is the time of a stage of a cell, rather than the whole cell cycle time, two-level pipelined systolic systems significantly improve throughput.

(3) *There are only a few types of simple cells.* To achieve performance goals, a systolic system is likely to use a large number of cells. The cells must be simple and of only a few types to curtail design and implementation costs, but exactly how simple is a question that can only be answered on a case by case basis. For example, if a systolic system consisting of many cells is to be implemented on a single chip, each cell should probably contain only simple logic circuits plus a few words of memory On the other hand, for board implementations each cell could reasonably contain a high-performance arithmetic unit plus a few thousand words of memory. There is, of course, always a trade-off between cell simplicity and flexibility.

(4) *Data and control flows are simple and regular.* Pure systolic systems totally avoid long-distance or irregular wires for data communication. The only global communication (besides power and ground) is the system clock. Of course, self-timed schemes can be used instead for synchronizing neighboring cells, but efficient implementations of self-timed protocols may be difficult. Fortunately, for any one-dimensional systolic array, a global clock parallel to the array presents no problems, even if the array is arbitrarily long. The systolic array (with data flowing in either one or opposite directions) will operate correctly despite the possibility of a large clock skew between its two ends.[30] However, large two-dimensional arrays may require slowdown of the global clock to compensate for clock skews. Except for this possible problem in the two-dimensional case, systolic designs are completely modular and expandable; they present no difficult synchronization or resource conflict problems. Software overhead associated with operations such as address indexing are totally eliminated in systolic systems. This advantage alone can mean a substantial performance improvement over conventional general-purpose computers. Simple, regular control and communication also imply simple, area-efficient layout or wiring—an important advantage in VLSI implementation.

In summary, systolic designs based on these criteria are *simple* (a consequence of properties 3 and 4), *modular* and *expandable* (property 4), and yield *high performance* (properties 1, 2, and 4). They therefore meet the architectural challenges for special-purpose systems. A unique characteristic of the systolic approach is that as the number of cells expands the system cost and performance increase proportionally, provided that the size of the underlying problem is sufficiently large. For example, a systolic convolution array can use an arbitrarily large number of cells cost-effectively, if the kernel size (that is, the number of weights) is large. This is in contrast to other parallel architectures which are seldom cost-effective for more than a small number of processors. From a user's point of view, a systolic system is easy to use—he simply pumps in the input data and then receives the results either on-the-fly or at the end of the computation.

Summary and concluding remarks

Bottlenecks to speeding up a computation are often due to limited system memory bandwidths, so called *von Neumann bottlenecks,* rather than limited processing capabilities per se. This problem can certainly be expected for I/O-bound computations, but with a conventional architectural approach, it may be present even for compute-bound computations. For every operation, at least one or two operands have to be fetched (or stored) from (or to) memory, so the total amount of I/O is proportional to the number of operations rather than the number of inputs and outputs. Thus, a problem that was originally compute-bound can become I/O-bound during its execution. This unfortunate situation is the result of a mismatch between the computation and the architecture. Systolic architectures, which ensure multiple computations per memory access, can speed up compute-bound computations without increasing I/O requirements.

The convolution problem is just one of many compute-bound computations that can benefit from the systolic approach. Systolic designs using (one- or two-dimensional) array or tree structures are available for the following regular, compute-bound computations.

Signal and image processing:

- FIR, IIR filtering, and 1-D convolution[2,3,31];

- 2-D convolution and correlation[6,8,10,24,25];
- discrete Fourier transform[2,3];
- interpolation[24];
- 1-D and 2-D median filtering[32]; and
- geometric warping.[24]

Matrix arithmetic:
- matrix-vector multiplication[5];
- matrix-matrix multiplication[5,27];
- matrix triangularization (solution of linear systems, matrix inversion)[5,29];
- QR decomposition (eigenvalue, least-square computations)[28,29]; and
- solution of triangular linear systems.[5]

Non-numeric applications:
- data structures—stack and queue,[34] searching,[15,35,36] priority queue,[7] and sorting[7,15];
- graph algorithms—transitive closure,[26] minimum spanning trees,[37] and connected components[38];
- language recognition—string matching[1] and regular expression[39];
- dynamic programming[26];
- encoders (polynomial division)[23]; and
- relational data-base operations.[4,40]

In general, systolic designs apply to any compute-bound problem that is regular—that is, one where repetitive computations are performed on a large set of data. Thus, the above list is certainly not complete (and was not intended to be so). Its purpose is to provide a range of typical examples and possible applications. After studying several of these examples, one should be able to start designing systolic systems for one's own tasks. Some systolic solutions can usually be found without too much difficulty. (I know of only one compute-bound problem that arises naturally in practice for which no systolic solution is known, and I cannot prove that a systolic solution is impossible.) This is probably due to the fact that most compute-bound problems are inherently regular in the sense that they are definable in terms of simple recurrences. Indeed, the notion of systolicity is implicit in quite a few previously known special-purpose designs, such as the sorting[41] and multiply designs.[22] This should not come as a surprise; as we have been arguing systolic structures are essential for obtaining any cost-effective, high-performance solution to compute-bound problems. It is useful, however, to make the systolic concept explicit so that designers will be conscious of this important design criterion.

While numerous systolic designs are known today, the question of their automatic design is still open. But recent efforts show significant progress.[27,42] Leiserson and Saxe, for instance, can convert some semi-systolic systems involving broadcasting or unbounded fan-in into pure-systolic systems without global data communication.[42] A related open problem concerns the specification and verification of systolic structures. For implementation and proof purposes, rigorous notation other than informal pictures (as used in this article) for specifying systolic designs is desirable.

With the development of systolic architectures, more and more special-purpose systems will become feasible—especially systems that implement fixed, well-understood computation routines. But the ultimate goal is effective use of systolic processors in general computing environments to off-load regular, compute-bound computations. To achieve this goal further research is needed in two areas. The first concerns the system integration: we must provide a convenient means for incorporating high-performance systolic processors into a complete system and for understanding their effective utilization from a system point of view. The second research area is to specify building-blocks for a variety of systolic processors so that, once built, these building blocks can be programmed to form basic cells for a number of systolic systems. The building-block approach seems inherently suitable to systolic architectures since they tend to use only a few types of simple cells. By combining these building-blocks regularly, systolic systems geared to different applications can be obtained with little effort. ■

Acknowledgment

Parts of this article were written while the author was on leave from Carnegie-Mellon University with ESL's Advanced Processor Technology Laboratory in San Jose, California, January-August 1981. Most of the research reported here was carried out at CMU and was supported in part by the Office of Naval Research under Contracts N00014-76-C-0370, NR 044-422 and N00014-80-C-0236, NR 048-659, in part by the National Science Foundation under Grant MCS 78-236-76, and in part by the Defense Advanced Research Projects Agency (DoD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539.

References

1. M. J. Foster and H. T. Kung, "The Design of Special-Purpose VLSI Chips," *Computer,* Vol. 13, No. 1, Jan. 1980, pp. 26-40.
2. H. T. Kung, "Let's Design Algorithms for VLSI Systems," *Proc. Conf. Very Large Scale Integration: Architecture, Design, Fabrication,* California Institute of Technology, Jan. 1979, pp. 65-90.
3. H. T. Kung, "Special-Purpose Devices for Signal and Image Processing: An Opportunity in VLSI," *Proc. SPIE, Vol. 241, Real-Time Signal Processing III,* Society of Photo-Optical Instrumentation Engineers, July 1980, pp. 76-84.
4. H. T. Kung and P. L. Lehman, "Systolic (VLSI) Arrays for Relational Database Operations," *Proc. ACM-Sigmod 1980 Int'l Conf. Management of Data,* May 1980, pp. 105-116.
5. H. T. Kung and C. E. Leiserson, "Systolic Arrays (for VLSI)," *Sparse Matrix Proc. 1978,* Society for Industrial and Applied Mathematics, 1979, pp. 256-282.
6. H. T. Kung and S. W. Song, *A Systolic 2-D Convolution Chip,* Technical Report CMU-CS-81-110, Carnegie-Mellon University Computer Science Dept., Mar. 1981.
7. C. E. Leiserson, "Systolic Priority Queues," *Proc. Conf. Very Large Scale Integration: Architecture, Design, Fabrication,* California Institute of Technology, Jan. 1979, pp. 199-214.
8. J. Blackmer, P. Kuekes, and G. Frank, "A 200 MOPS systolic processor," *Proc. SPIE, Vol. 298, Real-Time Signal Processing IV,* Society of Photo-Optical Instrumentation Engineers, 1981.

9. K. Bromley, J. J. Symanski, J. M. Speiser, and H. J. Whitehouse, "Systolic Array Processor Developments," in *VLSI Systems and Computations*, H. T. Kung, R. F. Sproull, and G. L. Steele, Jr., (eds.), Carnegie-Mellon University, Computer Science Press, Oct. 1981, pp. 273-284.
10. D. W. L. Yen and A. V. Kulkarni, "The ESL Systolic Processor for Signal and Image Processing," *Proc. 1981 IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Management*, Nov. 1981, pp. 265-272.
11. C. A. Mead and L. A. Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading, Mass., 1980.
12. R. N. Noyce, "Hardware Prospects and Limitations," in *The Computer Age: A Twenty-Year Review*, M. L. Dertouzos and J. Moses (eds.), IEEE Press, 1979, pp. 321-337.
13. I. E. Sutherland and C. A. Mead, "Microelectronics and Computer Science," *Scientific American*, Vol. 237, No. 3, Sept. 1977, pp. 210-228.
14. J-W. Hong and H. T. Kung, "I/O Complexity: The Red-Blue Pebble Game," *Proc. 13th Annual ACM Symp. Theory of Computing*, ACM Sigact, May 1981, pp. 326-333.
15. S. W. Song, *On a High-Performance VLSI Solution to Database Problems*, PhD dissertation, Carnegie-Mellon University, Computer Science Dept., July 1981.
16. A. Mukhopadhyay, "Hardware Algorithms for Nonnumeric Computation," *IEEE Trans. Computers*, Vol. C-28, No. 6, June 1979, pp. 384-394.
17. D. Cohen, *Mathematical Approach to Computational Networks*, Technical Report ISI/RR-78-73, University of Southern California, Information Sciences Institute, 1978.
18. D. A. Huffman, "The Synthesis of Linear Sequential Coding Networks," in *Information Theory*, C. Cherry (ed.), Academic Press, 1957, pp. 77-95.
19. K. Y. Liu, "Architecture for VLSI Design of Reed-Solomon Encoders," *Proc. Second Caltech VLSI Conf.* Jan. 1981.
20. W. W. Peterson and E. J. Weldon, Jr., *Error-Correcting Codes*, MIT Press, Cambridge, Mass., 1972.
21. L. Schirm IV, *Multiplier-Accumulator Application Notes*, TRW LSI Products, Jan. 1980.
22. R. F. Lyon, "Two's Complement Pipeline Multipliers," *IEEE Trans. Comm.*, Vol. COM-24, No. 4, Apr. 1976, pp. 418-425.
23. H. T. Kung, "Use of VLSI in Algebraic Computation: Some Suggestions," *Proc. 1981 ACM Symp. Symbolic and Algebraic Computation*, ACM Sigsam, Aug. 1981, pp. 218-222.
24. H. T. Kung and R.L. Picard, "Hardware Pipelines for Multi-Dimensional Convolution and Resampling," *Proc. 1981 IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Management*, Nov. 1981, pp. 273-278.
25. H. T. Kung, L. M. Ruane, and D. W. L. Yen, "A Two-Level Pipelined Systolic Array for Convolutions," in *VLSI Systems and Computations*, H. T. Kung, R. F. Sproull, and G. L. Steele, Jr. (eds.), Carnegie-Mellon University, Computer Science Press, Oct. 1981, pp. 255-264.
26. L. J. Guibas, H. T. Kung, and C. D. Thompson, "Direct VLSI Implementation of Combinatorial Algorithms," *Proc. Conf. Very Large Scale Integration: Architecture, Design, Fabrication*, California Institute of Technology, Jan. 1979, pp. 509-525.
27. U. Weiser and A. Davis, "A Wavefront Notation Tool for VLSI Array Design," in *VLSI Systems and Computations*, H. T. Kung, R. F. Sproull, and G. L. Steele, Jr. (eds.), Carnegie-Mellon University, Computer Science Press, Oct. 1981, pp. 226-234.
28. A. Bojanczyk, R. P. Brent, and H. T. Kung, *Numerically Stable Solution of Dense Systems of Linear Equations Using Mesh-Connected Processors*, Technical Report, Carnegie-Mellon University, Computer Science Dept. May 1981.
29. W. M. Gentleman and H. T. Kung, "Matrix Triangularization by Systolic Arrays," *Proc. SPIE, Vol. 298, Real-Time Signal Processing IV*, Society of Photo-optical Instrumentation Engineers, 1981.
30. A. Fisher and H. T. Kung, CMU Computer Science Dept. technical report, Jan. 1982.
31. P. R. Cappello and K. Steiglitz, "Digital Signal Processing Applications of Systolic Algorithms," in *VLSI Systems and Computations*, H. T. Kung, R. F. Sproull, and G. L. Steele, Jr. (eds.), Carnegie-Mellon University, Computer Science Press, Oct. 1981, pp. 245-254.
32. A. Fisher, "Systolic Algorithms for Running Order Statistics in Signal and Image Processing," in *VLSI Systems and Computations*, H. T. Kung, R. F. Sproull, G. L. Steele, Jr. (eds.), Carnegie-Mellon University, Computer Science Press, Oct. 1981, pp. 265-272.
33. E. E. Swartzlander, Jr., and B. K. Gilbert, "Arithmetic for Ultra-High-Speed Tomography," *IEEE Trans. Computers*, Vol. C-29, No. 5, May, 1980, pp. 341-354.
34. L. J. Guibas and F. M. Liang, "Systolic Stacks, Queues, and Counters," *Proc. Conf. Advanced Research in VLSI*, MIT, Jan. 1982.
35. J. L. Bentley and H. T. Kung, "A Tree Machine for Searching Problems," *Proc. 1979 Int'l Conf. Parallel Processing*, Aug. 1979, pp. 257-266. Also available as a Carnegie-Mellon University Computer Science Dept. technical report, Aug. 1979.
36. T. Ottmann, A. L. Rosenberg, and L. J. Stockmeyer, "A Dictionary Machine (for VLSI)," Technical Report RC 9060 (#39615), IBM T. J. Watson Research Center, Yorktown Heights, N.Y., 1981.
37. J. L. Bentley, *A Parallel Algorithm for Constructing Minimum Spanning Trees*," *Journal of Algorithms*, Jan. 1980, pp. 51-59.
38. C. Savage, "A Systolic Data Structure Chip for Connectivity Problems," in *VLSI Systems and Computations*, H. T. Kung, R. F. Sproull, and G. L. Steele, Jr., (eds.), Carnegie-Mellon University, Computer Science Press, Oct. 1981, pp. 296-300.
39. M. J. Foster and H. T. Kung, "Recognize Regular Languages With Programmable Building-Blocks," in *VLSI 81*, Academic Press, Aug. 1981, pp. 75-84.
40. P. L. Lehman, "A Systolic (VLSI) Array for Processing Simple Relational Queries," in *VLSI Systems and Computations*, H. T Kung, R. F. Sproull, and G. I. Steele, Jr. (eds.), Carnegie-Mellon University, Computer Science Press, Oct. 1981, pp. 285-295.
41. S. Todd, "Algorithm and Hardware for a Merge Sort Using Multiple Processors," *IBM J. Research and Development*, Vol. 22, No. 5, 1978, pp. 509-517.
42. C. E. Leiserson and J. B. Saxe, "Optimizing Synchronous Systems," *Proc. 22nd Annual Symp. Foundations of Computer Science*, IEEE Computer Society, Oct. 1981, pp. 23-36.
43. C. A. Mead et al., "128-Bit Multicomparator," *IEEE J. Solid-State Circuits*, Vol. SC-11, No. 5, Oct. 1976, pp. 692-695.

On Supercomputing with Systolic/Wavefront Array Processors

SUN-YUAN KUNG, SENIOR MEMBER, IEEE

Invited Paper

In many scientific and signal processing applications, there are increasing demands for large-volume and/or high-speed computations which call for not only high-speed computing hardware, but also for novel approaches in computer architecture and software techniques in future supercomputers. Tremendous progress has been made on several promising parallel architectures for scientific computations, including a variety of digital filters, fast Fourier transform (FFT) processors, data-flow processors, systolic arrays, and wavefront arrays. This paper describes these computing networks in terms of signal-flow graphs (SFG) or data-flow graphs (DFG), and proposes a methodology of converting SFG computing networks into synchronous systolic arrays or data-driven wavefront arrays. Both one- and two-dimensional arrays are discussed theoretically, as well as with illustrative examples. A wavefront-oriented programming language, which describes the (parallel) data flow in systolic/wavefront-type arrays, is presented. The structural property of parallel recursive algorithms points to the feasibility of a Hierarchical Iterative Flow-Graph Design (HIFD) of VLSI Array Processors. The proposed array processor architectures, we believe, will have significant impact on the development of future supercomputers.

I. INTRODUCTION

The increasing demands for high-performance signal processing and scientific computations indicate the need for tremendous computing capability, in terms of both volume and speed. The availability of low-cost, high-density, fast processing/memory devices will presage a major breakthrough in future supercomputer designs, especially in the design of highly concurrent processors.

Current parallel computers can be characterized into three structural classes: vector processors, multiprocessor systems, and array processors [13]. The first two classes belong to the general-purpose computer domain. The development of these systems requires a complicated design of control units and optimized schemes for allocation of machine resources. The third class, however, belongs to the domain of special-purpose computers. The design of such systems requires a broad knowledge of the relationship between parallel-computing algorithms and optimal-computing hardware and software structures.

It is this last class that we shall focus upon, since it offers a promising solution to meet real-time processing requirements. Especially, locally interconnected computing networks, such as systolic and wavefront arrays, are well suited to efficiently implement a major class of signal processing algorithms due to their massive parallelism and regular data flow [15], [17]. Such architectures promise real-time solutions to a large variety of advanced computational tasks.

This paper first discusses some important design considerations for massively parallel VLSI array processors and the algorithmic background of these array processors. This will lead to a systematic software/hardware design approach. Specifically, we discuss the methodology of imposing systolic architectures and/or data-flow computing onto signal-flow graph computing networks. The concept of computational wavefronts, which leads to systolic-type and wavefront-type architectures, is reviewed. A wavefront programming language is proposed to broaden the applications of the wavefront/systolic type arrays.

A. Architectural Considerations in Array Processor Design

There are many important issues in designing array processor systems, such as processor interconnection, system clocking, and modularity [19].

Interconnection in massively parallel array processors is the most critical issue of the system design, since communication is very expensive in terms of area, power, and time consumption [26]. Therefore, communication has to be restricted to *localized interconnections*. To avoid global interconnections, local and regular data movements are strongly preferred. This is the most salient characteristic of systolic and wavefront arrays.

The clocking scheme is also a very critical issue. In the globally synchronous scheme, there is a global clock network which distributes the clock signal over the entire array. For very large systems, the clock skew incurred in global clock distribution is a nontrivial factor, causing unnecessary slowdown in the clock rate [18], [10]. Under this circumstance, the self-timed scheme is more preferable.

Large design of layout costs suggest using repetitive modular structures, i.e., a few different types of simple (and often standard) cells. Thus we have to identify the primitives that can be implemented efficiently and optimally realize the potential of new device technologies.

Programmable processor modules (as opposed to dedicated modules) are favored due to cost-effectiveness considerations. The high cost of designing such modules may be amortized over a broader applicational domain. Indeed, a major portion of scientific computations can be reduced to a basic set of matrix operations and other related algo-

Manuscript received February 9, 1984; revised March 5, 1984. This research was supported in part by the Office of Naval Research under Contracts N00014-81-K-0191 and N00014-83-C-0377 and by the National Science Foundation under Grant ECS-82-12479.

The author is with the Department of Electrical Engineering, University of Southern California, Los Angeles, CA 90089, USA.

Reprinted from *Proceedings of the IEEE*, pp. 867-884, July 1984.

rithms. These should be exploited in order to simplify the hardware module.

B. Mapping Parallel Algorithms onto Locally Interconnected Computing Networks

There are quite a few software packages for scientific computation and image/signal processing algorithms available today. For example, LINPAK [9] and EISPAK [29] are popular packages for many scientific computations, especially those using various types of matrix operations (such as). However, the execution time of these algorithms, running on conventional computers, is often too slow for real-time applications. Fortunately, VLSI and other new techniques have made high-speed parallel processing economical and feasible. When mapping these algorithms onto parallel processors, typical questions often raised are: "How to fully utilize the inherent concurrency in these algorithms?" "How are these algorithms best implemented in hardware?" "What kind of array processor(s) should one turn to for a specific application?"

After examining most of the algorithms collected in the aforementioned packages, some prominent traits surface, such as localized operations, intensive computation, and matrix operands. The common features of these algorithms should be exploited to facilitate the design of array processors for signal processing applications.

An array processor is composed of an array of processor elements (PE) with direct (static) or indirect (dynamic) interconnections, including linear, orthogonal, hexagonal, tree, perfect-shuffle, or other types of structures. The most critical issue is communication, i.e., moving data between PE's in a large-scale interconnection network.

Correspondingly, a communication-oriented analysis on parallel algorithms will be most useful for mapping algorithms onto the arrays. To conform with the constraints imposed by VLSI, this paper will emphasize a special class of algorithms, i.e., *recursive* and *locally dependent* algorithms.[1]

For proper communication in an interconnected computing network, each PE in the array should know

a) **where** to send (or fetch) data, and
b) **when** to send (or fetch) data.

When mapping a locally recursive algorithm onto a computing network, it allows a simple solution to the question "a) **where** to send the data?" because the data movements can be confined to nearest neighbor PE's. Therefore, locally interconnected computing networks will suffice to execute the algorithm with high performance.

The conventional approach to the second question "b) **when** to send the data?" is to use a globally synchronous scheme, where the timing is controlled by a sequence of "beats" [30]. A prominent example is the systolic array [16], [15] However, locality can have two meanings in array processor designs: localized data transactions and/or localized timing scheme (i.e., using self-timed, data-driven control). In fact, the class of locally recursive algorithms permits both locality features; these should be exploited in the architectural designs. An example for such a design is the wavefront array [17].

II. Signal-Flow Graph (SFG) Computing Networks

The most useful graphical representation for scientific and signal processing computations is the signal-flow graph (SFG). While the graphical representations are most popularly used for signal processing flow diagrams, such as FFT and digital filters, etc., the SFG representations in fact cover a broad domain of applications, including linear and nonlinear, time-varying and time-invariant, and multidimensional systems. For convenience, this paper will treat only time-invariant SFG systems.[2]

Notations: In general, a **node** is often denoted by a circle representing an arithmetic or logic function *performed with zero delay*, such as multiply, add, etc. (cf. Fig. 1(a)). An **edge**, on the other hand, denotes either a function

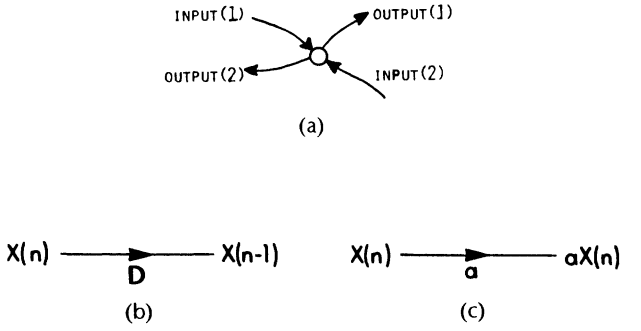


Fig. 1. Examples of SFG graphical denotations. (a) An operation node with (two) inputs and (two) outputs. (b) An edge as a delay operator. (c) An edge as a multiplier.

or a delay. Unless otherwise specified, for a large class of signal processing SFGs, the following conventions are adopted for convenience. When an edge is labeled with a capital letter D (or D', $2D'$, etc.), it represents a time-delay operator with delay time D (or D', $2D'$, etc.) (see Fig. 1(a)). On the other hand, if an edge is labeled with a lower case letter, such as a, a_i, b_i, it represents a multiplication by a constant a, a_i, b_i (see Fig. 1(b)).

When the concept of the SFG was originally conceived, there was little consideration given to the locality preferences in parallel-computing network design. Hence this paper addresses the issue of systematic approaches of mapping SFGs into locally interconnected parallel-array processors.

There are two major classes of SFGs: those with *local interconnections*, and those with *global interconnections*. A typical example of a global SFG is one representing the FFT algorithm. The principle of the (decimation-in-time) FFT is based on successively decomposing the data, say $\{x(i)\}$, into *even* and *odd* parts. This partitioning scheme will result in global communication between PEs. More precisely, the FFT recursions can be written as (using the "in-place" computing scheme [27])

[1] In a recursive algorithm, all processors do nearly identical tasks and each processor repeats a fixed set of tasks on sequentially available data.

[2] This incurs no loss of generality, since any internal time-varying parameters can be equivalently represented by an (external) input signal.

$$x^{(m+1)}(p) = x^{(m)}(p) + w_N^r x^{(m)}(q)$$
$$x^{(m+1)}(q) = x^{(m)}(p) - w_N^r x^{(m)}(q)$$

with p,q,r varying from stage to stage. The "distance" of the global communication involved will be proportional to $|p - q|$. An SFG for the (decimation-in-time) FFT, with space and time indices properly labeled, is shown in Fig. 2. Note

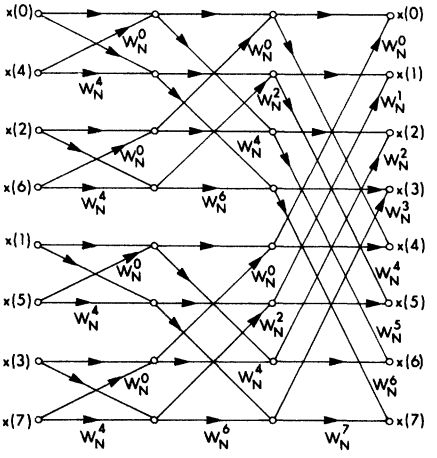


Fig. 2. A signal-flow graph (SFG) for the decimation-in-time FFT algorithm.

that in the last stage the maximum distance is $|p - q| = N/2$ (see [27]). For example, the maximum distance will be 512 units for a 1024-point FFT. Thus the FFT algorithm is a *global one*, since the recursion involves globally separated indices. Therefore, FFT computing structures will call for spatially global interconnections, and cannot be easily mapped onto systolic or wavefront arrays.

In contrast to the FFT algorithm, most other recursive signal processing algorithms are *local*, i.e., the spatial separations between nodes are within a certain limit. Therefore, the corresponding SFGs are "localizable." For examples, the SFGs for FIR and IIR filters can be easily implemented with spatially local SFGs. Generally, an IIR (infinite impulse response) filter is defined by the difference equation

$$y(k) = \sum_{m=1}^{N} x(k-m)b(m) + \sum_{m=1}^{N} y(k-m)a(m). \quad (1)$$

(Note that FIR filtering, linear convolution, and transversal filterings are simply special cases when $a(m) = 0$.)

A popular SFG[3] [27] for (1) is shown in Fig. 3.

We note that this SFG has spatially local interconnections. But it is not *temporally local*, since according to the SFG, propagating a datum "X" from, say, the left-most node to the right-most node uses "zero" time. More precisely, the SFG imposes the requirement that the datum "X" has to be **broadcast** to all the nodes on the upper path. This is certainly undesirable from a systolic design perspective

[3]For this section, a node (circle) commonly denotes addition when there are multiple outputs and single input to the node. It denotes a branching node if there is one input and multiple outputs.

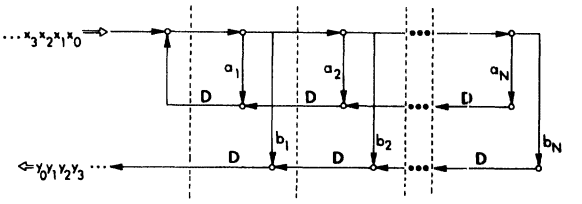


Fig. 3. Direct form design of ARMA (IIR) filter. The SFG is spatially localized but *not temporally localized.*

(and unrealistic for a circuit implementation). This will be the focus point of the next section.

III. Systolization of SFG Computing Networks

A. Systolic Array

Systolic processors [16], [15] are a new class of "pipelined" array architectures. According to Kung and Leiserson [16], "A systolic system is a network of processors which rhythmically compute and pass data through the system." For example, it is shown in [16] that some basic "inner product" PEs ($Y \leftarrow Y + A*B$) can be *locally* connected together to perform digital filtering, matrix multiplication, and other related operations. The systolic array features the important properties of modularity, regularity, local interconnection, a high degree of pipelining, and highly synchronized multiprocessing. The data movements in a systolic array are often described in terms of the "snapshots" of the activities [16].

There are no formal or coherent definitions of the systolic array in literature. In order to have a formal treatment of the subject, however, we shall adopt the following definition:

1) Definition: Systolic Array: A systolic array is a computing network possessing the following features:

a) *Synchrony:* The data are rhythmically computed (timed by a global clock) and passed through the network.

b) *Regularity (i.e., Modularity and Local Interconnections):* The array should consist of modular processing units with regular and (spatially) local interconnections. Moreover, the computing network may be extended indefinitely.

c) *Temporal Locality:* There will be at least one unit-time delay allotted so that signal transactions from one node to the next can be completed.

d) *Pipelinability (i.e., O(M) Execution-Time Speed-Up):* A good measure for the efficiency of the array is the following

$$\text{Speed-Up Factor} = \frac{\text{Processing Time in a Single Processor}}{\text{Processing Time in the Array Processor}}$$

A systolic array should exhibit a *linear-rate pipelinability*, i.e., it should achieve an $O(M)$ speed-up, in terms of processing rates, where M is the number of processor elements (PEs).

We note that a *regular* SFG, such as the canonic SFG for ARMA filters, is already very close to a systolic array. The major difference being that most SFGs are not given in temporally localized form. Therefore, it is important to be able to convert them into localized ones. The topic of imposing temporal locality into a computing network has been a focus point of several researchers, including a series

of publications by Fettweis [31], Leiserson [22]–[24], etc., and, more recently, the works reported in [20], [25], [14], [2]. The main advantage of the (cut-set) scheme proposed here (largely based on [20]) lies in its simplicity to use and its straightforward proof. Our proof is based on a graph-theoretical result—the *colored arc lemma*, which will be discussed momentarily.

2) Systolic Array for ARMA (IIR) Filter: Before discussing the general procedure, let us first take a look at an example. The canonic SFG for an ARMA Filter in Fig. 3 can be easily converted into a local-type one. Our first step in Fig. 4(a) is to rescale the time unit by setting $D = 2D'$. After shifting one of the two delays from the upper edges to the corresponding lower edges, a modified design can be derived as in Fig. 4(a).

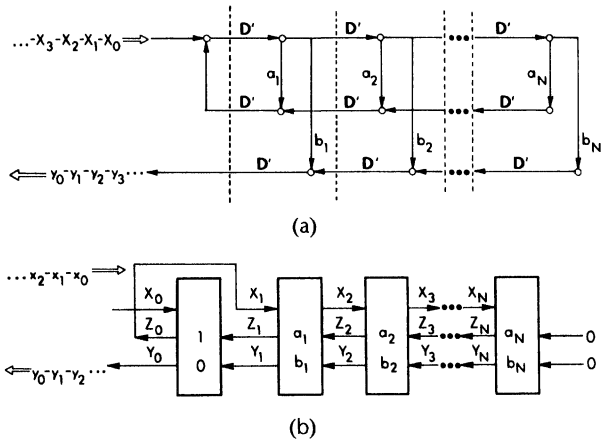


Fig. 4. (a) A modified SFG for an ARMA filter—a systolized version. (b) A systolic array for an ARMA (IIR) filter.

To verify that Fig. 4(a) yields the same transfer function as Fig. 3, one can simply check that the transfer function remains the same [20]. A more general proof will be discussed in a moment.

Let us now demonstrate how the modified form can be trivially converted into a systolic design. To do this, the operation time for one multiplication, one addition, and data transfer are merged with the uni-time delay D'. Therefore, the delay D', the multiplier, and the adder in each single section (defined by means of dashed lines) in Fig. 4(a) are all merged into an "inner product" processor. This leads to an overall systolic array configuration for IIR filter as shown in Fig. 4(b).

Now we are ready to discuss a general systematic procedure for converting SFGs into systolic arrays. First, we need a procedure to convert an SFG into a temporally localized SFG, which contains only nonzero-delay edges between modular sections.

Definition: Cut-Set:
A *cut-set* in an SFG is a minimal set of edges which partitions the SFG into two parts.

B. A Cut-Set (Temporal) Localization Procedure

The localization procedure is based on two simple rules:

Rule (i) Time-Scaling: All delays D may be scaled, i.e., $D \rightarrow \alpha D'$, by a single positive integer α. Correspondingly, the input and output rates also have to be scaled by a factor α (with respect to the new time unit D') (see Fig. 4(a)).

Rule (ii) Delay-Transfer: Given any cut-set of the SFG, we can group the edges of the cut-set into *inbound edges* and *outbound edges*, depending upon the directions assigned to the edges. Rule (ii) allows advancing k (D') time units on all the outbound edges and delaying k time units on the inbound edges, and *vice versa*. It is clear that, for a (time-invariant) SFG, the general system behavior is not affected because the effects of lags and advances cancel each other in the overall timing. Note that the input–input and input–output timing relationships will also remain exactly the same only if they are located on the same side. Otherwise, they should be adjusted[4] by a lag of $+k$ time units or an advance of $-k$ time units.

We shall refer to these two basic rules as the (cut-set) *localization rules*. Based on these rules, we assert the following:

Theorem:
All computable[5] SFG's are temporally localizable.

Proof of the Theorem: We claim that the localization Rules (i) and (ii) can be used to "localize" any (targeted) zero-delay edge, i.e., to convert it into a nonzero-delay edge. This is done by choosing a "good" cut-set and apply the rules upon it. A good cut-set including the "target edge" should not include any "bad edges," i.e., those zero-delay edges in the opposite direction of the target edge. This means that the cut-set will include only i) the target edge, ii) nonzero delay edges going in either direction, and iii) zero-delay edges going in the same direction. Then, according to Rule (ii), the nonzero delays of the opposite-direction edges can "give" one or more spare delays to the target edge (in order to localize it). If there are no spare delays to give away, simply scale all delays in the SFG according to Rule (i) to create enough delays for the transfer needed.

Therefore, the only thing left to prove is that such a "good" cut-set always exists. For this, we refer to Fig. 5., in which we have kept only all of the zero-delay *successor edges* and the zero-delay *predecessor edges* connected to the target edge, and removed all the other edges from the graph. In other words, Fig. 5 depicts the bad edges which should not be included in the cut-set. As shown by the dashed lines in Fig. 5, there must be "openings" between these two sets of bad edges—otherwise, some set of zero-delay edges would form a zero-delay loop, and the SFG would not be computable. Obviously, any cut-set "cutting" through the openings is a "good" cut-set, thus the existence proof is completed. (The author was later advised by a colleague that the existence proof discussed above is in fact a result known as the *colored arc lemma* in graph theory.) It is clear that repeatedly applying the localization Rule (ii) (and (i), if necessary) on the cut-sets will eventually lead to a temporally localized SFG.

[4]If there is more than one cut-set involved, and if the input and output are separated by more than one cut-set, then such adjustment factors should be accumulated.

[5]An SFG is meaningful only when it is computable, i.e., there exists no zero-delay loop in SFG.

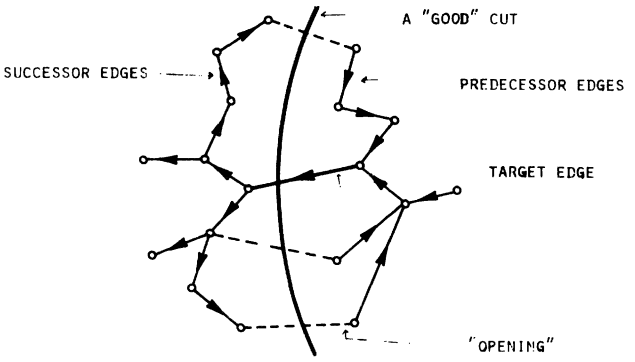


Fig. 5. "Openings" between bad edges ensure the existence of a "good" cut-set. This may be used as a clue for selecting a "good" cut-set.

C. Systolization Procedure

As we have claimed earlier, a regular SFG is *almost* equivalent to a systolic array and can be easily systolized. The systolization procedure, essentially based on the cut-set localization rules, is outlined below:

1) *Selection of Basic Operation Modules:* The choice may not be unique. In general, the finer the granularity of the basic modules, the more efficient (in speed) a systolic array will be. (A comparison of two possible lattice modules for a systolic array will be discussed in the next subsection.)

2) *Applying Localization Rules:* If the given SFG is regular, i.e., modular and spatially local, then regular cut-sets can be selected and the above rules can be used to derive a regular and temporally localized SFG.[6]

3) *Combination of Delay and Operation Modules:* To convert such an SFG into a systolic form, we need only to successfully introduce a delay into each of the operation modules, such as $A + B*C$. Combine the delay with the module operation to form a basic systolic element. All the extra delays will be modeled as pure delays without operations.

4) *Verification of the Arrays:* An SFG representation for linear, time-invariant systems is directly verifiable by the Z-transform technique. The correctness of the transformation to a systolic design is also guaranteed by the cut-set rules. Therefore, there is no need to display snapshots for the verification purpose. Nevertheless, one may find snapshots a simple and useful tool for a better appreciation of the movement of data.

IV. Examples of the Systolization Procedure

In this section, we shall apply the systolization procedure to some one- and two-dimensional SFGs.

A. Systolic Lattice Filters

For the one-dimensional case, a very interesting example is the systolic implementation of a digital lattice filter, which should have many important applications to speech and seismic signal processing. For this example, let us now apply the transformation rules to the SFG for an autoregressive (AR) lattice filter, as shown in Fig. 6(a). There are two possible choices of basic operation modules for the lattice array: (A) a lattice operation module,[7] and (B) a multiply/add (AM) basic module. Note that in each lattice operation there are two MA operations—implying that the lattice operation uses twice the time of MA.

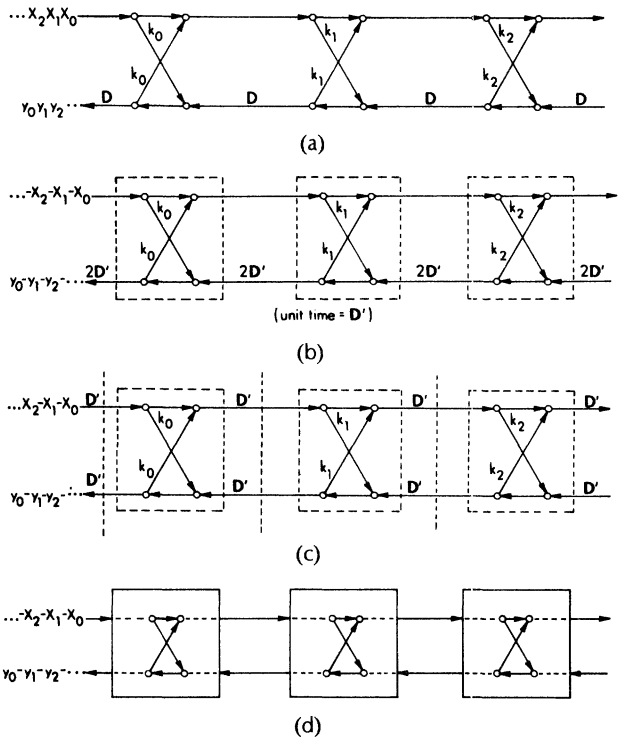


Fig. 6. (a) An SFG for AR lattice filters. (b) Time-rescaled SFG for AR lattice filters (type A). (c) "Localized" SFG for AR lattice filters (type A). (d) Systolic array for AR lattice filters (type-A).

1) Lattice Systolic Array (Type-A): By localization Rule (i), we first double each delay, i.e., $D \rightarrow 2D'$ as shown in Fig. 6(b). Apply (uniformly) the cuts to the SFG and subtract one delay from each of the left-bound edges and, correspondingly, add one delay to each of the right-bound edges in the cut-sets. This yields Fig. 6(c). Finally, by combining the delays with the lattice module, we have the final systolic structure, as in Fig. 6(d). Note that, because of the time-scaling, the input sequence $\{x(i)\}$ will be interleaved with "blanks" to match the adjusted delays. It is clear that $\alpha = 2$ and the array can yield an $M/2$ execution-time speedup.

2) Lattice Systolic Array (Type-B): By the localization Rule (i), we first triple each delay; i.e., $D \rightarrow 3D'$. (The resultant SFG is the same as what is shown in Fig. 6(b), but substituting $2D'$ by $3D'$.) Apply uniform cut-sets to the SFG as shown in Fig. 7(a). Subtract two delays from each left-bound edge, and, correspondingly, add two delays to every right-bound edge in the cut-sets. This yields Fig. 7(a). Now let us use a cut-set partitioning the upper edges from the lower edges as shown in Fig. 7(b). Transfer one delay from the down-going edges to the up-going ones. The result is depicted in Fig. 7(b). Finally, by combining the delays with

[6]In order to preserve the modular structure of the SFG (a basic feature of systolic design), the cut-set localization should be applied uniformly across the network. Otherwise, the resultant array may not be systolic.

[7]That is, the lattice operation is now treated as a single module—a desirable choice of CORDIC implementation.

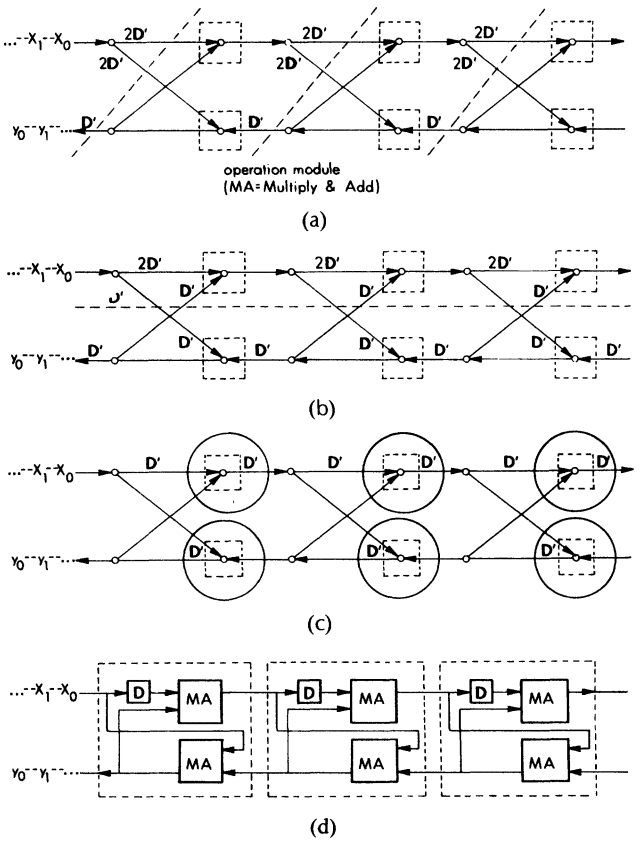


Fig. 7. (a) Time-rescaled SFG (and cut-sets) for AR lattice filters (type-B). (b) Partially localized SFG—first step. (c) "Localized" SFG—all operations are ready to merge with the corresponding uni-time delays D'. (d) Systolic array for AR lattice filters (type-B) with the small squares denoting pure delays.

the Multiply–Add Module (cf. Fig. 7(c)), a systolic structure is obtained as shown in Fig. 7(d). Note that because of time scaling the input sequence $\{x(i)\}$ will be interleaved with two "blanks" to match the adjusted rates. It is clear that the array can achieve a $2M/3$ execution-time speedup.[8] In terms of speed, this systolic design is superior to the Type-A design.

B. Two-Dimensional Systolic Arrays

The systolization procedures can be applied to two-dimensional networks. Although the descriptions of two-dimensional activities are often cumbersome; fortunately, the SFG representations of two-dimensional algorithms (especially, the (temporally) nonlocalized version) are often much easier to comprehend. With the procedures discussed in the previous section, the conversion of a two-dimensional SFG to a systolic array is straightforward.

A typical example used for illustrating a two-dimensional array operation is matrix multiplication. Let

$$A = [a_{ij}] \qquad B = [b_{ij}]$$

and

$$C = A \times B.$$

[8]Note that since the upper and lower MA modules in each PE are never simultaneously active, only one MA hardware module suffices to serve both functional needs.

Suppose that both A and B are nonsparse $N \times N$ matrices. The matrix A can be decomposed into columns A_i and the matrix B into rows B_j and, therefore,

$$C = [A_1B_1 + A_2B_2 + \cdots + A_NB_N]$$

where the product A_iB_i is termed "outer product." The matrix multiplication can then be carried out in N recursions (each executing one outer product)

$$c_{i,j}^{(k)} = c_{i,j}^{(k-1)} + a_i^{(k)}b_j^{(k)} \tag{2}$$

$$a_i^{(k)} = a_{ik}$$

$$b_j^{(k)} = b_{kj}$$

for $k = 1, 2, \cdots, N$ and there will be N sets of wavefronts involved.

1) Systolizing an SFG for Matrix Multiplication: The simplest matrix multiplication array design is one letting columns A_i and rows B_i be broadcast instantly along the square array as shown in Fig. 8(a). All outer products will then be sequentially summed via a loop with single delay. This design is not suitable for VLSI circuit design since it needs to use global communication. However, there is a rapidly growing interest in the developments of optical array processors, [12], [3]. From an optical interconnection perspective this SFG may be directly implementable.

If local interconnection is preferred, the proposed procedure in Section III can again be used to systolize the SFG. Let us apply Rule (ii) to the cut-sets shown in Fig. 8(a). The systolized SFG will have one delay assigned to each edge and thus represent a localized network. According to Rule (ii), the inputs from different columns of B and rows of A will have to be adjusted by a certain number of delays before arriving at the array. By counting the cut-sets involved in Fig. 8(a), it is clear that the first column of B needs no extra delay, the second column needs one delay, the third needs two (i.e., attributing to the two cut-sets separating the third column input and the adjacent top-row processor), etc. Therefore, the B matrix will be skewed as shown in Fig. 8(c). A similar arrangement can be applied to A.

2) Multiplication of a Banded Matrix and a Full Matrix: Let us look at a slightly different, but commonly encountered, type of matrix multiplication problem. This involves a banded-matrix A, $N \times N$, with bandwidth P, and a rectangular matrix B, $N \times Q$. This situation arises in many application domains, such as DFT and time-varying (multichannel) linear filtering, etc. In most applications, $N \gg P$ and $N \gg Q$, and this makes the use of $N \times N$ arrays for computing $C = A \times B$ very uneconomical.

Fig. 9(a) shows that, with slight modification to the SFG in Fig. 8(a), the same speedup performance can be achieved with only a $P \times Q$ rectangular array (as opposed to an $N \times Q$ array). Now, the left memory module will store the matrix A along the band-direction (see Fig. 9(a)) and the upper module will store B the same as before.

Note that the major modification to the array is that, between the recursions of outer products, there should be an upward shift of the partial sums. This is because the input matrix A is loaded in a skewed fashion. The final result (C) will also be outputted from the I/O ports of the top-row PEs.

Applying the systolization procedure leads to the data array as depicted in Fig. 9(b).

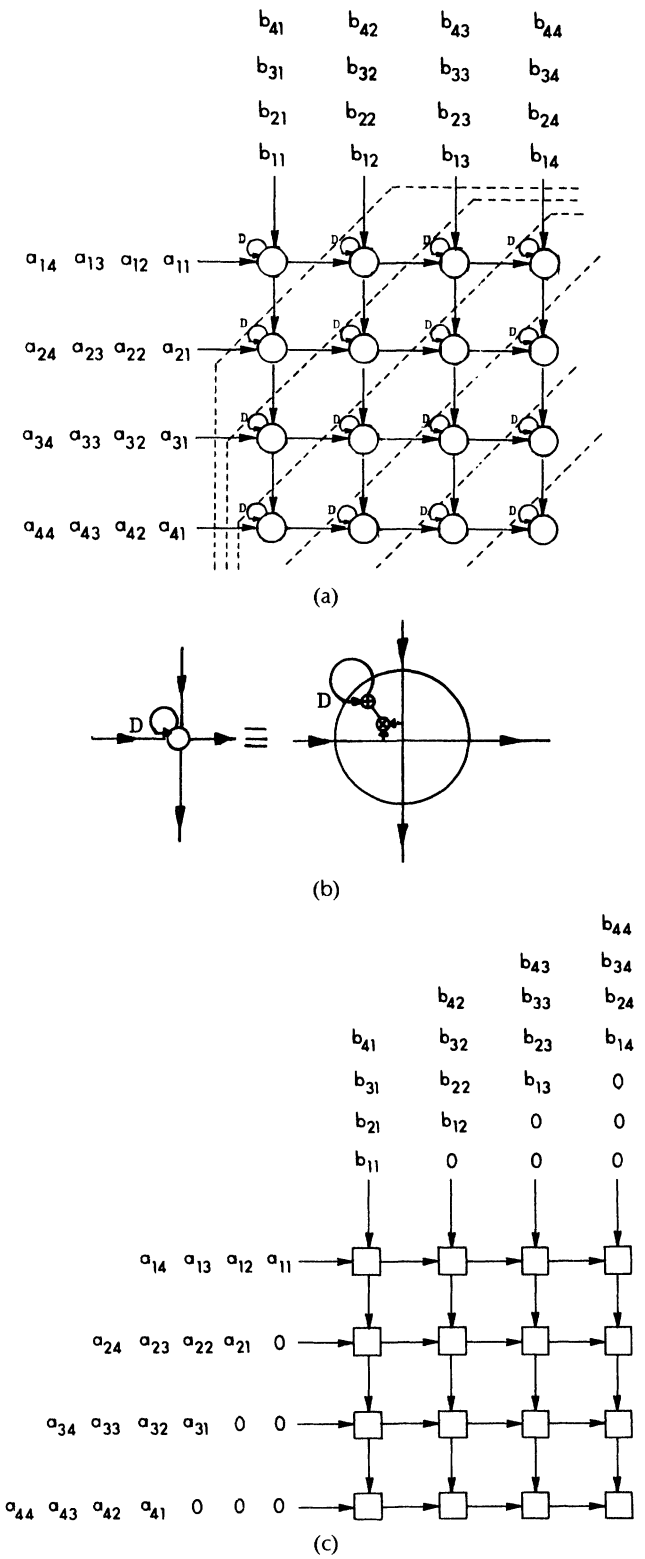


Fig. 8. (a) An SFG for matrix multiplication. (b) The detailed diagram of the processing nodes. (c) A systolic array for matrix multiplication.

3) Multiplication of Two Banded Matrices: Another interesting case is the situation when both A and B are banded matrices, with bandwidths P and Q, respectively. Let us assume that $N \gg P$ and $N \gg Q$, where P and Q are bandwidths for A and B, respectively. Then it is possible to achieve full parallelism with only a $P \times Q$ rectangular array (as opposed to an $N \times N$ array).

Now, the left- and upper memory modules will store the matrices A and B (respectively) along the band direction (see Fig. 10(a)). The delayed feedback edge (with partial sum of the outer products) will be along the diagonal direction (to the N–W direction). This is because both A and B are stored in the skewed version of Fig. 10(a). Applying the systolization procedure to the cut-sets as shown in Fig. 10(a) will call for a triple scaling of $D \rightarrow 3D'$. (This is because each north–west-bound delay edge is "cut" twice.) The procedure leads to an array configuration depicted in Fig. 10(b), which is topologically equivalent to the two-dimensional hexagonal array proposed in [16], [15]. Similarly to what happens in the hexagonal array, the output data (of the matrix C) are also pumped from the both sides.

4) Systolizing an SFG for LU Decomposition: In LU decomposition, a given matrix C is decomposed into

$$C = A \times B$$

where A is a lower- and B an upper-triangular matrix. The recursions involved are

$$C_{ij}^{(k)} = C_{ij}^{(k-1)} - a_i^{(k)} * b_j^{(k)} \tag{3}$$

where

$$a_i^{(k)} = \frac{1}{c_{kk}^{(k)}} C_{i,k}^{(k-1)}$$

$$b_j^{(k)} = C_{k,j}^{k-1}$$

for $k = 1, 2, \cdots, N$; $k \leqslant i \leqslant N$

$k \leqslant j \leqslant N$.

The SFG representation for the above iterations is shown in Fig. 11. Note that it bears a great similarity to the SFG for multiplication of two banded matrices. Therefore, by almost the same systolization procedure as shown in Fig. 10, a systolic array for LU decomposition can be obtained. The resultant configuration of the array (not shown here) is very similar to Fig. 10(b).

C. Linear-Rate Pipelinability

Although the above systolization procedure is essentially complete, it is useful to find out how well the operations of an algorithm can be pipelined through the array (i.e. the pipelinability). The answer is rather straightforward:

We assert that the array has a linear-rate pipelinability *if and only if* α remains constant with respect to M, where M is the number of processor elements (PEs). It is clear that if the total time-scaling factor is α (i.e., $D = \alpha D'$), then the data input rate is slowed down by α. Consequently, in average only one of α PEs can be active. This implies that the full processing rate speedup M reduces to $\alpha^{-1}M$. If α remains constant with respect to M, then the array is pipelinable with a linear-rate speedup.

For most practical computational models the scaling factor α is 1, 2, or 3. For example: in the ARMA and lattice (type-A) systolic arrays, $\alpha = 2$, and in the lattice (type-B) array, $\alpha = 3$, regardless of how large M is. The same is true for most two-dimensional graphs, e.g., the SFGs for matrix multiplications, and LU decomposition, etc. However, there are examples of SFGs that, when localized, lead to nonconstant α. The arrays then cannot exhibit $O(M)$ execution-time speedups.

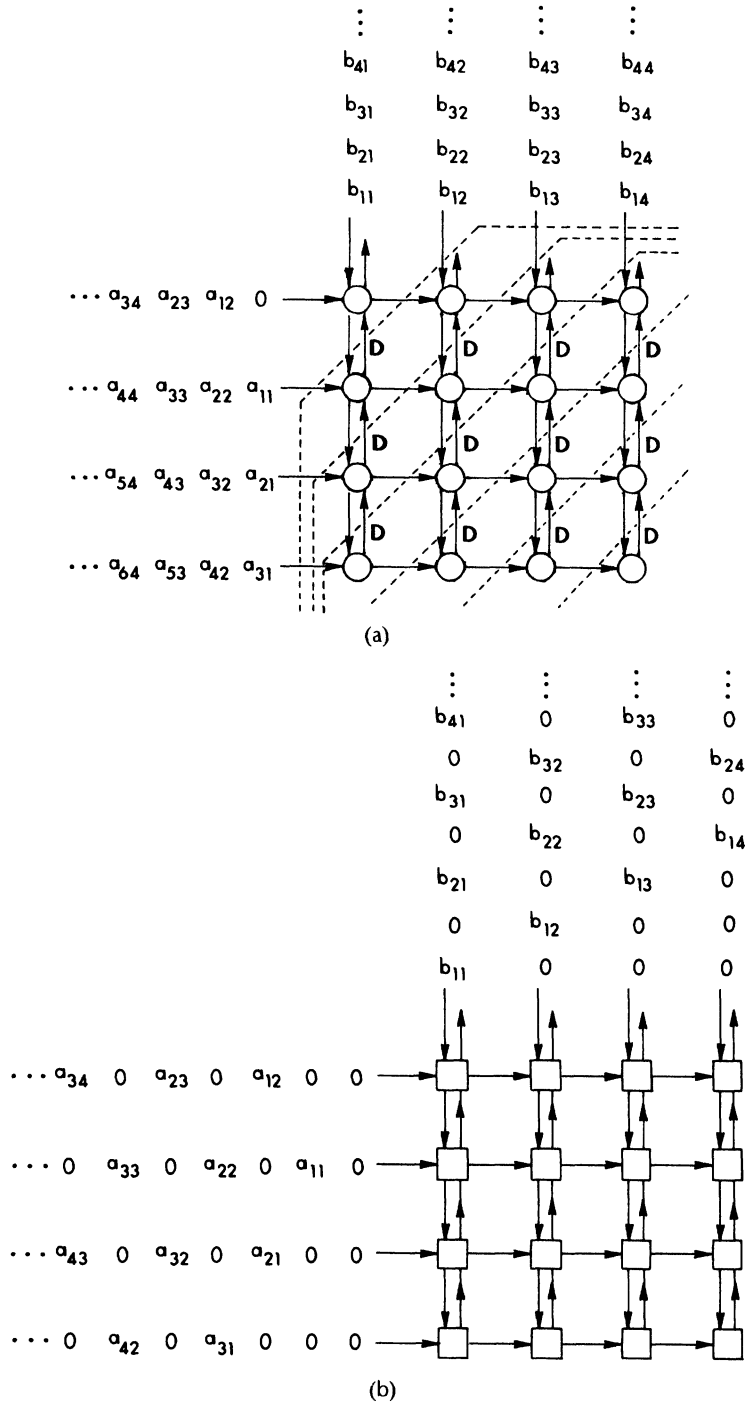


Fig. 9. (a) An SFG for matrix multiplication with one banded matrix. (b) Systolic array for matrix multiplication with one banded matrix.

1) An Example of Nonpipelinable SFGs: As an example of a regular but nonpipelinable SFG, let us look at the SFG shown in Fig. 12(a), which is originally due to Dewilde [8], [14]. Note that there exist no simple, uniform cuts with which to proceed the conversion. In fact, applying the localization rules with nonuniform cut-sets as shown in Fig. 12(a) (where the numbers in parentheses indicate the order of the cut-sets to apply the localization rules) will lead to a temporally localized array as depicted in Fig. 12(b). Note that the final time-scaling factor α turns out to be linearly proportional to M. This means that the speed performance of the "parallel" array processor is basically no different from a sequential computer, because no efficient pipelining is possible. Therefore, the array is said to be nonsystolic.

D. Improving Processing Speed and Utilization Efficiency

1) Multirate Systolic Array—Improving Processing Speed [20]: Note that due to the recaling of time units, the input data $\{x_i\}$ have to be interleaved with "blank" data (see Figs. 4, 6, 7), and the throughput rate becomes $(\alpha T)^{-1}$. This

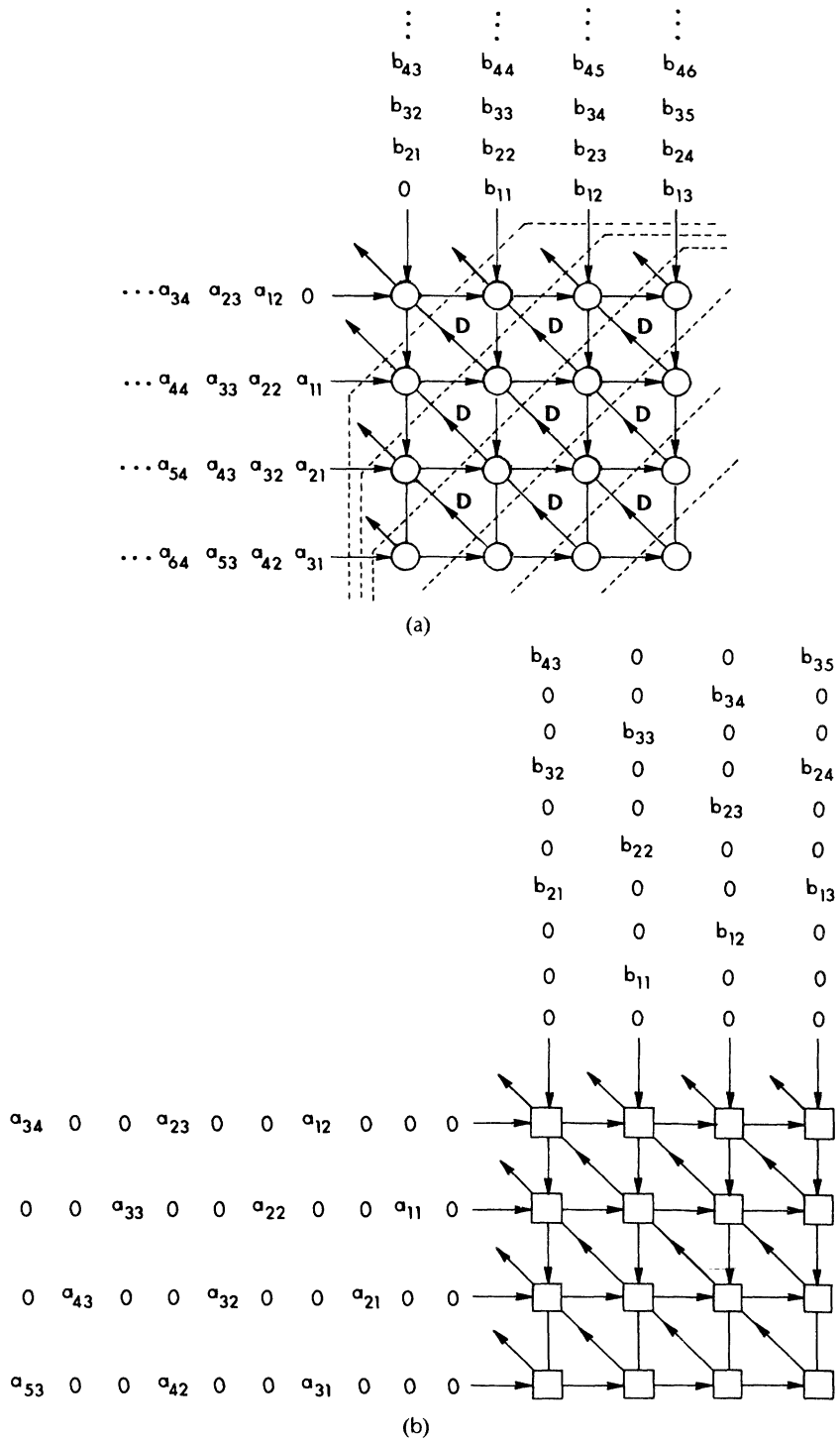
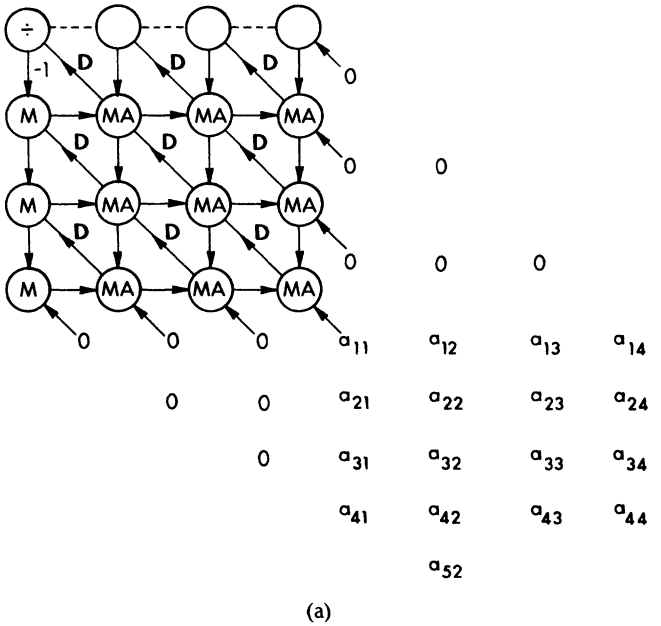


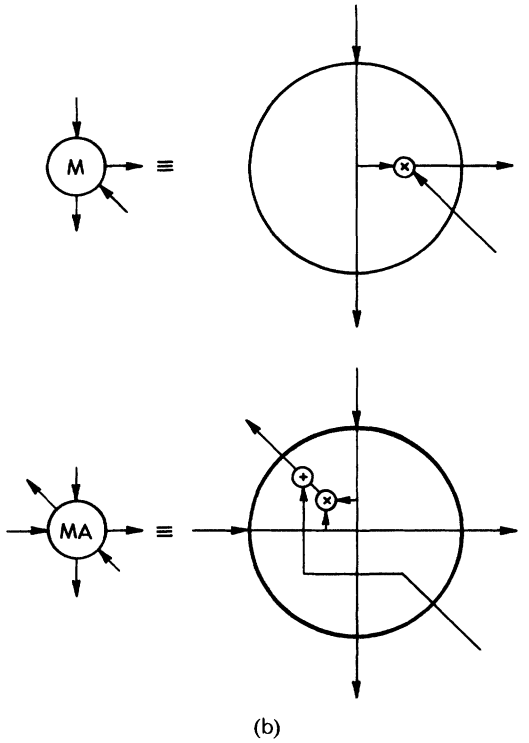
Fig. 10. (a) An SFG for multiplication of two banded matrices. (b) Systolic array for multiplication of two banded matrices.

rate is slower than that of the direct form design ($1.0T^{-1}$), because the data-transfer operation ($X \rightarrow X'$) alone consumes the same time as a multiply-and-add operation, an unnecessary delay. There are two solutions to this problem: one is to use a multirate systolic array and the other is to use a wavefront array based on asynchronous data-driven computing.

A multirate systolic array is a generalized systolic array, allowing different operations to consume different time units. As we have mentioned earlier, the finer the granularity in defining the basic module, the better the efficiency. For maximal efficiency, the granularity has to go all the way down to the bit level for a data-transfer operation, while the arithmetic operation may remain at the word level. For the ARMA filter design example, we can assign Δ as the time unit for a data transfer and T for a multiply-and-add. Consequently, in the circuit representation in Fig. 4(a), we replace D' on the feedforward path (for X) by Δ, and D' on



(a)



(b)

Fig. 11. (a) An SFG for LU decomposition. (b) The detailed diagram of the processing nodes.

the feedback paths (for Y and Z) by Δ. This means that the X data are pumped to the right with a delay of Δ, while the data Y and Z are transferred (to the left) with a (much longer) delay T. These modifications lead to a multirate systolic array as shown in Fig. 13. Since the original basic delay interval (D) is now replaced by $\Delta + T$, therefore, the input/output sequences $\{x_0, x_1, x_2, \cdots,\}$ and $\{y_0, y_1, y_2, \cdots,\}$ have to be pumped in and out by an interval $(T + \Delta)$, and attain a throughput rate of $1/(T + \Delta)$. In fact, a multirate systolic array is equivalent to a synchronized version of the wavefront array discussed in the next section.

2) Sharing Operation Modules—Improving Utilization Ef-

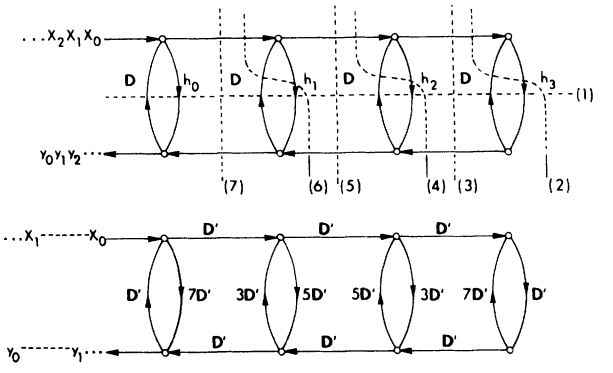


Fig. 12. (a) A nonsystolizable SFG example. (b) Localized but nonpipelinable SFG.

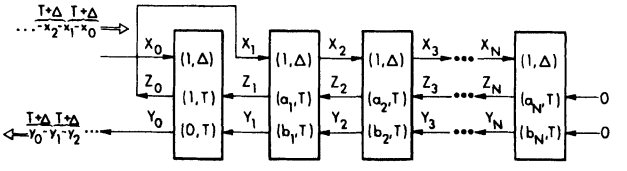


Fig. 13. A multirate systolic array for ARMA (IIR) filter.

ficiency: It is possible to improve the processor utilization rate by as much as α times, where α is the time-scaling factor used in the localization process. The scheme is straightforward, noting that the interval between data will have to be α units apart, and therefore only one of α consecutive processor modules will be active at any instant. Therefore, a group of α consecutive PEs can *share a common arithmetic unit* without compromizing the throughput rates. Now let us use an example for a better illustration. Note that, according to the snapshots for the lattice systolic array (B) as depicted in Fig. 14, *only one upper MA module is active* in every three PEs at any time instant, and the same is true for the lower MA module. Therefore, as shown in Fig. 14, the three PEs can be combined into a (macro)PE and share the two common MA modules (one upper and one lower). A special-purpose ring register (with *period* = α) can be designed to handle the resource scheduling.

V. Data-Flow Principle and Wavefront Array [17]

One problem associated with the systolic is that the data movements are controlled by global timing-reference "beats." In order to synchronize the activities in a systolic array, extra delays are often used to ensure correct timing. However, the price of this is an unnecessary slowdown in throughput rates. More critically, the burden of having to synchronize the entire computing network will eventually become intolerable for very- (or ultra-) large-scale arrays. The solution to the problem is to substitute the need for correct "timing" by correct "sequencing," as is used in data-flow computers and wavefront arrays. This leads to a completely different way of tackling the question "b) **when** to send data?" as posed in Section I-B. This time the answer lies in a data-driven, self-timed approach.

A. Data-Flow Multiprocessor

A data-flow multiprocessor [7] is an asynchronous, data-driven multiprocessor which runs programs expressed in

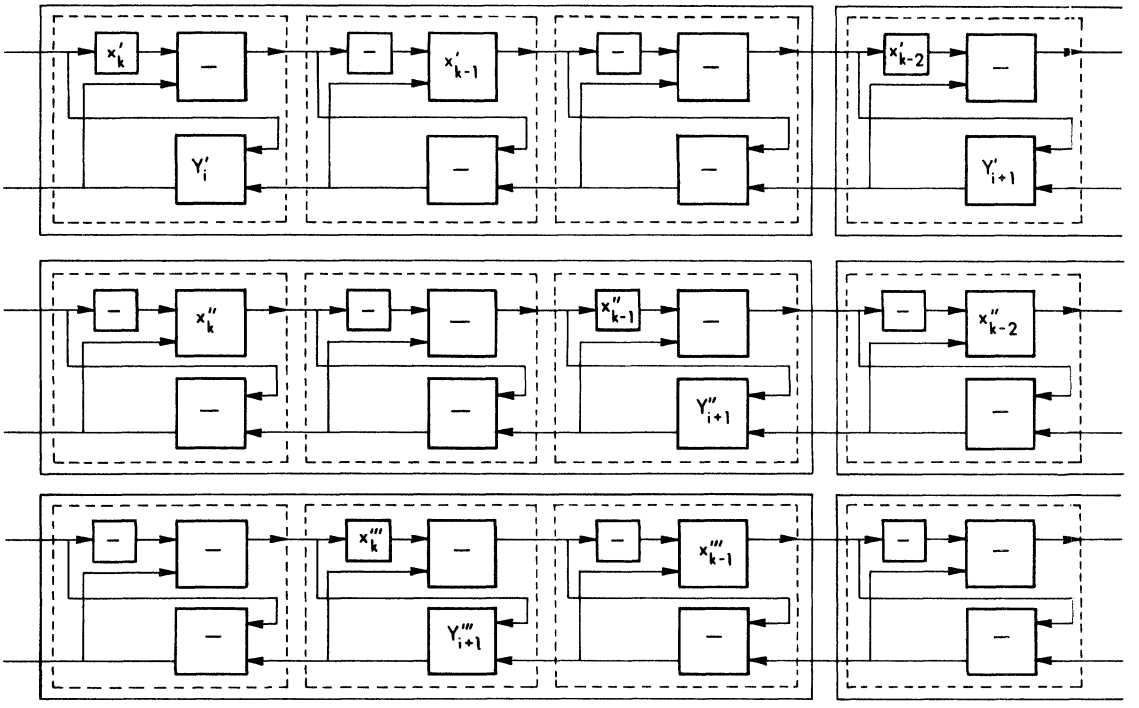


Fig. 14. Snapshots and time sharing of three PEs within a (macro)PE. (a) Snapshot at $t = i$. (b) Snapshot at $t = i + 1$. (c) Snapshot at $t = i + 2$.

data-flow graph form. Since the execution of its instructions is "data-driven," i.e., the triggering of instructions depends only upon the availability of operands and resources required, unrelated, instructions can be executed concurrently without interference. The principal advantages of data-flow multiprocessors over conventional multiprocessors are simple representation of concurrent activity, relative independence of individual PEs, greater use of pipelining, and reduced use of centralized control and global memory. However, for a general-purpose data-flow multiprocessor, the interconnection and memory conflict problems remain very critical. Such problems can be eliminated if the concepts of *modularity* and *locality* are imposed onto data-flow multiprocessors. This idea is the key motivation leading to concept of Wavefront Arrays.

B. Wavefront Arrays

Definition: Wavefront Array: A Wavefront Array is a computing network possessing the following features:

1) *Self-Timed, Data-Driven Computation:* No global clock is needed, since network is self-timed.

2) *Modularity and Local Interconnection:* Basically the same as in a systolic array. However, the wavefront array can be extended indefinitely without having to deal with the global synchronization problem.

3) *O(M) Speedup and Pipelinability:* (Similar to the systolic array.)

Note that the major difference distinguishing a wavefront array from a systolic array is the data-driven property. Consequently, the temporal locality condition (see 3 in the definition of Systolic Array) is no longer needed, since there is no explicit timing reference in the wavefront arrays. By relaxing the strict timing requirement, there are many advantages gained, such as speed and programming simplicity.

C. Incorporating Data-Flow Computing into Computing Networks

Our main goal here is to demonstrate that *all SFG computing networks can be converted into data-driven computing models.* Therefore, by properly incorporating the data-flow feature, every regular and modular SFG can be converted into a wavefront array.

In a self-timed system, the exact timing reference is ignored; instead, the central issue is sequencing. *Getting a data token in a self-timed system is equivalent to incrementing the clock by one time-unit in a synchronous system.* Therefore, the delay operators D will be replaced by self-timed delays, i.e., handshaked "separator" registers.[9] In other words, the conversion of an SFG into a data-driven system involves substituting the delay D with implicit or explicit separators, and replacing the global clock by data handshaking. This process incorporates the data-flow principle into SFG's or systolic arrays.

Theorem: (Equivalence Transformation between SFG's and DFG's)

The computation of any SFG can be equivalently executed by a self-timed, data-driven machine with a data-flow graph (DFG) identical to the SFG, apart from substituting every time-delay operator D (controlled by a global clock) in the SFG with a *separator* ($\diamond$) that is locally controlled by handshaking.

Proof: What needs to be verified is that the global timing in the SFG can be (comfortably) replaced by the corresponding sequencing of the data tokens in the DFG.

[9]A handshaked separator is a device, symbolized by a diamond $\diamond$, which prevents any incoming data from directly passing through *until* the handshaking flag signals a "pass."

Note that the transfer of the data tokens is now "timed" by the processing node. This ensures that the relative "time" between data tokens received at the node is the same as it was in the SFG, as far as that individual node is concerned. By mathematical induction, this can be extended to show the correctness of the sequencing in the entire network.

For convenience, we shall term this trivial transformation the *SFG/DFG Equivalence Transformation.*

Example: Linear Phase Filter Design: As an example, let us now apply the rules to the SFG for a very popular linear phase filter, as shown in Fig. 15(a).[10] By the SFG/DFG

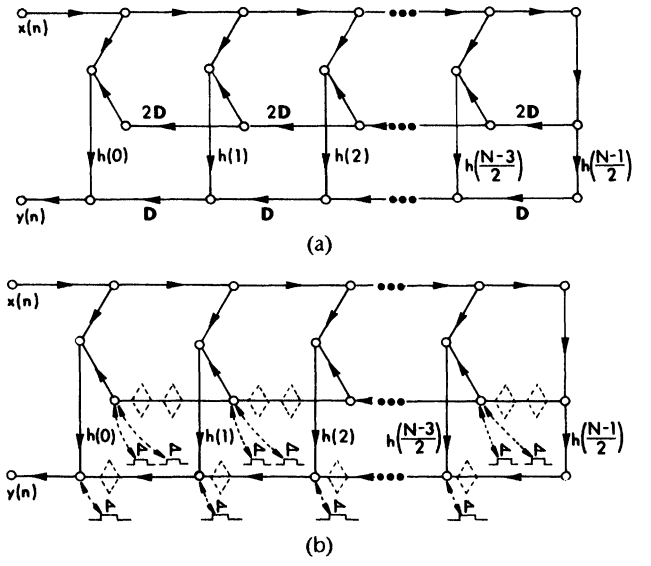


Fig. 15. (a) SFG for linear-phase filters. (b) Data-driven model (delta-flow graph) for the linear-phase filter.

equivalence transformation, the data-flow graph is derived as in Fig. 15(b). Now note that there are two separators inserted into every middle-level edge with handshaking (symbolized by "flags") to the branching node immediately succeeding them. In other words, the transfers occurring in the two separators are synchronized. In order to ensure the correct sequencing of data, the *W* data should propagate twice as slowly as the *Y* data do. Note that the separators play the role of ensuring such a correct sequencing.

Initial States: The general principle is that, all the separators are assigned initial values (regarded as data token), which are the same as those assigned to the delay registers in the corresponding SFG. We note that the initial state assignment under the SFG/DFG transformation is straightforward. This simplicity compares very favorably with the initial state reassignments in the systolization rule, where (because of the retiming involved in the systolization procedure) such reassignments may sometimes become rather complicated.

[10]Linear phase filters are very often used in one- and two-dimensional convolutions. They have two key features: one, that they have a symmetrical impulse response function, i.e., $h(n) = h(N-1-n)$, and two, they do not add phase distortion to the signal. Fig. 15(a) shows an SFG which takes advantage of the symmetry property, and reduces the amount of multiplier hardware by one half.

For a detailed illustration on the relationship between the initial states and the correctness of sequencing of data transfers in DFGs, let us again look into the linear phase filter example. Note that one initial zero is assigned to the separator of each *Y*-data edge; and two initial zeros are assigned to the two separators in each *W*-data edge. The "0" of the *Y*-data separator, when requested by the *Y*-summing node, will be passed to meet the *V* data arriving from the upper node. When the operation is done, a "data-used" flag will then be sent to the separator, clearing the way for sending the next *Y* data from the right-hand PE. The situation is similar for the *W* summing node, but only one "0" is "used" and the *W* data are still one separator away from meeting the "*X*" data in the summing node. It will have to wait until the *Y* data and the second "0" meet in the lower summing node. This explains why the propagation of *W* is slower than *Y*. (This is just what is needed to ensure a correct sequencing of data transfers.) Note also that there will be handshaking circuits needed for the nodes in the upper branches. Since there are no associated delays, there will be no initial data-tokens for the nodes.

1) Converting SFG's into Wavefront Arrays: The SFG/DFG equivalence transformation helps establish a theoretical footing for the wavefront array as well as provide more insights towards the programming techniques. The transformation implies that all regular SFGs can be easily converted into wavefront arrays, making modularly designed wavefront processing elements very attractive to use. Furthermore, because there is no concept of (global) time in a self-timed system, temporal locality is no longer an issue of concern. Therefore, the procedure of converting an SFG into a wavefront array is simpler than that of systolizing an SFG.

Another important feature is that data-flow graphs often provide useful clues for programming the data-driven wavefront arrays. An exemplificative program for the wavefront processing for linear phase filters, written in MDFL—Matrix Data Flow Language, will be discussed later.

VI. Wavefront Arrays and Computational Wavefronts

For further illustration, let us apply the SFG/DFG equivalence transformation to several one- and two-dimensional computing networks, e.g., ARMA and lattice filters, matrix multiplication, LU decomposition, etc.

A. One-Dimensional Wavefront Arrays

1) Wavefront Array for ARMA (IIR) Filter: Following the conversion strategy, an asynchronous wavefront model is derived as shown in Fig. 16. Therefore, at each node in Fig. 16, the operation is executed when and only when the required operands (data tokens) are available. An immediate advantage of this model is that a data transfer operation ($X \rightarrow X'$) uses only negligible time, Δ, compared with the time needed for an arithmetic operation. More precisely, the throughput rate achieved by the wavefront array is approximately $1.0(T + \Delta)^{-1}$, i.e., almost twice that of the pure systolic array in Fig. 4(b).

It is important to note that the pipelining in a wavefront array is different from the traditional idea of pipelining. Under the wavefront notion, $X^{(k)}$ is initiated at the leading PE ($n = 0$), and then propagated rightward across the processor array, activating the MA operations in all of the data-driven PEs. The updated data $\{Y_{n+1}^{(k)}, Z_{n+1}^{(k)}\}$ in the sum-

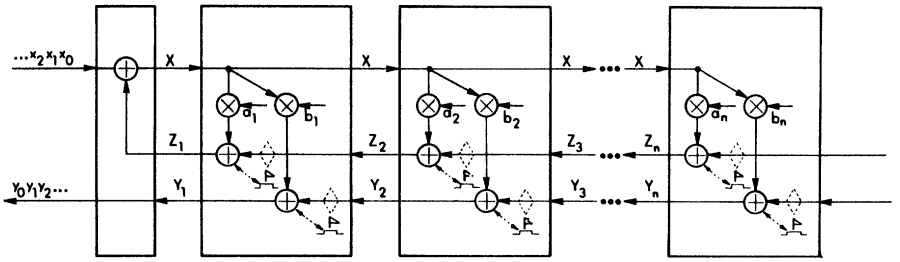


Fig. 16. Wavefront array for ARMA (IIR) filter. (Asynchronous, data-driven model, i.e., operations take place only on availability of appropriate data.)

ming nodes are fed back leftward, ready for the next wavefront. In this case, a reflection of the wave plays an interesting role. For convenience, we shall call such reflection a "ripple" wave. As illustrated in Fig. 17, a ripple from the kth wavefront in the $(n+1)$th PE will be needed (and expected) by the $(k+1)$th wavefront in the nth PE.

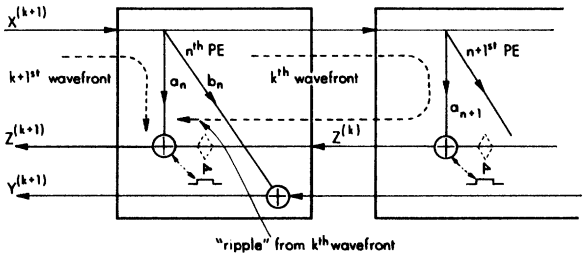


Fig. 17. A "ripple" wave from $(n+1)$th PE to nth PE.

2) Wavefront Array for Lattice Filter: In general, the data-driven model is not only faster (with maximized pipelining) but also has a simpler (self-timed) design, since a global timing reference is no longer required. For example, the (data-driven) wavefront model for the AR lattice as shown in Fig. 18 (compatible with the sytolic array Type-B) is considerably simpler than its systolic counterpart. Due to the data-driven nature of a wavefront array, it is guaranteed that the operation on the X data will have to wait until the Y data operation is done and the result transferred to the upper MA module. Therefore, the appropriate delays naturally fall into place, yielding the correct sequencing of the data. In contrast, in the (Type-B) systolic structure, the two kinds of signals, X (right-bound) and Y (left-bound), are propagating at different speeds, as shown in the snapshots in Fig. 14. Therefore, the (pure) systolic version is more complex than the wavefront solution, due to the timing of the complicated "ripple" effect.

B. Pipelining of Two-Dimensional Computational Wavefronts

From an algorithmic analysis perspective, the notion of computational wavefronts offers a very simple way to appreciate the wavefront computing. The separators are the handshaking device ensuring that the computational wavefronts are orderly following, instead of overtaking, their previous fronts.[11] We shall illustrate the wavefront concept and the related architecture and language designs with the matrix multiplication example. The computational wavefront for the first recursion in matrix multiplication will now be examined.

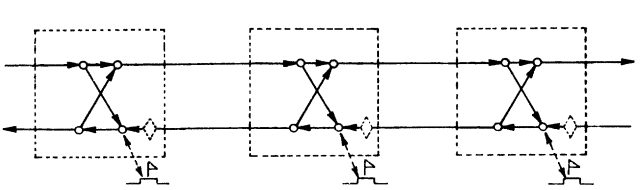


Fig. 18. Wavefront array for lattice filter.

The application of conversion rules to the (original) SFG is fairly straightforward. Basically, imposing handshaking upon all cut-sets will ensure correct sequencing. To see that this is true, a general configuration of computational wavefronts traveling down a processor array is illustrated in Fig. 19.

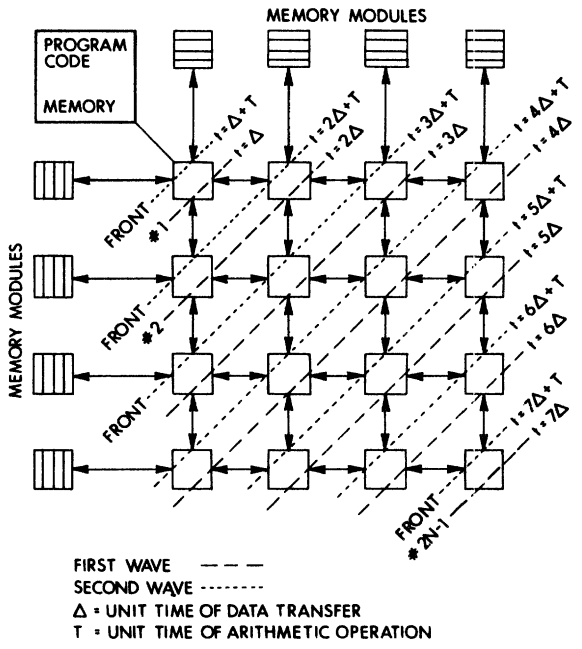


Fig. 19. Propagation of two-dimensional computational wavefronts.

Suppose that the registers of all the processing elements (PEs) are initially set to zero

$$C_{ij}^{(0)} = 0, \qquad \text{for all } (i,j)$$

the entries of A are stored in the memory modules on the left (in columns), and those of B in the memory modules

[11] In fact, applying the data-flow concept along uniform cuts will lead to a self-timed, regular, and locally interconnected array—a wavefront array. As a matter of fact, the "wavefronts" will correspond to the cuts.

on the top (in rows). The process starts with PE (1,1):

$$C_{11}^{(1)} = C_{11}^{(0)} + a_{11} * b_{11}$$

is computed. The computational activity then propagates to the neighboring PEs (1,2) and (2,1), which will executive in parallel

$$C_{12}^{(1)} = C_{12}^{(0)} + a_{11} * b_{12}$$

and

$$C_{21}^{(1)} = C_{21}^{(0)} + a_{21} * b_{11}.$$

The next front of activity will be at PEs (3,1), (2,2), and (1,3), thus creating a computational wavefront traveling down the processor array. It may be noted that wave propagation implies localized data flow. Once the wavefront sweeps through all the cells, the first recursion is over (see Fig. 19).

As the first wave propagates, we can execute an identical second recursion in parallel by pipelining a second wavefront immediately after the first one. For example, the (i,j) processor will execute

$$C_{ij}^{(2)} = c_{ij}^{(1)} + a_{i2} * b_{2j}$$

and so on.

1) Why the Name "Wavefront Array"?: The principle of wavefront processing is to successively pipeline the computational wavefronts as fast as resource and data availability allow, according to the concept of data-flow computing. As a justification for the name "wavefront array," we note that the computational wavefronts are similar to electromagnetic wavefronts (they both obey Huygens' principle) since each processor acts as a secondary source and is responsible for the propagation of the wavefront. In addition, wave-propagation implies localized data flow as well as localized control (handshaking). The pipelining is feasible because the wavefronts of two successive recursions will never intersect (Huygens' wavefront principle), thus avoiding any contention problems. (From the hardware perspective, the desired "separation" between two consecutive wavefronts is reaffirmed by the "separators" with proper handshaking.)

In other words, it is possible to have wavefront propagating in several different fashions. In the extreme case of nonuniform clocking, the wavefronts are actually crooked. However, what is important is that the order of task sequencing must be correctly followed. The correctness of the sequencing of the tasks is ensured by the wavefront principle [17].

2) Wavefront Array for LU Decomposition: By tracing backwards through the iterations in (3), we note that

$$C = c_{ij}^{(0)} = \sum_{k=1}^{N} a_i^{(k)} b_j^{(k)} = AB \qquad (4)$$

where $A = \{a_{mn}\} = \{a_m^{(n)}\}$, and $B = \{b_{mn}\} = \{b_n^{(m)}\}$ are the outputs of the array.

In comparison with (2) and (4), (3) is basically a reversal of the matrix multiplication recursions. Therefore, its wavefront processing should be similar to what is shown in Fig. 19. In fact, by converting the SFG as shown in Fig. 11 into a DFG, such a wavefront array can be directly obtained.

3) Least Square Error Solution and SVD: In many applications, we will be faced with solving a least square solution of an overdetermined linear system, as opposed to an exact solution of a nonsingular linear system. In this case, QR decomposition will prove to be much more useful than LU decomposition. The natural topology associated with QR decomposition is a square interconnect pattern. This can be shown by looking into the mathematical iterations and the corresponding SFGs. Similar systolic and wavefront arrays can be obtained by carrying out the systolization procedure or SFG/DFG equivalence transform. The details (omitted here) will be published in a later report.

The eigenvalue and SVD (singular value decomposition) problems are considerably more complicated. However, in [11], [21], it is shown that the notion of computational wavefronts can be employed to track down the activities in a square or linear array for computing eigenvalues or singular values.

VII. Wavefront Array Software/Hardware (WASH)

A. Programming Array Processors

The actual implementation of systolic or wavefront arrays can be either dedicated or programmable processors. Programmable arrays are preferred, due to the high cost of hardware implementation and the increasing varieties of application demands. Therefore, it is equally important to develop a complete set of software packages for most wavefront/systolic-type processing. For that, a formal algorithmic notation and programming language will be indispensable.

General guidelines for algorithmic notations for array processors are problem orientation, executability, and semantic simplicity. More importantly, an adequate language criteria must take into account the characteristics and the constraints of the arrays. Examples for appropriate array processing notations, which incorporate the language criteria for systolic/wavefront array processors, are CRYSTAL, [4] data space notation, [5] and the wavefront language (MDFL) [17].

B. Wavefront Language and Software Development

The effectiveness of programming in a processor array is directly related to the algorithm analysis technique. Our description of parallel algorithms hinges upon the notion of a computational wavefront. This leads to a special-purpose, wavefront-oriented language, termed Matrix Data Flow Language (MDFL) [17]. This denotation is in many ways very similar to the data space notation, which is based on the notion of applicative state transition systems described in [1], [5]. Among other commonalities shared by the two notations is, in particular, that they are both based on the data-flow principle [6].

The wavefront language is tailored towards the description of computational wavefronts and the corresponding data flow for the class of algorithms which exhibit the recursion and locality properties. Rather than requiring a program for each processor in the array, MDFL allows the programmer to address an entire front of processors. The wavefront idea can facilitate the description of parallel and pipelined algorithms and drastically reduce the complexity of parallel programming. To translate the global MDFL notation into microinstructions for the PEs, a preprocessor

is needed. For a wavefront array, the design of such a preprocessor is relatively easy, since we do not have to consider the timing problems associated with a synchronous systolic array.

As an example, let us now take a look at the computation of

$$C = A \times B.$$

The matrix multiplication can be carried out in N (outer product) recursions (see (1)). A general configuration of computational wavefronts traveling down a processor array is illustrated in Fig. 19. An example of an MDFL program for the corresponding array processing of the matrix multiplication is given in Fig. 20. (For the time being, please ignore the bracketed instructions.)

```
BEGIN
SET COUNT N;
REPEAT;
  WHILE WAVEFRONT IN ARRAY DO
    BEGIN
      {[FETCH C, DOWN;]}
        FETCH A, LEFT;
        FETCH B, UP;
        FLOW A, RIGHT;
        FLOW B, DOWN;
            (* NOW FORM C:= C + A × B)
        MULT A, B, D;
        ADD C, D, C;
      {[FLOW C, UP;]}
    END;
  DECREMENT COUNT;
UNTIL TERMINATED;
ENDPROGRAM.
```

Fig. 20. An MDFL program for matrix multiplication.

Note that, initially matrix A is stored (row by row) in the left Memory Module (MM). Matrix B is in the top MM and is stored column by column. The final result will be in the C registers of the PEs. This example illustrates the typical simplicity of the MDFL programming language.

1) Flexibilities with the Wavefront Programming: To demonstrate the flexibility of wavefront-type programmability, let us look at the multiplication of a banded-matrix A, $N \times N$, with bandwidth P, and a rectangular matrix B, $N \times Q$. Only a slight modification to the program in Fig. 20 is needed. First, the data storage in the memory modules will be the same as in Fig. 16(a). The major modification on the wavefront propagation is that, between the recursions of outer products, there should be an upward shift of the partial sums. (This is because the input matrix A is loaded in a skewed fashion.) Therefore, the program (see Fig. 20) remains almost the same, except for the two added bracketed instructions to shift partial sums upward.

Another major flexibility offered by the wavefront programming technique is that software reconfigurability can be used to map a linear or bilinear array onto a square array hardware. Therefore, a (hardwired) square array may be used for purpose of linear (or bilinear) wavefront array processing.

2) An Example on Linear Phase Filtering: In order to demonstrate the simplicity of programming based on the DFG representation, an MDFL program implementing the DFG for the linear phase filter (cf. Fig. 15(b)) is shown in Fig. 21.[12] The simple mapping between the DFG and the programming codes suggests a potentially significant impact of the SFG/DFG equivalence transformation to both the hardware and software developments of array processors.

```
BEGIN
REPEAT;
  WHILE WAVEFRONT IN ARRAY DO
    BEGIN
        FETCH X, LEFT;
        FLOW X, RIGHT;
        TRANSFER W2 TO W1;
        FLOW W1, LEFT;
        FETCH W2, RIGHT;
        (! NOW COMPUTE V:= (W1 + X) + × H(K))
        ADD W1, X, U;
        MULT U, H(K), V;
        FETCH Y, RIGHT;
        (! NOW COMPUTE Y:= Y + V)
        ADD Y, V, Y;
        FLOW Y, LEFT;
    END;
  DECREMENT COUNT;
UNTIL TERMINATED;
ENDPROGRAM.
```

Fig. 21. An MDFL program for linear-phase filter.

The power and flexibility of the wavefront array and MDFL programming are best demonstrated by the broad range of the applicational algorithms suitable for the wavefront array [17]. Such algorithms can be roughly classified into three groups:

1) Basic Matrix Operations: such as a) Matrix Multiplication, b) Banded-Matrix Multiplication, c) Matrix–Vector Multiplication, d) LU Decomposition, e) LU Decomposition with Localized Pivoting, f) Givens Algorithm, g) Back Substitution, h) Null Space Solution, i) Matrix Inversion, j) Eigenvalue Decomposition, and k) Singular Value Decomposition.

2) Special Signal Processing Algorithms: a) Toeplitz System Solver, b) One- and Two-Dimensional Linear Convolution, c) Circular Convolution, d) ARMA and AR Recursive Filtering, e) Linear Phase Filtering, f) Lattice Filtering, g) DFT, and h) Two-Dimensional Correlation (image matching).

3) Other Algorithms: PDE (partial difference equation) solution.

Note that, if the communication constraint is relaxed, our technique for converting an SFG for a given application into a wavefront array will also work with other global-type algorithms, such as the FFT algorithm, the Householder transformation, the Kalman filter network, or other non-regularly interconnected arrays. The only additional requirement lies in routing the physical connections between PEs.

It is worth noting that the cut-set rules can be potentially very useful for designing fault-tolerant arrays. For systolic arrays without feedback, it has been shown in [32], [33] that a retiming along cut-sets allows a great degree of fault tolerance. The discussion in Section III-B should offer a theoretical basis for improving fault-tolerance of arrays with feedback via the cut-set retiming procedure. More interest-

[12]Note that the separators in the DFG are implemented simply by adding three lines of (internal register–transfer) code to the program, as opposed to adding a separate buffer register external to the PE.

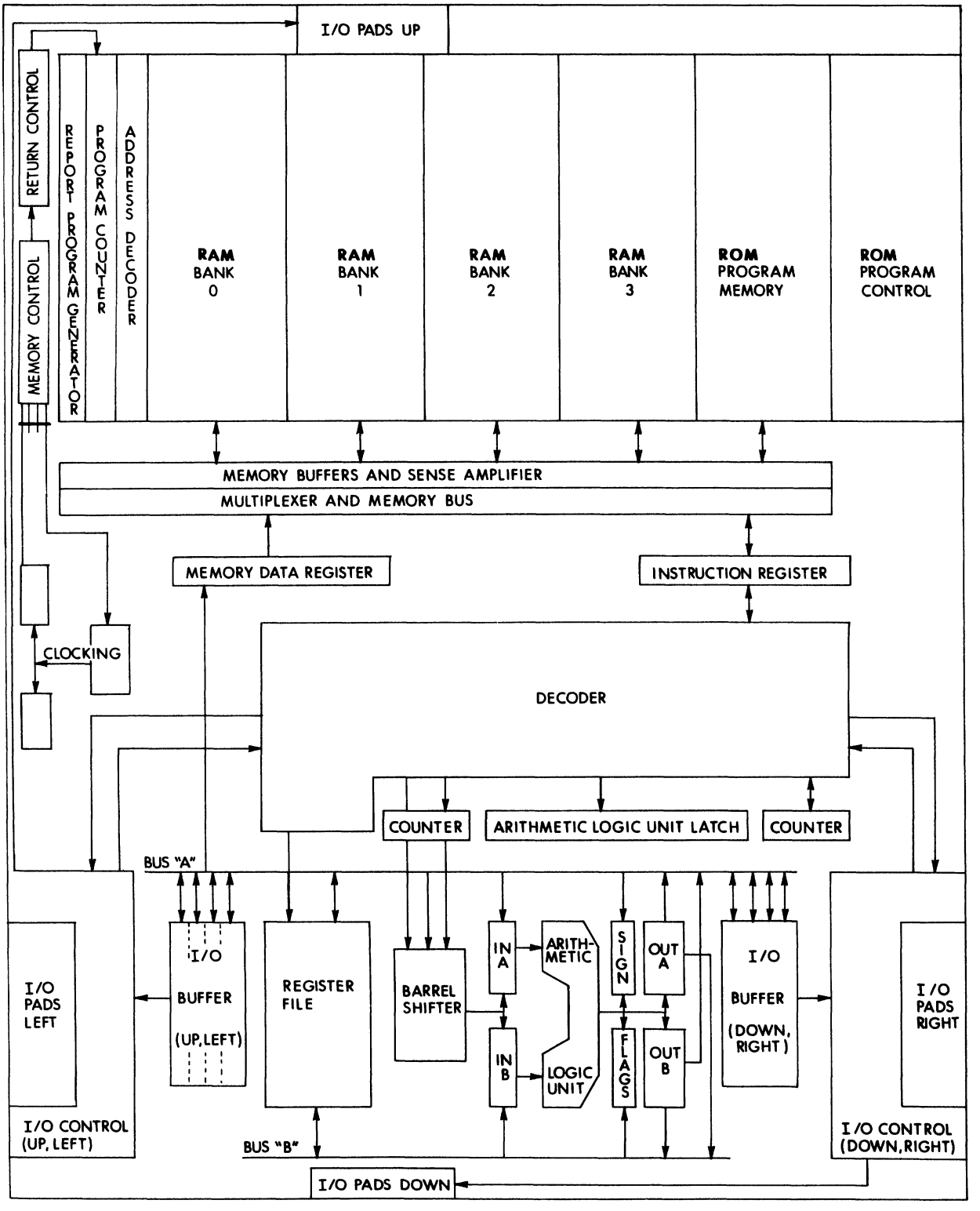


Fig. 22. Functional block diagram of wavefront PE.

ingly, with a slight modification, the self-timed feature of wavefront arrays offers a way of achieving the same fault-tolerance efficiency without any need of retiming.

In summary, in the first phase of the software development project, we 1) define the application/ algorithm domain, 2) develop a language tailored to the application, and 3) design a (language-based) wavefront architecture. In order to maximize the application algorithm domain (with minimal hardware overhead), the next phase is to develop a complete software library of all algorithms suitable for systolic/wavefront-type parallel processing. This software library, combined with design automation tools such as silicon compilers, will facilitate the construction of future VLSI systems design. The success of the project will demand joint and cohesive efforts from all related disciplines. For this end, we welcome any suggestions from interested readers and colleagues.

C. Hardware Design

In this subsection we will give an overview of the architecture of a PE, to be used as a basic module in a programmable array processor. A typical example will be the design of a PE to be used in a wavefront array. The basic wavefront array is either a square array of $N \times N$ PEs, a linear array of $1 \times N$ PEs, or a bilinear array of $2 \times N$ PEs. The PEs are orthogonally connected and are identical. The hardware of the PE is designed to support the features of the Matrix Data Flow Language (MDFL) introduced previously. Given the current state of the process technology, with a minimum feature size of 2 μm or less, we estimate the area of the chip taken by a PE to be 6×6 mm^2.

1) Architectural Outline: The PE that we have designed is a special-purpose microprocessor. The functional block diagram of the PE is shown in Fig. 22. The main functional blocks are datapath, program memory, I/O control units, and instruction decoder.

Our design objective is to limit the complexity of the datapath, preferring a regular and easy layout design. We have adopted a 32-bit-wide datapath for fixed-point computations. Moreover, the ALU in the PE is designed to support the operations that are of major importance for signal processing applications, such as multiplication and rotation. To speed up the throughput of the PE, we used a two-level pipelining scheme.

The PE can simultaneously perform data transfers in four directions. The transfer of data is controlled by an I/O controller, one for each of the four directions, which handles the two-way handshaking functions.

2) Instruction Set: The instruction set of the PE was selected to optimize the performance of the wavefront array as a whole. To reduce the complexity of the control unit, in a manner similar to that used in the RISC design [28], we wanted each instruction to take exactly one clock cycle. This implies that complex instructions should be decomposed into sequences of simpler (primitive) instructions. An example is the multiplication instruction, which is decomposed into three instructions: one for initializing the processor registers with the correct data, one that does the main multiplication step, and the last one which transfers the result back to the register file. The instruction set is divided into arithmetic instructions, register transfer instructions, conditional and unconditional jump instructions, and program loading instructions.

3) Design Specification and Verification: The PE described in this section is currently being specified and verified using the ISPS language. The ISPS language allows not only the specification of the design of a single PE, but also the simulation of an entire wavefront array. More importantly, it facilitates the verification of the correctness and suitability of the architecture of the PE before designing the lower (logic, circuit, layout) levels of the PE. It will also be an important tool for the development of the host interface, the memory units, etc.

VIII. Conclusions

The rapid advance in VLSI device technology and design techniques have encouraged the development of massively parallel-array processors. We have stressed the importance of modularity, communication, and system clocking in the design of VLSI arrays. For signal processing applications, a large number of algorithms possess the properties of recursiveness and locality. These properties naturally led to the systolic and wavefront arrays.

The two types of arrays share the important common feature of using a large number of modular and locally interconnected processors for massive pipelined and parallel processing. However, in several key aspects, the wavefront array is noticeably distinctive from the systolic array: First, it uses data-driven computing and thus gets around the burden of having to synchronize a (potentially ultra-) large-scale array. [18] Second, it maximizes the pipelining efficiency and offers a speed achievable only by multirate systolic arrays. Third, the data-flow principle allows a simpler language design facilitating a formal description of the activities. Finally, it can easily cope with the variations of communication delays in dynamically interconnected systems, such as reconfigurable waferscale integration designs.

In conclusion, we have shown that both the systolic and data-flow principles will play a major role in future supercomputing, especially for number crunching problems. Most computing networks described in signal-flow graphs (SFGs) can be systematically converted into systolic or wavefront arrays, following the procedures proposed. This should encourage more practitioners to develop advanced hardware and software for massively parallel-array processors. The impacts of the novel architectures upon future supercomputer designs cannot be overestimated.

Acknowledgment

The author wishes to thank his colleagues in the VLSI signal processing group at the University of Southern California, for their very valuable contributions to the Wavefront Array Software/Hardware (WASH) Project. He is also grateful to Dr. H. Lev-Ari of the Stanford University for many useful comments which were incorporated into this final draft.

References

[1] J. Backus, "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs," *Commun. ACM*, vol. 21 pp. 613–641, 1978.

[2] C. Caraiscos and B. Liu, "From digital filter flow-graphs to systolic arrays, " submitted to *IEEE Trans. Acoust., Speech, Signal Processing*, 1984.

[3] H. J. Caulfield, W. T. Rhodes, M. J. Foster, and S. Horvitz, "Optical implementation of systolic array processing," *Opt. Commun.*, vol. 40, pp. 86–90, Dec. 1981.

[4] M. Chen and C. Mead, "Concurrent algorithms as space-time recursion equations," to appear in *VLSI and Modern Signal Processing*, S. Y. Kung *et al.*, Eds., Englewood Cliffs, NJ: Prentice Hall.

[5] A. B. Cremers and T. N. Hibbard, "The semantic definition of programming languages in terms of their data spaces," *Informatik-Fachberichte*, vol. 1, pp. 1–11 (Berlin, Springer-Verlag), 1976.

[6] A. B. Cremers and S. Y. Kung, "On programming VLSI concurrent array processors," in *Proc. IEEE Workshop on Languages for Automation* (Chicago, IL, 1983), pp. 205–210; also in *INTEGRATIONS* (The VLSI Journal), vol. 2, no. 1, Mar. 1984.

[7] J. B. Dennis, "Data flow supercomputers," *IEEE Computer*, vol. 13, pp. 48–56, Nov. 1980.

[8] P. Dewilde, personal communication, 1983.

[9] J. J. Dongarra *et al.*, *LINPACK USER'S GUIDE*. Philadelphia, PA: SIAM PUB., 1979.

[10] M. Franklin and D. Wann, "Asynchronous and clocked control structures for VLSI based interconnection networks," presented at the 9th Annual Symp. on Computer Architecture, Apr. 1982, Austin, TX.

[11] R. J. Gal-Ezer, "The wavefront array processor," Ph.D. dissertation, Dept. of Electrical Engineering, University of Southern California, Dec. 1982.

[12] J. W. Goodman, F. Leonberger, S. Y. Kung, and R. Athale, "Optical interconnections for VLSI systems," this issue, pp. 850–866; also prepared as an ARO Palantir Meeting Report.

[13] K. Hwang and F. Briggs, *Computer Architectures and Parallel Processing*. New York: McGraw-Hill, 1984.

[14] J. V. Jagadish, T. Kailath, G. G. Mathews, and J. A. Newkirk, "On pipelining systolic arrays," presented at the 17th Asilomar Conf. on Circuits, Systems, and Computers, Pacific Grove, CA, Nov. 1983, also "On hardware description from block diagrams," in *Proc. IEEE ICASSP* (San Diego, CA, Mar. 1984).

[15] H. T. Kung, "Why systolic architectures," *IEEE Computer*, vol. 15, no. 1, Jan. 1982.

[16] H. T. Kung and C. E. Leiserson, "Systolic arrays (for VLSI)," in *Proc. Sparse Matrix Symp.* (SIAM), pp. 256–282, 1978.

[17] S. Y. Kung, K. S. Arun, R. J. Gal-Ezer, and D. V. Bhaskar Rao, "Wavefront array processor: Language, architecture, and applications," *IEEE Trans. Comput.* (Special Issue on Parallel and Distributed Computers), vol. C-31, no. 11, pp. 1054–1066, Nov. 1982.

[18] S. Y. Kung and R. J. Gal-Ezer, "Synchronous vs. asynchronous computation in VLSI array processors," in *Proc. SPIE Conf.* (Arlington, VA), 1982.

[19] S. Y. Kung and J. Annevelink, "VLSI design for massively parallel signal processors," *Microprocessors and Microsystems* (Special Issue on Signal Processing Devices), vol. 7, no. 4, pp. 461–468, Dec. 1983.

[20] S. Y. Kung, "From transversal filter to VLSI wavefront array," in *Proc. Int. Conf. on VLSI 1983*, IFIP (Trondheim, Norway), 1983.

[21] S. Y. Kung and R. J. Gal-Ezer, "Eigenvalue, singular value and least square solvers via the wavefront array processor," in *Algorithmically Specialized Computer Organizations*, L. Snyder *et al.*, Eds. New York: Academic Press, 1983.

[22] C. E. Leiserson, "Area-efficient VLSI computation," Ph.D. dissertation, Carnegie-Mellon University, Pittsburgh, PA, Oct. 1981.

[23] C. E. Leiserson, F. M. Rose, and J. B. Saxe, "Optimizing synchronous circuitry by retiming," in *Proc. Caltech VLSI Conf.* (Pasadena, CA), 1983.

[24] C. E. Leiserson and J. B. Saxe, "Optimizing synchronous systems," *J. VLSI Comput. Syst.*, vol. 1, no. 1, pp. 41–67, 1983.

[25] H. Lev-Ari, "Modular computing networks: A new methodology for analysis and design of parallel algorithms/architectures," Tech. Memo ISI-29, Integrated Systems Inc., Palo Alto, Ca., 1983.

[26] C. Mead and L. Conway, *Introduction to VLSI Systems*. Reading, MA: Addison-Wesley, 1980.

[27] A. Oppenheim and R. Schafer, *Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1975.

[28] D. A. Patterson and C. H. Sequin, "A VLSI RISC," *IEEE Computer*, vol. 14, Sept. 1981.

[29] B. T. Smith *et al.*, *Matrix Eigensystem Routines, EISPACK Guide*, vol. 6 of *Lecture Notes in Computer Science*, 2nd ed. New York: Springer-Verlag, 1976.

[30] J. D. Ullman, *Computational Aspects of VLSI*. Rockville, MD: Computer Science Press, 1984.

[31] A. Fettweis, "Realizability of digital filter networks," *AEU* (Electronics and Communication), vol. 30, pp. 90–96, 1976.

[32] H. T. Kung and M. S. Lam, "Fault-tolerant VLSI systolic arrays and two-level pipelining," in *Proc. SPIE Symp.*, vol. 431, pp. 143–158, Aug. 1983.

[33] ____, "Fault-tolerance and two-level pipelining in VLSI systolic arrays," in *Proc. Conf. on Advanced Research in VLSI* (MIT, Cambridge, MA, Jan. 1984), pp. 74–83.

Wafer-Scale Integration and Two-Level Pipelined Implementations of Systolic Arrays*,†

H. T. KUNG AND MONICA S. LAM

Department of Computer Science, Carnegie–Mellon University, Pittsburgh, Pennsylvania 15213

Two important issues in systolic array designs are addressed: How is fault tolerance provided in systolic arrays to enhance the yield of wafer-scale integration implementations? And, how are efficient systolic arrays with two levels of pipelining designed? (The first level refers to the pipelined organization of the array at the cellular level, and the second refers to the pipelined functional units inside the cells.) The fault-tolerant scheme proposed replaces defective cells with clocked delays. This has the distinct characteristic that data can flow through the array with faulty cells at the original clock speed. It is shown that both the defective cells under this fault-tolerant scheme and the second-level pipeline stages can simply be modeled as additional delays in the data paths of "generic" systolic designs. The mathematical notion of a *cut* is introduced to solve the problem of how to allow for these extra delays while preserving the correctness of the original systolic array designs. The results obtained by applying these techniques are encouraging. When applied to systolic arrays without feedback cycles, the arrays can tolerate large numbers of failures (with the addition of very little hardware) while maintaining the original throughput. Furthermore, all of the pipeline stages in the cells can be kept fully utilized through the addition of a small number of delay registers. However, adding delays to systolic arrays with cycles typically induces a significant decrease in throughput. In response to this, a new class of systolic algorithms has been derived in which the data cycle around a ring of processing cells. The *systolic ring* architecture has the property that its performance degrades gracefully as cells fail. Use of the cut theory and ring architectures for arrays with feedback gives effective fault-tolerant and two-level pipelining schemes for most systolic arrays. As a side effect of developing the ring architecture approach, several new systolic algorithms have been derived. These algorithms generally require only one-third to one-half of the

*A preliminary version appeared in *Proceedings of the Conference on Advanced Research in VLSI*, MIT, January 1984.

†This research was supported in part by the Office of Naval Research under Contracts N00014–76–C–0370, NR 044–422 and N00014–80–C–0236, NR 048–659, and in part by the Defense Advanced Research Projects Agency (DoD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615–81–K–1539.

number of cells used in previous designs to achieve the same throughput. The new systolic algorithms include ones for LU-decomposition, QR-decomposition, and the solution of triangular linear systems.

1. Introduction

In recent years many systolic algorithms have been designed and several prototypes of systolic array processors have been constructed [1–4]. Major efforts are currently devoted to building systolic arrays for large, real-life applications. In this paper, we consider two implementation techniques for building high-performance systolic arrays: wafer-scale integration (WSI) and fabrication using pipelined components.

Fabrication flaws on a wafer are inevitable. It is necessary for a WSI circuit to be "fault tolerant" so that wafers with defective components can still be used. A common approach is to include redundant circuitry in the design and avoid defects by programming the interconnection of the constituent elements. In particular, the laser-programming technology has been applied successfully to program the redundant circuitry in VLSI RAMs as a yield enhancement measure [5]. The MIT Lincoln Laboratory [6] has also been experimenting with the use of laser-programmable links to build wafer-scale processor arrays.

Systolic arrays are well suited to the application of wafer-scale integration. They consist of large numbers of small and identical (thus interchangeable) cells and their regular and localized interconnections greatly simplify the problem of routing around defective cells. On the other hand, systolic architectures guarantee full exploitation of their constituent cells to achieve maximum parallelism. The more cells an array has, the more powerful it is. Wafer-scale integration has the potential to provide a very cost-effective and reliable way of implementing high-performance systolic systems.

Before WSI systolic arrays can become a reality, we must solve the problem of how to construct fault-tolerant arrays. After the cells are tested (by wafer probing, for example), how do we route around the defects to build a functional array? (See Figure 1.1a.) This paper describes a *systolic* approach which provides fault tolerance at a very low cost and admits of a graceful degradation in performance as the number of defects increases.

The use of pipelined components for implementing cells of systolic arrays

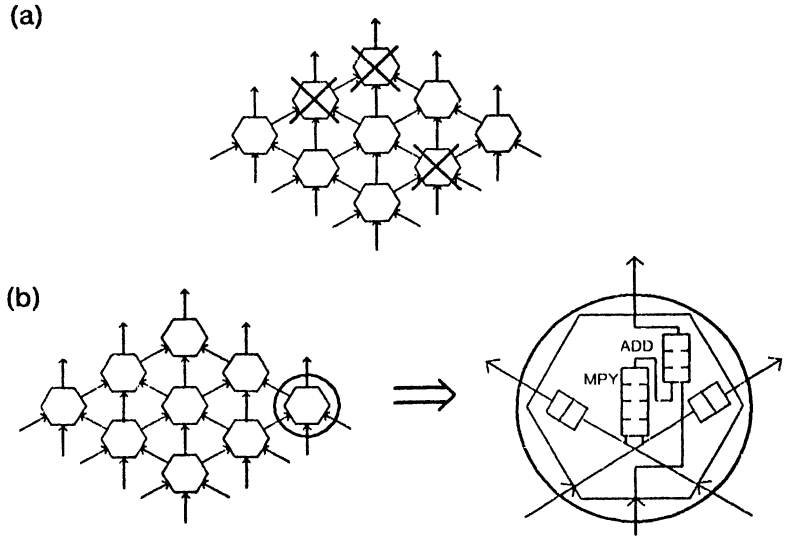


FIG. 1.1. Two problems addressed in the paper: (a) fault tolerance for arrays with faulty cells, and (b) two-level pipelining.

is especially attractive for applications requiring floating-point operations. Commercially available floating-point multiplier and adder chips can deliver up to 5 MFLOPS per device. To achieve such high throughput, they typically have three or more pipeline stages [7]. These components, when used to implement systolic cells, form a *second level of pipelining* [8], the first being the pipelined organization of systolic arrays at the cellular level. While this additional level of pipelining can increase the system throughput, it considerably complicates the design of systolic array algorithms. Our solution to this problem is to devise a methodology to transform existing systolic designs which assume single-stage cells to arrays consisting of pipelined cells.

We will show that both the "fault-tolerance" and the "two-level pipelining" problems can be solved by the same mathematical reasoning and techniques. Our results imply that once a "generic" systolic algorithm is designed, other versions of the algorithm (for execution on arrays with failed cells, or for implementation using different pipelined processing units) can be systematically derived. The techniques of this paper can also be applied to other computation structures, such as FFT processor arrays.

In the next section we introduce our approach to the problems, using as an example the simplest type of systolic arrays—unidirectional linear arrays. As we will see, systolic arrays without feedback admit of a much simpler solution and they are discussed in Section 3. In Section 4, we propose a new architecture, the "systolic ring," which can be used in place of many systolic arrays with feedback cycles and are much more amenable to fault-tolerant measures. Section 5 includes a summary and some concluding remarks.

2. Fault Tolerance and Two-Level Pipelining for Unidirectional Linear Arrays

Figure 2.1 depicts a systolic array [9] for the convolution computation with four weights $w_1, \ldots, w_4$. In this array the data flow only in *one* direction; that is, both x_i and y_i move from left to right (with x_i going through an additional "delay register" following each cell). This is an example of a systolic array without feedback cycles—an array where none of the values in any data stream depends on the preceding values in the same stream. (For an example of an array with feedback cycles, see Figure 4.1a.)

Depicted in Fig. 2.2a is an example of a 5-cell array with one faulty element. The defective cell in the middle is replaced with two "bypass" registers (drawn in dashed lines)—one for the x-data stream and one for the y-data stream. It can easily be shown that this array correctly solves the same problem as the array of Fig. 2.1. For example, y_1 picks up $w_4 \cdot x_4$, $w_3 \cdot x_3$, and $w_2 \cdot x_2$ at the first, second, and fourth cell, respectively. The degradation in performance due to the defect is slight. The maximum convolution computed by this array in one pass can have only 4 rather than 5 weights, and the latency of the solution is increased by one cycle. However, the computational throughput, often the most important factor in performance, remains the same at one output per cell cycle. Figure 2.2b depicts the cell specification for this fault-tolerant scheme, using reconfigurable links. Note that the input/output register in a systolic cell can be used as a bypass register in case the cell fails. Therefore no extra registers are needed to implement this fault-tolerant scheme.

A basic assumption of this paper is that the probability of the interconnecton links and registers failing is very small and thus negligible. This is reasonable because these components are typically much simpler and smaller than the cells themselves. Furthermore, they can be implemented conservatively and/or with high redundancy to increase the yield.

In the proposed scheme data move through all the cells. At failed cells, data items are simply delayed with bypass registers for one cycle, and no computation is performed (Figure 2.3a). We call fault-tolerant schemes of this type *systolic* in view of the fact that data travel *systolically* in a defective array from cell to cell, at the original clock speed.

For unidirectional linear arrays, the systolic fault-tolerant scheme proposed here has the advantages that it utilizes *all* the live cells and maintains the

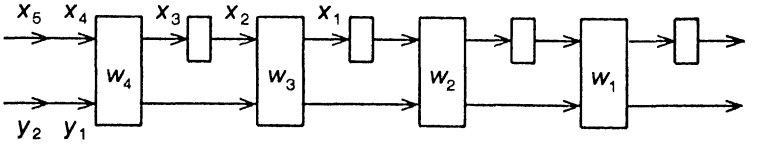


FIG. 2.1. Unidirectional linear systolic array for convolution.

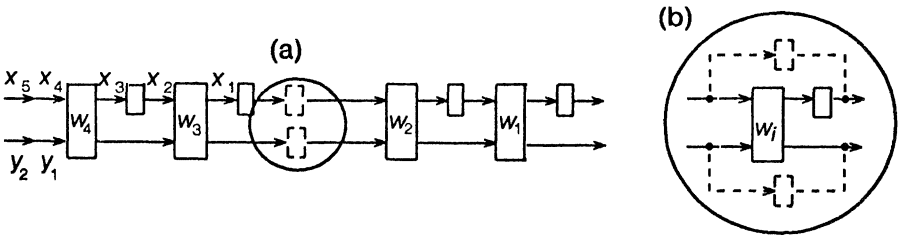


FIG. 2.2. (a) Defective cell replaced with registers, and (b) cell specification.

throughput rate of a flawless array (Figure 2.3a). As illustrated in Figure 2.3b, other fault-tolerant schemes previously proposed in the literature either suffer from low utilization of live cells [10–13] or reduced throughput due to a slower system clock required by the fact that the communication between logically adjacent cells can now span a large number of failures [14–17]. Moreover, as will be shown in the next section, our systolic fault-tolerant technique can be generalized to two-dimensional arrays.

We now examine more carefully the idea behind our fault-tolerant scheme for the linear array of Fig. 2.2. Because of the unit delay introduced by the bypass registers, all the cells after the failed one receive data items one cycle later than they normally would. Since both the x- and y-data streams are

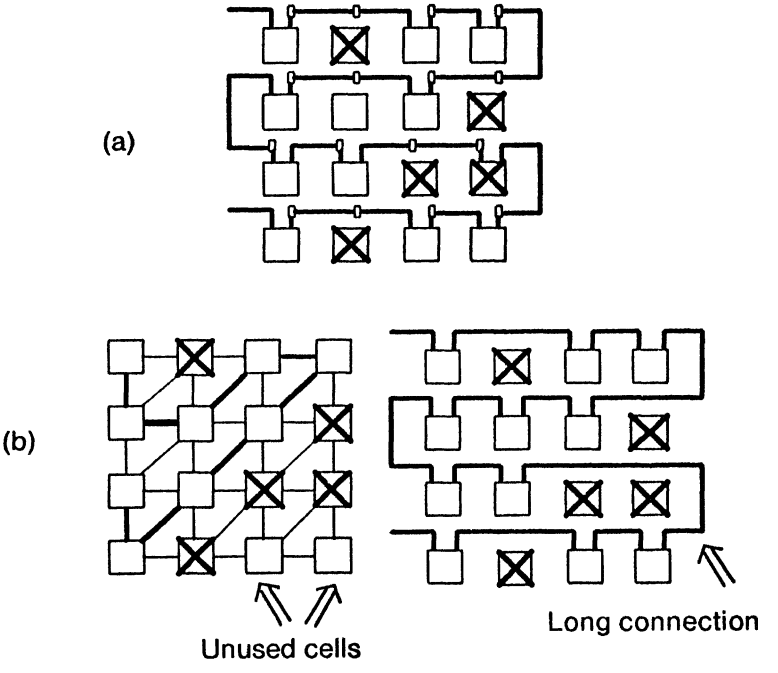


FIG. 2.3. (a) Systolic and (b) previous fault-tolerant schemes for unidirectional linear arrays.

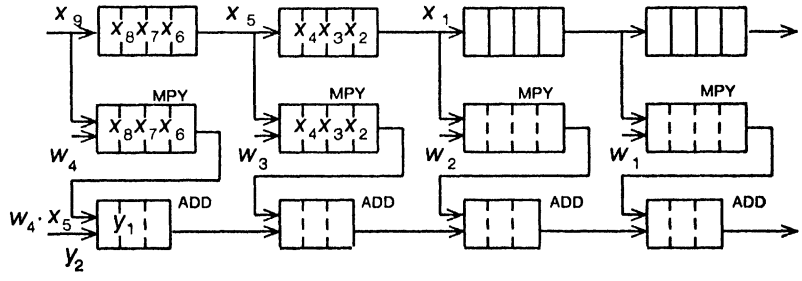


FIG. 2.4. Two-level pipelined systolic array for convolution, using pipelined arithmetic units of Fig. 1.1b.

delayed by the same amount, the relative alignment between the two data streams remains unchanged. Thus, all the cells after the third one receive the same data and perform the same function, with a one-cycle delay, as would the cell preceding it in a normal array. For this reason, an n-cell, unidirectional, linear array with k defective cells will perform the same computation as a perfect array of $n - k$ cells.

The above reasoning also implies that the correctness of a unidirectional linear array is preserved, if the *same* delay of any length of time is introduced uniformly to *all* the data streams between two adjacent cells. This result is directly applicable to the implementation of two-level pipelined arrays. We can interpret the stages in a given pipelined processing unit as additional delays in the communication between a pair of adjacent cells.

Consider, for example, the problem of implementing the systolic array of Fig. 2.1 using the pipelined multiplier and adder of Fig. 1.1b. Since the adder is now a three-stage pipeline unit instead of a single-stage unit, two additional delays are introduced in the y-data path. Thus each cell requires a total number of four delay registers be placed in the x-data path—one is implicit in the original cell definition, the second is the delay register in the original algorithm design, and the last two are to balance the two new delays in the y-data stream. The resulting two-level pipelined array is depicted in Fig. 2.4. This design has been proposed previously [8], but it is reproduced here as a special example of a general theory.

3. SYSTOLIC ARRAYS WITHOUT FEEDBACK CYCLES

From the previous section we see that both the defective cells in a fault-tolerant array and the pipeline stages in systolic cells can simply be modeled as additional delays in the data paths. Thus by solving the one problem of how, if possible, to allow for additional delays in systolic designs, we can transform generic systolic designs to fault-tolerant or two-level pipelined

designs. A general theory of adding and removing register delays to a system has been proposed by Leiserson and Saxe [18] in the context of optimizing synchronous systems.

3.1. The Cut Theorem

We model a systolic array as a directed graph, with the nodes denoting the combinational logic and the edges the communication links [19]. The edges are weighted by the number of registers on the links. We say that two designs are *equivalent* if, given an initial state of one design, there exists for the other design an initial state such that (with the same input from the host, i.e., the outside world) the two designs produce the same output values (although possibly with a constant delay). In other words, as far as the host is concerned, the designs are interchangeable provided the possible differences in the timing of the output are taken into account.

We define a *cut* to be a set of edges that "partitions" the nodes in a graph into two disjoint sets, the *source set* and the *destination set,* with the property that these edges are the only ones connecting nodes in the two sets and are all directed from the source to the destination set.

We say that a systolic design is a "delayed" system of another design if the former differs from the latter by having additional delays on some of the communication links. Thus the graph representations of the two designs are the same except for the weights on the edges that correspond to the communication links with additional delays.

THEOREM 1. (Cut Theorem). *For any design, adding the same delay to all the edges in a cut and to those pointing from the host to the destination set of the cut will result in an equivalent design.*

Proof. Let S be the original design partitioned by a cut into sets A and B, the source and the destination set, respectively. Let S' be the same as S (with its corresponding sets A' and B'), with the difference that d delays are now added onto the edges in the cut. We will show that by properly initializing S' (at t_0), the output values from A and A' will be identical and that the output values from B are the same as those from B', but lagging behind by d clock cycles.

We define the initial state of A' to be identical to the state of A at time t_0. Since none of the edges in the cut feed into A', directly or indirectly, nodes in A' behave exactly the same way as the corresponding ones in A and thus produce the same outputs.

Let $r_1(e'), \ldots, r_d(e')$ be the delay registers on any edge in the cut, e', with $r_1(e')$ being closest to the source node and $r_d(e')$ closest to the destination node. First, we assign the initial state of B' to be identical to the state of B at time $t_0 - d$. We then initialize the registers $r_1(e'), \ldots, r_d(e')$ with the values of the data on the corresponding edge in S at time $t_0 - 1$, $t_0 - 2, \ldots, t_0 - d$, respectively. In this way, the input data received by the

nodes in set B' from time t_0 to $t_0 + d - 1$ are identical to those received by B from $t_0 - d$ to $t_0 - 1$ and so the configuration of B' at $t_0 + d$ and that of B at t_0 are identical. Since the outputs from A' are the same as those from A, all the inputs arriving at B' starting from time $t_0 + d$ are the same as those arriving at B, except that they lag behind by d cycles due to the additional delay registers. Therefore the nodes in B' will behave the same way as the corresponding ones in B with a d cycle delay. ■

We say that a delayed system S' is *derivable* from S if there exists a set of cuts $C_1, C_2, \ldots, C_n$ with their cut delays $d_1, d_2, \ldots, d_n$, such that

$$\forall\, e' \in S', \text{ number of additional delays on } e' = \sum_{\{i \mid e' \in C_i\}} d_i.$$

Since equivalence is associative, the cut theorem implies that if a "delayed" design is derivable from the original design then the two designs are equivalent.

Since a cut partitions the nodes of a graph into two sets with data flowing unidirectionally between them, it cannot cross any feedback cycle. On the other hand, for any given edge not in a feedback cycle, we can always construct a cut set that contains it. Therefore any number of delays on the data paths in a graph without feedback can always be incorporated if we have the option of inserting other delays into the system.

3.2. Linear Arrays without Feedback

We will now apply the above results to the examples we discussed previously. As depicted in Figure 3.1a, the edges between any two adjacent cells of a unidirectional linear array form a cut. Hence by the cut theorem, we can see immediately that both the defective array of Fig. 2.2a and the two-level pipelined array of Fig. 2.4 are equivalent to the original array of Fig. 2.1. Figure 3.1b depicts a less obvious cut, consisting entirely of all the output edges from the multipliers. This implies that the convolution array will function correctly regardless of the number of pipeline stages present in the multipliers (provided the number is the same for all the multipliers in the array). For instance, if all the four-stage multipliers in Fig. 2.4 were replaced with ten-stage multipliers, the resulting systolic convolution array would still be correct.

3.3. Two-Level Pipelining for Two-Dimensional Systolic Arrays

It is just as simple to apply the cut theorem to two-level pipelined arrays of two dimensions. Let us consider the example of a hexagonal systolic array that can perform band matrix multiplication [20] (Fig. 3.2a). Two results follow directly from the cut theorem:

1. The edges under each dashed line in Fig. 3.2a define a cut. All vertical

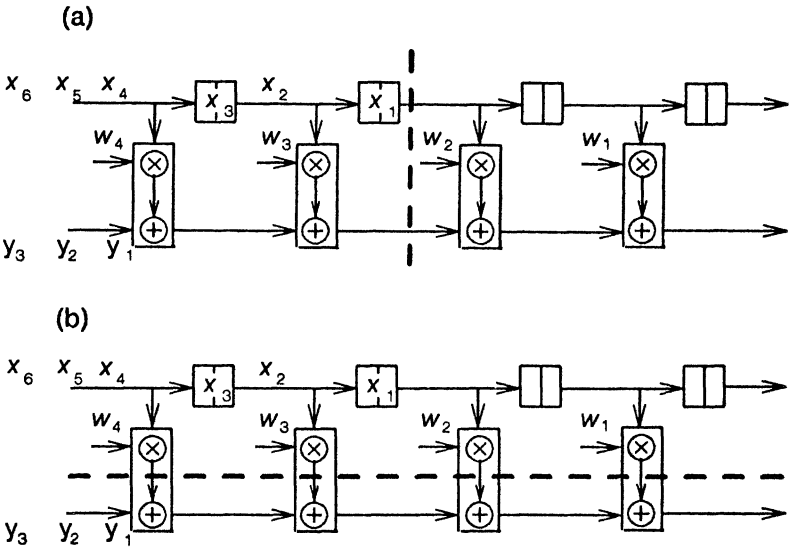


FIG. 3.1. Two types of cuts for a unidirectional linear systolic array for convolution.

edges, each representing an adder's output (Fig. 3.2b), intersect two dashed lines while all the other edges intersect only one. Thus by the cut theorem, if the number of pipeline stages in all the adders is increased by $2k$, then for each cell, k delays must be added to the other data paths. Figure 1.1b depicts the case when $k = 1$.

2. Consider the output edges of all the multipliers in the array. Like those in the unidirectional linear convolution array (Fig. 3.1b), these edges define a cut since none of the outputs from the adders are fed back into the multipliers. By the cut theorem, we conclude that these systolic cells can be implemented using pipelined multipliers of any number of stages without any further modification, provided the number of stages is the same for all the multipliers.

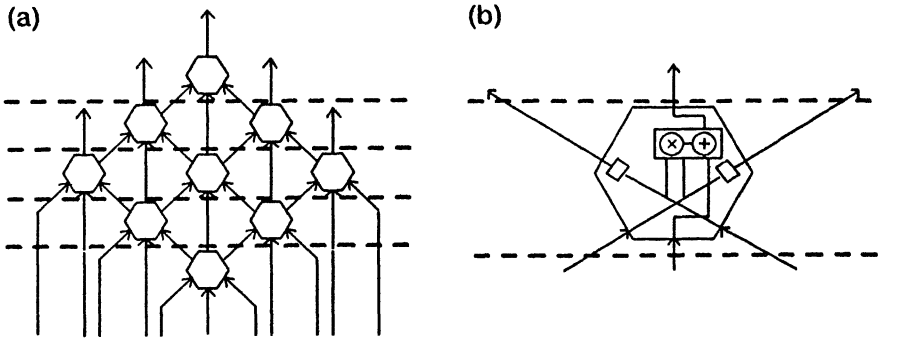


FIG. 3.2. (a) Hexagonal systolic array without feedback loops, and (b) original cell definition.

3.4. *Two-Level Pipelining for FFT Processor Arrays*

The cut theorem can be applied to two-level pipelined designs for any processor arrays without cycles. We consider here, as an example, the well-known processor array for computing fast Fourier transforms (FFTs). For an n-point FFT, the array has $\log_2 n$ stages of $n/2$ processors for performing butterfly operations. The data are shuffled between any two consecutive stages according to a certain pattern [21, 22]. Figure 3.3 depicts the so-called constant geometry version of the FFT algorithm (for $n = 16$), that allows the same pattern of data shuffling to be used for all stages. In the figure the processors for butterfly operations are represented by circles, and the number h by an edge indicates that the result associated with the edge must be multiplied by ω^h, where ω is a primitive nth root of unity.

A butterfly operation,

$$\begin{aligned}(a_{\text{real}} + ja_{\text{imag}}) \pm (b_{\text{real}} + jb_{\text{imag}}) \cdot (w_{\text{real}} + jw_{\text{imag}}) \\ = [a_{\text{real}} \pm (b_{\text{real}} \cdot w_{\text{real}} - b_{\text{imag}} \cdot w_{\text{imag}})] \\ + j[a_{\text{imag}} \pm (b_{\text{real}} \cdot w_{\text{imag}} + b_{\text{imag}} \cdot w_{\text{real}})],\end{aligned}$$

involves four real multiplications and six real additions. Figure 3.4a depicts a straightforward processor implementation for the butterfly operation using four multipliers and six adders. The time that the processor takes to perform a butterfly operation is the total delay of one multiplier and two adders.

To increase the throughput for calculating butterfly operations, we implement the processors with pipelined multipliers and adders. Suppose that these functional units each have five pipeline stages, as in the case of some recent floating-point chips [7]. By the cut theorem, the pipeline delays on the b_{real} and b_{imag} data paths have to be balanced by the same number of delays on the

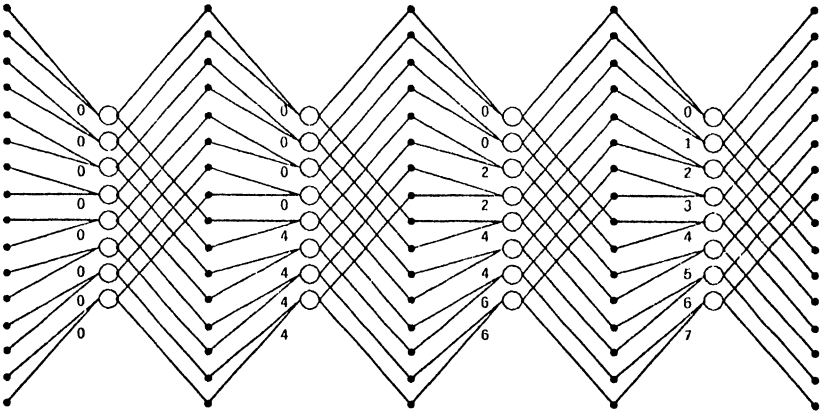


FIG. 3.3. Constant geometry version of the FFT algorithm.

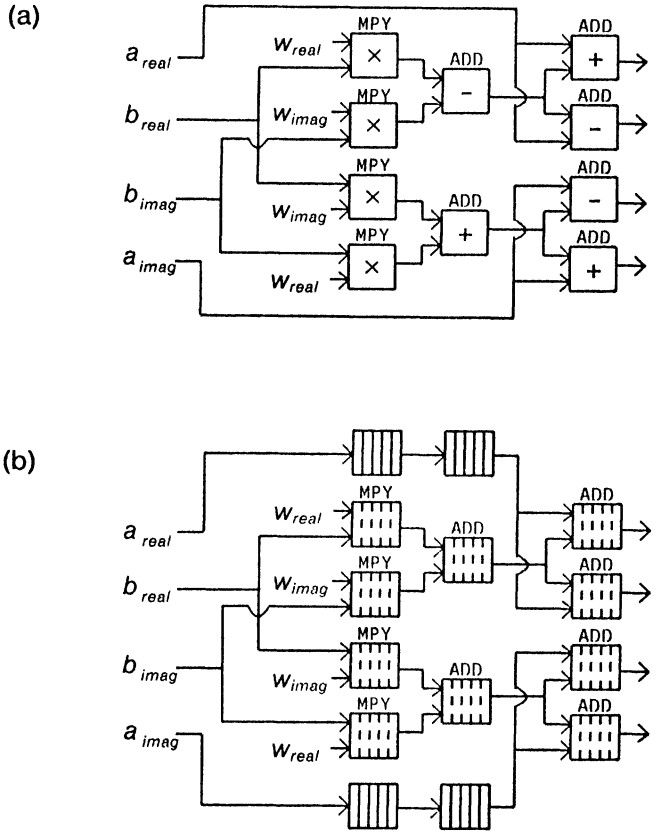


FIG. 3.4. (a) Processor for the butterfly operation, and (b) corresponding two-level pipelined processor.

a_{real} and a_{imag} input lines. The two-level pipelined design of the processor is shown in Figure 3.4b.

3.5. Systolic Fault-Tolerant Schemes for Two-Dimensional Arrays

Let us consider as an example the rectangular array of Figure 3.5a, where the data move forward and downward. Among many other applications, this array can perform matrix multiplication with either an operand or the partial result matrix stored in the array during the computation. We will first discuss the constraints that a correct implementation must satisfy and then we will study several redundancy schemes.

3.5.1. *The Local Correctness Criterion*

By exploiting the regularity in systolic arrays, the following theorem reduces the problem of establishing equivalence between two designs to smaller problems which can be solved using only "local information".

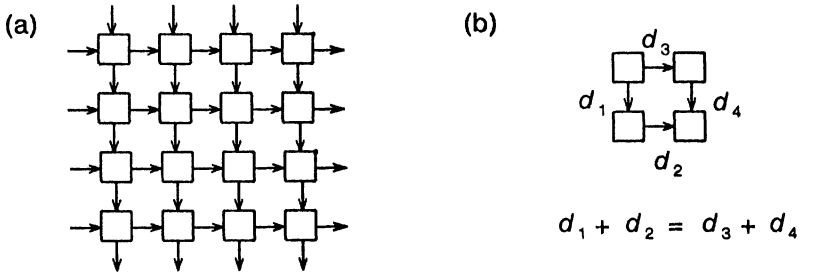


FIG. 3.5. (a) Rectangular systolic array without feedback loops, and (b) local correctness criterion.

THEOREM 2. *Let S be a mesh-connected systolic design without feedback and S' be a "delayed" version. S' is equivalent to S if for each square of adjacent cells in the grid, the number of delays on each of the two paths joining the two diagonally opposite corners is the same.*

Proof. Let V_i and E_i be the nodes and (vertical) edges in the ith column in grid S'. We form two subgraphs G_1' and G_2', such that G_1' contains all the nodes and edges to the left of the ith column and G_2' contains all those to the right, and in addition, they each contain V_i and E_i. We will first show that graph S' is derivable from S if subgraphs G_1' and G_2' are derivable from the corresponding subgraphs in S, G_1 and G_2, respectively.

Let C be a cut in subgraph G_1'. If C does not intersect E_i, all the nodes in V_i must belong to the destination set of the cut. Since there are no direct links between any nodes in the source set and the nodes in $S' - G_1'$, C is also a cut in S'. By the same token, any cut in subgraph G_2' that does not contain any edges in E_i is a cut in S'.

It is obvious that a cut can have at most one edge in E_i. Suppose the cuts C_1 in G_1' and C_2 in G_2' both contain the same edge e in E_i. For both subgraphs, all the nodes in V_i that are above e belong to the source set and those below belong to the destination set. We observe that $C_1 \cup C_2$ partitions the nodes of S' also into a source set and a destination set with the former being the union of the source sets in the two subgraphs and the latter the union of destination sets. Therefore, $C_1 \cup C_2$ is a cut in S'.

Without loss of generality, let the delay associated with all the cuts be 1. (A cut with d delays is equivalent to d identical cuts, each with 1 delay.) If G_1' and G_2' are derivable from G_1 and G_2, respectively, then for each edge $e \in E_i$ with $d(e)$ delays, there exist exactly $d(e)$ cuts containing e in each of the two subgraphs. Therefore all the cuts containing edges in E_i in the two subgraphs can be paired up to form cuts in S'. We have already shown that the cuts in the subgraph that do not contain any edges in E_i are also cuts in S'. Therefore if G_1' and G_2' are derivable from G_1 and G_2, respectively, then S' is also derivable from S.

The above result implies that we can cut up the grid S' into vertical strips and show that S' is equivalent to S by proving the equivalence of each of the strips. By applying the same argument on the horizontal links, we can further subdivide the strips into squares, each containing only four cells. The equivalence problem is now reduced to solving the equivalence for each of the squares. An edge from each of the two paths that connect the two diagonally opposite corners constitutes a cut. Therefore if the number of delays on each of the two paths of a square is the same, then the square is derivable from its counterpart in S. If this condition holds for each square, then S' is derivable from, and thus equivalent to, S. ■

The criterion for correctness as derived from this theorem is represented graphically in Figure 3.5b.

This theorem can be generalized to any array where we can find paths that partition the graph representing the array into disjoint subgraphs. For example, in the case of a hexagonal array without feedback cycles (Fig. 3.2a), the constraints for equivalence are simply reduced to the local criterion that for each unit triangle of three adjacent cells, the number of delays on each of the two paths connecting two of the corners of the triangle has to be the same.

3.5.2. *Redundancy Schemes*

The utilization of live cells for the rectangular systolic array of Fig. 3.5a depends on the availability of two hardware resources: delay registers in the live cells and the channel width. The results of Section 3.1 imply that if sufficient delay registers are available in the cells, the "systolic" approach can fully utilize all the live cells without any penalty to the throughput rate of the system. In general, a lower utilization can be expected with a smaller number of delay registers. The other factor that might decrease the utilization is the channel width. If there are not sufficient tracks in the channels, we might not be able to implement the interconnection desired.

We have conducted several experiments to study the trade-off between the utilization of live cells and the required hardware resources. We implemented four heuristic programs modeling different redundancy schemes. We ran Monte Carlo simulations on three different array sizes and cell failure rates ranging from 5 to 65%. The distribution of defects is assumed to be identical for all the cell locations on the wafer. The different schemes are described in the following and their examples are illustrated in Fig. 3.6.

1. No additional hardware. Because of the limitation in routing, we resort to a simple scheme where for each defective cell, we skip either the row or the column that contains the cell. The criterion of correctness is trivially satisfied. A greedy algorithm is used here; the row or column containing the most failures is eliminated first.

2. No delay register and unlimited channel width. In this scheme, all the cells in the final array are chosen so that the links only point in the forward

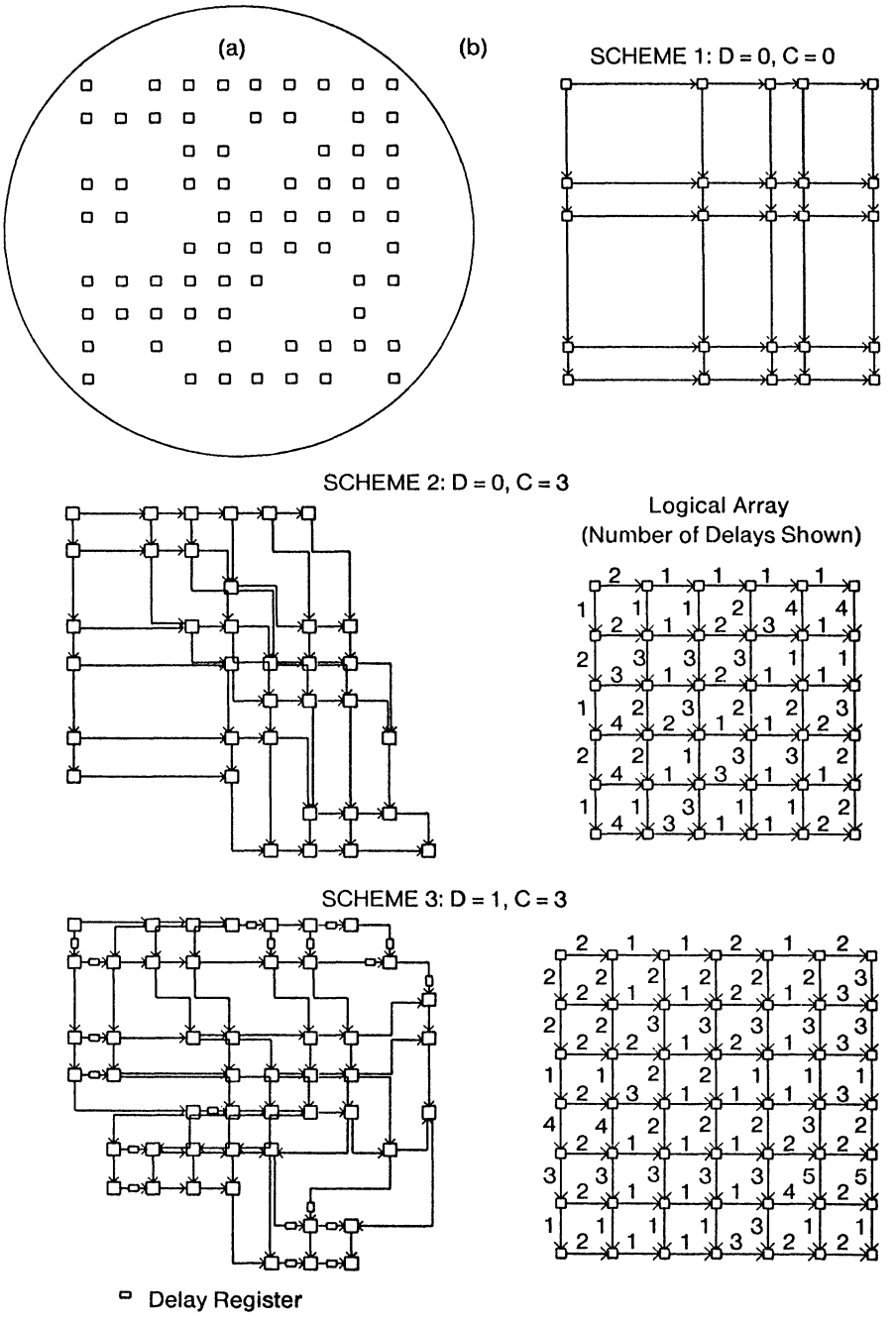


FIG. 3.6. (a) Live cells (72 out of 100 cells), and (b) array configurations under different redundancy schemes (D represents number of delay registers and C the redundant tracks in channel).

or downward direction. This guarantees that the number of delays on each link is equal to the manhattan distance between the two endpoints of the link, and thus the local correctness criterion is satisfied for each unit square of the array. Our basic strategy is to build the array row by row, picking as the next cell the one that satisfies the criterion and excludes the least number of live cells from being used. A simple maze runner is also implemented to determine the number of tracks required for interconnection.

3. One delay register per data path and unlimited channel width. The additional delay increases the flexibility in the assignment scheme, but it also complicates the algorithm of the program. We modified the program in scheme 2 such that if necessary, a delay register may be added to the new edges being created and to the old edges that are in the same row or column as the new ones, provided, of course, they do not have delay registers on them already.

4. Unlimited delay registers and unlimited channel width. How many delay registers are necessary to achieve 100% utilization? The scheme we chose requires delay registers be placed *only* on the *logically* vertical connections and none on the horizontal ones. The n live cells are partitioned into horizontal strips, each containing $\sqrt{n}$ cells. The cells in each strip are connected to form the rows and then connected to the corresponding cells in their neighboring rows. Delays are then assigned only to the logically vertical connections to satisfy the correctness criterion.

The empirical results are shown in Fig. 3.7. Each data point represents the average value over 100 trials. These results indicate that unless the cell yield is exceptionally high, redundancy is essential (see Fig. 3.7a). The channel width is generally not a bottleneck. While low yields and poor utilization increase the length of the path between two logically neighboring cells, they also open up more space for routing. For the range of array sizes and cell yields in our simulations, three redundant tracks are found to be sufficient for schemes 2 and 3, and five for scheme 4.

The expected utilizations with zero and one delay register are shown in Figs. 3.7b and c. The larger the array is, the more hardware delay registers are needed to get the same utilization. This is obvious since the set of constraints that has to be satisfied by a larger array is a superset of those satisfied by a smaller array. We have to bear in mind, however, that the cells in a larger array are typically smaller and thus have lower failure rates. From Fig. 3.8, we see that the maximum number of delay registers required on the logically vertical links to achieve 100% utilization is approximately equal to the number of cells on a side of a wafer. We note that for systolic arrays composed of programmable cells such as the CMU Programmable Systolic Chip (PSC) [23, 24], implementing programmable delay is straightforward and requires no extra circuitry.

These experiments give us a general idea of the expected efficiency of the different redundancy schemes using the systolic approach. In-depth studies

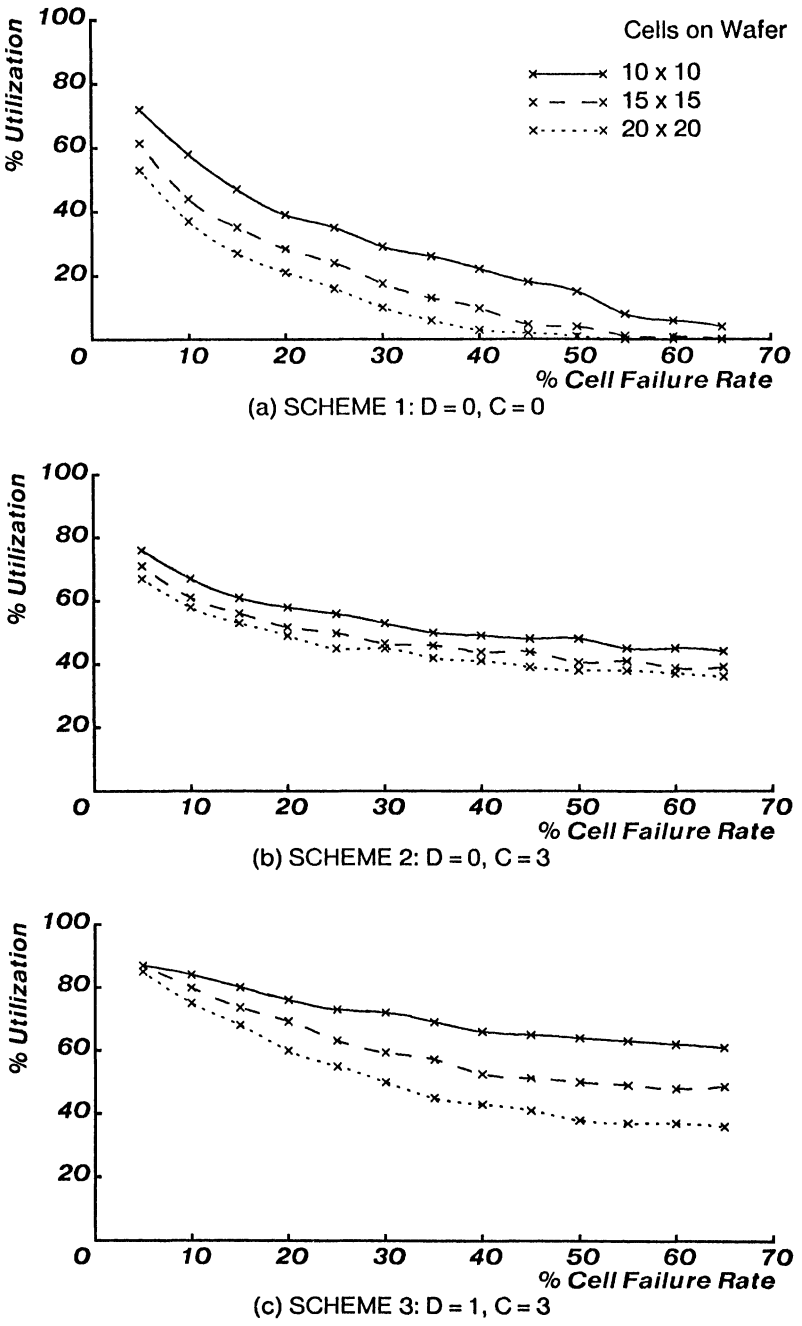


FIG. 3.7. Utilization under different redundancy schemes (D represents number of delay registers and C the redundant tracks in channel).

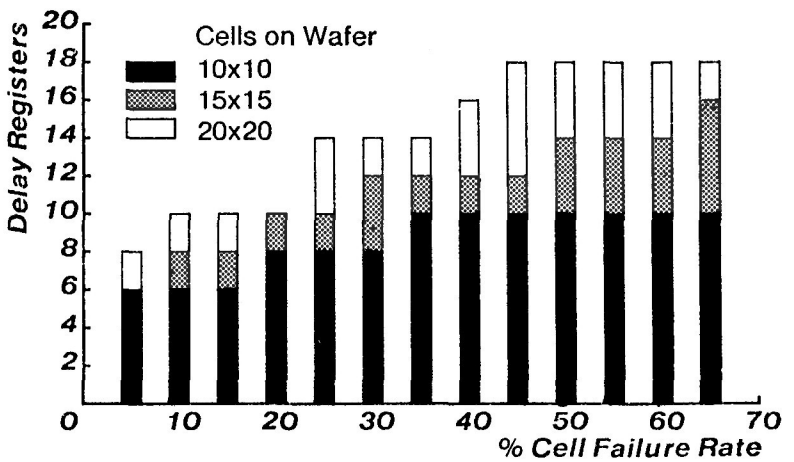


FIG. 3.8. Maximum number of delays required by 98% of the trials to achieve 100% utilization using scheme 4.

using a more precise model are necessary to determine the optimal or near-optimal redundancy scheme for any particular application. Probabilistic analyses [16, 17] have been performed for other fault-tolerant schemes where utilization is limited by the maximum length of interconnection allowed.

4. Systolic Arrays with Feedback Cycles

In this section we describe a new technique for treating systolic arrays with feedback cycles. Such arrays include systolic designs for LU-decomposition [25], QR-decomposition [26], triangular linear systems [25], and recursive filtering [27].

4.1. Computation of Simple Recurrences—An Example of Cyclic Systolic Arrays

To illustrate the basic ideas, we consider the computation of the following simple recurrence of size $n - 1$:

given: the initial values $\{y_{-n+2}, y_{-n+1}, \ldots, y_0\}$

compute: the output sequence $\{y_1, y_2, \ldots\}$ as defined by $y_i = \sum_{j=1}^{n-1} y_{i-j}$

Although summation is used here, the computation structure presented below generalizes to any associative operator. An n-cell systolic array with feedback cycles [27] is capable of performing this simple recurrence computation of

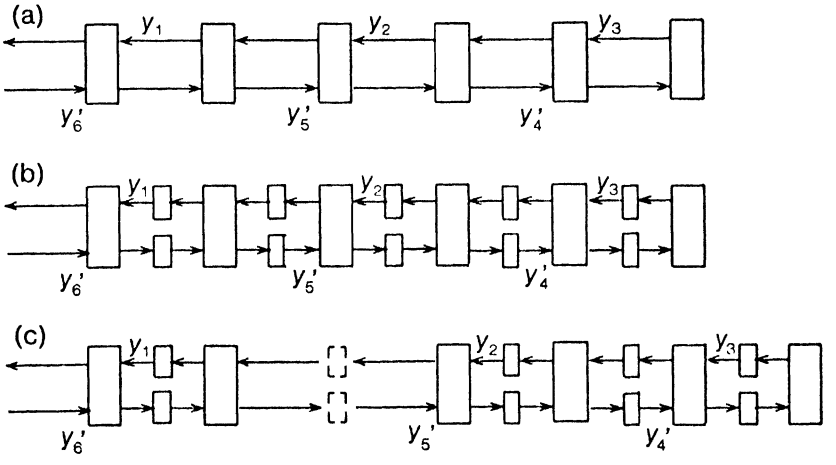


FIG. 4.1. Linear array with feedback: (a) original array, (b) reduced throughput, and (c) single failure.

size up to $n - 1$. Depicted in Fig. 4.1a is such an array where $n = 6$. The partial sums, y_4', y_5', y_6', move down the array from left to right picking up the completed sums that are moving in the opposite direction, y_1, y_2, y_3. The computation of each sum is completed when it reaches the end of the array. Note that this is a 2-slow [19] system, in the sense that only half its cells are active at all times.

A naive attempt at achieving fault tolerance involves slowing the system down even further. In the array of Fig. 4.1b data pass through an extra register per cell. This is a 4-slow system, performing the same computation as the 2-slow version, but at half its throughput. Suppose that the third cell from the left were to fail. The original function of the array could be preserved by simply allowing cells 2 and 4 to communicate through a bypass register (as illustrated in Fig. 4.1c). A drawback of this approach is that the performance of the array degrades rapidly with respect to the number of consecutive failed cells that need to be tolerated. Note that systolic arrays with feedback cycles are initially 2- or 3-slow in general, and in order to tolerate k consecutive failures, the throughput must be further decreased by a factor of $k + 1$.

The recurrence of size $n - 1$ computed by an n-cell bidirectional linear array (illustrated in Fig. 4.1a) can also be implemented on an $n/2$-cell ring with unidirectional data flow (as in Fig. 4.2a). The systolic ring works as follows. The $n/2$ most recently computed results are stored in each of the $n/2$ cells, while the next $n/2$ partial sums travel around the ring to meet these stored values. Every two cycles, a sum is completed and a new computation begins. For example, at time 0 in Fig. 4.2a, y_4' is ready to pick up its last term y_3 while y_6' is ready for its first term y_1. The final value of y_4 then travels to cell "a" to replace y_1. At time 2, y_5' and y_7' will pick up their last and first

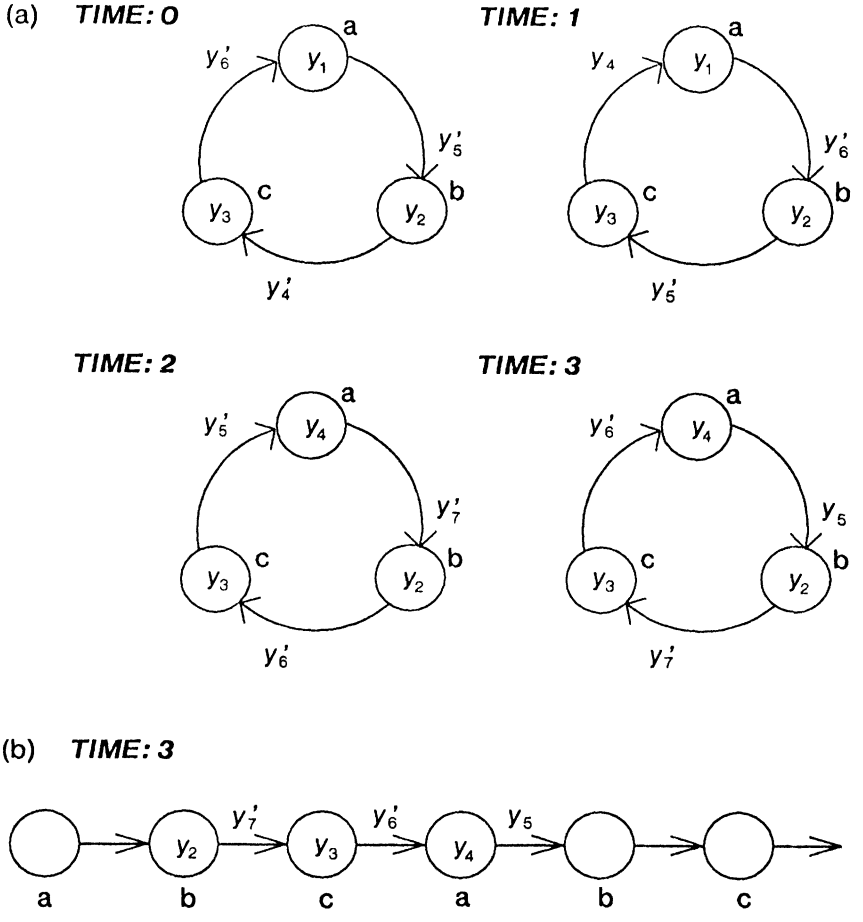


FIG. 4.2. (a) Four consecutive snapshots of a systolic ring, and (b) its unrolled structure.

terms, respectively. Like the bidirectional systolic array of Fig. 4.1a, this systolic ring has a computational rate of one output every two cycles. However, all its cells are active at any time; therefore only half as many cells are needed.

4.2. Fault-Tolerant Systolic Rings

Systolic rings require not only fewer cells than other designs solving the same problems; they also degrade gracefully as the number of defective cells increases.

Each cell in the systolic ring computes with a stored result for a period of $2n$ cycles before the result is replaced by a new value. The ring can be unrolled to form a linear array where each cell stores only one result in its

whole lifetime, as shown in Fig. 4.2b. This transformation reduces the ring structure to one without feedback, and thus allows us to analyze its fault-tolerant behavior using the results of the preceding section.

Figure 4.3a shows an example of a 4-cell systolic ring with one defect and Fig. 4.3b shows its unrolled version. A defect in the ring of m cells translates to a defect in every block of m cells in the linear array. Recall that in an array without feedback, the bypass registers corresponding to the defects will cause a delay in the action of the cells but not the functionality of the array. It is therefore the case that the defective ring computes the recurrence correctly. However, due to the delay through the defective cell, the $m - 1$ live cells produce results at a reduced rate of $m - 1$ outputs every $2m - 1$ $(= 2[m - 1] + 1)$ cycles.

Although a defective systolic ring solves problems at a slower rate than a flawless ring with the same number of live cells, it can solve larger problems. The additional delay through the defective cell means that the live cells have an extra clock cycle before they have to store a new result. This cycle can be effectively used to compute with one more recurrence term. Figure 4.3a shows the ring of $m - 1$ live cells solving a maximum size problem with $2m - 2$ $(= [2(m - 1) - 1] + 1)$ recurrence terms. The following theorem summarizes the result of this section.

THEOREM 3. *A perfect ring of size m can solve recurrences of sizes up to* $2m - 1$ *at a throughput rate of* $\frac{1}{2}$. *If k cells fail, it can solve problems of sizes up to* $2m - k - 1$ *at a throughput rate of* $(m - k)/(2m - k)$. *In other words, the reduction in throughput due to the k failures is only* $k/(2m - k)$ *of the original.*

4.3. Two-Level Pipelining for Systolic Rings

By going through an argument similar to that above for the two-level pipelined array, we can obtain the following result:

THEOREM 4. *A systolic ring of m p-stage pipelined cells can solve recurrences of sizes up to* $(p + 1)m - 1$ *at a throughput rate of* $1/(p + 1)$. *If k of the m cells fail, this ring can solve problems up to size* $(p + 1)m - pk - 1$ *at a throughput rate of* $(m - k)/[(p + 1)m - pk]$. *In other words, the reduction in throughput is only* $k/[(p + 1)m - pk]$ *of the original.*

4.4. Other Examples of Systolic Ring Architectures

We have shown in the previous section that the ring structure is suitable for solving simple recurrences where each result is dependent on a fixed number of previous results. This characterizes many of the problems solved by systolic arrays with feedback. We describe some of the examples in this section.

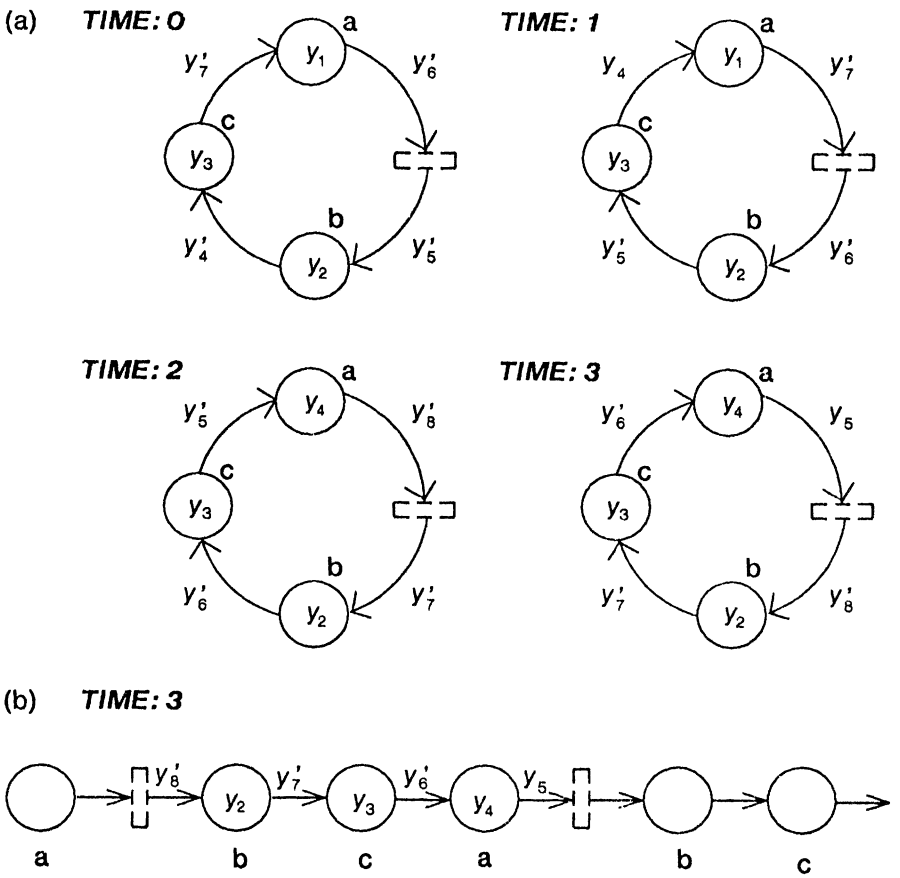


FIG. 4.3. (a) Four consecutive snapshots of a systolic ring with one failure, and (b) its unrolled structure.

4.4.1 *Solution of Triangular Linear Systems*

Let $A = (a_{ij})$ be a nonsingular $n \times n$ band, lower triangular matrix with bandwidth q. Suppose that A and an n-vector $b = (b_1, \ldots, b_n)^T$ are given. The problem is to solve $Ax = b$ for $x = (x_1, \ldots, x_n)^T$. This can be viewed as a recurrence problem of size $q - 1$. A ring of $q/2$ cells is sufficient to solve the problem at a throughput of one result every two cycles. As a comparison, the previous bidirectional linear systolic array [25] has the same throughput, but it uses twice as many cells. The ring is also more robust—with k failures in a ring of m cells, the throughput is only reduced from $\frac{1}{2}$ to $(m - k)/(2m - k)$.

Figures 4.4 and 4.5 illustrate the data flow pattern of a perfect 3-cell ring and a 4-cell ring with one failure, respectively, when solving a triangular

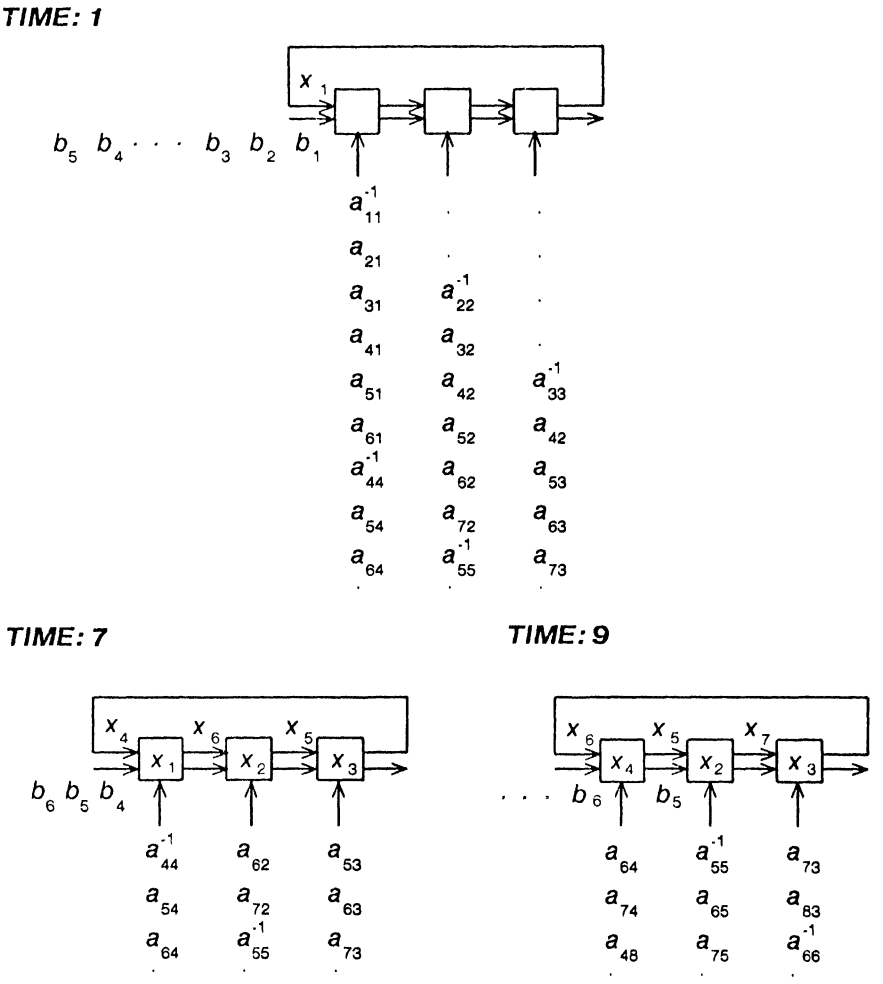


FIG. 4.4. Systolic ring for solving triangular linear systems.

linear system with bandwidth $q = 6$. While this problem size is the largest the former ring can handle in one pass, the latter can solve linear systems with bandwidth up to $q = 7$. As a result, the cells in the defective ring of Fig. 4.5 are idle one-seventh of the time. In the figure, a cell is assumed to be idle for one cycle if the input has a "don't care" value, denoted by $\times$.

The final step in the computation of each result (x_i) involves a subtraction (from b_i) and a division (by a_{ii}). This needs to be performed by every cell. To avoid having to provide each cell with a division capability and an external data path, we precompute the reciprocals of the diagonals outside the ring and send the additional input (b_i) to the cells via a systolic path.

The layout of a ring of processors is very straightforward, as shown in Fig.

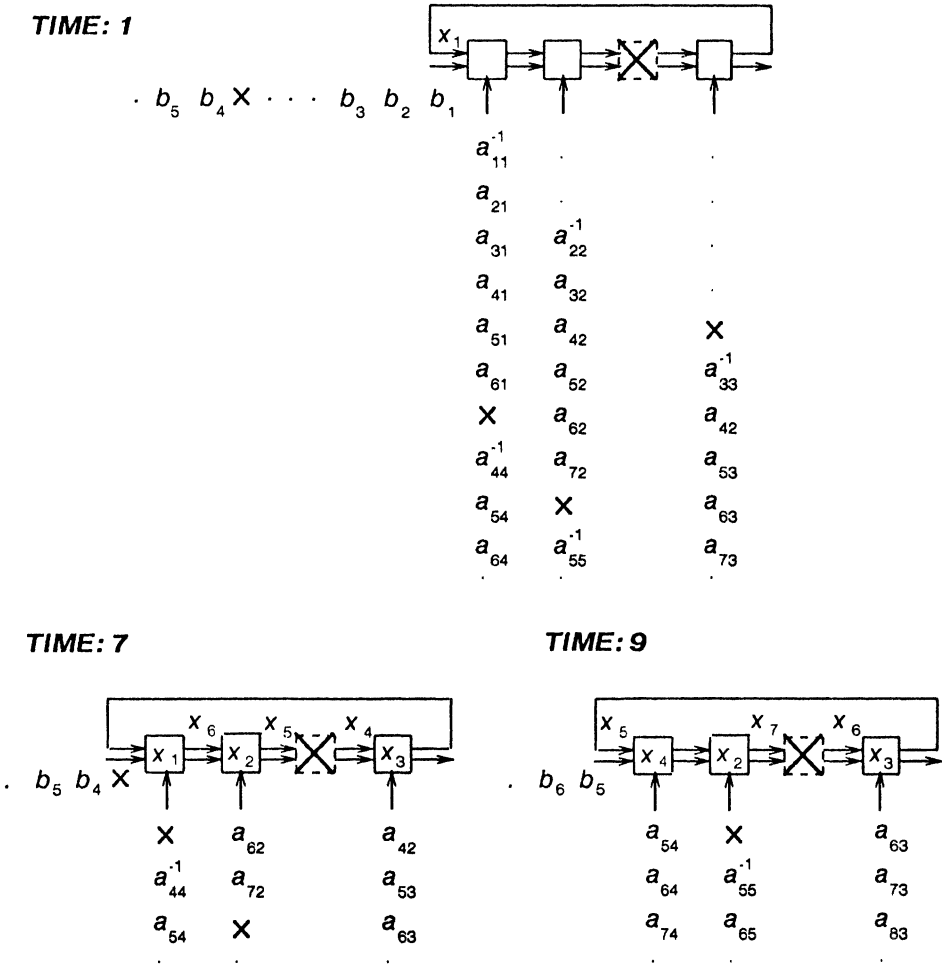


FIG. 4.5. A single failure in a systolic ring for solving triangular linear systems.

4.6. Similar to the unidirectional one-dimensional array, defects on a wafer can simply be bypassed via the cells' input/output registers.

4.4.2 *Triangularization of a Band Matrix*

The usefulness of the systolic ring approach is not limited to linear array solutions—Fig. 4.7a depicts a two-dimensional ring structure for triangularizing a band matrix A, with bandwidth $w = 6$ and $q = 3$ subdiagonals. This ring structure can perform the QR-decomposition, an important computation for linear least-squares approximation, and it can also solve linear systems using the stable computational technique of neighbor pivoting [28].

Each ring in the structure of Fig. 4.7 eliminates a subdiagonal, with the

t

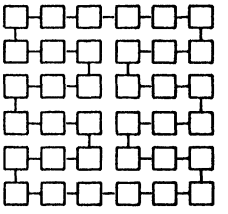


FIG. 4.6. Layout of a systolic ring.

bottommost ring handling the bottommost subdiagonal. The operations of a ring are illustrated by Fig. 4.7b. The parameters needed for performing the elimination (e.g., Givens rotations for QR-decomposition) pass around the ring after they are generated. Suppose p_i is the parameter generated by the element eliminated in row i and the element above it. If the data input a_{ij} is not on the subdiagonal to be eliminated, it is updated on the arrival of p_i. It stays in the cell for one cycle to compute with p_{i+1} and then moves on to the next ring. If a_{ij} is to be eliminated, it is computed with the stored value, a_{i-1j}, to get p_i, which is then passed down the ring. The output of each ring is the result obtained by eliminating the last subdiagonal of the input array. The uppermost ring outputs the entries of the triangular matrix that we want to compute. Note that corresponding to the elimination of each subdiagonal, a new superdiagonal is created. In the systolic ring, the new elements for this superdiagonal take the place previously occupied by the elements of the eliminated subdiagonal.

Unlike the data values circulating the rings in the previous examples, the p_i, are computed before they are passed around. However, they have the same property that they are produced every two cycles and need to meet with $w - 1$ input values before they can be discarded. Therefore, from our previous analysis, q rings of $w/2$ cells each are required for triangularizing a band matrix with bandwidth w and q subdiagonals. This architecture requires about half the amount of hardware and achieves the same throughput of a previous solution of QR-decomposition [26]. An efficient layout of this ring architecture is shown in Fig. 4.8a. Every two consecutive rows correspond to one ring.

The analysis of the fault-tolerant behavior of this ring structure is very similar to that of the one-dimensional ring. A system with n rings can be unrolled to form a mesh-connected acyclic array with n cells on one side and an "unbounded" number on the other. The throughput rate is reduced from $\frac{1}{2}$ to $n/(2n + k)$ if k defects are tolerated in each of the n rings in the final array. Also, by applying Theorem 2, we can simplify the correctness constraints on the final configuration to get a local criterion that has to be satisfied by each unit square in the logical grid. This criterion is depicted in Fig. 4.8b.

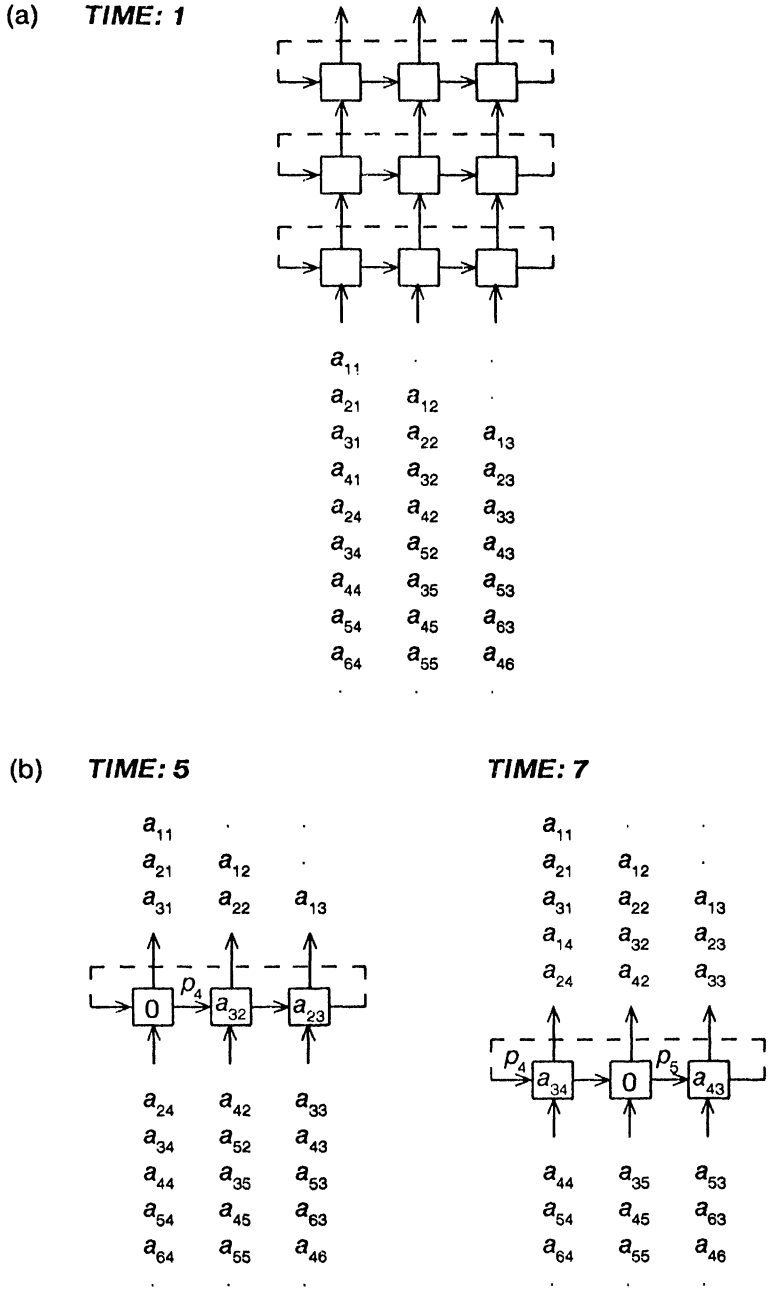


FIG. 4.7. (a) Two-dimensional systolic ring structure for matrix triangularization, and (b) two snapshots of the bottommost ring.

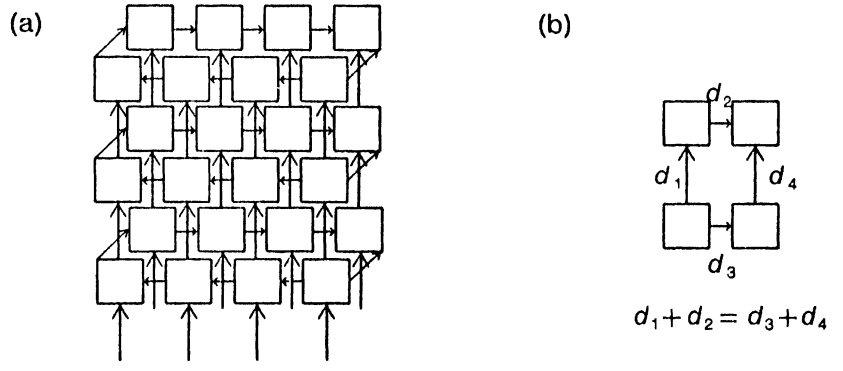


FIG. 4.8. (a) Layout of the two-dimensional ring structure for matrix triangularization, and (b) the local correctness criterion.

4.4.3. *LU-Decomposition of a Band Matrix*

Figure 4.9 depicts a two-dimensional systolic ring architecture for the LU-decomposition of a band matrix, $A = LU$. For a given matrix A with bandwidth $2q - 1$ we need to use $q/3$ rows of cells, with q cells in each row. The $q/3$ most recently computed rows of u_{ij}'s are stored in the cells as they are generated, while the l_{ij}'s are passed down the rows. Figure 4.10 shows the

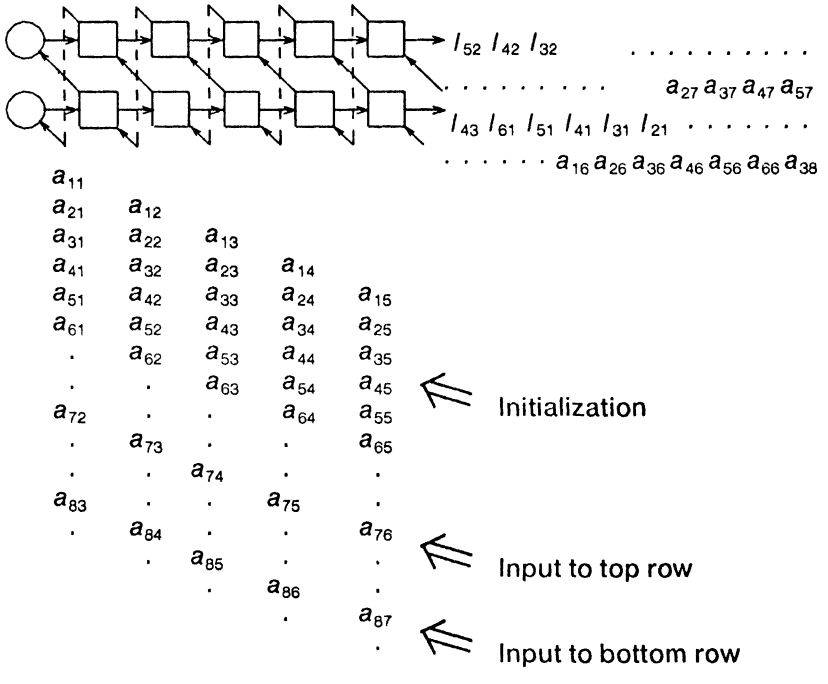


FIG. 4.9. Systolic ring architecture for LU-decomposition.

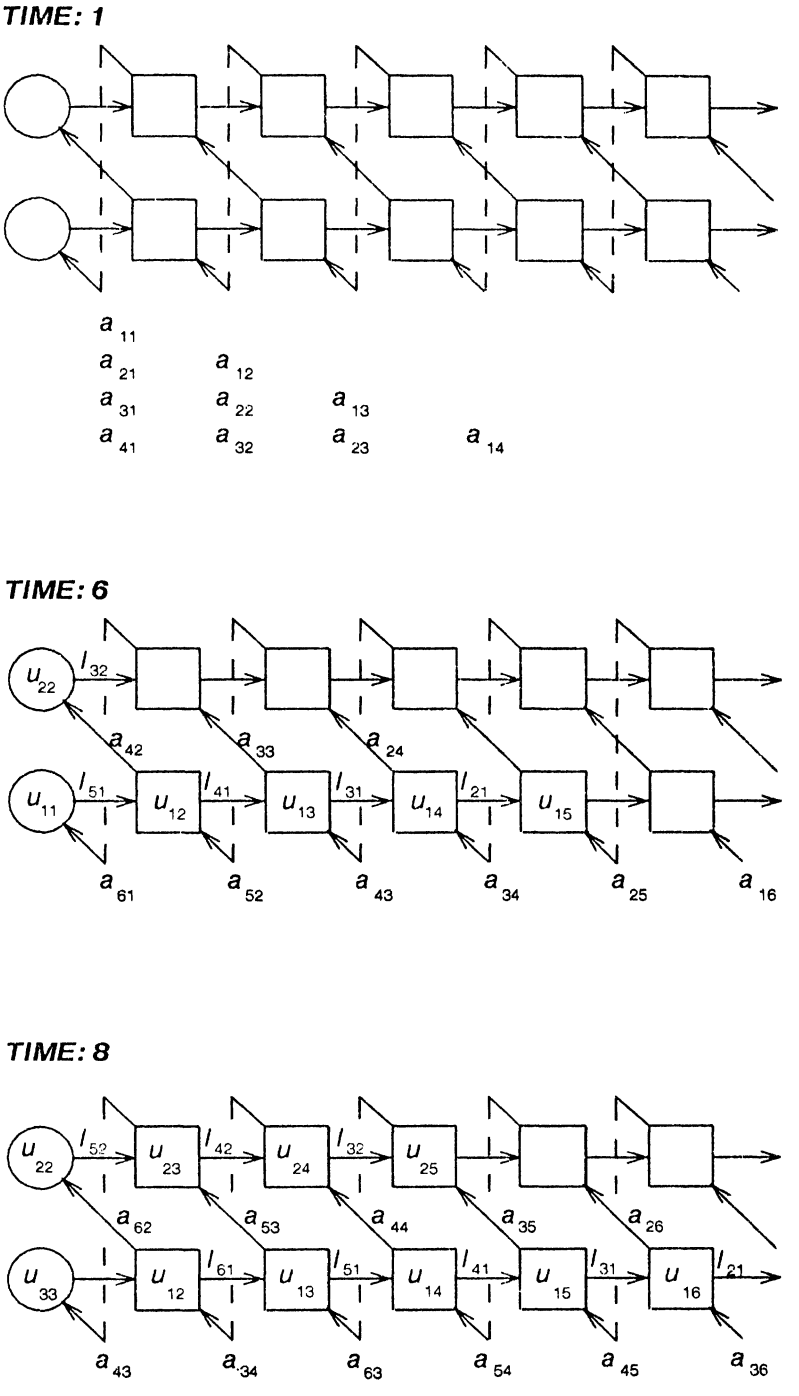


FIG. 4.10. Snapshots of a ring architecture for LU-decomposition.

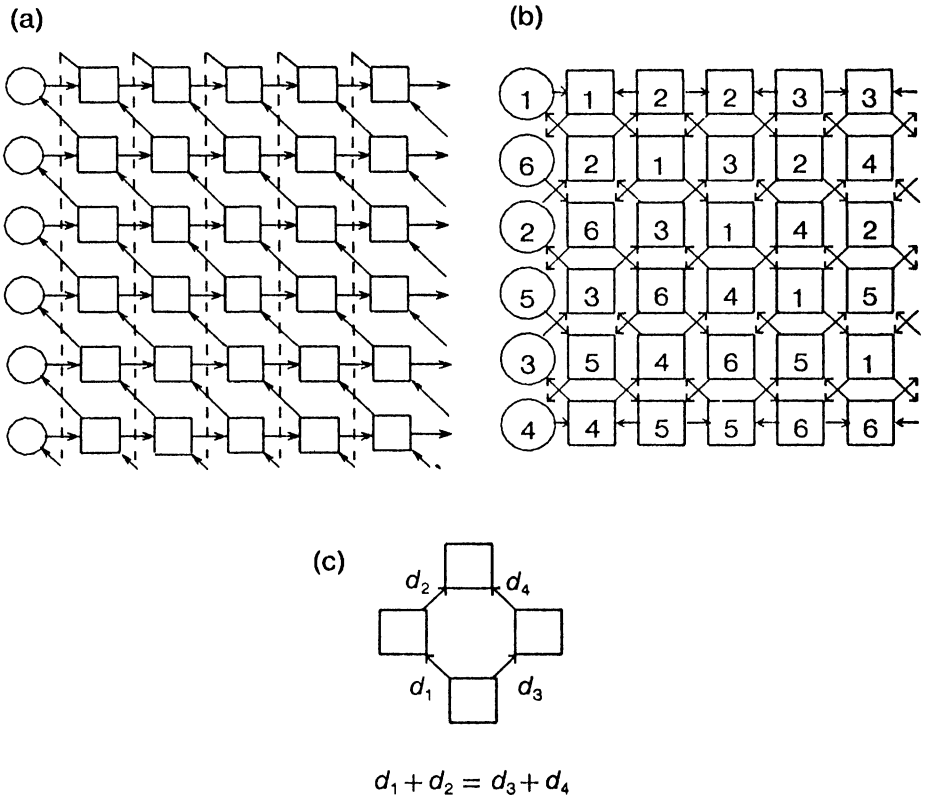


FIG. 4.11. (a) Two-dimensional ring architecture for LU-decomposition, (b) its layout, and (c) the local correctness criterion.

snapshots of this structure at various stages in the computation. By viewing this structure as an array of rings, its performance can be analyzed using the result of Theorem 4 with parameter $p = 2$. The throughput of this array is the same as that of the previous design [25], which, however, uses three times as many cells.

This two-dimensonal ring architecture admits of a surprisingly efficient layout. See Figure 4.11b. The numbers on the cells indicate the original row the cells are in. This layout can be obtained by the following method. Starting with the original architecture (Fig. 4.11a), we first bring the top and bottom rows together and get a cylindrical structure. We then expand the space between rows by one cell length, so that if we flatten out the cylinder, the consecutive rows in the "front" and "back" surfaces will be interleaved. But before we flatten out the cylinder, we first "twist" it by one cell length per row in the direction that results in shorter inter-row links.

By going through the analysis of the unrolled structure, we get the follow-

ing results. If k faults are bypassed in the communication links in each of the column of connections in Fig. 4.11b, then the throughput is reduced from $\frac{1}{3}$ to $n/(3n + k)$, where n is the number of rows in the final array. Also, the local criterion that has to be satisfied by each group of four cells is illustrated in Fig. 4.11c.

4.5. General Remarks on Systolic Rings

The systolic ring architecture has some disadvantages over other systolic architectures, but they are compensated for by its superior fault-tolerance performance. One of the possible disadvantages is that we need to provide an additional data path to unload the values during the computation, as the computed results are continuously stored in the ring. This, however, is not the case for the triangularization schemes of Section 4.4.2.

In many of the conventional cyclic algorithms, only one or a few boundary cells may require special processing capability and extra input/output bandwidth. However, with some ring architectures, every cell is required to assume the role of a boundary cell. Algorithm-dependent methods can sometimes be used to alleviate the problem of having to provide each cell with special functionality. For instance, in the previous example of solving triangular linear systems, instead of providing each cell with the capability to divide, we precompute the reciprocals of the diagonals.

5. Summary and Concluding Remarks

The fault-tolerant approach proposed in this paper is tailored to systolic arrays. By using the additional information about systolic data flows we are able to design schemes that are usually more effective than other schemes designed for general processor arrays. Our systolic fault-tolerant scheme has the characteristic that the maximum interconnection length is not increased. This eliminates a source of inefficiency, such as increased system cycle time or driver area, common to most other approaches.

For unidirectional linear arrays, our systolic fault-tolerant technique achieves 100% utilization of live cells, without extra registers or interconnection links. For two-dimensional arrays without feedback cycles, the utilization of live cells on a wafer increases with the number of redundant channels and delay registers available in the cells. The number of delay registers needed to achieve the same utilization also increases with the cell failure rate and the size of the original array on the wafer. Our empirical studies indicate that for a wafer with $n \times n$ cells, approximately n delay registers per cell are needed to achieve 100% utilization.

Although many systolic algorithms with feedback have been proposed, some of the same problems which these algorithms address can also be solved

by systolic arrays without feedback. Examples of such problems include convolution, graph connectivity and graph transitive closure [9, 29, 30]. Acyclic implementations usually exhibit more favorable characteristics with respect to fault tolerance, two-level pipelining, and problem decomposition in general.

For problems that have been solved exclusively by systolic arrays with feedback cycles, this paper introduces a new class of systolic algorithms based on a ring architecture. These systolic rings have the property that the throughput degrades gracefully as the number of failed cells in the rings increases. Furthermore, as a byproduct of the ring architecture approach, we have derived several new systolic algorithms which require only one-third to one-half of the cells used in previous designs while achieving the same throughput.

We have shown that the two-level pipelining problem in systolic arrays can be solved by the same techniques used to solve the fault-tolerance problem. An important task left for the future is the development of software to solve both problems automatically.

REFERENCES

1. Evans, R. A., *et al.* A CMOS implementation of a systolic multi-bit convolver chip. In Anceau, F., and Aas, E. J. (Eds.). *VLSI '83*. North-Holland, Amsterdam, Aug. 1983, pp. 227–235.
2. Kung, H. T. On the implementation and use of systolic array processors. *Proc. International Conference on Computer Design: VLSI in Computers,* IEEE, Nov. 1983, pp. 370–373.
3. Symanski, J. J. NOSC systolic processor testbed. Tech. Rep. NOSC TD 588, Naval Ocean Systems Center, June 1983.
4. Yen, D. W. I., and Kulkarni, A. V. Systolic processing and an implementation for signal and image processing. *IEEE Trans. Comput.* **C-31,** 10 (Oct. 1982), 1000–1009.
5. Smith, R. T., *et al.* Laser programmable redundancy and yield improvement in a 64K DRAM. *IEEE J. Solid-State Circuits* **SC-16,** 5 (Oct. 1981).
6. Lincoln Laboratory. Semiannual technical summary: Restructurable VLSI program. Tech. Rep. ESD-TR-81-153, MIT Lincoln Lab, Mar. 1981.
7. Woo, B., *et al.* ALU, multiplier chips zip through IEEE floating-point operations. *Electronics* **56,** 10 (May 19, 1983), 121–126.
8. Kung, H. T., Ruane, L. M., and Yen, D. W. L. Two-level pipelined systolic array for multidimensional convolution. *Image and Vision Computing* **1,** 1 (Feb. 1983), 30–36. An improved version appears as a CMU Computer Science Department technical report, Nov. 1982.
9. Kung, H. T. Why systolic architectures? *Comput. Mag.* **15,** 1 (Jan. 1982), 37–46.
10. Aubusson, R. C., and Catt, I. Wafer scale integration—A fault tolerant procedure. *IEEE J. Solid-State Circuits* **SC-13,** 3 (June 1978), 339–344.

11. Fussel, D., and Varman, P. Fault-tolerant wafer-scale architectures for VLSI. *Proc. 9th Annual Symposium on Computer Architecture,* April 1982, pp. 190–198.

12. Koren, I. A reconfigurable and fault-tolerant VLSI multiprocessor array. *Proc. 8th Annual Symposium on Computer Architecture.* May 1981, pp. 425–442.

13. Manning, F. B. An approach to highly integrated, computer-maintained cellular arrays. *IEEE Trans. Comput.* **C-26,** 6 (June 1977), 536–552.

14. Rosenberg, A. L. On designing fault-tolerant arrays of processors. Tech. Rep. CS-1982-14, Duke University, 1982.

15. Rosenberg, A. L. The Diogenes approach to testable fault-tolerant networks of processors. Tech. Rep. CS-1982-6, Duke University, 1982.

16. Leighton, F. T., and Leiserson, C. E. Wafer-scale integration of systolic arrays. *Proc. 23rd Annual Symposium on Foundations of Computer Science,* IEEE, Oct. 1982, pp. 279–311.

17. Greene, J. W., and El Gamal, A. Configuration of VLSI arrays in the presence of defects. Tech. Rep., Information Systems Lab, Stanford University, May 1983.

18. Leiserson, C. E., and Saxe, J. B. Optimizing synchronous systems. *J. VLSI and Computer Systems* **1,** 1 (1983), 41–68.

19. Leiserson, C. E. *Area-Efficient VLSI Computation.* Ph.D. dissertation, Carnegie–Mellon University, 1981. (Published by MIT Press, Cambridge, Mass., 1983.)

20. Weiser, U., and Davis, A. A wavefront notation tool for VLSI array design. In Kung, H. T., Sproull, R. F., and Steele, G. L., Jr. (Eds.). *VLSI Systems and Computations,* Computer Science Department, Carnegie–Mellon University, Oct. 1981. Computer Science Press, Rockville, Md., pp. 226–234.

21. Rabiner, L. R., and Gold, B. *Theory and Application of Digital Signal Processing.* Prentice–Hall, Englewood Cliffs, N.J., 1975.

22. Stone, H. S. Parallel processing with the perfect shuffle, *IEEE Trans. Comput.* **C-20** (Feb. 1971), 153–161.

23. Fisher, A. L., Kung, H. T., Monier, L. M., and Dohi, Y. Architecture of the PSC: A programmable systolic chip. *Proc. 10th Annual Symposium on Computer Architecture,* June 1983, pp. 48–53.

24. Fisher, A. L., Kung, H. T., Monier, L. M., Walker, H., and Dohi, Y. Design of the PSC: A programmable systolic chip. In Bryant, R. (Ed.). *Proc. Third Caltech Conference on Very Large Scale Integration,* California Institute of Technology, Mar. 1983. Computer Science Press, Rockville, Md., pp. 287–302.

25. Kung, H. T., and Leiserson, C. E. Systolic arrays (for VLSI). In Duff, I. S., and Stewart, G. W. (Eds.). *Sparse Matrix Proceedings 1978.* Society for Industrial and Applied Mathematics, 1979, pp. 256–282. A slightly different version appears in *Introduction to VLSI Systems,* by C. A. Mead and L. A. Conway. Addison–Wesley, Reading, Mass., 1980, Sect. 8.3, pp. 37–46.

26. Heller, D. E., and Ipsen, I. C. F. Systolic networks for orthogonal equivalence transformations and their applications. *Proc. Conference on Advanced Research in VLSI,* Massachusetts Institute of Technology, Cambridge, Mass. Jan. 1982, pp. 113–122.

27. Kung, H. T. Let's design algorithms for VLSI Systems. *Proc. Conference on Very Large Scale Integration: Architecture, Design, Fabrication,* California Institute of Technology, Jan. 1979, pp. 65–90. Also available as a CMU Computer Science Department technical report, Sept. 1979.

28. Gentleman, W. M., and Kung, H. T. Matrix triangularization by systolic arrays. *Proc. SPIE Symposium,* Vol. 298, *Real-Time Signal Processing IV.* Society of Photo-Optical Instrumentation Engineers, Aug. 1981, pp. 19–26.

29. Guibas, L. J., Kung, H. T., and Thompson, C. D. Direct VLSI implementation of combinatorial algorithms. *Proc. Conference on Very Large Scale Integration: Architecture, Design, Fabrication,* California Institute of Technology, Jan. 1979, pp. 509–525.

30. Tchuente, M., and Melkemi, L. Systolic arrays for connectivity problems and triangularization for band matrices. Tech Rep. R.R. No. 366, IMAG, Institut National Polytechnique de Grenoble, March 1983.

On the Design of Algorithms for VLSI Systolic Arrays

DAN I. MOLDOVAN, MEMBER, IEEE

Abstract—This paper is concerned with the mapping of cyclic loop algorithms into special-purpose VLSI arrays. The mapping procedure is based on the mathematical transformations of index sets and data dependence vectors. Necessary and sufficient conditions for the existence of valid transformations are given for algorithms with constant data dependences. Two examples of different algorithms are given to illustrate the proposed mapping procedure; first is the LU decomposition of a matrix which leads to constant data dependence vectors, and secondly is the dynamic programming which leads to dependences which are functions on the index set and are more difficult to be mapped into VLSI arrays.

I. INTRODUCTION

THE TECHNOLOGY available to produce central processor units (CPU) and computer memories has always influenced the architecture of computers. Improvements in technological processes resulted in higher computer performances. The present semiconductor technology has reached already the level of maturity beyond which no significant breakthroughs are expected for switching speeds. The level of integration, however, continues to grow and in the next ten years it will be possible to incorporate one million logic gates on a die of a chip. This very large scale integration (VLSI) is a new technological environment which requires new ideas in computer organization, theory of computing, and other related fields.

In this paper we are concerned with the development of algorithms for special-purpose VLSI arrays. As will be seen below, while the VLSI technology offers remarkable advantages to the system designer, it also imposes restrictions on the design of algorithms. The most important of these restrictions is the necessity for reduced communication complexity.

The paper is organized as follows: in Section I we discuss the implications of the VLSI technology on computer architectures and algorithm design. Sections II and III contain the main results of the paper; first, a technique is proposed to transform algorithms with loops into highly parallel forms suitable for VLSI devices; then, a procedure is proposed to map these transformed algorithms into VLSI systolic arrays. In Section IV, algorithms for the LU decomposition of a matrix and dynamic programming are used as examples to show how previously proposed architectures can be formally derived using appropriate algorithm transformations.

A. VLSI Algorithms and Architectures

The main advantages offered by the VLSI technology are: large amount of hardware available at very low cost, reduced power consumption and physical size, and increased reliability at the circuit level. Additionally, the high level of integration can conceivably eliminate the need to physically separate processors from memory, thus eliminating the bottleneck between them. Parallelism and pipelining are two classical concepts without which the efficient utilization of the large hardware resources offered by VLSI is not possible. Parallelism implies the operation of many units at the same time. Pipelining also requires a multitude of resources, but in contrast with parallelism, the resources work in a chain allowing data to flow only from one unit to the next one. Both, parallelism and pipelining, can be seen at different logic levels. The first level of parallelism is offered by partitioning a computational task into smaller computational modules. The second level of parallelism is found within each computational module. The last level of parallelism is offered by the simultaneous processing of all the bits in a word; and this level is present in almost all computers. The focus of this paper is the parallelism at the second level.

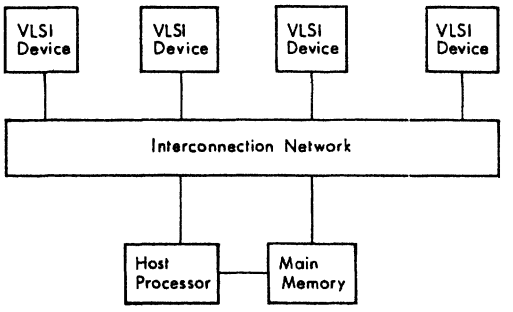


Fig. 1. Organization of a computer system containing several special-purpose VLSI processor arrays, interconnection network, host processor, and main memory.

The exploitation of parallelism at the first level is often necessary because computational problems are larger than a single VLSI device can process at a time. If a parallel algorithm is structured as a network of smaller computational modules, then these modules can be assigned to different VLSI devices. The communications between these modules and their operation control dictates the structure of the VLSI system and its performances. In Fig. 1, a simplistic organization of a computer system consisting of several VLSI devices, main memory, and an interconnection network are shown. Each VLSI device has a number of processors working in parallel.

The I/O bottleneck problem in VLSI systems presents a serious restriction imposed on the algorithm design. The challenge is to design parallel algorithms which can be partitioned such that the amount of communication between modules is as small as possible. Moreover, data entering the VLSI

Manuscript received April 22, 1982; revised October 20, 1982. This research was partially supported by the National Science Foundation under Grant ECS 8119509.

The author is with the Electrical Engineering Department—Systems, University of Southern California, Los Angeles, CA 90089.

Reprinted from *Proceedings of the IEEE*, pp. 113-120, January 1983.

device should be utilized exhaustively before passing again through the I/O ports.

Another potential problem which can deteriorate the performance is the data communication within the VLSI device. The interconnections between logic gates are as expensive as the logic itself and the signal propagation is comparable with the logic switching time. An efficient utilization of silicon area, time, and energy is achieved only if the hardware contains local interconnections. The solution to this problem is to design algorithms which, when mapped into VLSI hardware, require only local data transfers. In the next two sections it is shown that some algorithms can be transformed to meet these requirements.

Systolic array architectures have been proposed by Kung [1], [2] and others as a possible solution to these VLSI problems. In the systolic concept, VLSI devices consist of arrays of interconnected processing cells with a high degree of modularity. Each processor operates on a string of data that flow regularly through the network. If the I/O problem is ignored, the throughput of such computational structure is proportional to the number of cells.

In order to match better the characteristics of algorithms with the characteristics of computer architectures, and consequently to increase the efficiency of computation, a careful mapping of the computational problem to the machine is necessary. The mapping of algorithms into systolic arrays is different than the mapping of algorithms into architectures with fixed number of processors and interconnections. In the case of systolic arrays, one has to deal with issues ranging from the organization of the network of cells to the detailed operation of the cells. In fact, the mapping is nothing but the design of the VLSI array according to the properties of the algorithm and a set of design goals.

A number of special-purpose VLSI architectures have been proposed in the last few years. Kung's early work in parallel algorithms for VLSI has stimulated a considerable interest. He proposed systolic arrays for matrix–vector and matrix–matrix multiplications, LU decompositions, recurrence evaluations, etc., [1], [2]. The VLSI implementation of some combinational algorithms has been investigated by Guibas *et al.* [3]. Algorithms for solving systems of equations have been proposed by Kung [4], Hwang and Cheng [5], and Preparata [6]. The special-purpose VLSI computing structures have found immediate application in signal processing where many algorithms have regular structures [7], [8].

B. A VLSI Model of Computation

A model of the VLSI computing structure is needed in order to relate the features of an algorithm to the realities of the hardware. Tradeoffs are possible between various parameters of the VLSI device in order to improve one performance or another. The approach taken here is to distinguish between the operation of the systolic system at the array level and the activities taking place inside the processing cells. The array level is called the global level, and the processor level is called the local level. At both levels, the operation should be examined in time and space. Fig. 2 shows the main steps involved in the design of a special-purpose VLSI chip.

In this paper, we will focus only on the step from the parallel algorithm to the global model. A model of the processing cell and the transition from the global model to the local model can be found in [9]. The organization and the operation of the VLSI array can be described by the network geometry G, the functions F performed by the processing cells, and the network timing T.

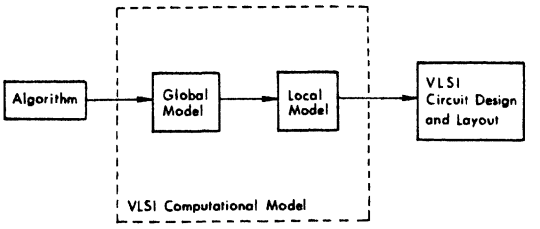


Fig. 2. Main steps in the design of a special-purpose VLSI device.

The assumptions about the VLSI systolic network are as follows:

a) The network consists of a planar mesh connected network of processing cells.

b) The cells can be of different types and perform different functions.

c) The interconnections between cells are buses which transfer parallel words.

d) The operation of the network is synchronous.

The *network geometry* G refers to the geometrical layout of the network. The position of each processing cell in the plane is described by its Cartesian coordinates. By choosing the grid arbitrarily small it is possible to represent these coordinates by integers. Then, the interconnections between cells can easily be described by the position of the terminal cells. These interconnections support the flow of data through the network; a link can be dedicated only to one data stream of variables or it can be used for the transport of several data streams at different time instances. A simple and regular geometry is desired.

The *functions* F associated to each processing cell represent the totality of arithmetic and logic expressions that a cell is capable to perform. We assume that each cell consists of a small number of registers, ALU, and control logic. Several different types of processing cells may coexist in the same network; however, one design goal should be to reduce the number of cell types.

The *network timing* T specifies for each cell the time when the processing of functions F occurs and when the data communications take place. A correct timing assures that the right data reach their destination at the right time. The speed of the data streams through the network is given by the ratio between the distance of communication link over the communication time. Networks with constant data speeds are preferable because they require a simpler control logic.

In summary, the global model of the VLSI array can be formally described by a set of 3-tuples (G, F, T). The more regular the network is the simpler these functions become. This model is quite general and it is sufficient for the purpose of this paper of developing a methodology for designing VLSI algorithms.

In the next section, a technique is developed to study and to modify computational algorithms for the purpose of mapping them into VLSI processing arrays. While any algorithm can be analyzed using this technique, only some algorithms can be mapped directly into simple systolic arrays.

II. Transformations of Algorithms for VLSI Systems

A. Data Dependences

The intention of mapping computational algorithms into VLSI circuits implies first a transformation of algorithms into equivalent but more appropriate forms for VLSI. The basic structural features of an algorithm are dictated by the data and control dependences. These dependences refer to precedence

relations of computations which need to be satisfied in order to compute the problem correctly. The absence of dependences indicates the possibility of simultaneous computations. These dependences can be studied at several distinct levels: blocks of computations level, statement (or expression) level, variable level, and even bit level. In this paper, since we concentrate on algorithms for VLSI systolic arrays, we will focus only on data dependences at the variable level which is the lowest possible level before the bit level.

The analysis of data dependences in high-level language (HLL) programs for the purpose of detecting concurrency of operations has received considerable attention in the last decade. Muraoka [10] and Kuck *et al.* [11] have studied the parallelism of simple loops and have introduced the notion of dependence relations between assignment statements. Towle [12], Banergee [13], and Banerjee *et al.* [14] extended the methodology of transforming ordinary programs into highly parallel forms. Based upon dependences between statements, they have provided algorithms for exploiting parallelism in loops. Techniques such as loop freezing, wave from method, the splitting-lemma, and loop interchanging have been introduced. Recently, Kuhn [15] has proposed a methodology to analyze data dependences using transformations on convex index sets. Results on program transformations were also reported by Lamport [16].

All these results were aimed mostly towards program speed-up and compiler design. Although the program transformations proposed before contain many basic results, they are not adequate enough for VLSI implementations. In addition to a high degree of parallelism, VLSI arrays suggest pipelining and reduced communication distance and time.

Previous work in algorithm transformations has focused on deciding whether a pair of occurrences is dependent or not. The present approach is based not only on the detection of dependences but also on their modification. The very structure of algorithm interconnections has to be modified in order to increase the "locality" of communications and to meet other VLSI requirements.

In what follows, algorithms written in HLL are considered. Other forms to express algorithms are possible, but the results would be similar. Consider a Fortran loop structure of the form

$$\begin{aligned}
&\text{DO 10 } I^1 = l^1, u^1\\
&\text{DO 10 } I^2 = l^2, u^2\\
&\cdots\\
&\text{DO 10 } I^n = l^n, u^n\\
&S_1(\bar{I})\\
&S_2(\bar{I})\\
&\cdots\\
&S_N(\bar{I})\\
&\text{10 CONTINUE}
\end{aligned} \tag{1}$$

where l^j and u^j are integers value linear expressions involving $I^1, \cdots, I^{j-1}$ and $\bar{I} = (I^1, I^2, \cdots, I^n)$. $S_1, S_2, \cdots, S_N$ are assignment statements of the form $X = E$ where X is a variable and E is an expression of some input variables.

Let $\mathcal{F}$ denote the set of all integers and $\mathcal{F}^n$ denote the set of n-tuples of integers. The index set of loop (1) is a subset of $\mathcal{F}^n$ and is defined as

$$\mathcal{L}^n(\bar{I}) = \{(I^1, \cdots, I^n)\colon\ l^1 \leqslant I^1 \leqslant u^1, \cdots, l^n \leqslant I^n \leqslant u^n\}.$$

When loop (1) is executed, the elements of $\mathcal{L}^n$ are ordered in a lexicographical ordering. This is an induced ordering which is not essential and can be modified. Let f and g be two integer functions defined on the set $\mathcal{L}^n$. Denote X and Y two variables whose indices are f and g; we write $X(f(I))$ and $Y(g(\bar{I}))$. Variables X and Y are generated in statements $S(\bar{I}_1)$ and $S(\bar{I}_2)$, respectively. Variable $Y(g(\bar{I}))$ is said to be dependent on variable $X(f(\bar{I}))$ and write $X(f(\bar{I})) \to Y(g(\bar{I}))$ if

a) $I_1 < \bar{I}_2$ (throughout the paper "$<$" means "less than" in lexicographical sense)

b) $f(\bar{I}_1) = g(\bar{I}_2)$

c) $X(f(\bar{I}))$ is an input variable in statement $S(\bar{I}_2)$.

The vector $\bar{d} = \bar{I}_2 - \bar{I}_1$ is called the data dependence vector. An algorithm has a number of such dependence vectors. In general, the dependence vectors are functions of the elements of set $\mathcal{L}^n$, i.e., $\bar{d} = \bar{d}(\bar{I})$, as will be seen in Section III-B. There is, however, a large class of algorithms with fixed, or constant data dependence vectors.

B. Transformation of Index Set and Data Dependences

Denote the ordering imposed by the data dependences on set $\mathcal{L}^n$ with R. The elements of $\mathcal{L}^n$ and ordering R form together a well-defined algebraic structure $\langle \mathcal{L}^n, R\rangle$. We seek now a transformation T such that

$$T\colon \langle \mathcal{L}^n, R\rangle \to \langle \mathcal{L}^n_T, R_T\rangle \tag{2}$$

with the following properties:

a) T is a bijection and a monotonic function (2a)

b) The data dependences of the new structure $\langle \mathcal{L}^n_T, R_T\rangle$ can be selected by us. (2b)

Since T is a bijection, the two structures are said to be isomorphic, and since T is monotonic with respect to R and R_T

$$\bar{d} > 0 \to \bar{\delta} = T(\bar{d}) > 0.$$

It simply means that the transformation T preserves the sense of the data dependences. The meaning of the second condition will soon become clear. The transformation T is partitioned in two functions as follows:

$$T = \begin{bmatrix} \Pi \\ S \end{bmatrix}. \tag{3}$$

The mapping Π is defined as

$$\Pi\colon\ \mathcal{L}^n \to \mathcal{L}^k_T, \qquad n > k$$

$$\Pi(I^1, I^2, \cdots, I^n) = (J^1, J^2, \cdots, J^k), \qquad \text{with } \bar{J} \in \mathcal{L}^n_T.$$

The mapping S is defined as

$$S\colon\ \mathcal{L}^n \to \mathcal{L}^{n-k}_T$$

$$S(I^1, I^2, \cdots, I^n) = (J^{k+1}, J^{k+2}, \cdots, J^n).$$

The dimensionality of functions Π and S is marked by k; and k is selected such that Π alone establishes the ordering R_T. The first k coordinates of elements $\bar{J} \in \mathcal{L}^n_T$ can now be related to time and the last $n - k$ coordinates be related to the geometrical properties of the algorithm. In other words, the time is associated to the new lexicographical ordering imposed on the elements $\bar{J}$ and this is given only by their first k coordinates. The last $n - k$ coordinates can be chosen by us to satisfy our expectations about the geometrical properties of the algorithm. For all elements $\bar{I} \in \mathcal{L}^n$ for which $\Pi(\bar{I}) =$ constant, the first k coordinates of $\bar{J}(\bar{I})$ are also constant. It follows that all such $\bar{I} \in \mathcal{L}^n$ can be processed concurrently. $\Pi(\bar{I}) =$ constant represents hypersurfaces with property to contain elements which are not data dependent.

The freedom of selecting $(J^{k+1}, \cdots, J^n)$ can be used advantageously to satisfy property (2b); that is, to localize the data communications in the VLSI system.

Consider the case when an algorithm of form (1) with n loops provides m constant data dependence vectors. These are grouped in a matrix $D = [\bar{d}_1, \bar{d}_2, \cdots, \bar{d}_m], D \in R^{n \times m}$. A linear transformation T is sought, i.e., $\bar{J} = T \cdot \bar{I}$. Since T is linear $T(\bar{I} + \bar{d}_j) - T(\bar{I}) = T\bar{d}_j = \bar{\delta}_j$ for $1 \leqslant j \leqslant m$. These equations can be written as

$$TD = \Delta \tag{4}$$

where $\Delta = [\bar{\delta}_1, \bar{\delta}_2, \cdots, \bar{\delta}_m]$. The matrix Δ represents the modified data dependences in the new index space $\mathcal{L}^n$, and according to the requirements (2b) they are assumed known.

The interesting question now is under what conditions can such T exist? System (4) represents $n \times m$ diophantine equations with n^2 unknowns. T exists if system (4) has solution and the solution consists of integers. The following theorem indicates the necessary and sufficient conditions for valid linear transformations and it can be used as a tool to preselect Δ.

Theorem 1: For an algorithm with a constant set of data dependences D, the necessary and sufficient conditions that a valid transformation T exists are as follows:

i) The new data dependence vectors $\bar{\delta}_j$ are congruent to the dependences $\bar{d}_j$ modulo c_j, where c_j is the greatest common divisor (gcd) of the elements of $\bar{d}_j$

$$\bar{\delta} \equiv \bar{d}_j (\text{mod } c_j). \tag{5}$$

ii) System (4) can be solved for T.

iii) The first nonzero element of vector $\bar{\delta}_j$ is positive.

Proof: Sufficient: Condition i) indicates that the elements of $\bar{\delta}_j$ are multiples of the gcd of the elements of respective $\bar{d}_j$. This is a necessary and sufficient condition that each of the $n \times m$ diophantine equations can be solved for integers [17]. According to ii) system (4) has solution. Since the first nonzero elements of $\bar{\delta}_j$ are positive it follows that $\Pi\bar{d}_j > 0$, thus T is a valid transformation. *Necessary:* Transformation T is a bijection and consists of integers, hence i) and ii) are required conditions. Because T preserves the ordering R_T, $\bar{\delta}_j > 0$ and this implies that the first nonzero element is positive. QED

In the selection of Δ one should choose the smallest possible integers for its elements. In this way, the processing time and the communication requirements of the transformed algorithm are being optimized.

C. Mapping Algorithms into Hardware

The transformation of the index sets described above is the key towards an efficient mapping of an algorithm into a special-purpose VLSI array. It is shown in what follows how the global model of the VLSI device introduced in Section I can be derived directly from the transformed algorithm.

The functions F performed by the cells are derived directly from the mathematical expressions indicated in the algorithm. An algorithm of form (1) contains assignment statements in one loop body which is executed repeatedly for all iteration points in set $\mathcal{L}^n$. This implies that all the processing cells can be made identical. The peripheral cells performing input/output operations are, of course, different than the rest. If the mathematical expressions inside the loop involve too many computations, the loop can be split into several simpler loops. Algorithms with several distinct loop bodies normally require different processing cells.

The network geometry G refers to the physical underlying of the network, and it is derived from the mapping $S: \mathcal{L}^n \to \mathcal{L}_T^{n-k}$. A processing cell is assigned to each distinct element of $\mathcal{L}_T^{n-k}$. Assuming that in algorithm (1), $u_i - l_i = O(N)$ for all $i = 1, 2, \cdots, n$ where N is the size of the problem, it follows that the total number of processing cells is $O(N^{n-k})$. The position, or the identification number of each cell is given by $S(\bar{I}) = (J^{k+1}, \cdots, J^n)$. The interconnections between cells necessary for the data communication are derived directly from the last $n - k$ components of the modified data dependence vectors $\bar{\delta}_j^S = S(\bar{I} + \bar{d}_j) - S(\bar{I})$, which becomes $S\bar{d}_j$ for linear transformations. For each cell, the vectors $\bar{\delta}_j^S$ indicate the relative destination of the variable associated to that dependence vector. These interconnections are then replicated to the entire network. Although three-dimensional and multilayer VLSI networks may be attempted, the most practical is the planar arrangement. If $n - k > 2$, an additional one-to-one mapping S' is necessary $S': \mathcal{L}_T^{n-k} \to \mathcal{L}_T^2$.

The network timing T is derived from the mapping $\Pi: \mathcal{L}^n \to \mathcal{L}_T^k$. The exact time when the processing related to an element $\bar{I} \in \mathcal{L}^n$ occurs is simply $\Pi(\bar{I})$. The communication time for a data stream associated with a dependence vector $\bar{d}$ is given by $\Pi(\bar{I} + \bar{d}) - \Pi(\bar{I})$, which in the case of a linear transformation reduces to $\Pi\bar{d}$. The total running time for VLSI algorithm is $[\max \Pi(\bar{I}) - \min \Pi(\bar{I})]$. It can be seen that linear transformations yield a running time $O(N^k)$, while higher order mappings Π will normally lead to higher order processing times. Notice that the running time includes only the computation time and the communication time and not the input/output time. Another observation is that keeping k as small as the transformation permits should be one goal in designing VLSI algorithms. This will increase the concurrency of operations at the expense of the number of processors.

It remains to demonstrate that indeed the VLSI model executes the algorithm correctly. Since we consider here only the global model, it will be sufficient to show that the data flow through the VLSI network is correct. We say that the data flow through the network is correct if all the variables necessary to compute the mathematical expressions of the algorithm are available at the proper time at the proper cell. The following theorem refers to linear transformations.

Theorem 2: A transformation

$$T = \begin{bmatrix} \Pi \\ S \end{bmatrix}$$

of an algorithm which satisfies Theorem 1 maps that algorithm into a systolic array in which the data flow is correct.

Proof: Consider a typical assignment statement $x = E(v_1, v_2, \cdots, v_r)$ executed at $\bar{I} \in \mathcal{L}^n$. From the definition of data dependence vectors we have

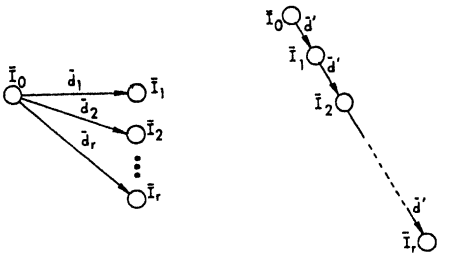
$$\bar{I} = \bar{I}_1 + \bar{d}_1 = \bar{I}_2 + \bar{d}_2 = \cdots = \bar{I}_r + \bar{d}_r \tag{6}$$

where $\bar{I}_i \in \mathcal{L}^n$ and $\bar{d}_i$ correspond to the generation of variables v_i. Apply the linear operators Π and S to (6)

$$\Pi\bar{I} = \Pi\bar{I}_1 + \Pi\bar{d}_1 = \Pi\bar{I}_2 + \Pi\bar{d}_2 = \cdots = \Pi\bar{I}_r + \Pi\bar{d}_r \tag{7}$$

$$S\bar{I} = S\bar{I}_1 + S\bar{d}_1 = S\bar{I}_2 + S\bar{d}_2 = \cdots = S\bar{I}_r + S\bar{d}_r. \tag{8}$$

If the computations at $\bar{I}_i \in \mathcal{L}^n$ produce correctly v_i, then it follows from (7) and (8) that all the input variables will be available for $\bar{I} \in \mathcal{L}^n$ at the same time and at the same processing cell. For each v_i it corresponds a $\bar{d}_i$ and Δ can be selected as desired. It follows that there is no overlap in the flow of



(a) (b)

Fig. 3. (a) Broadcasted variables. (b) Pipelined variables.

the data streams and no cell is required to perform more than one operation at any one time. QED

D. Procedure for Mapping Algorithms into VLSI Systolic Arrays

In this subsection, a procedure is proposed which summarizes the technique introduced above. This procedure will then be used in the next section to discuss some examples.

Step 1) Pipeline all variables in the algorithm.
Step 2) Find the set of data dependence vectors.
Step 3) Identify a valid transformation for the data dependence vectors and the index set.
Step 4) Map the algorithm into hardware.
Step 5) Prove correctness and analyze performances.

Explanation: The role of the first step is to eliminate all possible data broadcasts which may exist in the original algorithm. Consider, for example, that a variable v is generated at some element $\bar{I}_0 \in \mathcal{I}^n$ and used at several other elements $\bar{I}_i \in \mathcal{I}^n$, $i = 1, 2, \cdots, r$. There are r data dependence vectors for variable v, as shown symbolically in Fig. 3.

The highest parallelism for variable v is achieved when all the use statements $S(\bar{I}_i)$, $i = 1, 2, \cdots, r$, are performed in parallel, provided that there are no other restrictions; thus the generated variable v needs to be broadcasted at once. However, it is likely that in the VLSI implementation, this algorithm will be communication saturated. The goal is then to reduce the number of the original data dependences. The solution to this problem is to pipeline the propagation of variable v for the r usage statements, as shown in Fig. 3(b). New data dependences have been created by arbitrarily ordering the usage elements. If the new arrangement offers fewer dependences, then this will eventually translate into fewer communication requirements.

Typically, broadcasts are signaled by missing indices of variables in the loop. In order to avoid broadcasts and to increase pipelining we first complete all the missing indices and introduce new artificial variables such that for each generated variable there is only one destination.

Example: Consider the following loop which implements a matrix multiplication $C = AB$, where $A, B \in R^{n \times n}$.

```
DO 10  k = 1 to n
DO 10  i = 1 to n
DO 10  j = 1 to n                                   (9)
c(i, j) = c(i, j) + a(i, k) · b(k, j)
10 CONTINUE.
```

This loop can be written in an equivalent form in which variables a, b, and c are pipelined

```
      DO 10  k = 1 to n
      DO 10  i = 1 to n
      DO 10  j = 1 to n
S1 :  a^{j+1}(i, k) = a^j(i, k)                     (10)
S2 :  b^{i+1}(k, j) = b^i(k, j)
S3 :  c^{k+1}(i, j) = c^k(i, j) + a^j(i, k) · b^i(k, j)
   10 CONTINUE.
```

The data dependence vectors of the algorithm can now be found using their definition. All possible pairs of generated (output) and used (input) variables are formed and their indices are equated. This is equivalent to writing $\bar{I}_1 + \bar{d} = \bar{I}_2$, from which the data dependence vector $\bar{d}$ can be found directly. It is possible that two different pairs of variables lead to the same data dependence vector. Caution should be exercised to identify only valid generated-used pairs of variables.

As an example, consider loop (10). For variable c, S_3 is both the generate and the use statement. The pair $\langle c^{k'+1}(i, j), c^k(i, j)\rangle$ is formed (in this example we use $'$ for the indices of the generated variable). This yields a dependence vector $\bar{d}_1 = (k - k', i - i', j - j')^t = (1, 0, 0)^t$.

The generated variable $a^{j'+1}(i', k')$ in S_1 is used in S_1 and S_3. However, only one distinct pair can be formed for variable a, i.e., $\langle a^{j'+1}(i', k'), a^j(i, k)\rangle$. It follows that $\bar{d}_2 = (k - k', i - i', j - j')^t = (0, 0, 1)^t$. Similarly, the data dependence vector is found for variable b, $\bar{d}_3 = (0, 1, 0)^t$. For the matrix multiplication algorithm pipelined as in loop (10), there are only three data dependence vectors $\bar{d}_1$, $\bar{d}_2$, and $\bar{d}_3$. Note that these vectors are independent of the index set.

Steps 3) and 4) have been discussed only for the case of linear transformations. As will be seen in the next section, the dynamic programming algorithm requires a more complex transformation. Step 5) is necessary in order to validate the mapping process and to ensure that the performances obtained are satisfactory.

III. Examples

A. LU Decomposition of a Matrix A

Consider a matrix A which can be decomposed into a lower and upper triangular matrices by Gaussian elimination without pivoting. VLSI computing structures for the LU decomposition problem have been proposed by Kung [1], Kung [4], Hwang and Cheng [5], and others. In this example it is shown that previously proposed architectures can be formally derived by using appropriate algorithm transformations.

The algorithm for the LU decomposition of a matrix $A = [a_{ij}]$ is expressed by the program written in Pidgin ALGOL

```
for  k ← 0 until n - 1 do
     begin
     u_kk ← 1/a_kk
     for j ← k + 1 until n - 1 do
         u_kj ← a_kj
     for i ← k + 1 until n - 1 do                   (11)
         l_ik ← a_ik u_kk
     for i ← k + 1 until n - 1 do
         for j ← k + 1 until n - 1 do
             a_ij ← a_ij - l_ik · u_kj
     end.
```

This program can be rewritten into the following equivalent form in which all the variables have been pipelined and all the data broadcasts have been eliminated.

TABLE I
DATA DEPENDENCES FOR LU DECOMPOSITION ALGORITHM

The pairs of generated-used variables	Date dependences k-k'	i-i'	j-j'
$\langle a^{k'}_{i'j'}, a^{k-1}_{ij} \rangle$	(1	0	0)^T = $\bar{d}_1$
$\langle u^{i'}_{k'j'}, u^{i-1}_{kj} \rangle$	(0	1	0)^T = $\bar{d}_2$
$\langle l^{j'}_{i'k'}, l^{j-1}_{ik} \rangle$	(0	0	1)^T = $\bar{d}_3$

```
    for k ← 0 until n - 1 do
      begin
1:      i ← k;
        j ← k;
        u^i_kj ← 1/a^k_ij
        for j ← k + 1 until n - 1 do
2:        begin
            i ← k;
            u^i_kj ← a^k_ij
          end
        for i ← k + 1 until n - 1 do
3:        begin
            j ← k;                                   (12)
            u^i_kj ← u^{i-1}_kj;
            l^j_ik ← a^k_ij · u^i_kj
          end
        for i ← k + 1 until n - 1 do
          for j ← k + 1 until n - 1 do
4:          begin
              l^j_ik ← l^{j-1}_ik;
              u^i_kj ← u^{i-1}_kj;
              a^k_ij ← a^{k-1}_ij - l^{j-1}_ik u^{i-1}_kj
            end
      end.
```

This algorithm is similar to the matrix multiplication (9). Indeed, both algorithms yield the same data dependences. The only three distinct pairs of generate and use variables and their respective data dependence vectors are summarized in Table I.

The data dependences for this algorithm have the nice property that $D = [\bar{d}_1 \bar{d}_2 \bar{d}_3] = I$. There are several other algorithms which lead to these simple data dependences, and they were among the first to be considered for the VLSI implementation.

Following the methodology of Section II, the next step (step 3) is to identify a linear transformation of form (3). This transformation must have the following properties: $\Pi \bar{d}_i > 0$, it offers the maximum concurrency, and T is a bijection. According to Theorem 1, T exists, and furthermore, $T = \Delta$. Denote

$$T = \begin{bmatrix} t_{11} & t_{12} & t_{13} \\ \hline t_{21} & t_{22} & t_{23} \\ t_{31} & t_{32} & t_{33} \end{bmatrix}.$$

In this case, it is possible to have $k = 1$, thus $\Pi \bar{d}_i = t_{1i} > 0$. The smallest possible positive integers are $t_{11} = t_{12} = t_{13} = 1$. The first two conditions are satisfied; and Π is unique. In the selection of mapping S we are now restricted only by the fact that T must be a bijection and consists of integers. A large number of possibilities exist, each leading to different network

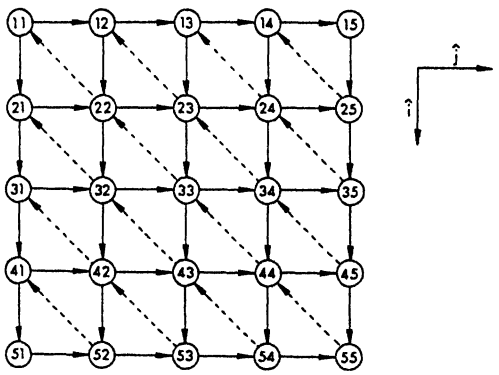


Fig. 4. VLSI array for the LU decomposition algorithm.

geometries. We choose

$$T = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad (13)$$

$$\begin{bmatrix} \hat{k} \\ \hat{i} \\ \hat{j} \end{bmatrix} = T \begin{bmatrix} k \\ i \\ j \end{bmatrix}.$$

The original indices k, i, j are transformed by T into $\hat{k}$, $\hat{i}$, $\hat{j}$. The organization of the VLSI array, for $n = 5$ generated by the transformation (13) is shown in Fig. 4.

In this architecture variables a^k_{ij} do not travel in space, but are updated in time. Variables l^k_{ij} move along the direction $\hat{j}$ (east) with a speed of one grid per time unit, and variables u^k_{ij} move along the direction $\hat{i}$ (south) with the same speed. The network is loaded initially with the coefficients of A, and at the end the cells below the diagonal contain L and the cells above the diagonal contain U.

The processing time is $\Pi_{max} - \Pi_{min} = 3n - 5$. All the cells have the same architecture. However, their functions at one given moment may differ. It can be seen from the program (12) that some cells may execute loop 4, while others execute loops 2 or 3. If we wish to assign the same loops only to specific cells, then the mapping S must be changed accordingly. For example, the transformation

$$T = \begin{bmatrix} 1 & 1 & 1 \\ -1 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix}$$

introduces a new data communication link between cells, toward north-west. These new links will support the movement of variables a^k_{ij}. According to this new transformation, the cells of the first row always compute loop 2, the cells of the first column compute loop 3, and the rest compute loop 4. The reader can now easily identify some other valid transformations which will lead to new organizations. By applying Theorem 2 to this example, one can prove the validity of an architecture.

In the next example, we will see data dependences which are no longer fixed, and this presents a challenge for finding a proper transformation.

B. Dynamic Programming

Many problems in computer science and engineering can be solved by the use of dynamic programming techniques. We consider here the VLSI implementation of an optimal paren-

TABLE II
DATA DEPENDENCES FOR THE DYNAMIC PROGRAMMING ALGORITHM

Pairs of generated-used variables	Data dependences $l-l'$	$i-i'$	$k-k'$
$\langle m^{l'}_{i'k'}, m^{l-1}_{ik} \rangle$	1	0	$0)^T = \bar{d}_1$
$\langle m^{l'}_{k'+1,\, i'+l'}, m^{l-1}_{k+1,i+l} \rangle$	1	-1	$0)^T = \bar{d}_2$
$\langle m^{l'}_{i',\, i'+l'}, m^{l-1}_{ik} \rangle$	1	0	$f)^T = \bar{d}_3$
$\langle m^{l'}_{i',\, i'+l'}, m^{l-1}_{k+1,\, i+l} \rangle$	1	-1	$g)^T = \bar{d}_4$

thesization algorithm based on dynamic programming. A string of n matrices are multiplied

$$M = M_1 \times M_2 \times \cdots \times M_n.$$

Let $r_0, r_1, \cdots, r_n$ be the dimensions of the n matrices with r_{i-1} and r_i dimensions of M_i. Denote by m_{ij} the minimum cost of computing the product $M_i \cdot M_{i+1} \cdots M_j$. The algorithm which finally produces m_{1n} is written as follows [18]:

```
for i ← 1 to n do m_ii ← 0
for l ← 1 to n - 1 do
    for i ← 1 to n - l do
      begin                                                    (14)
        j ← i + l
        m_ij ← MIN_{i≤k≤j} (m_ik + m_{k+1,j} + r_{i-1} r_k r_j)
      end.
```

Following the methodology of Section II, this program is transformed into the following equivalent form:

```
for i ← 1 to n do m_ii ← 0.
  for l ← 1 to n - 1 do
    for i ← 1 to n - l do
      for k ← i to i + l - 1 do
        begin                                                  (15)
          m^l_ik ← m^{l-1}_ik
          m^l_{k+1,i+l} ← m^{l-1}_{k+1,i+l}
          m^l_{i,i+l} ← MIN (m^{l-1}_ik + m^{l-1}_{k+1,i+l} + r_{i-1} r_k r_{i+l})
        end.
```

We will assume that the input data r are loaded on the network before the computations start, so we can neglect the dependences caused by the constant data r. This assumption is made only for the purpose of simplifying the explanation; in fact, the dependences caused by data r are similar to those generated by variable terms.

The data dependences derived from the above algorithm are shown in Table II. There are only four possible distinct pairs of used-generated variables.

The data dependence vectors for the first two pairs of generated-used variables are easily derived in the same manner as for the previous examples (see Table II). The last two dependences, however, require more attention. Consider, first, the pair $\langle m^{l'}_{i',i'+l'}, m^{l-1}_{ik} \rangle$; it yields that $l - l' = 1, i - i' = 0$, and $k = i' + l'$. Form program (15), k' takes values between i' and $i' + l' - 1$. It follows that $k - k' = l - 1, l - 2, \cdots, 1$. Similarly, the pair $\langle m^{l'}_{i',i'+l'}, m^{l-1}_{k+1,i+l} \rangle$ yields $l - l' = 1, i' + l' = i + l$, and $i' = k + 1$. From the first two equalities it results that $i - i' = -1$, and finally, since $k' = i', \cdots, i' + l' - 1$ it follows that $k - k' = -1, -2, \cdots, -l + 1$. Therefore, for both $\bar{d}_3$ and $\bar{d}_4$, $k - k'$ can take many possible values

$$f = l - 1, l - 2, \cdots, 1$$

$$g = -1, -2, \cdots, -l + 1.$$

The difference $k - k'$ is not fixed because the order in which the minimization in loop k is performed is not specified. For instance, in program (15) if $m^l_{i,i+l}$ is generated when k takes the largest value, then $f = 1$ and $g = -l + 1$. Notice that, if the minimization procedure in loop k is performed sequentially, then either f or g will depend on the value of l. This fact constitutes an obstacle in finding a linear transformation for the dynamic programming problem. Fig. 5 shows the dependences between the iteration elements for $n = 6$. Each column corresponds to a different value of l and each group in the column corresponds to a different loop k, in which the order is not specified yet. For example, element 512 receives data from elements 412 and 422, but element 511 receives m_{26}, the result of elements 422, 423, 424, and 425.

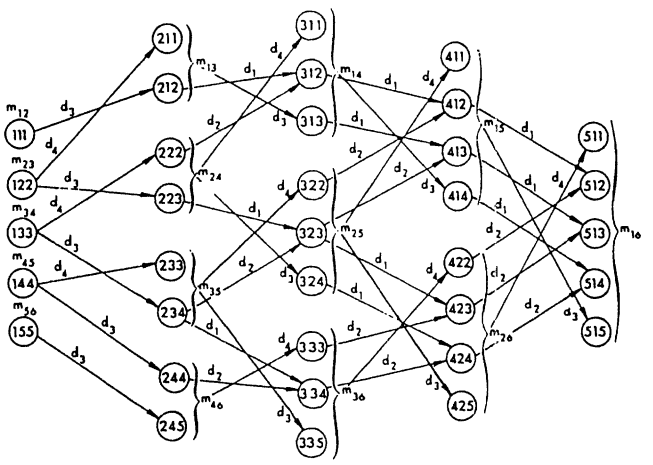


Fig. 5. Data dependence for the dynamic programming algorithm ($n = 6$). The encricled numbers correspond to elements of the index set l i k.

The following mapping Π is proposed for the dynamic programming algorithm:

$$\Pi(l, i, k) = \max \begin{cases} \Pi_1(l, i, k) = 2l + i - k \\ \Pi_2(l, i, k) = l - i + k + 1. \end{cases}$$

The index set, which constitutes the domain of the mapping function Π, is separated into two disjoint sets, one for Π_1 and the other for Π_2. This mapping Π has the advantage that by exploiting possible concurrencies within loop k, it provides a processing time $O(n)$. The first half of any loop k uses Π_1 and the second half uses Π_2. Because of this new concurrency between the first and the last index elements of loop k, the dependences $\bar{d}_3$ and $\bar{d}_4$ are transformed respectively in $(1\ 0\ 1)^t$ and $(1\ -1\ -1)^t$. This is possible because Π_1 applies to $\bar{d}_4$'s while Π_2 applies to $\bar{d}_3$'s. The only inconvenience created by mapping Π is that the data flow in data streams does not have a constant speed. This is easily seen from the fact that $\Pi_1 \bar{d}_1 = 2 \neq \Pi_2 \bar{d}_1 = 1$ and $\Pi_1 \bar{d}_2 = 1 \neq \Pi_2 \bar{d}_2 = 2$.

The mapping S is selected such that the resulting VLSI architecture will be simple and regular.

$$S = \begin{bmatrix} 1 & 1 & 0 \\ 0 & -1 & 0 \end{bmatrix}$$

$$S \cdot [\bar{d}_1 \bar{d}_2 \bar{d}_3 \bar{d}_4] = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}.$$

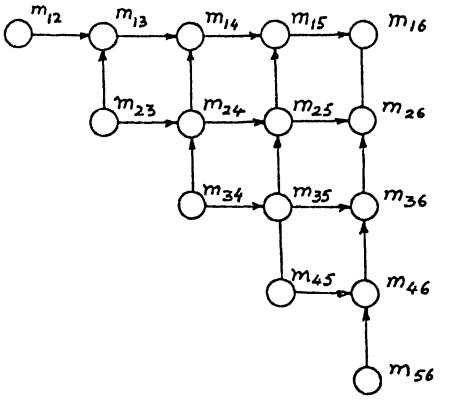


Fig. 6. VLSI array for the dynamic programming.

The mapping Π together with the mapping S form a transformation T: $(l, i, k) \rightarrow (\hat{l}, \hat{i}, \hat{k})$

$$T: \begin{bmatrix} \hat{l} \\ \hat{i} \\ \hat{k} \end{bmatrix} = \begin{bmatrix} \max(2l + i - k, l - i + k + 1) \\ l + k \\ -k \end{bmatrix}.$$

This transformation leads to the VLSI architecture shown in Fig. 6. This architecture was first proposed by Guibas *et al.* [3]. All the processing cells perform the same functions, and no memories are required. There are $O(n^2)$ cells. The operation of this network and the proof of correctness become now particular cases of Theorem 2.

IV. Conclusions

The design of special-purpose VLSI devices is a multistep process (see Fig. 2). In this paper we have concentrated only on the step concerned with the mapping of linear cyclic algorithms into high-level VLSI models. The VLSI devices are assumed to be two-dimensional array processors with local communications. The model resulting from the mapping procedure specifies the complexity of processors, interprocessor connections, and the timing of the data flow. Although we have concentrated in this paper only on a class of algorithms, the methodology proposed here can constitute the foundation of a unifying approach to the design of VLSI algorithms.

Perhaps the most important information about an algorithm is contained in its data dependences because they determine the algorithms's communication requirements. The basic idea of this paper is to modify the data dependences vectors such that the new algorithm satisfies the VLSI requirements, while remaining input/output equivalent to the original algorithm. Transformations of other classes of algorithms into parallel forms constitute a further research topic.

An important feature of the technique proposed in this paper is that the idea of data dependence vectors can be extended to the next step of the VLSI design, that is, the actual design of the processors. This is achieved by studying the dependences at the register level and the bit level.

The design of algorithmically specialized VLSI devices is at its beginning. The development of specialized devices to replace mathematical software is feasible but is still costly. Several important technical issues remain unresolved, and deserve further investigation. Some of these are: I/O communication in VLSI technology, partitioning of algorithms to maintain their numerical stability, and minimization of the communication among computational blocks. Also, a better understanding of the design of parallel algorithms starting directly from the computational problem is necessary.

Finally, the concepts introduced in this paper are not restricted only to VLSI systems; they can also be used for mapping algorithms into some other fixed parallel computer architectures.

References

[1] H. T. Kung, "Let's design algorithms for VLSI systems," in *Proc. Caltech Conf. on VLSI*, pp. 65–90, Jan. 1979.

[2] —, "The structure of parallel algorithms," *Adv. Comput.*, vol. 19, pp. 65–111, 1980.

[3] L. J. Guibas, H. T. Kung, and C. D. Thompson, "Direct VLSI implementation of combinational algorithms," in *Proc. Caltech Conf. on VLSI*, pp. 509–525, Jan. 1979.

[4] S. Y. Kung, "VLSI array processor for signal processing," presented at the Conf. on Advance Research in Integrated Circuits, MIT, Cambridge, MA, Jan. 28–30, 1980.

[5] K. Hwang and Y. H. Cheng, "VLSI computing structures for solving large scale linear system of equations," in *Proc. Parallel Processing Conf., 1980*, pp. 217–227.

[6] F. P. Preparata and J. Vuillemin, "Optimal integrated-circuit implementation of angular matrix inversion," in *Proc. Parallel Processing Conf., 1980*, pp. 211–216.

[7] J. M. Speiser, H. J. Whitehouse, and K. Bromley, "Signal processing applications for systolic arrays," in *Proc. 14th Asilomar Conf. on Circuits, Systems, and Computers*, Nov. 1980.

[8] J. G. Nash, S. Hansen, and G. R. Nudd, "VLSI processor array for matrix manipulation," presented at the Conf. VLSI Systems, Carnegie-Mellon Univ., Pittsburgh, PA, Oct. 1981.

[9] D. I. Moldovan, "Computational models for VLSI systems," Electrical Engineering Dep., Univ. of Southern California, Los Angeles, CA, Rep. DIM-82-3, 1982.

[10] Y. Muraoka, "Parallelism exposure and exploitation in programs," Ph.D. dissertation, Dep. Computer Sci., Univ. Illinois, Urbana-Champaign, Feb. 1971.

[11] D. J. Kuck, Y. Muraoka, and S. C. Chen, "On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup," *IEEE Trans. Comput.*, vol. C-21, pp. 1293–1310, Dec. 1972.

[12] R. Towle, "Control and data dependence for program transformations," Ph.D. dissertation, Dep. Computer Sci., Univ. Illinois, Urbana-Champaign, Mar. 1976.

[13] U. Banergee, "Data dependence in ordinary programs," M.S. thesis, Dep. Computer Sci., Univ. Illinois, Urbana-Champaign, Nov. 1976.

[14] U. Banerjee *et al.*, "Time and parallel processor bounds for Fortran-like loops," *IEEE Trans. Comput.*, vol. C-28, pp. 660–670, Sept. 1979.

[15] R. H. Kuhn, "Optimization and interconnection complexity for: parallel processors, single stage networks and decision trees," Ph.D. dissertation, Dep. Computer Sci., Univ. Illinois, Urbana-Champaign, 1980.

[16] L. Lamport, "The parallel execution of DO loops," *Commun. ACM*, pp. 83–93, Feb. 1974.

[17] L. J. Mordell, *Diophantine Equations*. New York: Academic Press, 1969, p. 30.

[18] A. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Reading, MA: Addison-Wesley, 1975.

Regular Iterative Algorithms and their Implementation on Processor Arrays

SAILESH K. RAO AND THOMAS KAILATH, FELLOW, IEEE

In this paper, we summarize some recent results concerning a class of algorithms known as Regular Iterative Algorithms, particularly with respect to their implementations on processor arrays. Regular Iterative Algorithms contain all algorithms executed by systolic arrays as a proper subclass and are therefore of considerable importance in real-time signal processing applications.

I. Introduction

The advent of VLSI has encouraged research into the design and development of special-purpose processor arrays for use in various applications, notable in signal processing. Such research was given a major impetus by the work of Kung and Leiserson [1], especially when it was reproduced in the pioneering textbook of Mead and Conway [2] on VLSI design. Kung and Leiserson introduced a class of parallel/pipelined arrays that they dubbed with the attractive name of "systolic" arrays. Since then systolic solutions have been derived for various problems: notably, in signal processing, numerical linear algebra, and in graph-theoretic applications. This intense research activity was motivated by the apparent simplicity of the hardware design: one merely patterns the layout for a single processing element in the array and replicates this pattern appropriately.

It may be noted that the idea of using a "regular iterative" array of processors for solving various problems actually dates back to the late 1950s and the early 1960s (see, e.g., Unger [3], [4], McCluskey [5], Hennie [6]). Moreover, such iterative layout patterns have already been effectively used for silicon memories, array multipliers, transversal filters, etc. The important contribution of Kung and Leiserson was in pointing out the applicability of these layout strategies for problems in signal processing, numerical computing, graph theory, and other areas.

However, despite fairly intense research activity in this field, the design of systolic arrays is still largely done on a heuristic and intuitive basis. This can be illustrated by discussing a set of natural questions that arise in this field:

Question 1. Given an algorithm, is there a systolic array implementation for it? The current answer is maybe, or maybe not. One has to try to find a systolic implementation by some means or other. If this can be done, then the answer, of course, is yes. On the other hand, one can hardly say no for any particular problem, because some more skillfull designer might find a systolic implementation. For example, in their original work, Kung and Leiserson [1] presented a systolic array for solving linear equations of Gaussian elimination, but without pivoting. Of course, pivoting is essential for obtaining meaningful numerical results for most problems, with the notable exception of equations with positive-definite coefficient matrices. However, no one has yet succeeded in finding a systolic array for Gaussian elimination with pivoting. It would be nice to have a test for whether a particular algorithm has a systolic implementation or not, without having to first find an actual implementation. A reason that such a test has not been available so far is that the existing "definitions" of systolic arrays have been too qualitative. For example: "A systolic array is a one- or two-dimensional array, the body of which is composed of identical functional modules arranged in a geometric lattice each interconnected to its nearest neighbors and utilizing common control synchronous timing." The problem with such a definition is that it is not too hard to find examples of arrays that meet all of the stated requirements but that most people would refuse to accept as systolic, because they failed to have some property (e.g., efficiency) not explicitly stated above. We could elaborate on this theme further, but let us continue with our questions.

Question 2. If the answer to Question 1 is yes, is there more than one systolic implementation? More specifically, how many different implementations are there, really? As an example, a much studied problem is that of matrix multiplication, and by now about four or five different systolic implementations have been presented [1], [7]–[9]. Are there more? The point, of course, is that these alternative implementations might have better properties of one kind or

Manuscript received August 31, 1986; revised June 25, 1987. This work was supported in part by the Joint Services Program at Stanford University (U.S. Army, U.S. Navy, and U.S. Airforce) under Contract DAAG29-85-K-0048, the National Science Foundation under Grant DCI-84-21315-A1, the U.S. Army Research Office under Contract DAAL03-86-K-0045, and the Air Force Office of Scientific Research, Air Force Systems Command, under Contract AF83-0228.

S. K. Rao is with the VLSI Systems Research Department, AT&T Bell Laboratories, Holmdel, NJ 07733, USA.

T. Kailath is with the Information Systems Laboratory, Stanford University, Stanford, CA 94305, USA.

IEEE Log Number 8716427.

Reprinted from *Proceedings of the IEEE*, pp. 259-269, March 1988.

another, e.g., higher throughput, lower latency, fewer processors, etc.

Question 3. Following up Question 2, the big question is whether there are systematic procedures for finding one or all systolic implementations? There are some partial answers to this question [10]–[19], but many gaps still remain, including, in particular, whether some "reasonable" generalization of the concept of systolic arrays might exist that would still be appropriate for VLSI technology.

In our recent efforts [9], [20]–[22], we have begun to find some satisfactory answers to the above questions. For example, a mathematical formulation has been obtained that shows, *inter alia*, that there is no true systolic implementation for Gaussian elimination with pivoting, but that there is an array of identical processors together with register pipelines and Last-In-First-Out (LIFO) buffers that can implement the pivoting algorithm. We have called such more general arrays *Regular Iterative Arrays* (*RIAs*) and have shown that systolic arrays, precisely defined, form a proper subclass.

In this paper we shall briefly describe the general theory we have developed for RIAs and point out their relationships to earlier work, especially a seminal paper of Karp, Miller, and Winograd [23].

In Section II, some general concepts concerning the design of parallel architectures will be introduced and the importance of devising special techniques that exploit any available structure in the algorithm will be highlighted.

Section III contains a generic description of the existing methodologies for the systematic design of systolic arrays. Using some simple examples, the limitations of these methods will be brought out.

In Section IV, a formal methodology is proposed that overcomes the difficulties in the existing procedures. This methodology is based on some fundamental concepts and results contained in [23].

Finally, Section V contains the concluding remarks.

II. On the Exploitation of Parallelism in Algorithms

An algorithm takes concrete form only through the language expressing it. It has been recognized for some time that for the purpose of extracting the parallelism in an algorithm, standard sequential programming languages such as Fortran and Pascal are ill-suited vehicles for expressing the algorithm. An algorithm written in these languages has a built-in ordering of the computations which most often obscures any parallelism present in the algorithm. Furthermore, ever since the days when core memory was a costly resource to be sparingly used, one has been conditioned to think in terms of minimizing the storage required by the program, and hence encouraged to overwrite on variables as much as possible. Such overwriting further compounds the problem of extracting the parallelism from the program.

The so-called *Single Assignment Language* [24], for example, provides the means for overcoming the difficulties mentioned above by requiring that every variable defined in the program take on a unique value during the course of the computation. Thus assignment statements of the form "$(a := a + b)$" are not allowed since a appears on both sides of the statement. If an algorithm is expressed as a *Single Assignment Algorithm*, viz., as a program in the single assignment language, then one can conceive of automated procedures for extracting the parallelism in the algorithm, with no further effort required of the user.

Given a single assignment algorithm, it is possible to capture the information regarding the parallelism in the algorithm by means of a *dependence graph*. This graph has one node for each of the variables in the algorithm and a directed arc from node x to node y if and only if variable y is computed using the value of x in the algorithm. The dependence graph of a single assignment algorithm specifies a *partial ordering* among the computations in the algorithm; that is, if there is a directed path in the dependence graph from node x to node y, then the computation represented by node y must be executed *after* the computation represented by node x is completed, no matter how many processors are brought to bear upon the problem. In such a case, one would say that y is dependent upon x, and if a path from x to y is an edge, this dependence is direct. From this observation, one can infer that the *length of the longest path* in the dependence graph is a lower bound for the total time required for executing the algorithm, independent of the number of processors used in this execution.

Suppose that one wishes to obtain an implementation of the algorithm that is optimal with respect to the total time required for executing the algorithm. One simple and brute-force method for achieving this objective is to use a distinct processor for executing the computation represented by every node in the dependence graph. This, in general, leads to a very inefficient use of the computational resources, since each processor is active only for a constant period of time, which could be a minute fraction of the time required for completing the algorithm. To achieve a better utilization of these resources, it is necessary to reuse the same procesor for handling a large number of computations. In general, the set of computations can be arbitrarily partitioned and assigned to different processors.

In determining an implementation for the algorithm, one must not only specify the processor at which each computation is to be performed, but also assign a time at which it is to be executed by the processor. This mapping of computations into time slots is referred to as the construction of a *schedule* for the computations. A schedule must satisfy the precedence constraints imposed by the dependence graph of the algorithm and must also be such that no two computations assigned to the same processor are expected to be executed at the same time. A schedule must also take into account the communication constraints among the processors. That is, if variable x is computed by processor p_x and if x is required as an input to the computation of variable y at processor p_y, then the schedule for the execution of y must include the time required to communicate the value of x from p_x to p_y. If the processors are arranged in a general-purpose communication fabric, such as a crossbar switch, then the time required for this communication will also depend upon the prevalent congestion in the switch. Clearly, for different partitions of the nodes in the dependence graph, the interprocessor communications required will differ in general.

The problem of determining an optimal schedule, i.e., one that minimizes the total time for the execution of the algorithm, is extremely hard even if the interprocessor communication among the processors is assumed to be instantaneous. Indeed, it has been proved to be NP-complete even

in the presence of many simplifying assumptions (see, e.g., Ullman [25]). If communication constraints have to be taken into account as well, then the problem becomes even more intractable, thereby forcing one to seek ways of exploiting any available structures in the algorithm.

Regular Iterative Algorithms (RIAs) are a special subclass of single assignment algorithms for which many of these difficulties can be successfully overcome. Indeed, for a Regular Iterative Algorithm, one can ensure that all computations assigned to the same processor can be described by the same simple instruction. For instance, if this instruction is a multiply operation, then one can replace this processor by a simple serial multiplier element. Furthermore, for a Regular Iterative Algorithm, one can ensure that the interprocessor communication required is fixed and can be implemented using a few dedicated links. A further attraction of this class is that the schedule for the algorithm can be constructed to be "periodic" so that the necessary delays on the interprocessor links can be implemented using shift registers and Last-In-First-Out buffers alone, without any additional control circuitry. Finally, Regular Iterative Algorithms form an extremely useful subclass of single assignment algorithms, in so far as signal processing applications are concerned, as shown in [9].

III. An Informal Description of RIAs and their Implementations

A formal definition of Regular Iterative Algorithms is given in [9] (see Appendix). For the present discussion, it is best to explore the various features of an RIA by means of some simple examples.

Example 1

An urn contains N red balls and N green balls. The following experiment is conducted repeatedly until either the urn is empty or exactly one ball remains:

> Two balls are picked at random from the urn. If they are of the same color, then one of these is replaced in the urn. If they are of different colors, then both are discarded.

To determine the probability that the urn is empty at the end of the experiment, one can derive a recursive algorithm, using elementary counting arguments. Let $p(i, j)$ denote the probability that the urn becomes empty if there are i red balls and j green balls to begin with. Then

$$p(i, j) = \frac{i(i-1)\,p(i-1, j) + j(j-1)\,p(i, j-1) + 2ijp(i-1, j-1)}{(i+j)(i+j-1)}, \quad 1 \le i, j \le N \tag{1}$$

with $p(0, 0) = 1$, $p(i, 0) = 0$ for all $i > 0$ and $p(0, j) = 0$ for all $j > 0$.

Example 2

Consider the following simple sorting algorithm referred to as *selection sort* [26]. Given a list of N numbers $\{x(i)\}$, first determine the largest number in the list and delete it from the list. Then, from the $(N-1)$ numbers in the remaining list, delete the largest number and so on iteratively until the list is empty.

To write this algorithm in single assignment form, let the jth instance of the list be given by $\{x(i, j)\}$, where i ranges from 1 to $(N-j)$. In addition, let $m(i, j)$ be the largest number in the segment $\{x(k, j), k = 1 \text{ to } i\}$ of the list. Then

$$m(i, j) = \begin{cases} x(i, j), & \text{if } i = 1 \\ \max\{m(i-1, j), x(i, j)\}, & \text{otherwise} \end{cases}$$

$$x(i-1, j+1) = \begin{cases} \textit{undefined} & \text{if } i = 1 \\ \min\{m(i-1, j), x(i, j)\}, & \text{otherwise} \end{cases} \tag{2}$$

where the first equation follows from the definition of $m(i, j)$ and the second equation creates the $(j+1)$th instance of the list from the jth instance. The calculations in (2) must be carried out for $j = 1$ to $N-1$, and $i = 1$ to $(N-j)$.

Example 3

Given an image represented by the $(N+1 \times N+1)$ matrix U, find the filtered image Y such that

$$y_{ij} = \sum_{k=1}^{n} a_k y_{i-k, j-k} + \sum_{k=0}^{n} b_k u_{i-k, j-k}. \tag{3}$$

Many possible RIAs can be obtained for solving this problem, by applying a simple transformation on known one-dimensional filtering algorithms. However, most of these RIAs are known to be numerically unstable and can thus be ignored. Among the numerically stable algorithms, the ones due to Deprettere and Dewilde [32], Vaidyanathan and Mitra [33], and Fettweis [34] can be written in the form:

For $i = 0$ **to** n **do**

For $j, k = 0$ **to** N **do**

$$\begin{aligned} x(i, j+1, k+1) &= f_{x,i}(x(i, j, k), y(i, j, k), w(i, j, k)) \\ y(i+1, j, k) &= f_{y,i}(x(i, j, k), y(i, j, k), w(i, j, k)) \\ w(i-1, j, k) &= f_{w,i}(x(i, j, k), w(i, j, k)) \end{aligned} \tag{4}$$

where $f_{x,i}$, $f_{y,i}$, $f_{w,i}$ are some linear functions that are determined by a synthesis procedure.

This set of equations is initialized with

$$x(i, 0, k), x(i, j, 0) \text{ and } w(N, j, k) = 0, \quad \text{for all } j, k. \tag{5}$$

The actual inputs are made available as

$$y(0, j, k) = u_{jk} \tag{6}$$

and the output of the filter is obtained as

$$y(N+1, j, k) = y_{jk}. \tag{7}$$

All the algorithms described in the above examples are Regular Iterative Algorithms because they have the following features:

i) They can be easily verified to be in the single assignment format.

ii) Each variable in these RIAs is identified by a *label* (p in Example 1, for instance) and an *index vector* ($\mathbf{k} = [i, j]^T$, in Example 1). The *range* of the index vector, which in gen-

eral can be S-dimensional with $S \geq 1$, is known as the *index space*. For instance, in Example 1, the index space is two-dimensional and is described by an $(N \times N)$ square grid (Fig. 1), whereas in Example 2 it is triangular (Fig. 2). At each

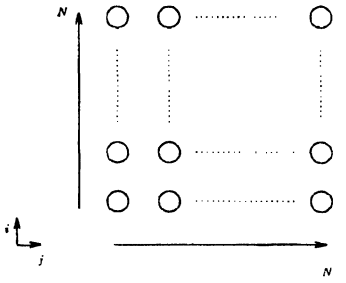


Fig. 1. The index space for the RIAs in Examples 1 and 3.

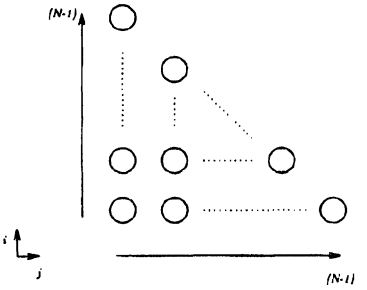


Fig. 2. The index space for the RIA in Example 2.

integer point in the index space, a set of V labels is used to denote the distinct variables ($V = 1, 2, 3$ for Examples 1-3, respectively). The number of integer points in the index space in the above examples is governed by the size parameter N (though in general, there may be several such size parameters).

iii) Finally, the main feature of these algorithms is the regularity of the direct dependences among the variables with respect to the index points. That is, if $x(\boldsymbol{k})$ is computed using the value of $y(\boldsymbol{k} - \boldsymbol{d})$, then the *index displacement vector* $\boldsymbol{d}$, corresponding to this direct dependence, is the same regardless of the index point $\boldsymbol{k}$. In Example 1 for instance, $p(i, j)$ is directly dependent on, say, $p(i - 1, j)$ irrespective of the particular value of i and j. As a consequence of this regularity, the dependence graph of an RIA has an iterative structure, which can be clearly demonstrated by drawing the dependence graph within the index space (see Figs. 3 and 4). Each node in this dependence graph can be identified as $(x, \boldsymbol{k})$ to represent the variable $x(\boldsymbol{k})$ and is (physically) located at the point $\boldsymbol{k}$ in order to exhibit this regularity.

Though the direct dependences among the variables in an RIA are required to be iterative, the actual computations carried out to evaluate these variables can depend upon the index point. This is reflected in the above examples, both through the use of conditional expressions and through the use of values of i and j in the instructions.

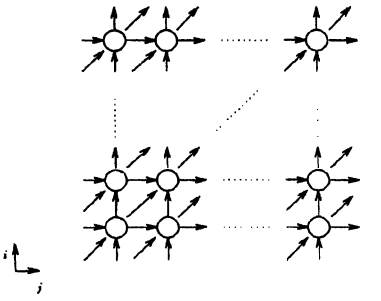


Fig. 3. The dependence graph of the RIA in Example 1.

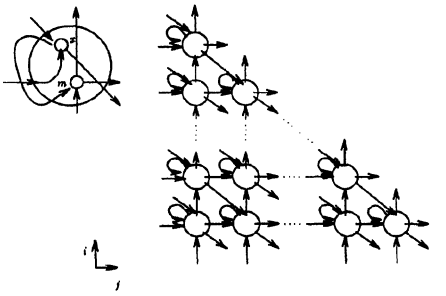


Fig. 4. The dependence of graph for the RIA for sorting in Example 2. While the main figure shows only the index-displacement vectors, the inset alongside reveals the fine structure of the dependence graph.

Implementations of RIAs

We shall now review, in our language, the present status of systematic methods for parallel implementation of the above algorithms. The three examples given above have been chosen to successively illustrate some of the limitations of the present methods.

Given an RIA, suppose that one wishes to exploit the maximum available parallelism in the RIA, while at the same time minimizing the computational resources used in the implementation. To achieve this globally optimal implementation, in general, the set of computations assigned to distinct processors might have to be arbitrary disjoint subsets of the set of all computations. However, the problem of optimally scheduling the computations that are assigned to a given processor becomes extremely hard if the partitioning is arbitrary. Hence, to render this problem tractable, one can restrict attention to *linear* partitions. Here, a set of parallel lines is drawn through the index space, so that all computations that correspond to index points that lie on the same line are handled by the same processor. In this manner, the processor array (including the interprocessor communication links) itself can be obtained by projecting the embedded dependence graph of the RIA along these lines on to a lower dimensional lattice of points known as the *processor space* (see Fig. 5). The direction along which this projection is made is represented by an integer vector $\boldsymbol{u}$, and is defined as the *iteration vector*.

Once the processor space is decided upon, one must attempt to schedule the computations that are mapped on to a given processor, i.e., assign some "time slot" for each variable (with respect to a global reference frame) during

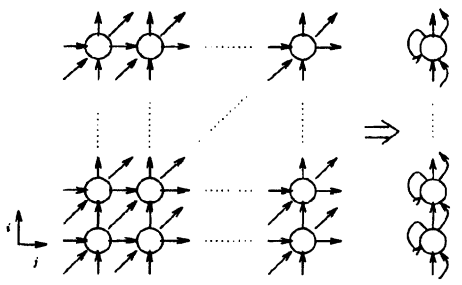


Fig. 5. A processor array obtained for the RIA in Example 1 using the projection method.

which its computation is performed by the processor. As noted before, the choice of the schedule is constrained both by the dependences in the algorithm and by the choice of the processor space. Once again, determining this schedule appears to be difficult, unless one imposes further restrictions on its nature. In keeping with the use of a linear projection to determine the processor space, one can attempt to determine a schedule that is *linear* with respect to the index vectors. In a linear schedule, a set of $(S-1)$-dimensional parallel hyperplanes must be drawn through the index space so that all computations that correspond to index points that lie on the same hyperplane are executed at the same time (necessarily, by different processors). Thus a linear schedule corresponds to *isotemporal* hyperplanes that are drawn through the index space, so that the progression of time is along the direction normal to these hyperplanes. For Example 1, S is two, and hence the linear schedule consists of a set of parallel isotemporal lines as shown in Fig. 6.

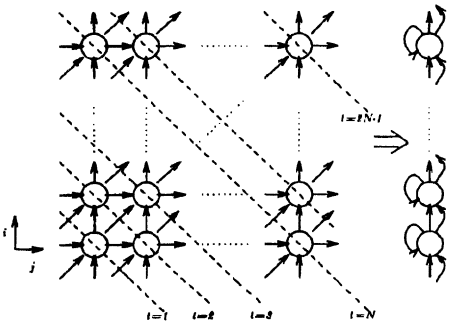


Fig. 6. A linear schedule for the RIA in Example 1, applicable to the processor array in Fig. 5.

Next, we note that a linearly scheduled RIA must be such that all edges in its dependence graph are oriented in directions along which time strictly increases. Thus if the normal to the isotemporal hyperplanes is given by the S-dimensional vector λ, then every distinct index displacement vector $\boldsymbol{d}$ in the RIA must satisfy $\lambda^T\boldsymbol{d} \geq 1$. Furthermore, in order to ensure that all computations assigned to the same processor are scheduled at different times, $\lambda^T\boldsymbol{u}$ must be required to be nonzero. Any vector λ^T that satisfies these constraints simultaneously, defines a valid linear schedule for the algorithm. This approach can be successfully applied for the RIA in Example 1 and the resulting linearly scheduled implementation is illustrated in Fig. 7.

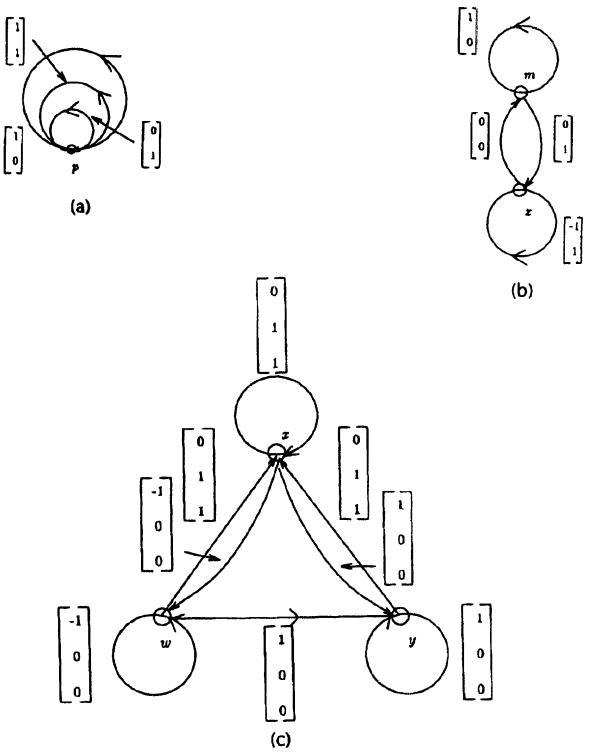


Fig. 7. The RDGs for the RIAs in Examples 1-3, displayed as (a), (b), and (c), respectively. In (c), all edges without any weights explicitly displayed have zero weight.

The procedure informally described above has been independently derived by several authors both in the geometric framework used here [10] and as an algebraic methodology [11]-[19]. Though it is intuitively appealing, it has some very serious drawbacks that severely limit its applicability. Consider, for instance, the RIA in Example 2, for which the distinct index displacement vectors are given by

$$\boldsymbol{d}_{xm} = \begin{bmatrix}0\\0\end{bmatrix} \quad \boldsymbol{d}_{mm} = \begin{bmatrix}1\\0\end{bmatrix} \quad \boldsymbol{d}_{mx} = \begin{bmatrix}0\\1\end{bmatrix} \quad \boldsymbol{d}_{xx} = \begin{bmatrix}-1\\1\end{bmatrix} \tag{8}$$

Surely, it is impossible to find a linear schedule for this RIA, since there exists no vector λ that can satisfy the constraint imposed by the first index displacement vector displayed above. One can, of course, overcome this difficulty simply by introducing the change of variables

$$\overline{m}(i, j) = m(i-1, j) \tag{9}$$

since in the new domain, the index displacement vectors are given by

$$\boldsymbol{d}_{x\overline{m}} = \begin{bmatrix}1\\0\end{bmatrix} \quad \boldsymbol{d}_{\overline{m}\overline{m}} = \begin{bmatrix}1\\0\end{bmatrix} \quad \boldsymbol{d}_{\overline{m}x} = \begin{bmatrix}-1\\1\end{bmatrix} \quad \boldsymbol{d}_{xx} = \begin{bmatrix}-1\\1\end{bmatrix} \tag{10}$$

and $\lambda^T = [1 \;\; 2]$ is a valid schedule for the algorithm. But then, how does one determine the necessary change of variables in general? Among the references listed above, only Quinton [12] recognizes the need for such modifications in the description of the RIA, but the subclass of RIAs studied by Quinton does not include the one in Example 2. Indeed the *Uniform Recurrence Equations* (UREs) of Quinton are

RIAs in which all variables except for a distinguished one, are computed using an assignment statement of the form

$$x(\boldsymbol{k}) = x(\boldsymbol{k} - \boldsymbol{d}) \tag{11}$$

and hence the RIA in Example 2 is not an URE.

The difficulty in this approach is even more serious in the case of the RIA in Example 3. Here, there are two distinct self-loops for which the index displacement vectors are the negative of each other, and thus any choice of the linear schedule, i.e., λ^T, will violate the constraints imposed by at least one of these vectors. It can be easily verified that in this case, a simple change of variables as suggested above cannot be used to overcome this difficulty.

The failure of the above procedure in these two examples may be traced to a certain implicit assumption that it is based upon: *all computations that belong to the same index point in the RIA can be scheduled to begin at the same time.* This may not be possible in general. What if there are directed paths in the dependence graph from some node at a particular index point to some other node at the same index point? Surely then, one cannot schedule these two computations at the same time and hence the second assumption can never be satisfied. Next, what if the coarse version of the dependence graph obtained by coalescing all distinct nodes in the original graph at every index point, is cyclic? Then also, the procedure is not applicable.

To rescue the above procedure, one must therefore pay attention to the fine structure at every index point in the dependence graph of a multivariable RIA ($V > 1$). Most systematic methodologies proposed thus far fail to take this into account, thereby implicity treating a multivariable RIA as a single-variable one. It must, however, be noted that as part of their analysis procedures, Karp, Miller, and Winograd [23] clearly pointed out this important distinction between single-variable RIAs and multivariable RIAs. Indeed, one must truly credit these authors for first pointing out the fact that single-variable RIAs can *always* be linearly scheduled as shown above and that this need not be the case for multivariable RIAs.

IV. A Formal Approach to the Design of Processor Arrays

In a multivariable RIA, each indexed variable defines a lattice of points by itself within the index space. Before defining the isotemporal lines within the index space, one may have to translate each of these lattices to a different origin, independently. Sometimes, mere translations may not suffice and one may also have to rotate each of the lattices independently. In the extreme case, translations and rotations may not suffice by themselves, at which point our geometrical interpretation breaks down. It must be noted here that this extreme case includes many useful, numerically stable algorithms [22], and is hence, not just of academic interest.

While the geometric approach is not powerful enough for our purposes, a formal algebraic framework has been proposed that can deal with *any* well-posed RIA and generate an *optimal* implementation for it [9]. Though this framework was developed recently, it is based on some fundamental concepts and techniques that were reported two decades ago in the seminal paper of Karp, Miller, and Winograd [23].

The fine structure in the dependence graph of an RIA is concisely captured in the concept of a *Reduced Dependence Graph* (RDG)[1] that was introduced in [23]. In general, the RDG of an RIA has V nodes, one for each of the indexed variables in RIA; it has a directed arc from node x to node y, if $y(\boldsymbol{k})$ is computed using the value of $x(\boldsymbol{k} - \boldsymbol{d})$ for some $\boldsymbol{d}$; finally, each directed arc is assigned a vector weight representing the displacement of the index point across the direct dependence. Thus in the above example, the arc from x to y has weight $\boldsymbol{d}$.

The RDGs for the RIAs in Examples 1–3 are shown in Fig. 7. Karp, Miller, and Winograd used this notation of an RDG representation in their study of RIAs defined over a semi-infinite index space. In their case, since the index space was assumed to be the set of all nonnegative integer vectors and was hence known to begin with, the RDG constituted a complete description of the dependence graph of the RIA. However, for many of our problems, the index space will naturally be finite, and hence the RDG together with the specification of the index space, will combine to form a complete description of the dependence graph. (The distinction between finite and semi-infinite index spaces is a nontrivial one; the latter assumption forces one to impose certain "causality" constraints on the dependence graph that cannot be met in several numerical algorithms.)

Given the RDG and a specification of the index space, one has all the information necessary to determine an efficient implementation for the RIA. Before developing the necessary algebraic framework, however, let us attempt to translate the geometric procedure informally described in the previous section into a more formal setting. Once this formalization is in place, one can then try to generalize it so as to be able to overcome the difficulties mentioned earlier.

The S-dimensional iteration vector $\boldsymbol{u}$, introduced earlier, defines the topology of the processor array completely: the computations at two index points $\boldsymbol{k}_1$ and $\boldsymbol{k}_2$ are mapped on to the same processor if and only if

$$\boldsymbol{k}_1 - \boldsymbol{k}_2 = \alpha \boldsymbol{u} \tag{12}$$

where α is some scalar integer. (*Interpretation:* This means that $\boldsymbol{k}_1$ and $\boldsymbol{k}_2$ map on to the same point when the index space is projected along the direction defined by $\boldsymbol{u}$.) Let $\boldsymbol{P}$ be any $(S - 1 \times S)$-dimensional integer matrix of rank $(S - 1)$ that is orthogonal to $\boldsymbol{u}$, i.e.,

$$\boldsymbol{P}\boldsymbol{u} = 0. \tag{13}$$

Then, the processor array is defined by the lattice of points obtained by mapping the index space according to

$$\boldsymbol{p} = \boldsymbol{P}\boldsymbol{k}, \qquad \boldsymbol{k} \in \text{index space} \tag{14}$$

where $\boldsymbol{p}$ defines the location of the processor that carries out the computation at the index point $\boldsymbol{k}$. Indeed, in the above mapping, two index points $\boldsymbol{k}_1$ and $\boldsymbol{k}_2$ are mapped on the same processor (i.e., the same $\boldsymbol{p}$ vector) if and only if they satisfy (12). The necessary interprocessor communication links are then defined by the vector weights on the

[1]This is our terminology. Karp, Miller, and Winograd refer to the RDG as the "dependence graph," which might cause considerable confusion at the present time. Incidentally, Waite [27] also independently introduced the RDG concept in a somewhat different setting in the same journal.

edges in the RDG: if $y(\boldsymbol{k})$ is dependent on $x(\boldsymbol{k} - \boldsymbol{d})$, then there must be a directed link in the array from

$$\boldsymbol{P}(\boldsymbol{k} - \boldsymbol{d}) \rightarrow \boldsymbol{P}\boldsymbol{k} \tag{15}$$

for all $\boldsymbol{k}$.

It must be noted here that while some authors use the iteration vector $\boldsymbol{u}$, in order to derive the processor array (e.g., Cappello and Steiglitz [10]), many prefer to use the transformation matrix $\boldsymbol{P}$ as originally proposed by Moldovan [11], [13]. Both these techniques are equivalent: given $\boldsymbol{u}$, one can determine the equivalent transformation matrix $\boldsymbol{P}$ to be any $(S - 1 \times S)$ matrix that forms a basis for the null space of $\boldsymbol{u}$; conversely, given $\boldsymbol{P}$, the corresponding iteration vector is the unique right null vector for $\boldsymbol{P}$. However, one must be careful in the latter case, since two matrices $\boldsymbol{P}_1$ and $\boldsymbol{P}_2$ represent the same processor array if they have the same row space. Of course, the same consideration applies to the former technique if one is dealing with multidimensional iteration spaces, rather than a one-dimensional iteration vector.

Once the processor array is obtained as above, one must now schedule the computations. If the processor array is to be implemented in a globally synchronous fashion, then this schedule might consist of specifying during which cycle of the global clock a particular computation must be started at the processor. But this requires a detailed knowledge of the capabilities of the processor—whether it is bit-serial or bit-parallel and whether it can begin several computations simultaneously or not. To avoid having to know these details and still be able to derive useful results, one can conceive of a schedule that partitions the computations into global steps with the following restrictions:

i) If variable $y(\boldsymbol{k})$ is computed using variable $x(\boldsymbol{k} - \boldsymbol{d})$, then the step $s_y(\boldsymbol{k})$ at which $y(\boldsymbol{k})$ is computed must be strictly larger than the step $s_x(\boldsymbol{k} - \boldsymbol{d})$ to which $x(\boldsymbol{k} - \boldsymbol{d})$ is assigned, i.e.,

$$s_y(\boldsymbol{k}) \geq s_x(\boldsymbol{k} - \boldsymbol{d}) + 1. \tag{16}$$

ii) All computations at step τ are completed by every processor in the array before step $(\tau + 1)$ is begun.

iii) At each step, each processor must be assigned a "small" number of computations.

The second restriction is only conceptual: a processor may begin step $(\tau + 1)$ when all its neighbors (from which it receives input values) have completed step τ without waiting for the rest of the processors in the array. Hence, even asynchronous implementations can be devised using this schedule. The third restriction needs to be clarified somewhat: we shall assume that a "small" number of computations is assigned at each step to every processor if and only if at most *one* of each of the V indexed variables in the algorithm is assigned to be computed at the processor. This restricts the number of computations to be at most V per step per processor.

In a linear schedule, as assumed in the previous section, the variable $x(\boldsymbol{k})$ is assigned to step $\boldsymbol{\lambda}^T\boldsymbol{k}$ for some constant vector $\boldsymbol{\lambda}$ that is independent of $x(\boldsymbol{k})$. That is

$$s_x(\boldsymbol{k}) = \boldsymbol{\lambda}^T\boldsymbol{k}, \quad \text{for all } x \tag{17}$$

which, as we demonstrated earlier, is not general enough to handle several interesting problems. Indeed, in this case, if $y(\boldsymbol{k})$ is computed using $x(\boldsymbol{k} - \boldsymbol{d})$, then from (16), we must have

$$\boldsymbol{\lambda}^T\boldsymbol{k} \geq \boldsymbol{\lambda}^T(\boldsymbol{k} - \boldsymbol{d}) + 1 \tag{18}$$

i.e.,

$$\boldsymbol{\lambda}^T\boldsymbol{d} \geq 1. \tag{19}$$

This must be true for every distinct index displacement vector $\boldsymbol{d}$ in the RIA, which may or may not be possible for the given RIA. In the sorting example, a zero index displacement vector exists for which it is impossible to satisfy (19).[2]

A natural generalization of the above linear scheduling strategy is a *uniform affine* schedule. Here, we hypothesize that

$$s_x(\boldsymbol{k}) = \boldsymbol{\lambda}^T\boldsymbol{k} + \gamma_x \tag{20}$$

where $\boldsymbol{\lambda}$ is a constant vector, independent of x, whereas γ_x is a scalar that is specific to x. Then, (16) translates into

$$\gamma_y - \gamma_x + \boldsymbol{\lambda}^T\boldsymbol{d} \geq 1 \tag{21}$$

and this must be true for all such dependences, i.e., for every directed edge in the RDG. When compiled together in matrix form, these constrains can be written as

$$\boldsymbol{\gamma}^T\boldsymbol{C} + \boldsymbol{\lambda}^T\boldsymbol{D} \geq [1\ 1 \cdots 1] \tag{22}$$

where

i) $\boldsymbol{C}$ is the familiar *edge-vertex incidence matrix* or the *connection matrix*, commonly found in many circuit analysis text books [28]. It has E columns, one for each of the edges in the RDG, and V rows, one for each of the nodes in the RDG. The (i, j)th element of $\boldsymbol{C}$ is $+1$ if edge j terminates in node i, is -1 if edge j originates from node i and is zero otherwise (if edge j both originates and terminates at node i, then also c_{ij} is zero). For example, the RDG for the RIA for sorting has a connection matrix given by

$$\boldsymbol{C} = \begin{bmatrix} 0 & 0 & -1 & 1 \\ 0 & 0 & 1 & -1 \end{bmatrix}. \tag{23}$$

ii) $\boldsymbol{D}$ is the $(S \times E)$-dimensional *index displacement matrix*, in which the jth column is the vector weight on the jth edge in the RDG. For the sorting example

$$\boldsymbol{D} = \begin{bmatrix} 1 & -1 & 0 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}. \tag{24}$$

iii) $\boldsymbol{\gamma}$ is the vector obtained by stacking $\{\gamma_x\}$ in the appropriate order, consistent with the arrangement of the rows in the connection matrix.

Before proceeding to attempt to solve the set of constraints in (22), a third restriction that a schedule must meet must be taken into consideration. If $x(\boldsymbol{k}_1)$ and $x(\boldsymbol{k}_2)$ are computed by the same processor, then they must not be assigned to the same step in the schedule. This implies that

$$\gamma_x + \boldsymbol{\lambda}^T\boldsymbol{k}_1 \neq \gamma_x + \boldsymbol{\lambda}^T\boldsymbol{k}_2, \quad \text{for all } (\boldsymbol{k}_1 - \boldsymbol{k}_2) = \alpha\boldsymbol{u} \tag{25}$$

which simplifies to

$$\boldsymbol{\lambda}^T\boldsymbol{u} \neq 0 \tag{26}$$

[2]As noted earlier, a shift in a particular index can be used to overcome this difficulty in the sorting example. However, no such trick exists for the problem in Example 3.

an additional constraint that is dependent upon our choice of the iteration vector $\boldsymbol{u}$. Fortunately, though, from elementary linear programming considerations, it can be shown that if there exists a feasible solution to the set of constraints in (22), then there *always* exists a feasible solution that meets the additional constraint in (26) as well [9], [20] (see Appendix).

An affine schedule exists as defined above if and only if the set of constraints in (22) has a feasible solution. Indeed, for the sorting example,

$$\gamma^T = [0\ 1]$$
$$\lambda^T = [1\ 2] \tag{27}$$

is a valid solution that is consistent with the choice

$$\boldsymbol{u}^T = [0\ 1] \tag{28}$$

which results in the processor array shown in Fig. 8. Note the need for a nonzero vector γ^T. In reference to our earlier

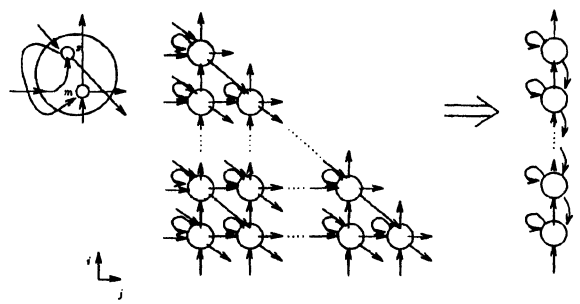


Fig. 8. The processor array for sorting obtained by projecting the dependence graph along the j direction.

remark concerning "change of variables," it can be easily verified that a nonzero γ^T specifies such a change of variables. Indeed, pick any integer vector $\boldsymbol{r}^T$ such that $\gamma^T\boldsymbol{r} = 1$. Then, consider the change of variables

$$x(\boldsymbol{k}) \rightarrow x(\boldsymbol{k} - \gamma_x\boldsymbol{r}) \tag{29}$$

for all x. Then, in the transformed domain, the RIA can be linearly scheduled simply because the corresponding transformation of the scheduling functions is given by

$$\gamma_x + \lambda^T\boldsymbol{k} \rightarrow \gamma_x - \gamma_x(\lambda^T\boldsymbol{r}) + \lambda^T\boldsymbol{k} = \lambda^T\boldsymbol{k}. \tag{30}$$

It can be shown that every *systolic array*, properly defined, executes an RIA for which a uniform affine schedule can be found as described above. Conversely, every such RIA can be implemented on a systolic array [9], [20] (the Appendix contains a partial exposition on this result).

What if, for the given RIA, there is no solution to the set of constraints in (22)? In this case, of course, one must further relax the restriction on the scheduling functions. It is easy to show that the RIA in Example 3 falls in this category, for which one must work harder in order to obtain a schedule. The interested reader can verify that

$$S_x(i, j, k) = nj + k$$
$$S_y(i, j, k) = i + nj + k$$
$$S_w(i, j, k) = -i + nj + k \tag{31}$$

constitute a valid set of scheduling functions for the RIA in this example; these scheduling functions are also affine, but for each x, the λ vector is different, i.e.,

$$S_x(\boldsymbol{k}) = \lambda_x^T\boldsymbol{k} + \gamma_x. \tag{32}$$

Due to space limitations, we will not discuss this case, but we must mention here that it has been proved that every RIA can be scheduled using affine scheduling functions, provided the λ-vectors are allowed to be different for different variables. Furthermore, the parameters $\{\lambda_x, \gamma_x\}$ must be allowed to depend upon the finite extent of the index space [9]. In addition, these affine scheduling functions can also be obtained by solving a series of integer linear programming problems defined on some specific subgraphs of the RDG of the RIA.

V. Concluding Remarks

With the brief overview presented in this paper, we wish to leave the reader with a list of the various issues that must be addressed in order to provide a complete theoretical framework for the design of processor arrays, from a given RIA description. While these issues have been fairly well resolved for the case of RIAs defined over a finite index space [9], many of them are as yet unsolved for the semi-infinite case, though some partial results can be found in [23].

Analysis Issues

a) Is the given RIA *computable*, that is, is there some variable in the RIA that is circularly defined? If there is such a variable, then the algorithm cannot be executed in the manner proposed, and hence must be discarded (this problem was completely solved in [23]).

b) Given a computable RIA, what is the minimum achievable execution time for completing the algorithm? This information is crucial for the designer as it lets him decide how many processors should be used for executing the algorithm. For instance, if the algorithm requires $O(N^3)$ computations and the minimum achievable execution time is $O(N)$, then one would wish to use $O(N^2)$ processors, each working on disjoint sets of $O(N)$ computations, to complete the algorithm in $O(N)$ time. On the other hand, if the minimum achievable execution time is $O(N^2)$, then a collection of $O(N)$ processors must be sought in order to achieve maximum utilization. (The iteration vector may now have to be generalized to an *iteration space*.)

c) How much storage should be provided for executing the algorithm? This would provide an indication of how much local memory should be associated with each processor in the array.

Implementational Issues

a) Given a computable RIA, how should one choose the processor space? Is this choice restricted by some properties of the RIA?

b) Once the processor space is chosen, how should one determine the schedule? Is it possible to choose the schedule so that every processor is active most of the time?

Synthesis Issues

a) Given a problem, is it possible to find an RIA for solving it? Is there a theoretical basis for determining such RIAs?

b) Presuming that there are a lot of RIAs for solving a given problem, is there some systematic method for weeding out the inefficient ones?

This paper has attempted to provide some insight into these issues, and to show the virtues of a formal analytical approach. While only an outline has been given here, a detailed presentation can be found in a forthcoming monograph [29]. However, as a partial indication of the formalization that is possible, the Appendix gives the formal description of systolic arrays and some deductions thereupon.

While RIAs themselves form a useful class of algorithms, some attempts have been made in [35], [36] to generalize this class. However, it can be shown that the so-called *affine recurrence equations*, defined therein, can be systematically transformed into an RIA format, and thus can be handled using the techniques outlined above [38].

Another question that has not been addressed in this paper is that of "optimality." First, there are many different criteria for optimality of the array that are not necessarily synergistic. Secondly, even if one were to fix upon some cost function for defining optimality, the best known techniques rely essentially upon exhaustive searches.

Appendix

This Appendix contains some formal definitions of the relevant concepts introduced in the paper and some deductions thereupon.

A *Regular Iterative Algorithm* is defined by the triple $\{I, X, F\}$ where

I is the *index space* which is the set of all lattice points enclosed within a specified region in S-dimensional Euclidean space,

X is the set of V variables that are defined at every point in the index space, where the variable x_j defined at the index point k will be denoted as $x_j(k)$ and takes on a unique value in any particular instance of algorithm, and

F is the set of *functional relations* among the variables, restricted to be such that if $x_i(k)$ is computed using $x_j(k - d_{ji})$, then

d_{ji} is a constant vector independent of k and the extent of the index space, and for every ℓ contained in the index space, $x_i(\ell)$ is computed using $x_j(\ell - d_{ji})$ (if $x_j(\ell - d_{ji})$ falls outside the index space, then this is an external input to the algorithm).

Note that the functional relations among the variables in F can involve conditional branches. However, in this case, the model essentially assumes that the dependence includes all the variables in every branch of the conditionals. While this assumption is a limitation in certain cases, it is not restrictive at all for a majority of useful algorithms.

Some related concepts can now be formalized as follows:

An *index vector* is any integer vector that represents a lattice point within the index space.

The *dependence graph* of an RIA is a directed graph in which the node set is defined by the ordered pair (x_j, k) where $x_j \in X$ and $k \in I$ and the edge set consists of all directed edges drawn from node $(x_j, k - d_{ji})$ to node (x_i, k) if and only if $x_i(k)$ is computed using $x_j(k - d_{ji})$ in the RIA.

The *Reduced Dependence Graph* of an RIA is a directed, vector edge-weighted graph $\mathcal{G} = \{V, E, D\}$ where

V is the set of V nodes in the graph, one for each of the indexed variables in the RIA,

E is the set of directed edges in the graph, such that a directed edge exists from node j to node i if and only if $x_i(k)$ is computed using $x_j(k - d_{ji})$ for some d_{ji},

D is the set of *index displacement vectors* defined on the edges in the graph so that the edge from j to i above has an index displacement vector of d_{ji}.

With these definitions in place, we now turn to the definition of a systolic array. Unfortunately, though, there is considerable confusion in the literature as to what exactly is a systolic array. Some authors, for instance, Leiserson, Rose, and Saxe [37], consider every synchronous circuit to be a systolic array, while others, e.g., Kung [30], define systolic arrays as a restricted class of processor arrays with certain attributable properties. The following formal definition is consistent with the four properties (modularity, spatial locality, temporal locality, and efficiency) qualitatively described in [30]. Based on this definition, one can then easily characterize the algorithm executed by a systolic array as a member of a proper subclass of Regular Iterative Algorithms.

Definition of a Systolic Array: A systolic array is characterized by the sets $\{P, \tau, X, D_p, F\}$ where

P is the *processor space* which is the set of all lattice points enclosed within a specified region in p-dimensional Euclidean space,

τ represents the beats of the *systolic clock*,

X is the set of V variables that is computed by every processor in the processor space and at every beat of the systolic clock during the execution of the array,

D_p is the set of *processor displacements* that defines the interconnecting links in the processor array so that

if d is a member of D_p, then there is an interconnecting link from the processor at location p to the processor at location $(p + d)$ irrespective of the particular value of p, and

if variable x computed at beat τ by the processor at location p is transferred across the link to the processor at location $(p + d)$, then this data transfer occurs regardless of the particular values of k and p, and

F is the set of functional dependences that relate the computation of a variable x at processor p during beat τ, as a function of the variables computed during the previous beat at the neighboring processors.

Once again, it must be noted that the functional dependences in F can involve conditional branches. Sometimes these conditional branches may be such that some of the data transfers involved in D may be unnecessary.

A detailed justification for this model can be found in [9], [20], [29]. For the present, we shall only be interested in characterizing the algorithm executed by a systolic array.

With these formalizations in place, any algorithm executed by a systolic array can be characterized as follows.

Theorem: A systolic array executes a Regular Iterative Algorithm that has a uniform affine schedule. Conversely, every Regular Iterative Algorithm with a uniform affine schedule can be implemented on a systolic array.

Proof: To show that a systolic array executes a Regular Iterative Algorithm, we define the index space to be

$$I = \left\{ k = \begin{bmatrix} p \\ \tau \end{bmatrix}, p \in P, \tau = \text{systolic beat} \right\}.$$

Next, let the variable x computed by the processor at location p at beat τ be denoted as $x(k)$. Then, by the definition of a systolic array, if $x(k)$ is computed using $y(\ell)$, then

$$k - \ell = \begin{bmatrix} d \\ 1 \end{bmatrix}$$

which is independent of k and the extent of the index space. Moreover, the systolic array operates according to the uniform affine schedule

$$\gamma = [0 \quad \lambda^T] = [0\ 0 \cdots 1].$$

To prove the converse statement, let $\{\gamma, \lambda\}$ constitute the parameters of a uniform affine schedule for the RIA. Then, if C be the connection matrix of the RDG of the RIA and D, its index displacement matrix, one must have

$$\gamma^T C + \lambda^T D \geq [1\ 1 \cdots 1].$$

Therefore, it must be possible to determine λ such that the greatest common divisor of its elements is 1. This implies (by the Bezoutian identity [31]) that there exists a vector u such that

$$\lambda^T u = 1.$$

Next, each indexed variable x can be redefined to be

$$\bar{x}(k) = x(k + \gamma_x u).$$

Then, the displacement matrix in the new domain can be written as

$$\bar{D} = D + u\lambda^T C.$$

Choose u to be the iteration vector and define the processor space according to

$$P = \{p : p = Pk\}$$

where P is defined as in (13). To complete the systolic array implementation, define

$$\tau = \lambda^T k$$

so that $\bar{x}(k)$ is computed by the processor at location Pk during the τth beat of the systolic clock.

The class of systolic algorithms, as characterized in the above theorem, is precisely the subclass of Regular Iterative Algorithms that have $O(N)$ I/O latency and constant storage requirements at each processor [29]. For instance, the RIA in Example 3 is not a systolic algorithm, since it does not meet the latency bound. In this case, one can show that the schedule given in (31) is "tight" and that the number of steps required to execute the RIA is at least nN regardless of the number of processors used. Thus the latency here has to be a polynomial of degree 2 in the size parameters n and N. Similarly, the RIA for Gaussian elimination with partial pivoting [22] is not systolic since it requires $O(N^2)$ time for its execution, no matter how many processors are used in its implementation.

Acknowledgment

The authors wish to thank H. V. Jagadish for his comments during the early course of this work.

References

[1] H. T. Kung and C. E. Leiserson, "Systolic arrays for VLSI," in *Sparse Matrix Proceedings*. Philadelphia, PA: Society of Industrial and Applied Mathematicians, 1978, pp. 245–282.

[2] C. A. Mead and L. A. Conway, *Introduction to VLSI Systems*. Reading, MA: Addison-Wesley, 1980.

[3] S. H. Unger, "A computer oriented towards spatial problems," *Proc. IRE*, vol. 46, pp. 1744–1750, Oct. 1958.

[4] ——, "Pattern recognition using two-dimensional bilateral iterative combinational switching circuits," *Proc. Polytechnic Institute of Brooklyn Symp. on Mathematical Theory of Automata*. Brooklyn, NY: Polytechnic Press, 1963.

[5] E. J. McCluskey, "Iterative combinational switching networks—General design consideration," *IRE Trans. Electro. Comput.*, vol. EC-9,. pp. 39–47, 1960.

[6] F. C. Hennie, *Iterative Arrays of Logical Circuits*. Cambridge, MA, and New York, NY: MIT Press and Wiley, 1961.

[7] U. Weiser and A. Davis, "A wavefront notational tool for VLSI Array design," in *CMU Conf. on VLSI on Systems and Computations*. Pittsburgh, PA: Comput. Sci. Press, Oct. 1981, pp. 226–234.

[8] S. Y. Kung, "VLSI array processors for signal processing," presented at the Conference of Advanced Research in Integrated Circuits, MIT, Cambridge, MA, 1981.

[9] S. K. Rao, "Regular iterative algorithms and their implementations on processor arrays," Ph.D. dissertation, Information Systems. Lab., Stanford University, Stanford, CA, Oct. 1985.

[10] P. R. Cappelo and K. Steiglitz, "Unifying VLSI array designs with linear transformations of space time," *Adv. Comput. Res.*, vol. 2, pp. 23–65, 1984.

[11] D. I. Moldovan, "On the analysis and synthesis of VLSI algorithms," *IEEE Trans. Comput.*, vol. C-31, pp. 1121–1126, Nov. 1982.

[12] P. Quinton, "The systmatic design of systolic arrays," INRIA Rep., Paris, 1983.

[13] D. I. Moldovan, "On the design of algorithms for VLSI systolic arrays," *Proc. IEEE*, vol. 71, pp. 113–120, Jan. 1983.

[14] M. C. Chen and C. A. Mead, "Concurrent algorithms as space-time recursion equations," in *Modern Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1984.

[15] M. S. Lam and J. Mostow, "A transformational model for VLSI systolic design," in *Proc. of the Int. Symp. on Hardware Description Languages and their Applications*, P. Uehara and M. Barbacci, Eds. Amsterdam, The Netherlands: North-Holland, 1983, pp. 65–77.

[16] G. J. Li and B. W. Wah, "The design of optimal systolic arrays," *IEEE Trans. Comput.*, vol. C-34, no. 1, pp. 66–77, Jan. 1985.

[17] L. Johnsson, S. Lennart, D. Weiser, D. Cohen, and A. Davis, "Toward a formal treatment of VLSI arrays," in *Proc. 2nd Caltech Conf. on VLSI*, Jan. 1981.

[18] W. L. Mirankler and A. Winkler, "Space-time representation of computational structures," *Computing*, vol. 32, pp. 93–114, 1984.

[19] J. A. B. Fortes and C. S. Raghavendra, "Gracefully degradeable processor arrays," *IEEE Trans. Comput.*, vol. C-34, no. 11, pp. 1033–1044, Nov. 1985.

[20] S. K. Rao, "What is a systolic algorithm?," *Proc. SPIE* (Highly Parallel Signal Processing Architectures), vol. 614, pp. 34–48, Jan. 1986.

[21] H. V. Jagadish, "Techniques for the design of parallel and pipelined VLSI systems for numerical computations," Ph.D. dissertation, Information Systems Laboratory, Stanford University, Stanford, CA, Dec. 1985.

[22] V. P. Roychowdury and T. Kailath, "Regular Processor Arrays for Matrix Algorithms with Pivoting," ISL Preprint, Stanford University, Stanford, CA, 1986.

[23] R. M. Karp, R. E. Miller, and S. Winograd, "The organization of computations for uniform recurrence equations," *J. ACM*, vol. 14, pp. 563–590, 1967.

[24] S. J. Celoni and J. L. Hennessy, "SAL—A single-assignment language for parallel algorithms," Tech. Rep., Computer Systems Laboratory, Stanford University, Stanford, CA, July 1981.

[25] J. D. Ullman, "NP-complete scheduling problems," *J. Comput. Syst. Sci.*, vol. 10, pp. 384–393, 1975.
[26] D. E. Knuth, *The Art of Computing Programming: Sorting and Searching*. Reading, MA: Addison-Wesley, 1973.
[27] W. M. Waite, "Path detection in multidimensional iterative arrays," *J. ACM*, vol. 14, pp. 300–310, 1967.
[28] W. H. Hayt and J. H. Kemmerly, *Engineering Circuit Analysis*. New-York, NY: McGraw-Hill, 1978.
[29] S. K. Rao, *The Design of Processor Arrays* (Prentice-Hall Systems Sciences Series), to be published.
[30] S. Y. Kung, "On supercomputing with systolic/wavefront array processors," *Proc. IEEE*, vol. 72, pp. 867–884, July 1984.
[31] C. C. MacDuffee, *The Theory of Matrices*. New York, NY: Chelsea Publishing, 1950.
[32] E. Deprettere and P. Dewilde, "Orthorgonal cascade realization of real multi-port digital filters," Tech. Report, Network Theory Section, Delft Univ. of Technol., Delft, The Netherlands, 1980.
[33] P. P. Vaidyanathan and S. K. Mitra, "A general theory and synthesis procedure for low sensitivity digital filter structures," ECE Rep., Dept. of Elec. and Comput. Eng., Univ. of California, Santa Barbara, Sept. 1982.
[34] A. Fettweis, "Digital filter structures related to classical filter networks," *Arch. Elek. Übertragung*, vol. 25, 1971.
[35] J-M. Delosme and I. C. F. Ipsen, "An illustration of a methodology for the construction of efficient systolic architectures in VLSI," in *Proc. 2nd Int. Symp. on VLSI Technology* (Taipei, Taiwan, 1985).
[36] ——, "Design methodology for systolic arrays," *Proc. SPIE.*, vol. 696, Nov. 1986.
[37] C. E. Leiserson, F. M. Rose, and J. B. Saxe, "Optimizing synchronous circuitry by retiming," in *Proc. 3rd CalTech Conf. on VLSI*, 1983.
[38] V. P. Roychowdhury and S. K. Rao, "Transformations of Regular Iterative Algorithms," ISL Preprint, Stanford University, Stanford, CA, 1987.

The Mapping of Linear Recurrence Equations on Regular Arrays

PATRICE QUINTON
IRISA, Campus de Beaulieu, 35042 Rennes-Cedex, France

VINCENT VAN DONGEN
PRLB, Av. van Becelaere, No. 2, Box 8, 1170 Brussels, Belgium

Received October 3, 1988, Revised May 1, 1989.

Abstract. The parallelization of many algorithms can be obtained using space-time transformations which are applied on nested do-loops or on recurrence equations. In this paper, we analyze systems of linear recurrence equations, a generalization of uniform recurrence equations. The first part of the paper describes a method for finding automatically whether such a system can be scheduled by an affine timing function, independent of the size parameter of the algorithm. In the second part, we describe a powerful method that makes it possible to transform linear recurrences into uniform recurrence equations. Both parts rely on results on integral convex polyhedra. Our results are illustrated on the Gauss elimination algorithm and on the Gauss-Jordan diagonalization algorithm.

1. Introduction

Designing efficient algorithms for parallel architectures is one of the main difficulties of the current research in computer science. As the architecture of super-computers evolves towards massive parallelism, it becomes necessary to design smart compilers that not only look for efficient vectorized programs, but also try to optimize the distribution of the algorithm among the processors. In order to obtain good performance, the distributed algorithm must minimize as much as possible the amount of communications between remote processors and balance the computational and the communication power of the processing units. One very promising methodology involves describing the algorithm using abstract specifications such as recurrence equations, as it was suggested by Karp, et al., [1] in 1967. The algorithm to be mapped is specified as a set of equations attached to integral points, and mapped using a regular scheduling and allocation scheme on the architecture. A similar approach was used later by Lamport [2] for the parallelization of do-loops, and became the basis of many studies on the synthesis of systolic arrays [3]–[11]. The main problems that were tackled were the scheduling of the computations, the mapping of the computations on regular architectures, partitioning schemes, and optimal organization of multistep algorithms.

This work was partially funded by the French Coordinated Research Program C^3 and by a Grant from the SOREP company

In this paper, we address the analysis and mapping of linear recurrence equations on parallel architectures. In Section 2, we recall informally the principles of recurrence mapping, on the example of the matrix multiplication. Section 3 is devoted to a formal definition and presentation of linear recurrences and algorithms. After defining a normal form of recurrence equations, we give a constructive necessary and sufficient condition for the existence of a linear timing function for such equations. This result extends the previously known results on uniform recurrence equations and makes it possible to relax the hypothesis that the target architecture is only locally connected. Our results are illustrated by the analysis of Gauss and Gauss-Jordan algorithms. Section 4 tackles the problem of transforming linear recurrence equations into uniform recurrence equations. Such a transformation is mandatory when the target architecture is locally connected, for example, a systolic array.

2. Principle of Recurrence Mapping

Consider the multiplication $C=AB$ of $N\times N$ matrices. Denote $a(i, j)$, $b(i, j)$, and $c(i, j)$ the elements of matrices A, B and C respectively. We can use the following system of recurrence equations to describe this algorithm:

$$1\le i\le N,\ 1\le j\le N,\ 1\le k\le N \rightarrow \qquad (1)$$
$$C(i, j, k) = C(i, j, k-1) + a(i, k) \times b(k, j)$$

$$1 \le i \le N,\ 1 \le j \le N,\ k = 0 \rightarrow C(i, j, k) = 0 \quad (2)$$

$$1 \le i \le N,\ 1 \le j \le N \rightarrow c(i, j) = C(i, j, N). \quad (3)$$

Each one of these equations represents a set of unique assignment statements attached to an integral point of the space. Equations (1) and (2) are attached to integral points of a cube of size N in $\mathbf{Z}^3$, and define an iterative calculation of the expression $\sum_{k=1}^{N} a(i, k)\, b(k, j)$, using an intermediate variable C. Equation (3) has indexes in $\mathbf{Z}^2$ and simply renames the results of the recurrence as c. Of course, many other recurrence forms would give the same result, and the choice which is made here, although reasonable, is arbitrary.

The mapping procedure, as it was described in [9] or [12], amounts to perform an index transformation, usually referred to as *space-time transformation,* that affects each statement to an instant of time and a processor location. Such a transformation must satisfy basically two conditions:

1. if a statement depends on the result of another statement, it should be executed after the result itself is available;
2. two statements allocated to the same processor should not be scheduled at the same instant of time.

If we restrict ourselves to the case when the index transformation is *linear,* it appears that the above conditions can be equivalently expressed as a linear programming problem, when one considers uniform recurrences. Linear transformations have proved to be very effective in order to design locally connected architectures such as systolic arrays. These are the main reasons why research in this domain has mainly focused on uniform recurrences and linear transformations.

Before discussing the limitations of uniform recurrences, let us recall informally the principles of the mapping procedure, in the case of uniform recurrence equations, on the matrix multiplication example. The following development is based on results that appeared in many papers including [9], [12].

First, we transform the equations in order to obtain uniform recurrences. Equation (1) is not uniform, as the indexes of variables a and b are not translations of (i, j, k). In this particular case, the index mapping is a *projection* and corresponds to the *broadcasting* of data a and b. Using a well-known technique called *pipelining,* we can replace equation (1) by:

$$1 \le i \le N,\ 1 \le j \le N,\ 1 \le k \le N \rightarrow \quad (4)$$
$$C(i, j, k) = C(i, j, k-1) + A(i, j, k) \times B(i, j, k)$$

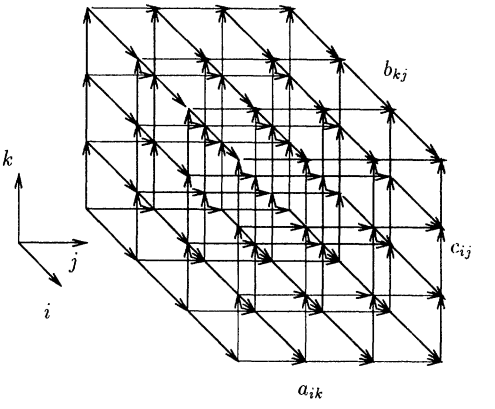
$$1 \le i \le N,\ 1 \le j \le N,\ 1 \le k \le N \rightarrow \quad (5)$$
$$A(i, j, k) = A(i, j-1, k)$$

$$1 \le i \le N,\ j = 0,\ 1 \le k \le N \rightarrow \quad (6)$$
$$A(i, j, k) = a(i, k)$$

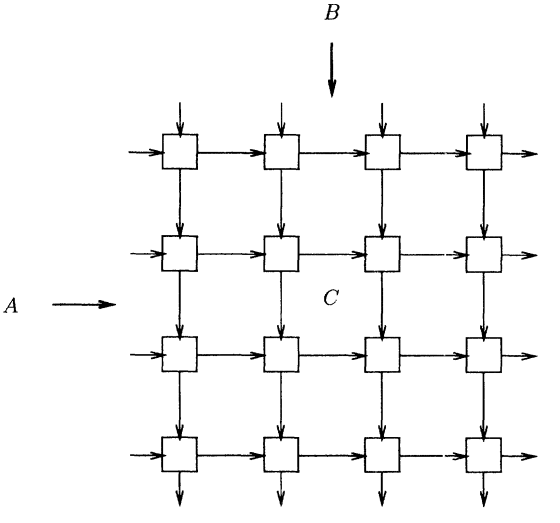
$$1 \le i \le N,\ 1 \le j \le N,\ 1 \le k \le N \rightarrow \quad (7)$$
$$B(i, j, k) = B(i-1, j, k)$$

$$i = 0,\ 1 \le j \le N,\ 1 \le k \le N \rightarrow \quad (8)$$
$$B(i, j, k) = b(k, j).$$

The detail of this transformation will be explained in Section 4.4. The new system is composed of equations (2) to (8). Equations (4), (5), and (7) are said to be uniform, since the index functions of all the variables



a)



b)

Fig. 1. The matrix muliplication dependence graph, and the systolic array obtained by a projection along the k axis.

are translations of (i, j, k). The other equations of the system are not uniform, but as far as the timing analysis is concerned, they can be ignored because they only serve defining inputs (equations (2), (6), (8)) or outputs of the system (3). The *dependence graph* that corresponds to this system is depicted in figure 1a for $N = 3$. The nodes of this graph are the points of the cube of $\mathbf{Z}^3$ of size 3, and edges represent dependences between the variables.

The next step of the mapping procedure is to find an affine schedule for the system, that is to say, a mapping

$$t: \mathbf{V}\times\mathbf{Z}^3 \rightarrow \mathbf{Z} \tag{9}$$

$$(V,(i, j, k)) \rightarrow \lambda_1 i+\lambda_2 j+\lambda_3 k+\alpha_v \tag{10}$$

where $\mathbf{V}$ denotes the set of variables of the system. The mapping t must be such that if computation of variable $V(i, j, k)$ depends on variable $V'(i', j', k')$, then $t(V,(i, j, k)) \geq t(V',(i', j', k')) + 1$, assuming that the execution of any statement takes exactly one step of time. However, as the equations are uniform, this condition can be replaced by a finite set of linear inequalities. Indeed, as $C(i, j, k)$ depends on $C(i, j, k - 1)$, we must have:

$$t(C,(i, j, k)) \geq t(C,(i, j, k - 1)) + 1$$

which gives

$$\lambda_3 \geq 1$$

Applying the same treatment to the other dependences, we obtain:

$\alpha_C - \alpha_A \geq 1$; $\alpha_C - \alpha_B \geq 1$	from equation (4)
$\lambda_2 \geq 1$	from equation (5)
$\lambda_1 \geq 1$	from equation (7).

A solution to this system of linear contraints is $\lambda_1 = \lambda_2 = \lambda_3 = 1$, $\alpha_A = \alpha_B = 0$, and $\alpha_C = 1$. Therefore, a valid schedule for the system is that $C(i, j, k)$ be computed at time $i + j + k + 1$, and $A(i, j, k)$ and $B(i, j, k)$ transmitted at time $i + j + k$.

The final step of the mapping procedure is to allocate the computations on a locally connected architecture. Basically, the method amounts to finding a linear mapping a from $\mathbf{Z}^n$ to $\mathbf{Z}^{n-1}$, called the *processor allocation function*, such that $a(z)$ is the number of the processor that executes the calculations attached to point z. The mapping a must be chosen in such a way that a processor has no more than one calculation to perform at a given instant. If a is chosen to be a *projection* of the space along a direction defined by a vector u, the only constraint on the choice of u is that u must not be parallel to the planes $i+j+k = T$ defined by the timing-function. In other words, $\lambda_1 u_1 + \lambda_2 u_2 + \lambda_3 u_3$ must be non-zero.

In the case of the matrix multiplication, the projection of the equations along the direction (0, 0, 1) produces the systolic architecture shown in figure 1b.

The interconnection pattern of the architecture is readily obtained by looking at the way the edges of the dependence graph are projected. Here, the elements of A and B enter the matrix, in a skewed fashion, respectively from the left and from the bottom of the array, and the partial results remain in the cells. Other architectures can be derived. For example, by projecting the dependence graph along (1, 1, 1), we obtain an array, similar to the well-known systolic array first proposed by Kung and Leiserson ([13]), where both data and partial results are moving. Moreover, the control of the cells of the systolic array is simple and can be derived automatically. The only difficulty is that a given cell may have to perform different calculations at different instant of time, as the equations are defined on subdomains. However, in his thesis, Rao ([11]) shows how one can replace conditions involving linear combinations of indexes by new boolean variables which are tested by the cells in order to decide which equation is to be applied. We will not develop this step further (see [9], [12], [14] for example).

Uniform recurrence equations present several limitations which call for an extension, at least to linear recurrences, as we shall see in the next section:

1. it is recognized that specifying an algorithm using uniform recurrences is often a difficult and tedious task. This task becomes much simpler if linear recurrences are allowed;
2. using uniform recurrences is somehow mandatory when one wants to implement an algorithm on a locally connected architecture, such as a systolic array. It is not possible to implement non-local communications in one instant of time. However, a lot of parallel architectures provide communication facilities via richer topologies: for example, hypercubes, shuffle-exchange networks, broadcast by bus, etc. Therefore, it is tempting to consider more complex recurrences, such as linear ones (other index functions would also be interesting).
3. in some cases, it is very difficult to replace linear dependences by uniform ones, especially when the calculations done by the algorithm depend recursively on its results. For example, this is the case in the recursive convolution or in the dynamic programming algorithm. Most of the recent work on the subject tries to overcome the problem, more or less formally. Therefore, it is interesting to try to solve this difficulty in its full generality.

In the following section, we study the scheduling of linear recurrences. For the sake of illustrating the paper, we will use, when necessary, results on allocation functions that were presented informally in this section.

3. Linear Recurrence Equations and Algorithms

3.1. Statement of the Problem

Definition 1. A *linear parameterized recurrence equation* is an equation of the form

$$z \in D_p \rightarrow U(z) = f[V(I(z)), \ldots] \qquad (11)$$

where:

1. z is a point of $\mathbf{Z}^n$, where $\mathbf{Z}$ denotes the set of integers,
2. $p = (p_1, p_2, \ldots, p_m)$ is a point of $\mathbf{Z}^m$ named *the size parameter* of the equation. We assume that p belongs to a convex polyhedron $\mathbf{P} \subset \mathbf{Z}^m$ (in most cases, $\mathbf{P} = \mathbf{N}^m$, where $\mathbf{N}$ denotes the set of non-negative integers).
3. D_p is the set of integral points belonging to a convex polyhedron of $\mathbf{Z}^n$, called the *domain* of the equation[1]. We assume that D_p is bounded[2] and is defined by a finite set of linear inequalities involving z and p.
4. I is an affine mapping from $\mathbf{Z}^n$ to $\mathbf{Z}^l$ called *index mapping; I* has the form

   $$I(z) = A.z + B.p + C$$

   where the constants A, B and C are integral matrices, $A \in \mathbf{Z}^l \times \mathbf{Z}^n$, $B \in \mathbf{Z}^l \times \mathbf{Z}^m$, and $C \in \mathbf{Z}^l$.
5. U and V are *variable names* belonging to a finite set **V**. Each variable is indexed with an integral index, whose dimension (called the *index dimension* of the variable in the following) is constant for a given variable. The variable $U(z)$ is called the *result* of the equation and $V(I(z))$ is an *argument.*
6. f is a single-valued function that depends *strictly* on its arguments; we assume that the function f has complexity $O(1)$.
7. the '. . .' means that there can be other arguments of the same form as $V(I(z))$.
8. the domains of two equations having the same variable as result are disjoints. *This* hypothesis ensures that a variable is not defined twice.

For a given p, equation (11) represents a finite set of *equation instances,* each one of which is associated with a particular point z of D_p.

A *system of linear recurrence equations* is a finite set of equations such as (11), having the same parameter set. Note that all equations need not be indexed in the same subspace, i.e., n is not necessarily the same for all equations.

We call *variable instance of the system* any term $U(t)$ that appears in an equation instance. The *index domain* of a variable is the set of indexes of the instances of the variable. Note that the index domain of a variable is not necessarily a convex polyhedron, but is a finite union of convex polyhedra. A variable instance that appears only in the left-hand side of an equation is called an *output.* Similarly, a variable instance that appears only in the right-hand side of an equation is called an input. Variable instances that are neither outputs nor inputs are called intermediate data. Variables whose instances are all inputs (respectively, outputs or intermediate data) are called input variables (respectively, output variables or intermediate variables).

A particular case of linear recurrence equations is when the index mapping reduces simply to a translation. The equation is then said to be *uniform.* The importance of uniform recurrence equations for expressing parallel computations was first noticed by Karp, *et al.* [1]. Our definition of uniform equations is, however, less restrictive than the definition in [1] which makes the assumption that all the equations of a system have the *same* domain.

3.2. Normal Form of a System of Equations

The purpose of this subsection is to show that one can replace a system of linear recurrence equations by a new equivalent system, which is more convenient for the analysis of the dependences. The goal is to separate clearly the inputs, the outputs, and the intermediate data, since only the dependences between intermediate data have to be considered for studying the schedule of the equations. Moreover, it is necessary that all equations be attached to integral points of the same space.

We say that the argument of the equation is fully indexed, if its index dimension is the same as the index dimension of the result of the equation. An equation is fully indexed if all its arguments are fully indexed. For example,

$$1 \leq i \leq n,\ 1 \leq j \leq n \rightarrow U(i, j) = V(i, j - 1)$$

is fully indexed, but

$$1 \leq i \leq n,\ 1 \leq j \leq n \rightarrow U(i, j) = V(i)$$

is not, because $V(i)$ is not fully indexed.

An equation is an input equation if f is the identity function, and the only argument of the equation is an

input variable. Similarly, an equation is an output equation it f is the identity function and U is an output variable. Finally, an equation is a computation equation if its result and arguments are all intermediate variables.

Definition 2. A system of linear recurrence equations is said to be in normal form if:

1. all the variables are either input, output, or intermediate variables,
2. all equations are either input, output, or computation equations,
3. all the computation equations are fully indexed and have the same index dimension.

The following theorem holds:

THEOREM 1. For any system of linear reucrrence equations, there exists an equivalent system which is in normal form.

A formal proof of the theorem can be found in [15]. Basically, it involves successively applying three transformations to the initial system of equations. Rather than describing formally these transformations, we illustrate them on examples.

Example 1. The first transformation, called *variable normalization*, modifies the system in such a way that only input, output, or intermediate variables remain. To illustrate this, consider the equation:

$$1 \leq i \leq N \rightarrow A(i) = A(i - 1).$$

The variable A has one input instance $A(0)$, one output instance $A(N)$, and intermediate instances $A(i)$, when $1 \leq i \leq N - 1$. Therefore, A is neither an input, an output, or an intermediate variable. We can rewrite this equation in the following way:

$$2 \leq i \leq N - 1 \rightarrow A_1(i) = A_1(i - 1)$$

$$i = 1 \rightarrow A_1(i) = A_2(0)$$

$$i = N \rightarrow A_3(i) = A_1(i - 1).$$

The domain of A is partitioned into three new domains, each one of which corresponds respectively to the intermediate, input, and output instances of A, renamed respectively A_1, A_2, and A_3.

Example 2. The purpose of the second transformation, called *input and output separation*, is to make input (respectively output) variables appear only in input (respectively output) equations. Consider, for example, the system

$$1 \leq i \leq N \rightarrow A(i) = A(i - 1) + B(3i) \quad (12)$$

$$i = 0 \rightarrow A(i) = x(i)$$

$$i = N \rightarrow C'(i) = A(i).$$

In this system, the input variable B appears in equation (12) which is not an input equation. However, the property holds when equation (12) is replaced by

$$1 \leq i \leq N \rightarrow A(i) = A(i - 1) + B'(i)$$

$$1 \leq i \leq N \rightarrow B'(i) = B(3i).$$

Example 3. Finally, the last transformation, called *full indexing*, aims at obtaining a fully indexed system. Consider the following system of equations, which is a somewhat simplified version of the dynamic programming algorithm:

$$1 \leq i < j \leq N,\ i < k < j \rightarrow \\ C(i, j, k) = f[C(i, j, k + 1), c(i, k)] \quad (13)$$

$$1 \leq i,\ i + 1 < j \leq N,\ k = j \rightarrow C(i, j, k) = w(i, j) \quad (14)$$

$$1 \leq i,\ i + 1 < j < N \rightarrow c(i, j) = C(i, j, i + 1) \quad (15)$$

$$1 \leq i,\ j = i + 1,\ j < N \rightarrow c(i, j) = w(i, j). \quad (16)$$

A straightforward method to obtain fully indexed computation equations is to change the index dimension of c, by adding a 0 as a third component. We then obtain the new system:

$$1 \leq i < j \leq N,\ i < k < j \rightarrow \\ C(i, j, k) = f[C(i, j, k + 1), c(i, k, 0)]$$

$$1 \leq i,\ i + 1 < j \leq N,\ k = j \rightarrow C(i, j, k) = w(i, j)$$

$$1 \leq i,\ i + 1 < j < N,\ k = 0 \rightarrow c(i, j, k) = C(i, j, i + 1)$$

$$1 \leq i,\ j < N,\ j = i + 1,\ k = 0 \rightarrow c(i, j, k) = w(i, j).$$

However this transformation is far from being optimal, in the sense that it amounts to place variables $c(i, j)$ arbitrarily in the index space, which may have consequences on the schedule of the equations.

In this particular example, a much better method is to substitute $c(i, k)$ in (13) by its definition. We obtain the new system:

$$1 \leq i < j \leq N,\ i + 1 < k < j \rightarrow \\ C(i, j, k) = f[C(i, j, k + 1), C(i, k, k + 1)] \quad (17)$$

$$1 \leq i < j \leq N,\ i + 1 = k < j \rightarrow \\ C(i, j, k) = f[C(i, j, k + 1),\ w(i, k)] \quad (18)$$

$$1 \leq i,\ i + 1 < j \leq N,\ k = j \rightarrow C(i, j, k) = w(i, j) \quad (19)$$

$$1 \leq i,\ j = i + 1,\ j < N \rightarrow c(i, j) = C(i, j, i + 1) \quad (20)$$

$$1 \leq i, j = 1, j < N \rightarrow c(i, j) = w(i, j) \quad (21)$$

where (17) is fully indexed.

The substitution method cannot always be used, in particular when the substituted variable depends on itself. In this case, the substitution creates a number of equations that depend on the parameter p. The problem of finding an index transformation which merges optimally several index spaces is therefore still open.

In the remaining of this paper, we will consider normal form systems. We define the *domain* $\mathfrak{D}$ of a (normal form) system of equations as the convex hull of the union of the domains of its computation equations (input and output equation domains excluded).

3.3. Dependence Vectors

Assume that $V(I(z))$ is an intermediate variable instance. Then, $V(I(z))$ is defined by another equation, attached to point $I(z)$. In a normal form system, the intermediate variable instances are fully indexed, and both z and $I(z)$ belong to $\mathbf{Z}^n$. Hence the vector $z - I(z)$ is defined and is called *dependence vector.* Dependences with results or data are not considered, as they do not correspond to effective computations. Given an argument $V(I(z))$ of an equation, we denote by $\Theta_{V(I(z))}$ (or simply Θ when there is no ambiguity) the dependence vector $z - I(z)$. More formally, $\Theta_{V(I(z))}$ is a function from $\mathbf{Z}^{n+m}$ to $\mathbf{Z}^n$ which maps a pair (z, p) to $z - I(z)$. In the following, we shall denote by *Range* $(\Theta_{V(I(z))})$ the range of $\Theta_{V(I(z))}$ when $p \in \mathbf{P}$ and $z \in D_p$.

The following proposition is a key result of our study:

Proposition 3.1. *Range* $(\Theta_{V(I(z))})$ is a convex polyhedron of $\mathbf{Z}^n$.

Proof. As I is linear in z and p, the dependence vector is an affine mapping:

$$J: \mathbf{Z}^{n+m} \rightarrow \mathbf{Z}^n$$

$$(z, p) \rightarrow z - I(z) = (\mathfrak{I} - A)z - Bp - C$$

where $\mathfrak{I}$ is the identity matrix. Clearly, J is affine. Moreover, for all p, D_p is a convex polyhedron, which depends linearly on p, and p itself belongs to a convex polyhedron of $\mathbf{Z}^m$. Therefore, the set $\{(z, p) | p \in \mathbf{P}, z \in D_p\}$ is a convex polyhedron of $\mathbf{Z}^{n+m}$. It results that *Range* $(\Theta_{V(i(z))})$, being the image of a convex polyhedron by an affine mapping, is also a convex polyhedron of Z^n.

As a consequence of proposition 3.1, any dependence vector $\Theta_{V(I(z))}$ can be expressed as the sum of a convex combination of the vertices of the convex set *Range* $(\Theta_{V(I(z))})$, of a positive combination of its extermal rays, and of a linear combination of its lines.[3]

3.4. Computability and Scheduling

The scheduling problem is to find a function that associates each variable instance $U(z)$ with a given instant of time t, in such a way that the arguments needed for the calculation of $U(z)$ are already calculated at time t. If such a mapping exists, the system is said to be *explicit* or *computable.* This is not always the case, as shown by the following equations:

$$A(i) = 1 \leq i \leq n \rightarrow A(i - 1) + B(i - 1)$$

$$B(i) = 1 \leq i \leq n \rightarrow A(i + 1).$$

Here, $A(i)$ depends on itself.

The property of computability has been investigated by Karp, *et al.* [1], Rao [11], Yaacoby and Capello [16], and Delosme and Ipsen [5]. Karp, *et al.*, have shown that this property can be checked, when all the equations are uniform, have the same (possibly infinite) domain, and use only *strict* functions. These assumptions exclude the case when equations are defined on subdomains, as described here. In his thesis, Rao [11] gives a procedure to check the computability of recurrence equations defined on different domains, but his procedure is based on the fact that all equations are extended to the same domain.

Depending on the assumptions that are made, different results which are summarized here can be obtained:

1. if the domains of the equations are bounded and not parameterized, it is obvious that this property can be checked by testing whether the dependence graph is acyclic or not.
2. if the domains of the equations are not bounded, and all equations are defined on the same domain, and the functions f are strict, then the computability can be checked (result of Karp, *et al.*).
3. concerning equations defined on unbounded domains, when the domains are not the same for all equations[4], it has been shown by Joinnault [17] that the computability of a system of uniform recurrence equation is undecidable. The proof amounts to show that any Turing machine can be encoded as a uniform recurrence system of equations and to show that the computability of uniform recurrence equations reduces to the termination of a Turing machine.
4. an interesting question is to find out whether a *parameterized* system of recurrence equations is

computable. In other words, when the domains of the equations depend on an integral size parameter, is it possible to check the computablity of all instances of the system? Recently, Saouter and Quinton ([18]) have shown that this property is also undecidable.

The last result shows that it is hopeless in general to find out if there exists an ordering of the calculations compatible with the dependences. However, as we shall see in the next section, results can be obtained in the special case of a *linear* ordering, which is of practical interest.

3.5. Linear Timing Functions

A particular case of ordering on the calculations is called an *affine timing function* [9], [10]. If such a function can be found, of course, the system is computable. As we shall see, we can determine automatically whether there exists such a timing function, all at once for a parameterized system of equations.

Consider a variable instance $U(z)$, and define for each variable a function $t(U, z)$ from $\mathbf{V} \times \mathbf{Z}^n$ to $\mathbf{Z}$ of the form:

$$t(U, z) = \lambda_1 z_1 + \ldots + \lambda_n z_n + \alpha_U$$

where $\lambda_1, \ldots, \lambda_n$ are integers independent of U. Denote $\lambda^t z = \lambda_1 z_1 + \ldots + \lambda_n z_n$.

Assume that the evaluation of each function of the system takes at least one unit of time. The following result gives a conservative means for obtaining the function t:

THEOREM 2. The numbers $\lambda_1, \ldots, \lambda_n, \alpha_U, U \in \mathbf{V}$ define a timing function for all p if:

(i) for all vertex σ of the dependence sets *Range* $(\Theta_{V(I(z))})$, $\lambda^t\sigma + \alpha_U - \alpha_V > 0$,

(ii) for all extremal ray ρ of the dependence sets *Range* $(\Theta_{V(I(z))})$, $\lambda^t\rho \geq 0$

(iii) for all line ν of the dependence sets *Range*$(\Theta_{V(I(z))})$, $\lambda_t\nu = 0$.

Proof. $\boxed{\Rightarrow}$ We have to prove that for all p and all $z \in D_p$, $t(U, z) - t(V, I(z)) \geq 1$, i.e.

$$\lambda^t(z - I(z)) + \alpha_U - \alpha_V \geq 1$$

As $\Theta_{V(I(z))} = z - I(z)$ belongs to *Range* $(\Theta_{V(I(z))})$, it can be decomposed using the vertices, rays, and lines of *Range*$\Theta_{V(I(z))}$) as

$$\Theta_{(V(I(z))} = \Sigma\alpha\sigma + \Sigma\beta\rho + \Sigma\delta\nu.$$

Using the hypotheses, we obtain:

$$t(U, z) - t(V, I(z)) \geq \lambda^t\Sigma\alpha\sigma + \alpha_U - \alpha_V$$

and as $\Sigma\alpha = 1$, we have

$$t(U, z) - t(V, I(z)) \geq \Sigma\alpha[\lambda^t\sigma + \alpha_U - \alpha_V] \geq 1.$$

$\boxed{\Leftarrow}$ Conversely, assume that $\lambda_1, \ldots, \lambda_n, \alpha_U, U \in \mathbf{V}$ define a timing function for all p. Consider a vertex σ of *Range* $(\Theta_{V(I(z))})$. There exist p and $z \in D_p$ and an equation

$$z \in D_p \rightarrow U(z) = f[V(I(z)), \ldots]$$

such that $z - I(z) = \sigma$. Therefore we must have:

$$\lambda^t\sigma + \alpha_U - \alpha_V \geq 1$$

which proves (i).

Let ρ be the ray of *Range* $(\Theta_{V(I(z))})$. Assuming that (ii) is not true, it is simple to show that there exists an equation such that $t(U, z) < t(V, I(z))$. A similar reasoning applies for (iii).

Note that Theorem (2) is a generalization of a result given by Rajopadhye and Fujimoto [19] to the case of parameterized recurrence equations. A similar condition was also presented, independent of ours, by Irigoin and Triolet [16], in the framework of Nested Loops analysis.

3.6. Gaussian Elimination and Gauss-Jordan Diagonalization

In the following, we shall use the Gaussian elimination and the Gauss-Jordan diagonalization as examples. We first present the Gauss-Jordan algorithm, from which the Gaussian elimination can be deduced immediately by removing one equation.

Let A be a $N \times N$ matrix and b be a $N \times 1$ vector. The problem is to solve the linear system $Ax = b$, by the Gauss-Jordan elimination algorithm. For the sake of commodity, we assume that A is an $N \times (N + 1)$ matrix whose last column is b. Elements of A are denoted $a(i, j)$. The basic step of the algorithm is as follows: at step k, element $a(k, k)$ is taken as the pivot, and is used to zero out elements $a(i, k)$, $1 \leq i \leq N$, $i \neq k$. At the end of the algorithm, i.e., when $k = N$, matrix A is the identity matrix and b is the solution x of the system. Let $A(i, j, k)$ denote the value of element (i, j) of matrix A at step k. The algorithm can be precisely specified by the following equations:

Input Equations:

$$k = 0,\ 1 \leq i \leq N,\ 1 \leq j \leq N \rightarrow A(i, j, k) = a(i, j) \tag{22}$$

$$k = 0,\ 1 \leq i \leq N,\ j = N + 1 \rightarrow A(i, j, k) = b(i) \tag{23}$$

Computation Equations:

$$1 \leq k \leq N,\ k \leq j \leq N+1,\ i = k \rightarrow \quad (24)$$
$$A(i, j, k) = A(i, j, k-1)/A(k, k, k-1)$$

$$1 \leq k \leq N,\ k \leq j \leq N+1,\ 1 \leq i \leq k \rightarrow \quad (25)$$
$$A(i, j, k) = A(i, j, k-1) - A(i, k, k-1)/A(k, k, k-1) \times A(k, j, k-1)$$

$$1 \leq k \leq N,\ k \leq j \leq N+1,\ k > i \leq N \rightarrow \quad (26)$$
$$A(i, j, k) = A(i, j, k-1) - A(i, k, k-1)/A(k, k, k-1) \times A(k, j, k-1)$$

Output Equations:

$$1 \leq i \leq N \rightarrow x(i) = A(i, N+1, N) \quad (27)$$

It can be readily seen that the system is in normal form. The Gauss-Jordan domain is shown in figure 2 and is:

$$\mathfrak{D} = \{(i, j, k) | 1 \leq i \leq N,\ k \leq j \leq N+1,\ 1 \leq k \leq N\}.$$

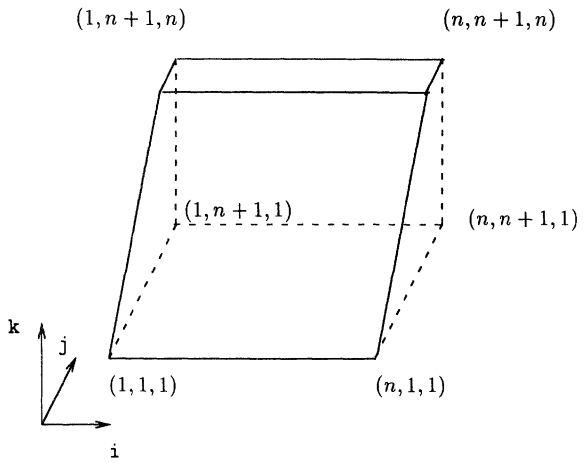


Fig. 2. Domain of the Gauss-Jordan elimination algorithm.

Table 1. Dependences in Jordan-Gauss elimination.

Equation	From	To	Vector	Note
(24)	$A(i, j, k)$	$A(i, j, k-1)$	(0, 0, 1)	
	$A(i, j, k)$	$A(k, k, k-1)$	$(0, j-k, 1)$	$i = k,\ j \geq k$
(25)	$A(i, j, k)$	$A(i, j, k-1)$	(0, 0, 1)	
	$A(i, j, k)$	$A(k, k, k-1)$	$(0, j-k, 1)$	$j \geq k$
	$A(k, j, k)$	$A(i, j, k-1)$	$(i-k, j-k, 1)$	$i < k$
	$A(k, j, k)$	$A(k, k, k-1)$	$(i-k, 0, 1)$	$i < k$
(26)	$A(i, j, k)$	$A(i, j, k-1)$	(0, 0, 1)	
	$A(i, k, k)$	$A(k, k, k-1)$	$(0, j-k, 1)$	$j \geq k$
	$A(k, k, k)$	$A(i, j, k-1)$	$(i-k, j-k, 1)$	$i > k$
	$A(k, j, k)$	$A(k, k, k-1)$	$(i-k, 0, 1)$	$i > k$

The Gaussian elimination differs from the Gauss-Jordan diagonalization in that elements above the pivot are not zeroed. Therefore, equation (25) has to be removed. At the end of the algorithm, matrix A is upper triangular, and it remains to perform a back substitution in order to solve the system.

Consider the Gauss-Jordan algorithm. Table 1 summarizes the dependences of the algorithm. Let us apply Theorem 2. Consider for example the dependence vector $(0, j-k, 1)$ from equation (24). When N ranges over $\mathbf{N}$, this vector belongs to the half-line Θ whose origin is (0, 0, 1) and direction is (0, 1, 0). In other words, Θ is a convex polyhedron with vertex (0, 0, 1) and (extremal) ray (0, 1, 0). Therefore, we must have:

$$\lambda_3 \geq 1 \text{ and } \lambda_1 \geq 0.$$

Applying the same analysis to the other dependence vectors gives the final set of constraints:

$$\lambda_1 \geq 0,\ \lambda_1 \leq 0,\ \lambda_2 \geq 0,\ \lambda_3 \geq 1.$$

Therefore, a possible timing function is $t(A, (i, j, k)) = k$. Notice that $t(A, (i, j, k)) = j + k$ is also valid, but the coefficient of i has to be 0. This is because the domain for the possible values of λ contains the line $j = k = 0$.

On the other hand, if we analyze the Gaussian elimination algorithm, the constraints are

$$\lambda_1 \geq 0,\ \lambda_2 \geq 0,\ \lambda_3 \geq 1$$

and $t(A, (i, j, k)) = i + j + k$ is now a valid schedule, as i is not constrained to be 0. This comes directly from the fact that equation (25) has been removed and, therefore, the constraint $\lambda_1 \leq 0$ is not necessary.

4. Uniformization

Unless it is uniform, a system of recurrence equations cannot be mapped directly on a systolic array. Indeed, as the dependence vectors are not constants, processors may have to communicate with an arbitrary large number of other processors, and this is not possible in a systolic array, where the processors are only connected to their neighborhood.

This Section is concerned with *uniformization*, that is to say, the transformation of linear recurrences into uniform ones. The uniformization problem has not yet received a full answer. It is tackled by Fortes and Moldovan [20], Wong and Delosme [21], Kuhn [22], Gachet, *et al.* [23], Van Dongen and Quinton [24], Joinnault [17], Rajopadhye [19], and others (see section 5). We begin by explaining informally the principle of our method (subsection 4.1). Then we precisely describe the uniformization method (subsection 4.2). In subsection 4.3, we sketch how the method can be applied in the case of the Gauss-Jordan algorithm.

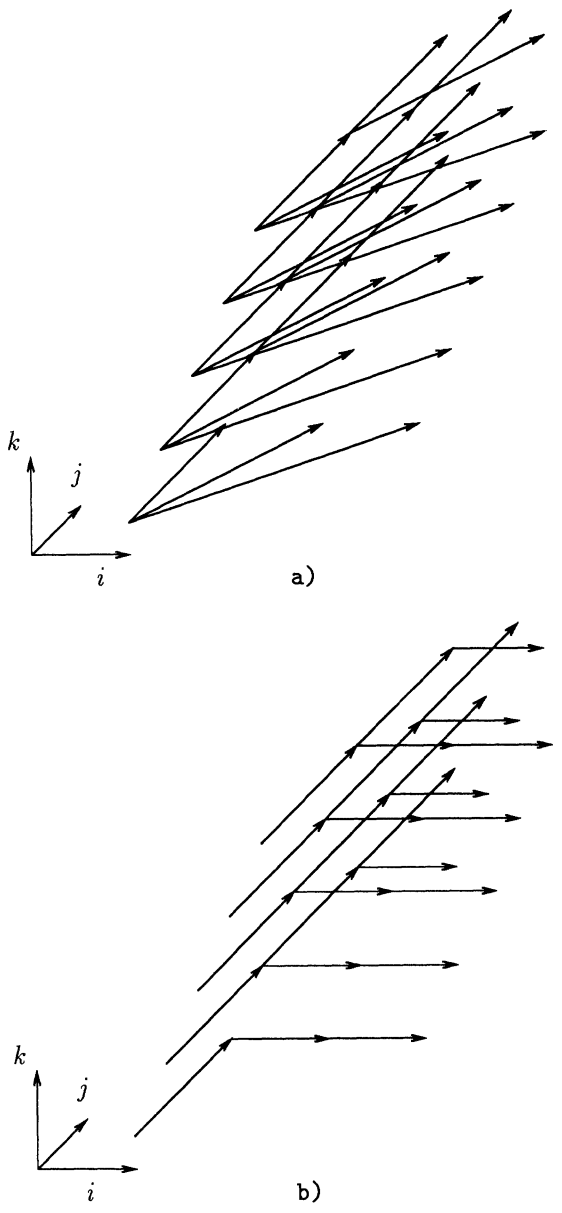


Fig. 3. The dependence $(i - k, D, 1)$ before and after uniformization.

4.1. Informal Description of the Uniformization Method

Let us briefly illustrate the uniformization process with the Gaussian elimination algorithm. Consider the non-uniform dependence $(i - k, 0, 1)$ between $A(i, j, k)$ and $A(k, j, k - 1)$ in equation (26) (figure 3). As explained in section 3.6, the associated domain *Range* $(\Theta_{V(I(z))})$ is infinite; it has one vertex (1, 0, 1) and a ray (1, 0, 0). To solve the uniformization problem, we express the dependence vector as a non-negative integral linear combination of these vectors, called *uniformization vectors*. A solution is:

$$(i - k, 0, 1) = (i - k - 1).(1, 0, 0) + (1, 0, 1). \quad (28)$$

The uniformization method involves rewriting the variable instance $A(k, j, k - 1)$ using a new variable A', which is defined by a (uniform) equation of the form $A'(i, j, k) = A'(i - 1, j, k)$, where the dependence vector is (1, 0, 0). The same transformation is repeated for all vectors of the decomposition, until the non-uniformity is removed. In our example, this transformation gives the uniform dependence graph of figure 3b.

The main difficulty of the method is to choose the uniformization vectors in such a way that the resulting uniform system of equations has a linear timing function. As we shall see, the choice of the uniformization vectors depends on several factors, among which the dimension of the domain of the initial equation, the dimension of its image by the index mapping I, and the value of the vectors of the decomposition.

4.2. Automatic Synthesis of the Uniformization Vectors

Given a set of linear recurrences in its normal form, we only consider here the uniformization of dependences resulting from the computation equations. The case of input equations will be solved in Section 4.4.

The organization of this section is the following. In subsection 4.2.1, we consider the case where pipelining vectors are needed, that is to say, when the null space of I and the linear space generated by the domain D of the equation where I appears to have a non-empty intersection. Then, in subsection 4.2.2, we consider the case where no pipelining is needed. Finally, we summarize the method in subsection 4.2.3.

4.2.1. Pipelining Vectors Consider an equation:

$$z \in D \rightarrow U(z) = f(\ldots, V(I(z)), \ldots). \quad (29)$$

Definition 3. Given a convex polyhedron D, we note *Vect*(D) the linear space parallel to D, that is to say the direction of the affine hull of D[5]. We say that a vector A is *parallel* to D if $A \in$ *Vect*(D). We note *dim*(D) the dimension of *Vect*(D). Given an affine mapping I, we note *Null*(I) the null space of the linear part of I.

In the remainder of this section, we will assume that the system of linear recurrence equations where equation (29) appears has a timing function and, moreover, that the ranges of all dependence vectors do not contain

the vector 0 and are enclosed in a pointed cone, named Θ^*. This excludes the case when the range of one of the dependence vectors contains a line.

Definition 4. A vector $\phi \neq 0$ is said to be a *pipelining vector* if $\phi \in Null(I) \cap Vect(D)$.

Proposition 4.1. If ϕ is a pipelining vector, then the following system is equivalent to equation (29):

$$z \in D \rightarrow U(z) = f(\ldots, V'(z), \ldots) \quad (30)$$

$$z \in D_1 \rightarrow V'(z) = V'(z - \phi) \quad (31)$$

$$z \in D_2 \rightarrow V'(z) = V(I(z)) \quad (32)$$

where V' is a new variable and

$$D_1 = \{z \in D | z - \phi \in D\}$$

$$D_2 = D \backslash D_1.$$

Moreover, the domain D_2 can be partitioned into a finite number of convex polyhedra of dimension $dim(D) - 1$, and for all point $z \in D_2$, $z - I(z)$ belongs to the dependence set $Range(\Theta_{V(I(z))})$ of equation (29).

Proof. For all $z \in D$, by repeated use of (31), there exists $k \in \mathbf{N}$ such that:

$$U(z) = f(\ldots, V'(z - k\phi), \ldots)$$

and $z - k\phi \in D_2$. Then, using (32),

$$U(z) = f(\ldots, V(I(z - k\phi)), \ldots).$$

However, as ϕ is a pipelining vector, $I(z - k\phi) = I(z)$. Therefore,

$$U(z) = f(\ldots, V(I(z)), \ldots)$$

which proves the equivalence between (29) and the above system of equations. Since $D_2 = \{z \in D | z - \phi \notin D\}$, it can be partitioned into a finite number (independent on p) of convex domains of dimension $dim(D) - 1$. Finally, as $D_2 \subset D$, the set $\{\Theta_{V(I(z))} | z \in D_2\}$ is included in $\{\Theta_{V(I(z))} | z \in D\}$.

Example 4. In the case of the Gaussian elimination, equation (28) gives an integral decomposition of the dependence vector $(i - k, 0, 1)$ on the vectors (1, 0, 0) and (1, 0, 1). In fact, vector (1, 0, 0) belongs both to the null space of I and to $Vect(D)$ and is therefore a pipeline vector. The pipelining transformation, when applied to equation (26), yields:

$$1 \leq k \leq N,\ k \leq j \leq N + 1,\ k < i \leq N \rightarrow \quad (33)$$
$$A(i, j, k) = A(i, j, k - 1) - A(i, k, k - 1) /$$
$$A'(i, j, k) \times A(k, j, k - 1)$$

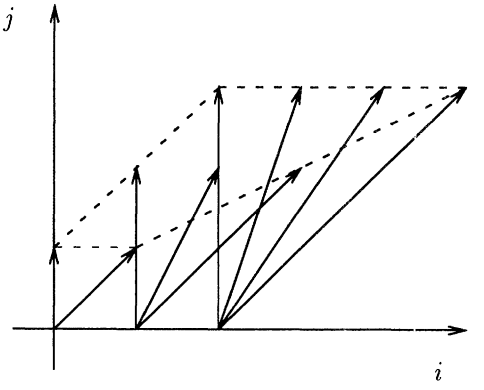


Fig. 4. Case when a pipeline vector does not belong to Θ^*

$$1 \leq k \leq N,\ k \leq j \leq N + 1,\ k + 1 < i \leq N \rightarrow (34)$$
$$A'(i, j, k) = A'(i - 1, j, k)$$

$$1 \leq k \leq N,\ k \leq j \leq N + 1,\ i = k + 1 \rightarrow (35)$$
$$A'(i, j, k) = A(k, j, k - 1).$$

One can check that the domain of equation (35) has dimension 2, one less than domain of (33).

When $Null(I) \cap Vect(D) \neq \{0\}$, it is necessary to propagate the data along pipelining vectors first, using Proposition 4.1. However, the pipelining vectors do not always belong to Θ^*, as shown by the following example.

Example 5. Consider the following system of equations, whose dependence graph is shown in figure 4, for $p = 3$.

$$1 \leq j \leq p,\ i \geq j - 1,\ i \leq 2j - 1 \rightarrow A(i, j) = B(j - 1, 0).$$

In this case, the domain of the dependence vector is the cone whose extremal rays are (1, 1) and (0, 1). On the other hand, the vector (1,0) is a pipelining vector and does not belong the the cone generated by the dependence vectors.

The following proposition shows that one can choose the pipelining vectors in such a way that the dependence vectors of the system after transformation still belong to a pointed cone. As a consequence, we will be guaranteed that a timing function still exists after transformation.

Lemma 1. Given a pointed cone C, and a linear space V, there exists a pointed cone C' which contains C and a basis of V.

Proof. Let $\{v_l\}_{1 \leq q \leq n}$ be a basis of V. We shall prove that there exists a set of values $\{\epsilon_i = \pm 1\}_{1 \leq q \leq n}$ such that $\{\epsilon_i v_i\}_{1 \leq q \leq n}$ and C generate a pointed cone C'. Let $\{c_k\}_{1 \leq k \leq K}$ be the set of extremal rays of C. Suppose first that the basis has one single vector v. Denote C_v

the cone generated by C and v. If C_v is pointed, the problem is solved with $\epsilon = 1$. Suppose that C_v is not pointed. Let us call *separating hyperplane of* C any hyperplane $h^t z = 0$ such that $h^t c_k > 0$ for all extremal ray c_k of C. As C is pointed, such an hyperplance exists. Moreover, as C_v is not pointed, for all separating hyperplane of C, $h^t v \leq 0$. If there exists such a hyperplane for which $h^t v > 0$, then $-h^t v > 0$, and C_{-v} is a pointed cone, and the proposition holds. If not, then $h^t v = -h^t v = 0$, and $-v \in C_v$. Therefore, $-v$ can be written as a positive combination of the extremal rays of C and of v, i.e:

$$-v = \sum_{k=1}^{K} \mu_k c_k + \nu v$$

where $\nu \geq 0$. A simple calculation shows then that

$$-v = \sum_{k=1}^{K} \frac{\mu_k c_k}{1 + \nu}$$

and therefore, $-v$ is a positive combination of the extremal rays of C, which implies that $-v \in C$. Again, C_{-v} is a pointed cone. The case when $q > 1$ is solved by applying the same idea successively to all the vectors of the basis.

Using Lemma 1, we have directly the following proposition:

Proposition 4.2. If Θ^* is a pointed cone, and if $Null(I) \cap Vect(D) \neq \{0\}$, there exists a basis of $Null(I) \cap Vect(D)$ which, together with Θ^*, generates a pointed cone.

As a direct result of the above development when $Null(I) \cap Vect(D) \neq \{0\}$, we can replace equation (29) by an equivalent system of equations, where none of the remaining non-uniformities involve pipelining vectors any more, the dependence vectors still belong to Θ^*, and the dimensions of D and of $I(D)$ are the same.

4.2.2. Routing Vectors We now consider the case when no pipelining vector exists. In the following, we shall say that a dependence vector $\Theta_{V(I(z))}$ has an *integral decomposition* on a set of vectors $\{A_i\}_{1 \leq i \leq q}$ in the domain D_p, if there exist q linear mappings $\alpha_i(z, p)$ such that for every $p \in \mathbf{P}$, and every $z \in D_p$

$$\Theta_{V(I(z))} = \sum_{j=1}^{q} \alpha_j(z, p).A_j \tag{36}$$

and $\alpha_i(z, p) \in \mathbf{N}$. Vectors A_i will be called *routing vectors* in the following.

To obtain such a decomposition, the idea is to use the extremal rays of the cone Θ^* as uniformization vectors. The following result, which is due to Delsarte ([8]), ensures that we can always find and compute such an integral decomposition:

Proposition 4.3. Given a rational convex polyhedral pointed cone C of dimension n, there always exists a pointed cone Γ whose extremal rays $R_1, \ldots, R_n$ form a unimodular basis, such that $C \subset \Gamma$.

Proof. We assume that the number of rays of C is $m \geq n$ (If $m > n$, one can transform the problem into one in $\mathbf{Z}^m$.) The polar cone C^0 of C, is the pointed cone defined as

$$C^0 = \{y \in \mathbf{Z}^n | x^t.y \leq 0, \ \forall x \in C\}.$$

We must find a pointed cone Γ^0 of extremal rays $R_1^0, \ldots R_n^0$ such that $\Gamma^0 \subset C^0$. Indeed, if such a cone exists, the polar cone Γ of Γ^0 solves the problem, as $C \subset \Gamma$, and the extremal rays $R_1, \ldots, R_n$ of Γ are such that

$$(R_1, \ldots, R_n) = -((R_1^0, \ldots, R_n^0)^{-1})^t.$$

Let M be the non-singular matrix $(M_1, \ldots, M_n)$ of the extremal rays of C^0. It can be shown that there always exists a rational positive upper triangular matrix P such that

$$M.P = U \tag{37}$$

where U is unimodular. Because of (37) and because P is positive, the unimodular U defines the extremal rays of Γ^0, i.e., $U = (R_1^0, \ldots, R_n^0)$.

The direct consequence of Proposition 4.3 is that the pointed cone Θ^* which contains the ranges of all the dependence vectors can be enclosed in another pointed cone Γ, whose extremal rays form a *unimodular basis.* Therefore, all the dependence vectors have an integral decomposition on the rays of Γ, which is simply their (unique) linear decomposition on the vectors of the unimodular basis. A simple change of basis provides the linear mappings $\alpha_i(z, p)$ which are sought.

It remains to show that we can remove the non-uniformities of the system of equations, using elementary transformations involving the routing vectors. We consider successively two cases depending on whether the routing vector belong to $Vect(D)$ or not.

Proposition 4.4. If A_i is a routing vector which does not belong to $Vect(D)$, there exists an affine mapping I' such that the following system of equations is equivalent to equation (29).

$$z \in D \rightarrow U(z) = f(\ldots, V'(z), \ldots) \quad (38)$$

$$z \in D_1 \rightarrow V'(z) = V'(z - A_i) \quad (39)$$

$$z \in D_2 \rightarrow V'(z) = V(I'(z)) \quad (40)$$

where V' is a new variable and

$$D_1 = \{z \in D | z \in D \wedge 0 \leq k < \alpha_i(z, p)\}$$

$$D_2 = \{z \in D | z \in D \wedge k = \alpha_i(z, p)\}$$

Proof. Clearly, if A_i is not parallel to D, then for all pair of points z and z' of D, and for all pair of integers k and k', $z - kA_i \neq z' - k'A_i$. Therefore, $D_1 \cap D_2 = \emptyset$, and each variable V' is defined only once. Moreover, the (linear) function which maps z to $z - \alpha_i(z, p)A_i$ is one-one, and there exists, therefore, an affine mapping J such that $z = J(z - \alpha_i(z, p)A_i)$. By taking $I'(z) = I(J(z))$ in equation (40), we obtain an equivalent system.

It remains to solve the case when a routing vector A is parallel to D. Before giving the solution, we show an example for which the above method fails.

Example 5. Figure 5 illustrates what happens when the routing vector is parallel to D. The equations are:

$$0 \leq i \leq p \rightarrow U(i) = U(i + p).$$

When p ranges over $\mathbf{N}$, the dependence vector $-p$ belongs to the half-line generated by -1. The transformation described by Proposition 4.4 gives the following system of equations:

$$0 \leq i \leq p \rightarrow U(i) = U'(i)$$

$$0 \leq i \leq 2p - 1 \rightarrow U'(i) = U'(i + 1)$$

$$p + 1 \leq i \leq 2p \rightarrow U'(i) = U(i).$$

It is clear that $U'(i)$ is defined twice for $p + 1 \leq i \leq 2p - 1$.

To solve the problem, we will show that we can augment the pointed cone Θ^* so that A is replaced by two other vectors which, together with Θ^*, generate a pointed cone.

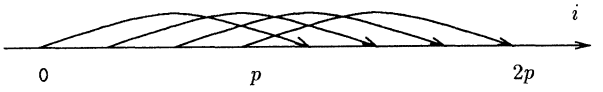


Fig. 5. An example where the routing transformation is not valid.

Proposition 4.5. If A is a routing vector which belongs to *Vect*(D), and if dim (D) $> n$, then there exist vectors u and v such that:

- neither u nor v belong to Vect(D)
- Θ^*, u and v generate a pointed cone.

Proof. We assume that A is an extremal ray of Θ^*. (If it is not the case, we can beforehand replace A by a positive combination of extremal rays of Θ^*.) As we have supposed that there are no pipelining vectors, $dim(D) = dim(I(D)) > n$. Therefore, the supplementary linear space S of $Vect(D) \cap Vect(I(D))$ is not reduced to 0. Let v be a non-zero vector of S. By Lemma 1, either Θ^* and v or Θ^* and $-v$ generate a pointed cone. Assume that this is true for v. We claim that Θ^*, v and $A - v$ still generate a pointed cone. Indeed, as Θ^* and v generate a pointed cone, and as v and A are not collinear, there exists a separating hyperplane $h^t z = 0$ of Θ^* such that $h^t A > h^t v$. Therefore, $h^t(A - v) > 0$, and $h^t z = 0$ is also a supporting hyperplane of the cone generated by Θ^*, v and $A - v$. The vectors v and $A - v$ are the new vectors we seek in order to replace A, as none of these vectors belongs to *Vect*(D).

The consequence of Proposition 4.5 is that one can replace a routing vector by two new vectors, in such a way that the transformation of Proposition 4.4 is possible and the new system still has a timing function. The following example illustrates this situation.

Example 6. Consider the following system of equations (see figure 6):

$$i = 0, 0 \leq j \leq n, 0 \leq k \leq n \rightarrow A(i, j, k) = A'(k, j + n, 0). \quad (41)$$

In this example the cone Θ^* has the extremal rays (0, 1, 0) and (1, 1, -1). The dependence vector is $(-k, -n, k) = (-n - k)(0, 1, 0) - k(1, 1, -1)$. However, as the first ray is parallel to D, the transformation using the routing vector (0, 1, 0) is incorrect. One can avoid the problem by replacing the first vector by a combination of the second one and the first one. Here we rewrite (0, 1, 0) = (1, 1, -1) + (-1, 0, 1). The new routing (see figure 6b) is done by $(-k, -n, k) = -(n + k)(-1, 0, 1) - (n + 2k)(1, 1, -1)$.

4.2.3. Summary of the Uniformization Method In summary, the uniformization method works as follows:

- First case: $Null(I) \cap Vect(D) = \{0\}$.
  - Assume first that $dim(D) > n$. We do not need pipeline vectors. Using Proposition 4.3, we find out a set at most n vectors A_i which form a unimodular basis generating a pointed cone enclosing Θ^*. Then, we process successively all the vectors A_i. If A_i is not paralled to D, then we apply the transformation defined in Proposition 4.4, which

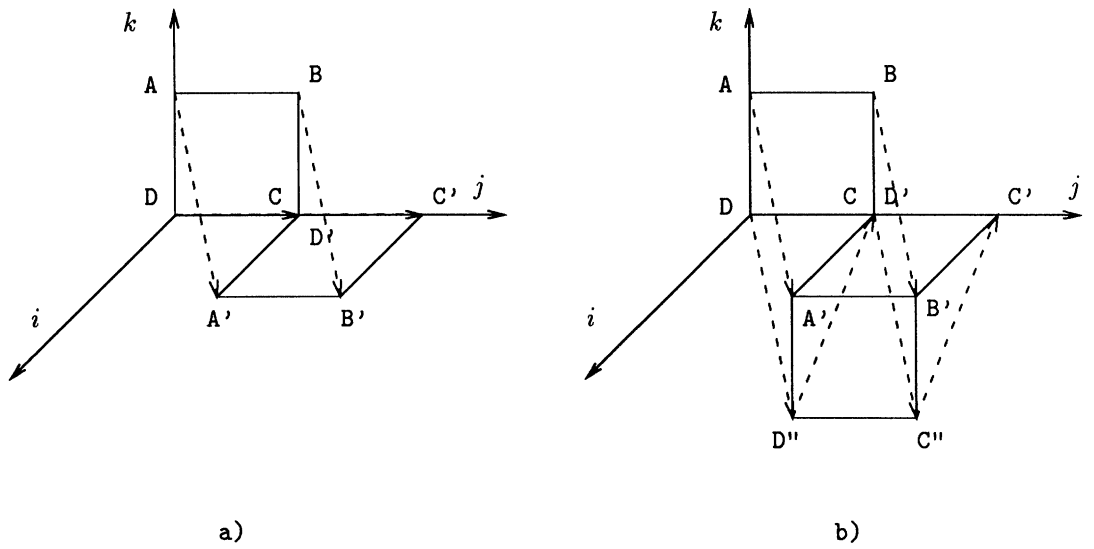


Fig. 6. Case when a uniformization vector is parallel to D.

gives a new domain D. If A_i is parallel to D, using Proposition 4.5, we replace A_i by two new vectors in such a way that these vectors still generate a pointed cone.

— If $dim(D) = n$, our method does not make it possible to obtain a uniform system in $\mathbf{Z}^n$. However, it is always possible to re-index the variables in $\mathbf{Z}^{n+1}$ and to apply the above theory. Indeed, the new system will need more processors, as the allocation function will map the system to $\mathbf{Z}^n$ instead of $\mathbf{Z}^{n-1}$. This is however not surprising: the case when $dim(D) = dim(I(D)) = n$ represents a bad situation when the dependence vectors are dense in $\mathbf{Z}^n$.

- If $Null(I) \cap Vect(D) \neq \{0\}$, then we can find an integral basis $\Phi = \{\phi_i\}_{1 \leq i \leq q \leq n}$ of $Null(I) \cap \text{Vect}(D)$ such that Φ and Θ^* still generate a pointed cone Θ^+. Then, by applying Proposition 4.1 successively to each vector of Φ, we eventually obtain a finite set of equations, whose domain has dimension $dim(D) - q$, and whose dependence vectors are still in Θ^+. We are back to the previous case.

All this discussion is summarized by the following theorem:

THEOREM 3. Given a set of linear parameterized recurrence equations whose dependence vectors are non-null and belong to a pointed cone, there exists an equivalent system of uniform recurrence equations which has a timing function.

It should be noted that although the above transformations involve solving some problems for which no efficient algorithms exist (such as, for example, the computation of the generating system of a convex polyhedral domain), our experience has shown that, in practice, the systems we handle lead to small-size problems which can be solved in a reasonable amount of time.

We also emphasize that our method has no pretention to be optimal. Clearly, at several steps, some choices may result in more or less efficient transformation. At least, however, we know that the resulting system can be scheduled.

4.3. Multistep Algorithms

The method described in the preceding section applies when Θ^* is pointed. We illustrate here what may be done when Θ^* is not pointed.

Consider the uniformization of the dependence $(i - k, 0, 1)$ (as above) in the Gauss-Jordan elimination algorithm. The set of dependences is defined by the line $k = 1$ (see 3.6). Hence, Θ^* is a cone defined by $k \geq 0$, $j = 0$ (a half-plane). This cone, shown in figure 7, is degenerated since it contains the line $k = 0$; one may verify that $(1, 0, 0)$, $(-1, 0, 0)$ and $(0, 0, 1)$ form an integral unimodular basis for this cone. The resulting uniform dependence graph is shown on figure 8. However, after uniformization, this algorithm has no timing-function, as there are opposite dependence vectors.

We now present an *ad-hoc* method for solving this problem. The opposite vectors were used in the uniformization of $A(k, k, k - 1)$ and of $A(k, j, k - 1)$. By looking at the recurrence equations, one observes that

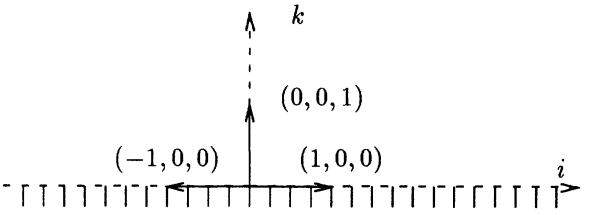


Fig. 7. A degenerated cone Θ^* (it contains the line $k = 0$) and a possible minimal basis.

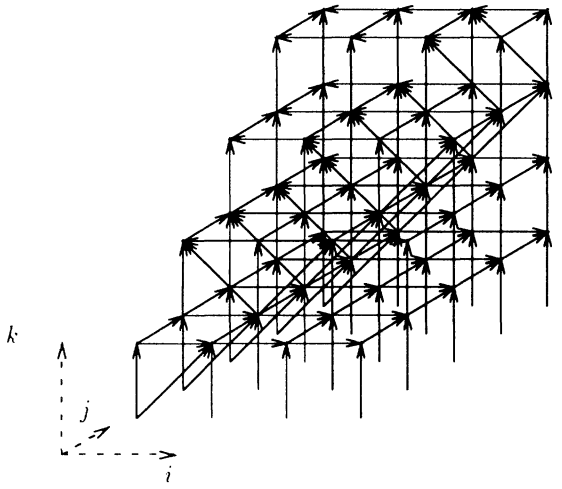


Fig. 8. A uniform dependence graph for Gauss-Jordan which cannot be mapped on a systolic array.

these two partial results are computed in the part of the domain where $i - k \geq 0$. Hence, the Gauss-Jordan algorithm can be considered as a two-step algorithm. In the first step, when $i - k \geq 0$, both $A(k, k, k-1)$ and $A(k, j, k-1)$ are computed. In the second one, both variables are used only as inputs.

Both steps can be uniformized independently with the above uniformization method. Then the two steps can be put together as described in [5] and [25]. By looking at the time cones of the two algorithms, one can compute a re-indexing transformation which gives a maximal global time cone. Finally, the translation of the second step can be computed by expressing the fact that all external dependences must be positive along the time. Figure 9 shows the resulting uniform dependence graph. The two steps of the algorithm are shown; one of them was translated from i to $i + N$

With this uniform dependence graph defined, one can now map it on various systolic arrays. Let us choose for example the mapping $a(i, j, k) = (i, k)$, i.e., a projection along axis j. This allocation function is authorized, as the direction (0, 1, 0) of the projection is not orthogonal to the vector λ, therefore, ensuring that two computations allocated to the same processor are not executed at the same instant of time (see [10]). The resulting architecture is shown in figure 10, and is described independently by Robert and Trystram [26] and Lewis and Kung [27][6]. The movement of the A coefficients is as follows: they enter the architecture in the

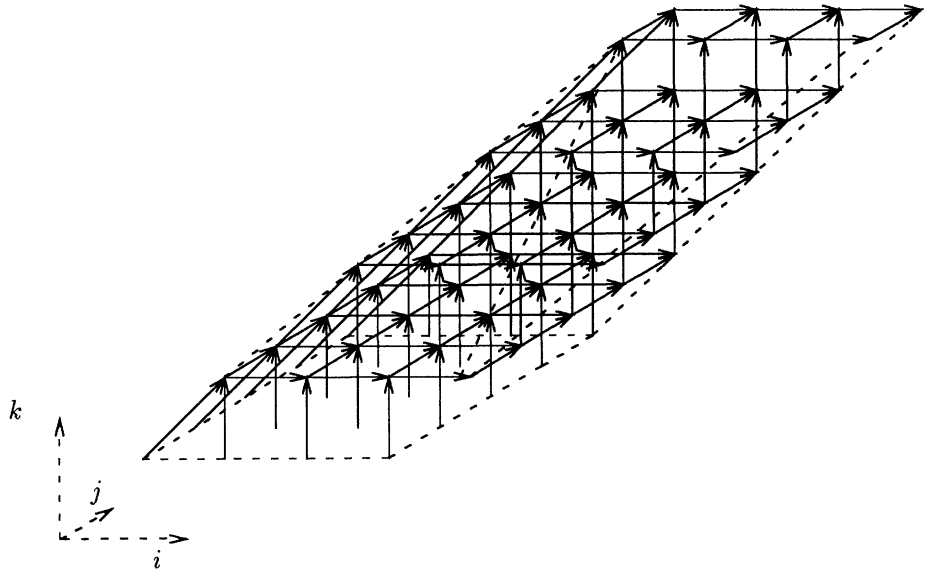


Fig. 9. A uniform dependence graph for Gauss-Jordan, when considered as a two-step algorithm. The edges of the domains of each step are in dashed lines.

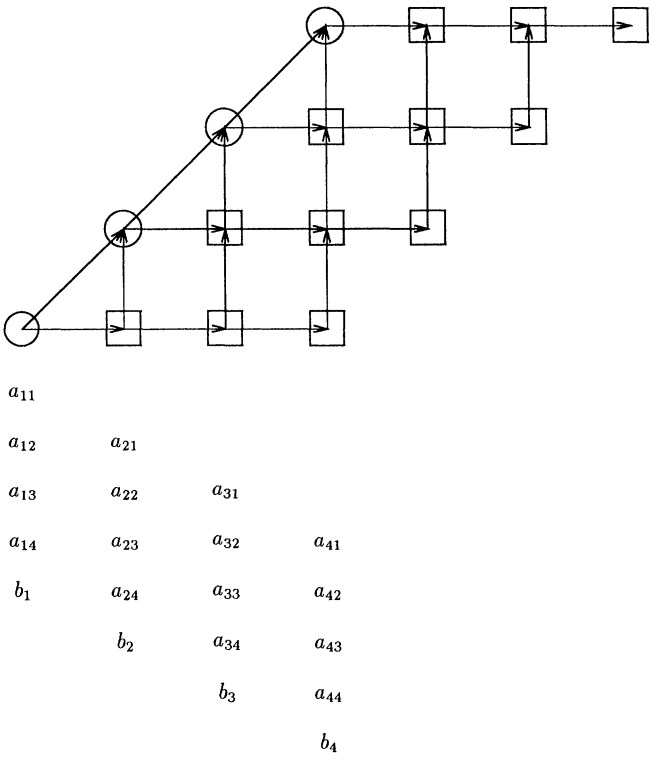


Fig. 10. Architecture obtained by projection along the j axis.

bottom row, then they are reflected by the left diagonal, move rightward to the right diagonal, where they are reflected again to the top. This movement is the projection of the path of the coefficients in the domain of figure 9.

4.4. *Uniformization of the Input Equations*

We now consider the uniformization of the *input equations*. The difference with what we saw previously is that there is a priori no location associated to the data: initially, a data can be anywhere. It can be considered as the transformation of data broadcasting into pipelining. Such a problem has been tackled already by several authors [17], [19]–[21]. We briefly present a method that uses some of the theory developed before. The way we solve the problem is illustrated on the input equation

$$A(i, j, k) = a(i, k) \tag{42}$$

of the matrix multiplication algorithm. The first step consists of defining the set of points (i, j, k) that will use a particular data $a(i_0, k_0)$. This is done by computing the kernel of the affine transformation that maps (i, j, k) on (i_0, k_0) and by intersecting this kernel with the domain of the index points (i, j, k). The kernel is the solution of the system

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \\ k \end{pmatrix} = \begin{pmatrix} i_0 \\ k_0 \end{pmatrix}. \tag{43}$$

Hence, the kernel is $(i, j, k) = (i_0, j_0, k_0) + k.(0, 1, 0)$. The intersection with the domain of the indices gives

$$(i, j, k) = (i_0, 0, k_0) + k.(0, 1, 0) \quad \text{with } k \in [1, N]. \tag{44}$$

The resulting domain is a convex polyhedron, whose vertices are $(i_0, 1, k_0)$ and (i_0, N, k_0). The second step is to choose an initial location in this convex, which defines where the data is first. In the systolic model, this location must be a vertex of the domain (otherwise there would be no affine schedule). Of the two possibilities, we arbitrary choose $(i_0, 1, k_0)$. Equation (42) can now be rewritten, with the use of a new variable U_α, as

$$A(i_0, j, k_0) = U_\alpha(i_0, 1, k_0) \tag{45}$$

$$U_\alpha(i_0, 1, k_0) = a(i_0, k_0). \tag{46}$$

Equation (45) being fully indexed, it can be transformed into a set of uniform recurrences with the method described before. Equation (46) is a new input equation. Yet, its kernel only contains one point; hence, nothing needs to be broadcasted and uniformized.

5. Related Work

In [16], Yaacoby and Capello are interested by the scheduling of affine recurrences. They give a sufficient condition for an affine recurrence to be computable, a necessary and sufficient condition for the existence of an affine schedule, and a constructive means for finding the affine schedule.

In [28], Mongenet and Perring present a methodology for the mapping of inductive problems on systolic arrays. In these problems, routing need to be performed for the partial results that are further used as data. Work in this direction was recently made by Clauss [29]. In their approach, they decompose non-uniform dependences into two sets of vectors, namely the generating vectors (in our terminology, the pipelining vectors) and the dependency vectors (some of which are the routing vectors). These vectors are automatically computed by looking at the domain of the dependences. Their approach, however, is restricted to inductive problems, a sub-class of linear recurrences. Also, because they do not consider the size parameters in the construction of their dependence domain, their resulting architecture may be non-modular.

In [30], Guerra and Melhem consider the synthesis of another class of affine recurrences, where the non-uniform dependences are of the form $\vartheta = (a_1, \ldots, a_{t-1}, i-j, a_{t+1}, \ldots, a_n)$, a_i are constants, and i and j are indices of the problem. Their synthesis is in two steps. First, they determine a coarse timing function by analyzing the set of dependence vectors (note that size parameters are not taken into account). In the second step, they perform a time-space index transformation, compatible with the coarse timing. They illustrate their methodology on the dynamic programming algorithm.

In [31], Chen presents a methodology for the derivation of systolic algorithms and architectures in a unified framework based on the language Crystal. First, the naive algorithm is transformed into a bounded order and bounded degree program (with bounded fan-in and fan-out degree). Then, it is further transformed by means of a space-time mapping. The method is illustrated with the dynamic programming algorithm. Clearly, the aim of fan-in and fan-out reductions is to uniformize the algorithm. The fan-in reduction is something we have not considered here. This problem occurs when a naive algorithm makes use of a census function (which is applied on a set of data). In a linear recurrence equation, this cannot be the case. The fan-out reduction is achieved by means of additional recurrence equations whose constant dependence vectors are, what we call, the uniformization vectors. For the dynamic programming algorithm, this reduction is quite simple.

In [20], Fortes and Moldovan discuss how and whether data broadcasts in an array processor with a given interconnection structure can either be eliminated or reduced by choosing an adequate linear schedule. They consider the class of linear recurrences whose variable instances $V(Az + B)$ are such that A is rank deficient. In order to have broadcast, the rank of A must be strictly less than n, the dimension of the index z. Also, the associated computations must be scheduled at the same time. For a given schedule, necessary conditions are given on the processor allocation so that no broadcast is required and the array interconnections support the necessary data communications. Finally, they discuss how linear schedules can explore the limited broadcasting capability offered by the array interconnections to implement large broadcasts. In the approach of Fortes and Moldovan, the interconnection structure of the array is fixed. In this paper, we do not consider this because the resulting array is more likely to be designed on a VLSI chip. It would be useful for us to extend the presented methodology to the case where the interconnection structure is fixed. It seems that such a problem can be solved by adding additional constraints related to this structure, as explained in [20].

In [21], Wong and Delosme also consider the elimination of broadcasting. Hence, as in [20], they consider dependences whose index matrix is rank deficient. Similar to our approach, they are interested by the synthesis of systolic arrays and they do not make a priori restriction on its interconnection structure. Their problem is similar to ours; non-uniform dependences must be expressed as a linear combination of a finite set of uniformization vectors. Their approach is different in the sense that no affine schedule is required on the resulting dependence graph. Thus, dependences can be opposite as in the graph of figure 9. First, the canonical vectors are chosen as candidates for the uniformization. They give a necessary and sufficient condition for the acceptance of such a canonical propagation. Secondly, it is generalized to allow the inclusion of non-canonical vectors. They show that all broadcasts are transformable when using these general propagations. This approach is

different since the schedule is not a priori taken into account. In the method that we present, the uniformization vectors are minimal in terms of the associated set of constraints on the affine schedule.

In [19], Rajopadhye and Fujimoto also consider the transformation of broadcasting into pipelining; hence, the index matrix A of the linear dependence is rank deficient. The pipelining vectors are also found by computing the null space of A. They show how to transform the linear recurrence into a set of conditional uniform recurrences, and how to propagate the control signals in the resulting architecture. They also illustrate their methodology on the dynamic programming algorithm (for optimal string parenthesization).

In [17], Joinnault considers both the pipelining problem and the routing problem. The first is solved by looking at the null space of the index matrix (as in [19], [20]). To solve the second problem, the non-uniform dependence is written in terms of n transvections and one translation. Yet, no systematic method is provided to find them.

In conclusion, several papers deal with the mapping of linear recurrences on regular arrays. Yet, they differ in the form of the linear recurrence equations, and on the requirements imposed on the uniformization vectors (existence or not of an affine schedule). Thus, comparing the results must be done with great care!

6. Conclusion

We have described the basics for the analysis of systems of linear recurrence equations and the principles of the mapping of such equations on parallel architectures. We also pointed out a few problems that still need to be solved. Once overcome, it will be possible to implement such methods into software packages that will produce efficient code. Some of the techniques presented here are already being used in prototype tools for systolic array design ([32], [33]). The presented approach is very promising as it makes it possible to apply safe transformations on the recurrence equations that would otherwise be very difficult to use. A complete methodology should probably include the results of several other methods, such as the ones presented in section 5. For example, a complete uniformization method should include automatic re-indexing transformations as the folding operation, which can be used for some particular types of affine function.

The material that we have presented in this paper has some commonalities with the work that has been done on restructuring compilers for super-computers. Merging both fields will probably provide new fruitful research ideas.

Acknowledgements

The authors would like to thank referee 3 for his very detailed and valuable comments. They are also greatly indebted to P. Delsarte who provided a proof of Proposition 4.3.

Notes

[1] The reader is referred to [34] for an introduction to convex polyhedra.

[2] This hypothesis is not compulsory, but simplifies the following presentation. In particular, it can be useful to assume that the domain has one infinite direction, in order to represent algorithms such as the convolution. All the results in this paper can be extended to cover such a case.

[3] Recall that a vertex of a convex polyhedron is an extremal point of the polyhedron, a ray is the direction of an infinite half-line of the polyhedron, and a line is the direction of a line of the polyhedron. Any convex polyhedron can be generated by a finite set of vertices, of rays and of lines. Again, the reader is referred to [34] for definitions.

[4] This is the assumption we have made in this paper.

[5] Given a set D, the affine hull of D is the smallest affine space which contains D. The direction of an affine space A is the unique linear space L such that A is a translation of L [35].

[6] Both papers deal with the Algebraic Path Problem, which covers as a particular case the Gauss-Jordan elimination.

References

1. R.M. Karp, R.E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *JACM*, 14(3): 563-590, July 1967.
2. L. Lamport. The parallel execution of Do-Loops. *Comm. ACM*, Vol. 17, N. 2, pp. 83-93, Feb. 1974.
3. P.R. Cappello and K. Steiglitz. Digital signal processing applications of systolic algorithms. In *VLSI Systems and Computation*, H.T. Kung, B. Sproull, and G. Steel editors, Computer Science Press (1981), 245-254.
4. M.C. Chen, Synthesizing systolic designs. *1985 International Symposium on VLSI Technology, Systems and Applications, Taipei, Taiwan, R.O.C.*, 8-10 May 1985.
5. J.M. Delosme and I.C.F. Ipsen. Efficient systolic arrays for the solution of Toeplitz systems: An illustration of a methodology for the construction of systolic architectures for VLSI. In W. Moore, A. McCabe, and R. Urquhart, editors, *International Workshop on Systolic Arrays*, pages 37-46, Adam Hilger, University of Oxford, UK, July 2-4 1986.

6. S.Y. Kung. VLSI array processors. *IEEE ASSP Magazine*, 2(3):4–22, July 1985.
7. G.H. Li and B.W. Wah. The design of optimal systolic arrays, *IEEE Trans. Computers* 34, 1 (1985), 66–77.
8. W.L. Miranker and A. Winkler. Space-time representations of systolic computational structures. *Computing*, 32:93–114, 1984.
9. D.I. Moldovan. On the analysis and synthesis of VLSI algorithms, *IEEE Trans. On Computer* C-31 (11), November 1982, pages 1121–1126.
10. P. Quinton. *The Systematic Design of Systolic Arrays*. Technical Report 193, Publication Interne IRISA, April 1983.
11. S.K. Rao, *Regular iterative algorithms and their implementations on processor arrays*, PhD Thesis, Information Systems Lab., Stanford Univerity, October 1985.
12. P. Quinton, Automatic synthesis of systolic arrays from uniform recurrent equations *Proc. IEEE 11th Int. Sym. on Computer Architecture*, Ann Arbor, MI, 1984, 208–214.
13. H.T. Kung. Let's design algorithms for VLSI systems. *Proc. of the Caltech conference on VLSI*, January 1979.
14. D.I. Moldovan and J.A.B. Fortes. Partitioning and mapping algorithms into fixed size systolic arrays. *IEEE Transaction on Computers*, C-35(1):1–12, January 1986.
15. V. Van Dongen and P. Quinton. The mapping of linear recurrence equations on regular arrays, Manuscript M 264, Philips Research Lab. Brussels, October 1988.
16. Y. Yaacoby and R. Cappello. Scheduling a system of affine recurrence equations onto a systolic array. *International Conference on Systolic Arrays*, San Diego, pages 373–381, May 1988.
17. P. Gachet and B. Joinnault. Conception d'alorithmes et d'architectures systoliques. Thèse de l'Université de Rennes I, Sept 1987.
18. Y. Saouter and P. Quinton. Computability of recurrence equations, IRISA Research Report, to appear, 1989.
19. S.V. Rajopadhye and R.M. Fujimoto. Systolic array synthesis by static analysis of program dependencies. In J.W. de Bakker, A.J. Nijman, and P.C. Treleaven, editors, *Parallel Architectures and Languages Europe*, pages 295–310, Springer-Verlag, June 1987.
20. J.A.B. Fortes and D.I. Moldovan. Data broadcasting in linearly scheduled array processors. *Proc. 11th Annual Symp. on Computer Architecutre*, pages 224–231, 1984.
21. Y. Wong and J.M. Delosme. Broadcast removal in systolic algorithms. *International Conference on Systolic Arrays*, San Diego, pages 403–412, May 1988.
22. R.H. Kuhn. Optimization and interconnection complexity for: Parallel processors, single-stage networks, and decision trees. Technical Report UIUCDCS-R-80-1009, Univeristy of Illinois at Urbana-Champaign, February 1980.
23. P. Gachet, B. Joinnault, and P. Quinton. Synthesizing systolic arrays using DIASTOL. In W. Moore, A. McCabe, and R. Urquhart, editors, *International Workshop on Systolic Arrays*, pages 25–36, Adam Hilger, University of Oxford, UK, July 2–4 1986.
24. V. Van Dongen and P. Quinton. Uniformization of linear recurrence equations: a step toward the automatic synthesis of systolic arrays. *International Conference on Systolic Arrays*, San Diego, pages 473–482, May 1988.
25. B.W. Wah, M. Aboelaze, and W. Shang. Systematic designs of buffers in macropipelines of systolic arrays. *Journal of Parallel and Distributed Computing* 5, pp. 1–25, 1988.
26. Y. Robert and D. Trystram. Systolic solution of the algebraic path problem. In W. Moore, A. McCabe, and R. Urquhart, editors, *International Workshop of Systolic Arrays*, pages 171–180, Adam Hilger, University of Oxford, UK, July 2–4, 1986.
27. P.S. Lewis and S.Y. Kung. Dependence graph based design of systolic arrays for the algebraic path problem. *20th Asilomar Conf.*, Nov. 10–12, 1986.
28. C. Mongenet and G.R. Perrin, Synthesis of systolic arrays for inductive problems. In J.W. de Bakker, A.J. Nijmann, and P.C. Treleaven, editors, *Parallel Architectures and Languages Europe*, pages 260–277, Springer-Verlag, June 1987.
29. P. Clauss. Contribution à la synthèse de réseaux systoliques. *Rapport de DEA*, Université de Besançon 1987.
30. C. Guerra and R. Melhem, Synthesizing non-uniform systolic designs. *Proc. of the 1986 International Conference on Parallel Processing*, pages 765–772, August 1986.
31. M.C. Chen. Synthesizing VLSI architectures: Dynamic programming solver. *Proc. of the 1986 International Conference on Parallel Processing*, pages 776–784, August 1986.
32. P. Gachet, P. Quinton, C. Mauras and Y. Saouter. Alpha du Centaur: A prototype environment for the design of parallel regular algorithms. IRISA Research Report number 439, November 1988.
33. V. Van Dongen. Presage, a tool for the design of low-cost systolic arrays. *ISCAS 88*, pages 2765–2768, 1988.
34. A. Schrijver. *Theory of linear and integer programming. Wiley-Interscience series in Discrete Mathematics*, John Wiley and Sons, 1986.
35. J. Dieudonné. *Algèbre linéaire et géométrie élémentaire*, Hermann, Paris, 1964.

The Use of Data Dependence Graphs in the Design of Bit-Level Systolic Arrays

JOHN V. McCANNY, JOHN G. McWHIRTER, AND SUN-YUAN KUNG, FELLOW, IEEE

Abstract—**The use of bit-level systolic array circuits as building blocks in the construction of larger word-level systolic systems is investigated. It is shown that the overall structure and detailed timing of such systems may be derived quite simply by using the dependence graph and cut-set procedure developed by Kung. This provides an attractive and intuitive approach to the bit-level design of many VLSI signal processing components. The technique may be applied to ripple-through and partly pipelined circuits as well as fully systolic designs. It therefore provides a means of examining the relative tradeoff between levels of pipelining, chip area, power consumption, and throughput rate within a given VLSI design.**

I. Introduction

THE USE of systolic arrays in the design and implementation of high performance digital signal processing equipment is now well established. Most of the research to date has concentrated on word-level systems where the typical processor is of single chip complexity (at least). Systolic arrays of this type have now been built at various laboratories throughout the world [1]–[3]. The systolic array concept can also be exploited at bit level in the design of individual VLSI chips and a number of commercial components have now been successfully developed using this design philosophy [4]. These include a bit slice correlator [5], a bit serial FIR filter [6], a chip for performing the Winograd Fourier transform [7], [8] and a rank order filter chip [9]. Bit-level systolic arrays exhibit a number of attractive features:

1) the basic processing element is small (typically a gated full adder) and an entire array of these may be integrated on a single chip;
2) the computation time for a single cell is small (typically 3–4 gate delays) and so the overall throughput which may be achieved using a given technology is very high;
3) the highly regular structure of the circuits renders them comparatively easy to design and test.

In a recent paper [10] we considered how bit-level systolic array circuits might be used as building blocks in the construction of world-level processing systems. Vector constraint equations were used to describe bit-level systolic arrays and it was shown that overall regularity could be achieved if the pattern of data flow within a bit-level building block is compatible with that required at the word level. We also suggested that the cut theorem [11] could be used to establish correct timing for this type of pipelined system but no attempt was made to justify that comment. In this paper, we develop the idea by showing how an approach developed by Kung [12] may be exploited in the systematic design of bit-level systolic building blocks. Suitable architectures are deduced by the projection of dependence graphs, and the detailed timing required for a pipelined implementation is then derived using a technique based on the cut set procedure. As in the previous paper, we consider the computation of inner products at the bit level (Sections II and III) and show how a variety of architectures may be derived for this purpose. We then describe how the ideas may be extended to matrix × matrix multiplication (Section IV) and discuss some of the systolic array structures which may be implemented using bit-level systolic building blocks. The advantages of this design approach are discussed in Section V and compared with the technique described in our previous paper.

Manuscript received May 26, 1987; revised May 5, 1989.

J. V. McCanny is with the Department of Electrical and Electronic Engineering, the Queen's University of Belfast, Belfast BT9 5AH, Northern Ireland.

J. G. McWhirter is with the Royal Signals and Radar Establishment, Malvern, Worcs. WR14 3PS, England.

S.-Y. Kung is with the Department of Electrical Engineering, School of Engineering/Applied Science, Princeton, NJ 08544.

IEEE Log Number 9034428.

II. Computation of Inner Products at the Bit Level

Numerous parallel processing architectures have been proposed for computing the inner product

$$s = \sum_{k=0}^{N-1} a_k b_k \tag{1}$$

of two vectors $\boldsymbol{a} = \{a_k\}$ and $\boldsymbol{b} = \{b_k\}$ ($k = 0, 1, \cdots, N-1$). By way of example we will consider first the use of a simple word-level systolic array of the type illustrated schematically in Fig. 1. It is assumed in this case that the numbers a_k reside on fixed processing sites while the numbers b_k are input in a staggered manner from the right with each of the black dots representing a delay. One pair of numbers is multiplied on each processing element and their product is added to the accumulating result which propagates vertically as indicated. This corresponds to a simple recursion of the form $s \leftarrow s + a_k b_k$.

A. Multiply and Accumulate at the Bit Level

In order to generate an efficient VLSI design for the type of processing system illustrated in Fig. 1 (as opposed

Reprinted from *IEEE Transactions on Acoustics, Speech, and Signal Processing*, pp. 787-793, May 1990.

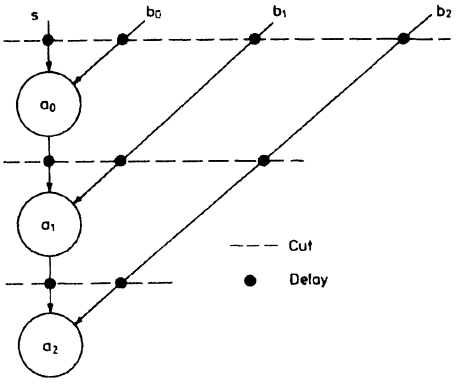


Fig. 1. Word-level systolic array for inner product computation.

to using an off-the-shelf multiply and accumulate chip for each node) it is essential to decide how the calculation is to be organized at the detailed bit level. One systematic approach is to employ the techniques developed by Kung [12] based on the use of dependence/signal flow graphs.

If a_k and b_k are both n bit binary (positive) numbers and s_k denotes their product, then the pth bit of s_k (denoted by $s_{k,p}$) is computed according to the equation

$$s_{k,p} = \sum_{q=0}^{n-1} a_{k,q} b_{k,p-q} + \text{carry bits} \qquad (2)$$

where $a_{k,q}$ and $b_{k,q}$ denote the qth bit of a_k and b_k, respectively. Following the approach adopted by Kung, this computation is expressed in single assignment form; i.e., a form such as that given in (3) where every variable is assigned one value only during the computation:

$$\bar{s}^1_{k,p} = 0 \qquad (3a)$$

$$s^q_{k,p} = s^{q-1}_{k,p} + a_{k,q} b_{k,p-q} + \text{carries} \qquad (3b)$$

$$s_{k,p} = s^{n-1}_{k,p}. \qquad (3c)$$

The computation can then be represented using a dependence graph of the type shown in Fig. 2 where, for simplicity, we have omitted the subscript k. In this representation the indices of the single assignment algorithm have been mapped onto a two-dimensional index space, as illustrated. For the purposes of Fig. 2, it has been assumed that both inputs are 3-b numbers and so a 3 × 3 array of index points is required. Each dependence relation then corresponds to an arc between the associated variables within the index space of the dependence graph. For example, in Fig. 2 we assume that each bit of the word a is common to all points having the same q index, i.e., in the same row of an array. Similarly, each bit of b is common to all points within a single column and so it interacts with each bit of a as required for binary multiplication. It is assumed that at each point of intersection (i.e., each index point) a computation occurs and this is represented by the open circle. The computation required in this example is the multiplication of two individual bits a and b followed by addition of the resulting partial product to the accumulating result s and an input carry bit. This may be accomplished in terms of logic circuits using an AND gate and a FULL ADDER. It should be noted that carry bits emerging from the third column of cells must be added to accumulating results of a higher significance. This is done by feeding them into the sum inputs of the cells in the row immediately above, as illustrated. It is worth noting that apart from the need for carry bits (which must propagate from the least to the most significant bit positions) the dependence graph in Fig. 2 also represents the process of convolving two 3-element sequences of binary digits.

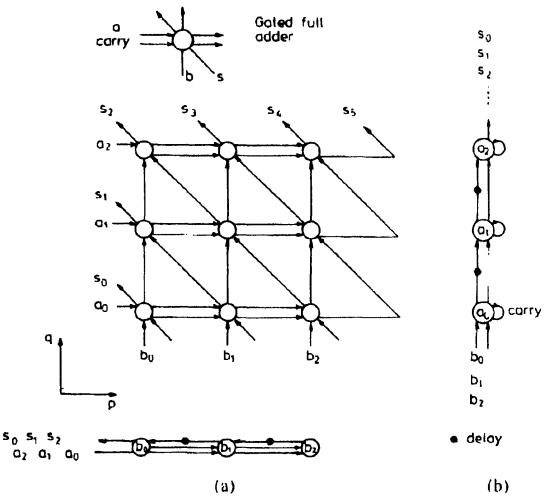


Fig. 2. Dependence graph for binary multiplication. For simplicity we have omitted the subscript k from the words a and b. The subscripts refer to individual bits within these words. (a) Graph derived by projecting dependence graph onto vertical axis. (b) Linear array derived by projecting dependence graph onto the horizontal axis.

An obvious way to map the computation represented in Fig. 2 onto physical hardware is to assign a separate processing element to each node in the dependence graph and allow the data bits to ripple through the resulting circuit as indicated. This corresponds to using the type of "ripple-through" parallel multiplier [13] which has become quite common in the design of signal processing equipment. The computation rate of a ripple-through circuit can be enhanced through the use of pipelining. A number of pipelined circuits can be derived from the graph in Fig. 2 by applying the cut-set procedure described by Kung [12]. A valid cut through a dependence graph is one which does not involve zero delay edges where computational parameters move in opposite directions. The number of valid cuts which can be applied to the graph in Fig. 2 is limited. One obvious cut is shown in Fig. 3 and referred to as cut 1. The effect of such a cut is to introduce delays (i.e., latches) along the data lines which have been cut. The resulting circuit then constitutes a pipelined shift and add multiplier where each bit of a is broadcast along one of the rows, the bits of b_k and the accumulating sum bits are clocked through latches from row to row and the carries ripple to higher significant bit positions within a given row.

The circuit can be fully pipelined (and thus takes the

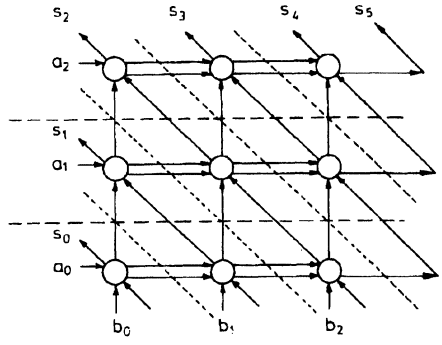


Fig. 3. The application of cuts to the dependence graph shown in Fig. 2.

form of a bit-level systolic array) by applying a second (diagonal) cut as shown in Fig. 3. This imposes a second delay on each bit of b as it passes from row to row and corresponds to a carry-save design of the type proposed by a number of authors including Myers [14], Sheeran [15], and Hoekstra [16].

So far, we have only considered bit parallel multiplication. Serial/parallel architectures may be obtained by projecting the dependence graph in Fig. 2 onto one dimension. For example, the linear array in Fig. 2(b) is derived by projecting the dependence graph onto the horizontal axis. The detailed timing of the resulting signal flow graph can then be derived by noting that on successive cycles this linear array must perform the computations assigned to successive rows within the two-dimensional array. The signal flow graph shown in Fig. 2(a) has been derived by projecting the dependence graph onto the vertical axis. In this case the timing has been derived by considering the computations required by successive groups of nodes which lie along the diagonals in the dependence graph. Fully systolic architectures can then be derived from these and similar signal flow graphs by applying cuts perpendicular to the line of processors. A number of well-known serial/parallel architectures may be derived in this way including those studied in detail by Danielson [17] and the ones proposed by Lyon [18] (Fig. 2(a)).

B. Alternative Multiply and Accumulate Architectures

The dependence graph in Fig. 2 represents a multiplication which starts by forming the product of the two least significant bits a_0 and b_0. An alternative dependence graph is shown in Fig. 4 corresponding to a single assignment computation of the form:

$$\bar{s}^1_{k,p} = 0 \tag{4a}$$

$$s^q_{k,p} = s^{q-1}_{k,p} + a_{k,n-1-q} b_{k,p-n+1+q} \tag{4b}$$

$$s_{k,p} = s^{n-1}_{k,p}. \tag{4c}$$

In this case, the partial products are summed in the opposite order to that defined in Fig. 2 as indicated by the reverse direction of the diagonal arrows in Fig. 4. One consequence of reordering the computation in this way is

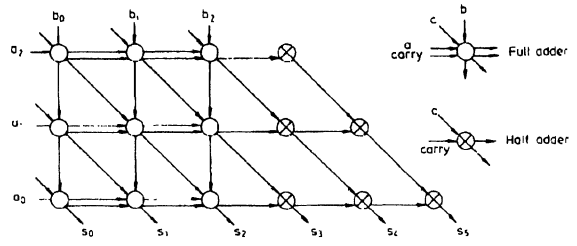


Fig. 4. An alternative dependence graph for binary multiplication.

that an extra triangular array of processing nodes (in the form of half adders) must be added to the dependence graph to ensure that all carry bits which emerge from the right-hand column are included in the final result. It is also necessary to apply different cuts to the dependence graph as illustrated in Fig. 5. For example, a vertical or horizontal cut will produce a semisystolic circuit. Alternatively, application of a cut at 45° (equivalent to a horizontal plus a vertical cut) yields the systolic multiplier which was described by McCanny and McWhirter [19].

C. Application to the Computation of Inner Products

One approach to designing an inner product processor at the bit level is to embed an appropriate bit-level dependence graph into each processing node of the preferred word-level circuit diagram. The resulting diagram can then be cut and/or projected as required to represent a wide range of alternative circuit implementations. In order to design a highly regular, pipelined circuit, it is necessary to align the flow of data in the bit-level dependence graph with that dictated by the overall word-level architecture. In Fig. 6, for example, we show how the data dependence graph of Fig. 2 may be embedded into a two stage inner product array of the type illustrated in Fig. 1. The dependence graphs have been rotated so that the bit-level computation flows in the direction required for the circuit architecture in Fig. 1. It will be noted that an extra column of three cells has been added to each of the dependence graphs in this figure. This is necessary to allow carry bits which are generated at the top left-hand edges of each array to be added to accumulating sum bits. Fig. 6 represents a word-level systolic array whose processing elements have been implemented using straightforward ripple-through parallel multipliers. However, by applying the type of cut illustrated in Fig. 3, it is possible to represent circuits which are pipelined at the bit level. A fully systolic circuit which uses the cuts in Fig. 3 is illustrated in Fig. 7. Note that the bits of b_1 are delayed relative to those of b_0 before being input to the array as one might expect in a systolic implementation.

The inner product processor array in Fig. 1 could also be implemented using serial parallel multipliers and various architectures of this type may be obtained by projecting the bit-level dependence graph of Fig. 6 onto one dimension. For example, a signal flow graph representing a linear chain of serial parallel multipliers of the type illustrated in Fig. 2(a) may be obtained by projecting Fig.

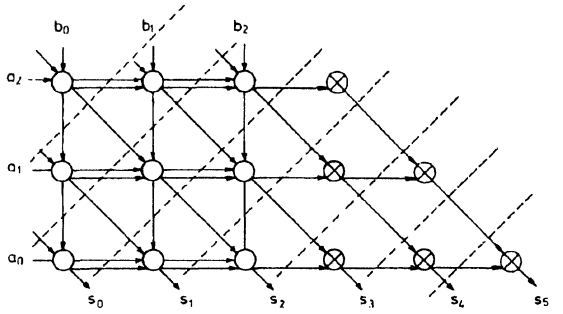


Fig. 5. The application of a valid cut to Fig. 4.

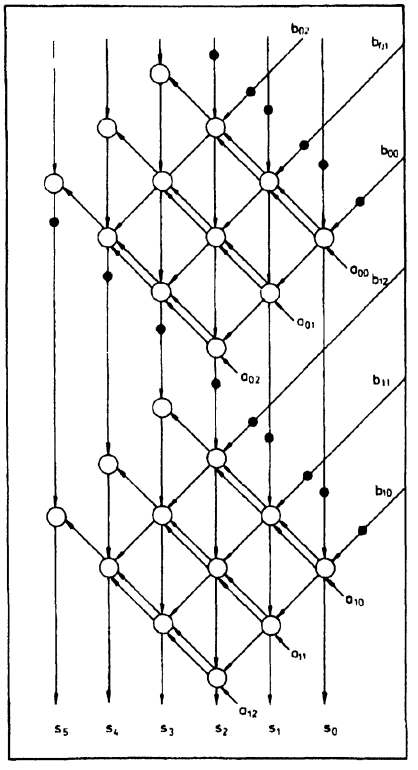


Fig. 6. Dependence graph for inner product computation based on Fig. 2.

6 in the corresponding direction. This projection allows the bits of each number a_k to remain on a fixed processing site as implied by the word-level diagram in Fig. 1. Cuts can then be applied perpendicular to the data lines in this graph to eliminate broadcasting if required.

III. An Alternative Approach

In the last section we assumed, in effect, that the inner product computation would be implemented using an array of multiply/accumulate processors. These were subsequently pipelined at the bit level to increase the overall throughput rate. This approach is often referred to in the literature as "two level pipelining"[11]. An alternative method is to consider the entire problem at bit level from the outset. This leads to some novel architectures and, in

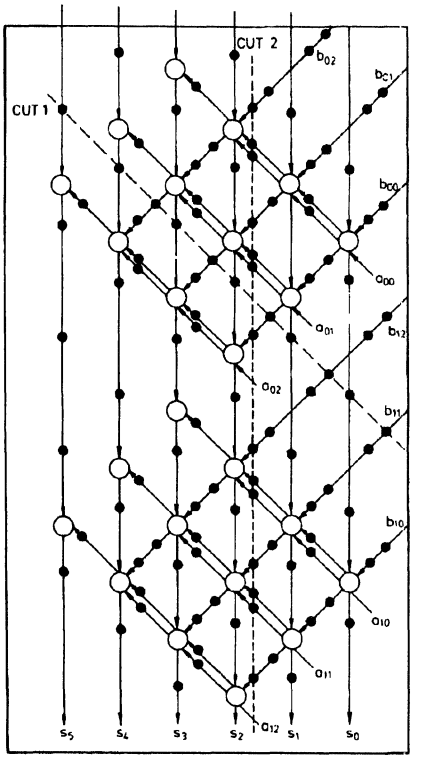


Fig. 7. Application of the cuts shown in Fig. 3 to Fig. 6.

fact, many of the bit-level systolic arrays proposed to date have been designed in this way.

Consider the computation in (1). The pth bit of the resulting inner product may be expressed in the form

$$s_p = \sum_{k=0}^{N-1} \sum_{q=0}^{n-1} a_{k,q}\, b_{k,p-q} + \text{carries}. \tag{5}$$

The traditional approach described in Section II corresponds to summing first over the index q and then over the index k. An alternative architecture may be obtained by reversing the order of summation and expressing the computation of partial product sums

$$s_{q,p-q} = \sum_{k=0}^{N-1} a_{k,q}\, b_{k,p-q} \tag{6}$$

in the single assignment form

$$\bar{s}^{1}_{q,p-q} = 0 \tag{7a}$$

$$s^{k}_{q,p-q} = s^{k-1}_{q,p-q} + a_{kq}\, b_{k,p-q} + \text{carries} \tag{7b}$$

$$s_{q,p-q} = s^{N-1}_{q,p-q}. \tag{7c}$$

The entire computation may then be represented by means of a three-dimensional dependence graph with axes (p, q, k) as illustrated in Fig. 8. Each horizontal plane of this graph represents the interaction of bits between one pair of numbers a_k and b_k. However, there are no diagonal connections within each plane in contrast to the graph in Fig. 2. Instead, there are connections in the vertical direction corresponding to the partial product summation in

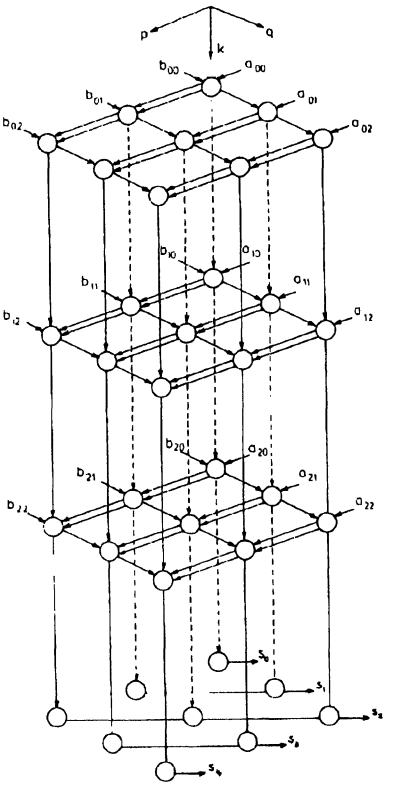


Fig. 8. An alternative dependence graph for inner product computation.

(6). In order to form each bit in the final result, all partial product sums of the same significance must be added together. This is represented by the bottom plane in the dependence graph where the diagonal interconnections correspond to the summation over q in (5). The arrows in Fig. 8 imply a specific ordering of the computation in accordance with (7). Other orderings may of course be derived by writing the basic recurrence in a different form. This will lead, in turn, to modified dependence graphs with arrows pointing in different directions from those in Fig. 8 and will thus give rise to different processor architectures.

Direct implementation of the dependence graph in Fig. 8 would of course require a three-dimensional VLSI circuit technology. A range of two-dimensional signal flow graphs can, however, be generated by projecting Fig. 8 onto various planes. The resulting diagrams can then be cut as required to produce circuits which are fully pipelined and systolic at the bit level. For example, if the axes in Fig. 8 are denoted (p, q, k) and we project onto the plane $(1, 1, 0)$ the resulting signal flow graph can then be cut in the horizontal and vertical directions to produce the bit-level systolic array design shown in Fig. 9 where the bits of a_k and b_k move in opposite directions (McCanny *et al.* [20]). For simplicity, we have omitted the bottom row of cells from Fig. 9. These constitute a simple accumulator which is defined by applying the same projections and cuts to the bottom plane in Fig. 8.

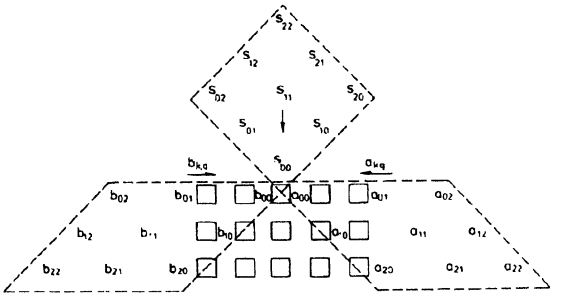


Fig. 9. Bit-level systolic array for the computation of inner products (after McCanny *et al.* [20]).

It is also possible to define an array in which the bits of one set of numbers remain on fixed processing sites by projecting onto either the $(1, 0, 0)$ plane (for fixed a) or the $(0, 1, 0)$ plane (for fixed b). An architecture of this type is shown in Fig. 10 and corresponds to the circuit design reported by Urquhart and Wood [21]. It is interesting to note that the dependence graph in Fig. 8 (omitting the bottom accumulator plane) is equivalent to a word-level dependence graph for matrix × matrix multiplication provided that the individual bits are interpreted as numbers and the processing elements are defined accordingly. Not surprisingly then, the bit-level systolic circuits which can be generated from Fig. 8 have direct analogs with the systolic array circuits generated from matrix × matrix multiply dependence graphs by authors such as Frison *et al.* [22] and Kung [12].

IV. Extension to Matrix × Matrix Multiplication

These ideas discussed above may be extended in a number of ways to other operations such as matrix × matrix multiplication. One approach is to take an established word level architecture such as shown in Fig. 11 (in which the matrices $\boldsymbol{X}$ and $\boldsymbol{Y}$ are multiplied to form the product matrix $\boldsymbol{Z}$ and represent each processing element by a bit-level dependence graph of the type shown in Fig. 2, for example. In this case, each column of Fig. 11 (which performs an inner product computation) would be represented again by the dependence graph in Fig. 6 and similar projections and cuts may be applied to the entire diagram to define the various hardware structures. This procedure serves to define all delays which are required to ensure that the movement of every bit in each element of the $\boldsymbol{X}$ and $\boldsymbol{Y}$ matrices is properly synchronized no matter what level of pipelining is incorporated. Note that the graph in Fig. 11 represents an orthogonal array but has been deliberately skewed for consistency with the bit-level inner product computation array described in Fig. 1.

An alternative approach, once again, is to start with a dependence graph which describes the entire computation at bit level before introducing any form of pipelining. For example, Fig. 12 depicts word-level dependence graph required for matrix × matrix multiplication. If each col-

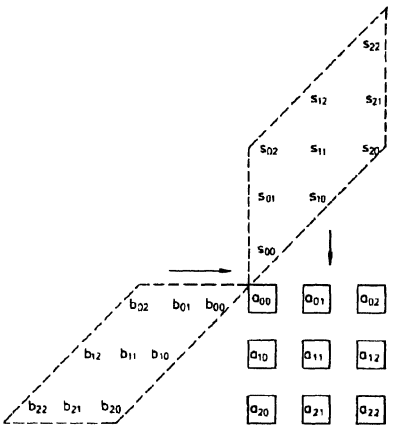


Fig. 10. Bit-level systolic array for the computation of inner products (after Urqhart and Wood [21]).

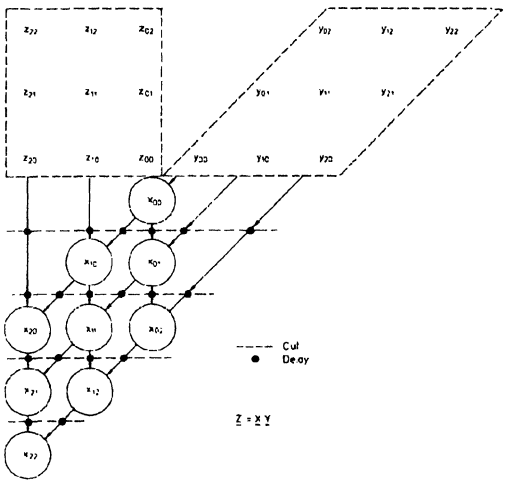


Fig. 11. Word-level systolic array for matrix × matrix multiplication.

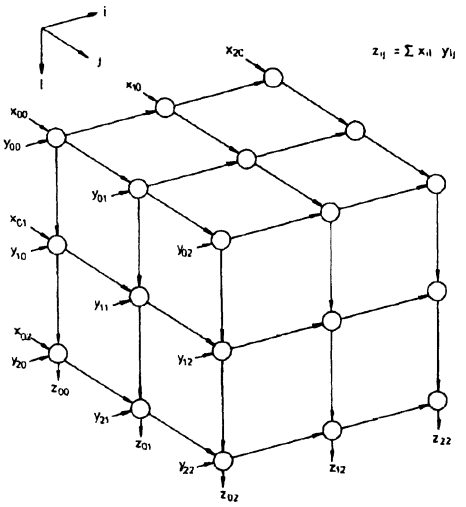


Fig. 12. Dependence graph for matrix × matrix multiplication implemented at word level.

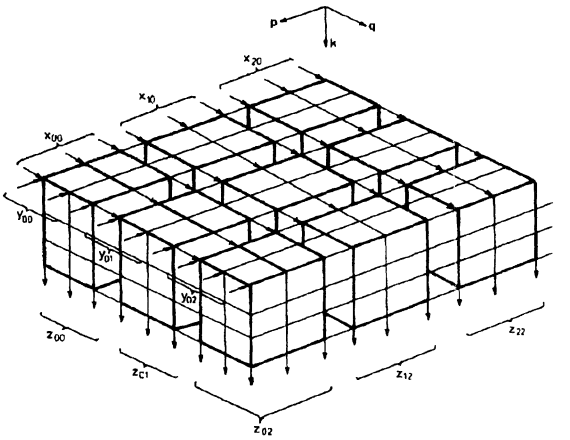


Fig. 13. Dependence graph for matrix × matrix multiplication implemented at bit level. In this figure we assume that each column of nodes in Fig. 12 has been replaced with the bit-level dependence graph shown in Fig. 8.

umn of this graph (which represents the inner product of two vectors) is replaced by the dependence graph in Fig. 8 then a rather complex three-dimensional graph is obtained which describes both the word-level and bit-level data interactions (Fig. 13). Similar projections and cuts to those described for Fig. 8 can be applied to this graph. Projecting onto the (1, 1, 0) plane and cutting the resulting signal flow graph both vertically and horizontally leads to the bit-level matrix × matrix multiplier proposed by McCanny *et al.* [20]. This is simply an expanded version of Fig. 9 in which more than one set of inner product computations occurs simultaneously. Projecting the graph onto the (0, 1, 0) or (1, 0, 0) planes leads once again to an array in which one set of coefficients remains static. In this case, the resulting matrix multiplier takes the form of a cascade of inner product arrays of the type shown in Fig. 10 and corresponds to the architecture described by Urqhart and Wood [21].

V. Discussion

As illustrated in the previous sections, the dependence graph and cut-set procedure developed by Kung [12] for describing systolic arrays at the word level can readily be extended to computations at the bit level. This approach is both simple and intuitive to use with the advantage that it allows many types of bit-level circuit to be described using the same graph. No supposition is made initially as to whether a design will be ripple through, partly pipelined, or fully systolic. It therefore provides a general tool which enables the VLSI chip designer to examine the relative tradeoffs between levels of pipelining, chip area, power consumption, latency, and throughput rate for a given application. It is important to retain this degree of flexibility within the design process since, for example, the effect of introducing pipeline delays is to increase the latency of a given circuit and this may be unacceptable for some applications even in signal processing. The approach developed in this paper may be also used to map

a specificied function onto other types of processor where computations are performed at the bit level. Examples include SIMD architectures such as the ICL DAP or the NCR GAPP machine.

Finally, it is interesting to compare this approach to bit-level systolic array design with the technique which we discussed in a previous paper based on the use of vector constraint equations as proposed by Li and Wah [23]. One advantage of the latter method is that the constraint equations, being vectorial in nature, do not imply a preferred ordering of computation and therefore describe a broader range of systolic architectures. One the other hand, designs generated using the dependence graph and cut-set procedure are highly dependent on the order of computation defined within the initial dependence graph. One disadvantage of the constraint equation approach is that it applies only to systolic circuits and not to corresponding ripple-through or partly pipelined circuits. Furthermore, it may be difficult to find all the solutions to the constraint equations and thus explore the range of architectures which may be derived.

References

[1] J. Blackmer, G. Franks, and P. Keukes, "A 200 million operations per second (MOPS) systolic processor," *Proc. SPIE Int. Soc. Opt. Eng.*, pp. 10-18, 1981.

[2] K. Bromley, J. J. Symanski, J. M. Speiser, and H. J. Whitehouse, "Systolic array processor developments," in *CMU Conf. VLSI Syst. Computations*, H. T. Kung, B. Sproull, and G. Steele, Eds. Computer Science Press, 1981, pp. 273-284.

[3] C. Ward and E. Davie, "The application and development of wavefront array processors for advanced front-end signal processing systems," in *Systolic Arrays* W. Moore, A. P. H. McCabe, and R. B. Urqhuart, Eds. Adam Hilger, 1987, pp. 295-301.

[4] J. W. McCanny and J. G. McWhirter, "Some aspects of systolic array research in the U.K.," *IEEE Comput. Mag.*, pp. 51-63, July 1987.

[5] J. C. White, J. V. McCanny, A. P. H. McCabe, J. G. McWhirter, and R. A. Evans, "A high speed CMOS/SOS implementation of a bit level systolic correlator," in *Proc. ICCASP '86* (Tokyo, Japan), 1986, pp. 1161-1164.

[6] R. A. Evans, D. Wood, K. Wood, J. V. McCanny, J. G. McWhirter, and A. P. H. McCabe, "A CMOS implementation of a multibit convolver chip," *VLSI'83*, 1983, pp. 227-235.

[7] J. S. Ward, J. V. McCanny, and J. G. McWhirter, "Bit level systolic array implementation of the Winogard Fourier transform algorithm," *Proc. Inst. Elec. Eng.*, pt. F, vol. 132, no. 6, pp. 473-47927, 1985.

[8] D. Healey, C. James, and J. Ward, "A bipolar systolic array for an eight point Winograd transform," presented at the IEEE/IEE Int. Workshop Systolic Arrays, University of Oxford, England, July 1986.

[9] Marconi Electronic Devices Ltd., "Signal stream product family," Preliminary Data Sheets, Nov. 1986.

[10] J. V. McCanny and J. G. McWhirter "The derivation and utilisation of bit level systolic array architectures," in *Systolic Arrays*, W. Moore, A. P. H. McCabe, and R. B. Urqhuart, Eds. Adam Hilger, 1987, pp. 47-59.

[11] H. T. Kung and M. Lam, "Fault tolerance and two level pipelining in VLSI systolic arrays," in *M.I.T. Conf. Advanced Research VLSI*, Jan. 1984, pp. 74-83.

[12] S. Y. Kung, *VLSI Array Processors*. Englewood Cliffs, NJ: Prentice Hall, 1988.

[13] TRW, "Multiplier accumulator," notes.

[14] D. J. Myers, "Multipliers for LSI and VLSI signal processing applications," M.Sc. Project Rep., MSP5 Edinburgh Univ., Edinburgh, U.K., Sept. 1981.

[15] M. Sheeran, "Designing regular array architectures using higher order functions," LNCS 201 (Functional Programming and Computer Architectures), 1985, pp. 22-237.

[16] J. Hoestra, "Systolic multiplier," *Electron. Lett.*, vol. 20, 1985, pp. 995-996.

[17] P. Danielson, "Serial/parallel convolvers," *IEEE Trans. Comput.*, vol. C-33, 1984, pp. 652-657

[18] R. F. Lyon, "Two's complement pipelined multipliers," *IEEE Trans. Commun.*, vol. COM-24, 1976, pp. 418-424.

[19] J. V. McCanny and J. G. McWhirter, "On the implementation of signal processing functions using one bit systolic arrays," *Electron. Lett.*, vol. 18, no. 6, 1982, pp. 241-243.

[20] J. V. McCanny, K. W. Wood, J. G. McWhirter, and C. J. Oliver, "The relationship between word and bit level systolic arrays as applied to matrix × matrix multiplication," *Proc. SPIE Int. Soc. Opt. Eng.*, vol. 495, pp. 114 120, 1984.

[21] R. B. Urqhart and D. Wood, "Systolic matrix and vector methods for signal processing," *Proc. Inst. Elec. Eng.*, pt. F, vol. 131, pp. 632-637, 1984.

[22] P. Frison, P. Gachet, and P. Quinton, "Designing systolic arrays with diastol," in *Proc. IEEE Workshop VLSI Signal Processing* (Los Angeles, CA), Nov. 1986.

[23] G. J. Li and B. W. Wah, "The design of optimal systolic arrays," *IEEE Trans. Comput.*, vol. C-34, pp. 418-424.

Matrix Computations on Systolic-Type Meshes

An Introduction to the Multimesh Graph Method

Jaime H. Moreno and Tomás Lang
University of California, Los Angeles

Matrix computations, characterized by having matrix operands and/or results, are a frequently used mathematical tool in modern scientific and engineering applications. For example, in a review of parallel-processing algorithms and architectures for real-time signal processing, Speiser and Whitehouse[1] indicated that the major computational requirements for many important real-time signal-processing tasks (such as adaptive filtering, data compression, beam-forming, and cross-ambiguity calculation) can be reduced to a common set of basic matrix computations. This set includes matrix-vector multiplication, matrix-matrix multiplication and addition, matrix inversion, solution of linear systems, eigensystems solution, matrix decompositions (LU-decomposition, QR-decomposition, and singular-value decomposition [SVD]), and the generalized SVD algorithm.

Matrix operations of the type mentioned above are computation intensive. Consequently, they require high computing rates to achieve acceptable execution speed and to meet the time constraints of many applications. Thus, parallel algorithms and architectures are often needed.[2] As Figure 1 shows, several classes of parallel architectures have been used for matrix computations. Vector computers, for example, exploit parallelism through vector instructions, extracted from sequential programs by vectorizing compilers. Array computers use a similar type of parallelism. Multiprocessor systems, on the other hand, exploit parallelism at several levels, including vector operations, concurrent execution of several loop iterations, and block methods. Because of their importance, matrix computations have become one of the preferred benchmarks for parallel architectures.[3]

Systolic-type arrays use both the fine-grain parallelism and the regularity of matrix computations effectively. The multimesh graph method for deriving these arrays is systematic, flexible, and easy to use.

Although the above-mentioned parallel architectures have demonstrated their effectiveness for executing matrix computations, they suffer from several degradation factors. These arise from the relative general-purpose character of the machines and from the need to adapt the algorithms to the specific hardware. Moreover, their general-purpose nature makes it necessary to include features that increase cost (for example, complex memory-addressing schemes) and also make the architectures less suitable for very large scale integration (VLSI) and wafer-scale integration (WSI) technology (broadcasting or complex interconnection networks, for example). These drawbacks led to the introduction of application-specific architec-

Reprinted from *IEEE Computer*, pp. 32-51, April 1990.

tures and, in particular, of systolic-type arrays, which are natural for matrix computations because they match the fine granularity of parallelism available in the computations and have very low overhead in communication and synchronization. (This contrasts with dataflow computers, which also use fine-grain parallelism but have high overhead.) In addition, the regular nature and nearest-neighbor connections of systolic-type arrays meet the requirements for effective use of VLSI and WSI.[4]

This article focuses on the execution of matrix computations on systolic-type arrays in an application-specific environment. We first present an extension to the concept of a systolic cell by incorporating a small, fixed amount of storage inside the cells, and we discuss the trade-offs this storage gives rise to. Then we review different approaches to decomposing (partitioning) large problems, highlighting their bandwidth requirements and their capabilities for using the storage in the cells. Finally, we discuss the basic characteristics of methods for the design of systolic-type arrays, describe the multimesh graph (MMG) design method, and illustrate its application to the transitive closure algorithm. Other examples, including LU-decomposition, QR-decomposition, and the Faddeev algorithm, are given elsewhere.[5]

Although applying the MMG method to the examples indicated above has produced arrays whose cells are simpler and better utilized than those in previously proposed structures for the same algorithms, this article concentrates on the method's capabilities rather than on the arrays obtained.

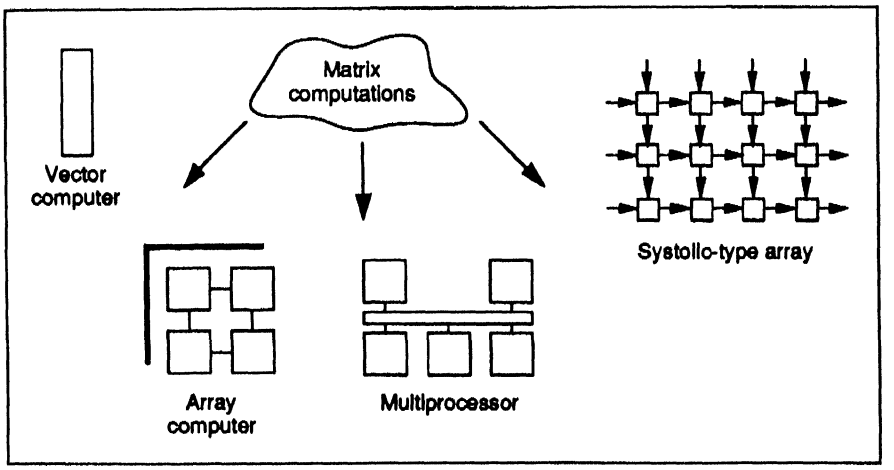


Figure 1. Parallel architectures for matrix computations.

Algorithms and systolic-type arrays

Several algorithms may exist for a given computation. Some of these algorithms are suited for sequential execution (that is, in a single processor), while others are better suited for particular types of parallel architectures. Matrix computations have properties that make them attractive for all the architectures mentioned earlier, and many algorithms have been developed for the different classes. For execution in systolic-type arrays, the algorithm should exhibit sufficient fine-grain parallelism. This is often characteristic of traditional algorithms used in sequential computers, but in other cases special algorithms have been developed.[6]

Depending on their range of applicability, there are two types of application-specific arrays: algorithm specific and class specific. Algorithm-specific arrays can execute only one particular algorithm. As Figure 2a illustrates, these structures may be the most appropriate solution for some applications because they offer the possibility of matching an array to a given algorithm and of fulfilling specific implementation requirements (such as speed, size, and power consumption). If the application consists of the matrix operation and other computations, the array should be combined with other modules to perform the complete task, composing a heterogeneous system. In addition, such an array is usually connected to a host that performs I/O and control functions. The realization of an algorithm-specific array consists of determining

- the topology of the array (triangular, linear, rectangular, etc.),
- the functionality of each processing element,
- the schedule of operations and data transfers, and
- the I/O schedule.

By contrast, in some situations the set of matrix computations is varied and not even predefined. In these cases a better solution is a class-specific array, that is, a general array that can be adapted (programmed) for a class of matrix algorithms, as Figure 2b illustrates. Programming a class-specific array consists of determining

- the assignment of operations to cells,
- the schedule of operations and data transfers, and
- the I/O schedule.

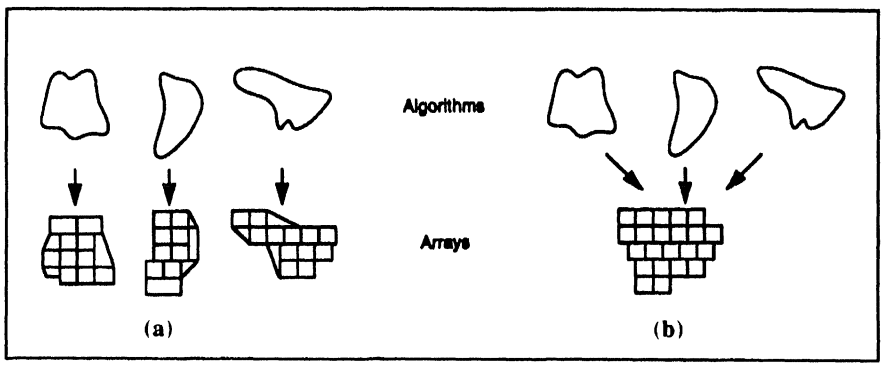


Figure 2. Classes of application-specific arrays for matrix computations: (a) algorithm specific, (b) class specific.

Glossary

Matrix computation — A computation having matrix operands and/or results.
Fine-grain parallelism — Parallelism at the level of basic (arithmetic) operations.
Array processor — A network of processing elements (PEs) that execute the same operation, synchronously, on different data elements; the operation is broadcast to all PEs.
Array (processor array) — A hardware network of processing elements with nearest-neighbor communication (no broadcasting).

Cell — A processing element.
Systolic-type array — A two-dimensional processor array with cells of similar complexity connected in regular nearest-neighbor manner, synchronized dataflow, and external I/O only at boundary cells.
Mesh array — A systolic-type array with cells connected by unidirectional orthogonal nearest-neighbor links.
Systolic cell — A cell with no local storage.
Pseudosystolic cell — A cell with a small, fixed amount of local storage.
Local-access cell — A cell with storage proportional to the size of problems.
Pipelined cell — A cell capable of performing several operations simultaneously by pipelining execution of operations through several computing stages within the cell.
Stage time — The time taken by a stage of a pipelined cell.
Optimal utilization of cells — A new operation is initiated in every computing cycle.

Application-specific array — An array designed for specific purposes (in contrast with general-purpose architectures, which are designed for [almost] any purpose).
Algorithm-specific array — An array designed for one algorithm.
Class-specific array — An array suitable for a class of selected algorithms.

Realization of an array — The design of an application-specific array for a given algorithm or class of algorithms (topology, functionality of cells, scheduling, data transfers, data I/O).
Mapping onto an array — Determining the execution of an algorithm on a predefined array (operations per cell, scheduling, data transfers, data I/O).

Partitioning — Decomposing a large problem for execution on a small array.

Both algorithm-specific and class-specific cases should meet the requirements of a particular application and should optimize relevant criteria. Consequently, the design of an algorithm-specific array and the programming of a class-specific structure have many aspects in common, and similar techniques can be used for both activities. We refer to these tasks as realization and mapping, respectively. In this article we concentrate on the realization of application-specific arrays for matrix algorithms. Mapping is discussed extensively elsewhere.[5]

Systolic and pseudosystolic mesh arrays

Let's examine the properties of an architectural model consisting of a mesh of processing elements (cells) with unidirectional orthogonal links. We refer to this architecture as a mesh array (see examples in Figure 3). These arrays are hardware networks of processing elements with the following basic characteristics:

- linear or two-dimensional structures with cells connected in nearest-neighbor manner (a linear array with K cells is a mesh of dimension K by 1);
- external I/O from a host only at the boundaries of the array;
- unidirectional communications between cells, that is, dataflow from cell to cell in one direction only, without data counterflow; and
- local communications only, that is, no capability for broadcasting or routing data through cells without using that data.

Analysis of a large class of matrix algorithms has shown them to consist of primitive operations with up to three operands and up to two results, as Figure 4a illustrates. Since cells of linear and two-dimensional mesh arrays have only two input and two output ports, the third operand is obtained from a feedback loop within the cell (Figure 4b). So, for ternary operations (those requiring three operands), two sources of data are off-cell and the third source is a feedback loop within the cell. On the other hand, the outputs from a cell can be either results computed within the cell or operands used in the cell and passed through without modification (that is, *transmitted data*[7]).

We assume that the execution time is the same for all operations and that the stage time is the same in pipelined cells. These assumptions, customarily used for the design of application-specific arrays, are highly implementation dependent.

In terms of the communication band-

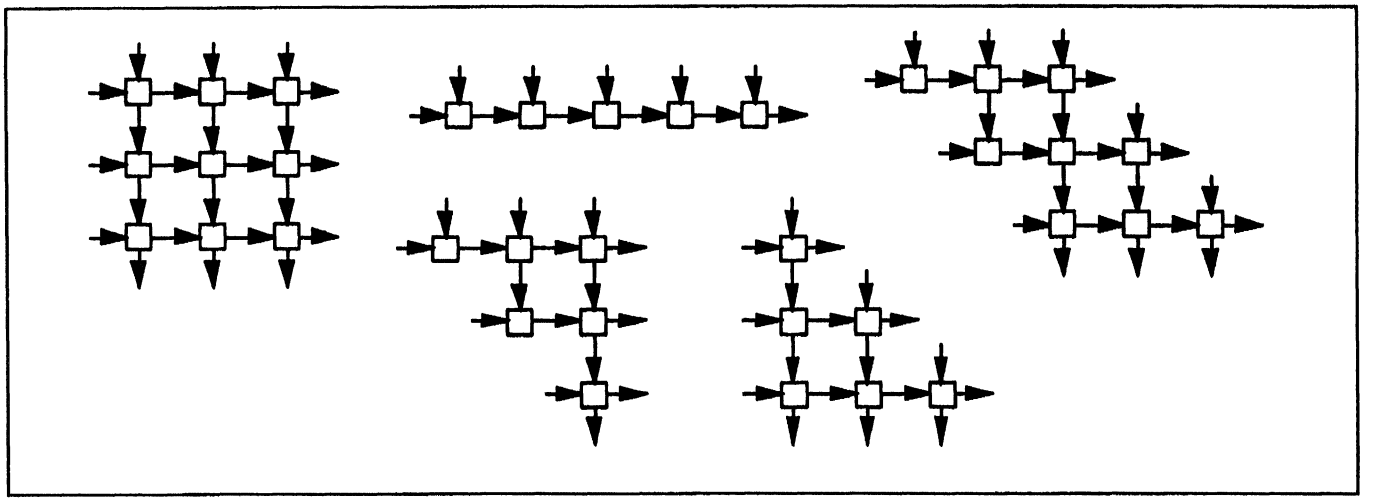


Figure 3. Examples of mesh arrays.

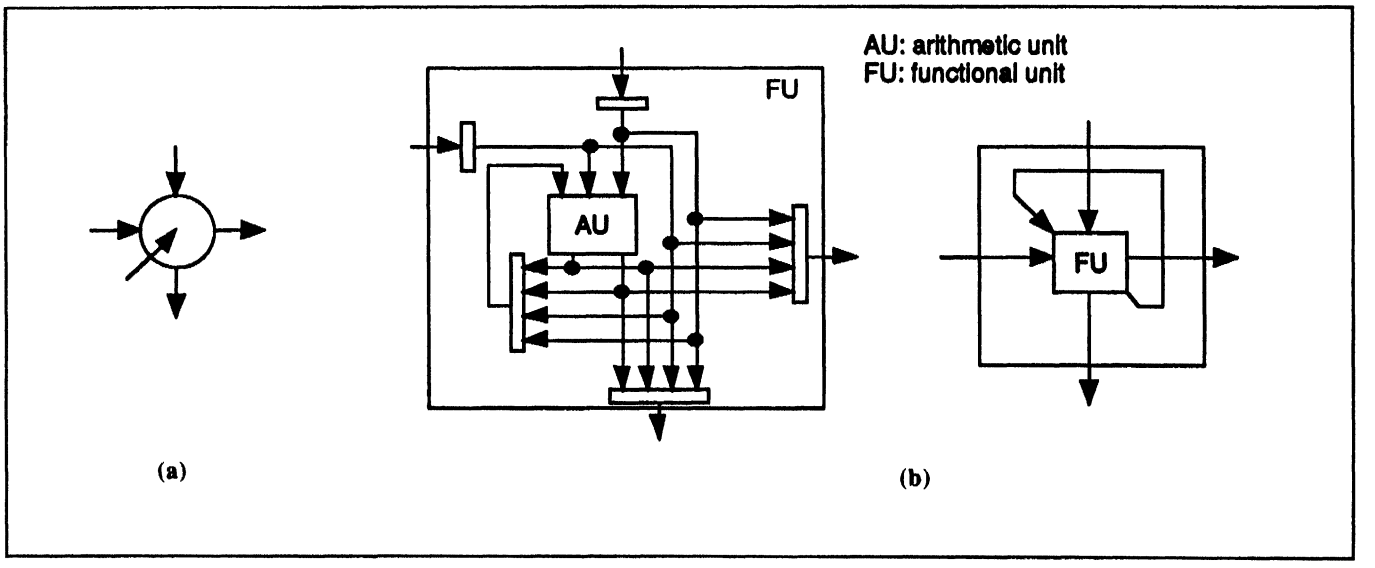


Figure 4. Primitive operation (a) and cell (b).

width, we consider three types of cells:

(1) *Systolic cell* — a cell with no local storage except for registers used to latch input operands (Figure 5a). Data flows through cells, so every operation in each cell requires one data transfer per data source. Consequently, the communication rate is the same as the computation rate of cells.

This type of cell is suitable for implementation in wafer-scale-integration technology because an entire array can be placed on a single wafer and there is no need to go off-wafer for communicating between cells. This is in contrast to very large scale integration technology, where only a few cells are placed on a chip, so that it is necessary to go off-chip to communicate between cells. The off-chip transfers degrade speed because of the lower bandwidth.

(2) *Pseudosystolic cell* — a cell with a small, fixed amount of storage (the amount of storage is independent of the size of problems to be solved in the array). This storage comprises two separate FIFO buffers, one for the vertical flow and the other for the horizontal flow of data through the array. Figure 5b depicts a pseudosystolic cell with its ports and buffers. The ports and/or local storage provide two sources of data for every operation; the third source is a feedback loop within the cell.

Since the size of the FIFO buffers is fixed and small, we assume that access time to this local storage matches the execution rate of the functional unit (that is, cell pipeline stage time or functional unit time) and that it is shorter than the time needed to transfer data between cells. This property is exploited by performing successive operations with data from the buffers, without accessing the ports. Consequently, pseudosystolic cells need not

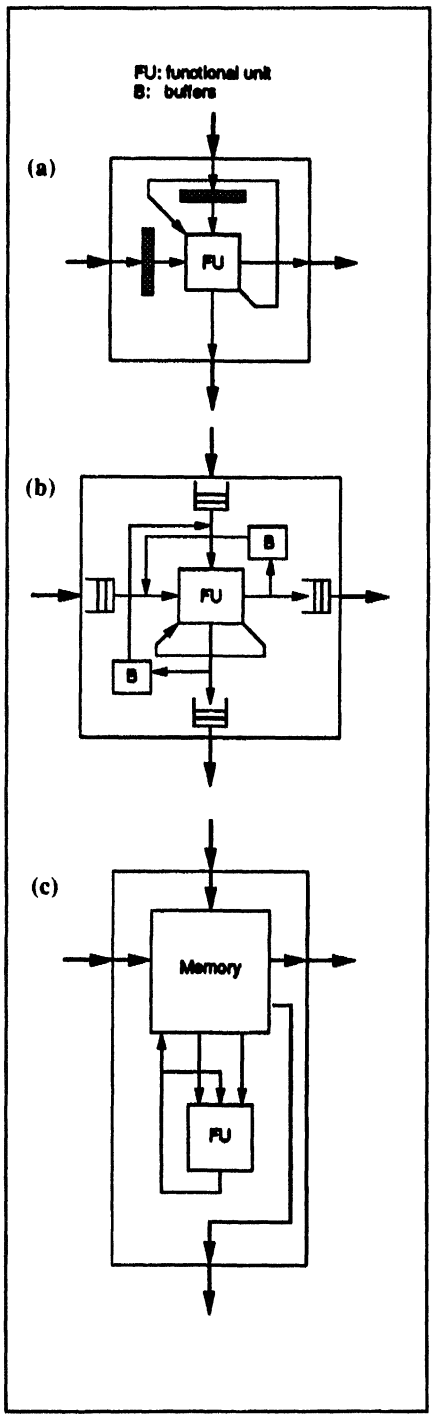


Figure 5. Different types of cells: (a) a systolic cell, (b) a pseudosystolic cell, (c) a local-access cell.

receive data through the ports at every cycle, so the communication bandwidth of pseudosystolic cells is lower than their computation rate. This lower communication rate is adjusted to the cell computation rate by FIFO queues attached to the ports.

Pseudosystolic cells are suitable for implementation as one cell per chip because they have only a small amount of local memory and the off-chip communication rate is lower than the on-chip computation rate. The amount of storage determines the relation between these rates, as we'll see later.

(3) *Local-access cell* — a cell with storage space proportional to the size of problems to be solved in the array (Figure 5c). Operations are performed in each cell with up to two operands obtained from local storage, so that data received from neighbor cells is stored before it is used. Another source of data is the feedback loop within the cell.

Local-access cells have sufficient memory to store a large portion of the data locally and to reduce communications between cells. Consequently, the communication rate is much lower than the computation rate (much less than one word per port per time-step). Local-access cells are suitable for implementation at the board level because they require a large local memory.

The remainder of this article will focus on systolic and pseudosystolic cells. A discussion on the use of local-access cells in arrays for matrix computations appears elsewhere.[5]

The model of computation used in mesh arrays consists of the synchronized flow of data through cells (Figure 6a), with operations performed in each cell. At each time-step, a cell reads operands from input ports, local storage, and/or the feedback loop; performs an operation; and delivers results to output ports, local storage, and/or the feedback loop. For pipelined cells the model of computation is similar, except that the results delivered to ports, the feedback loop, and local storage are from an operation previously initiated in the pipeline.

Size of problem and array

The relative size of the matrices and of the array significantly affects the design and operation of the computing structure. Two different cases can be identified. When the matrix size is fixed, the array can be designed to use the maximum parallelism achievable. When the matrix is much larger (its size may not even be predefined) than a cost-effective array, the computation is partitioned into subproblems, which are executed in sequence on the array (partitioned problems[8]). Consequently, the array is used many times while operating to solve a single large problem.

The model of computation described earlier is suitable for fixed-size algorithms executed repeatedly with different sets of input data (multiple-instance algorithms) or for partitioned algorithms. In either case an instance (or subproblem) can use a cell during several time-steps, and various instances (or subproblems) can execute concurrently throughout the array. Figure 6b depicts the flow of several instances through a mesh array.

There are different approaches to partitioning a problem, depending on the type of cells used. The approach suitable for systolic and pseudosystolic cells is a two-level scheme wherein the algorithm is decomposed into subproblems and each subproblem is decomposed into components. A subproblem is mapped onto the entire array, and each component is mapped onto a different cell. Subproblems are executed in pipelined fashion, according to a certain schedule. This sequential execution requires feedback of data and memory external to the array, but little or no storage inside the cells (depending on the components of the subproblems, to be described later). This approach produces good load balancing.

As an example, Figure 7 illustrates an algorithm partitioned into subproblems whose components exhibit a rectangular communication pattern (except at the boundaries of the algorithm). Consequently, subproblems are executed in a rectangular array. This type of partitioning is known as *cut-and-pile*.[8] It has also been referred to as *locally parallel globally sequential partitioning*.[7]

On the other hand, the approach suitable for local-access cells partitions the entire algorithm into a number of subproblems equal to the number of cells in the target array, and it maps each subproblem onto one cell. As a result, the dependencies among the subproblems should match the interconnection structure of the array, cells must have a large amount of local storage (enough to store all data for the corresponding subproblem), and cells need low bandwidth. However, this scheme requires a careful selection of subproblems to achieve good load balancing.

Figure 8 shows this technique, wherein an algorithm is partitioned into a number of communicating subproblems that are mapped onto the array. This type of parti-

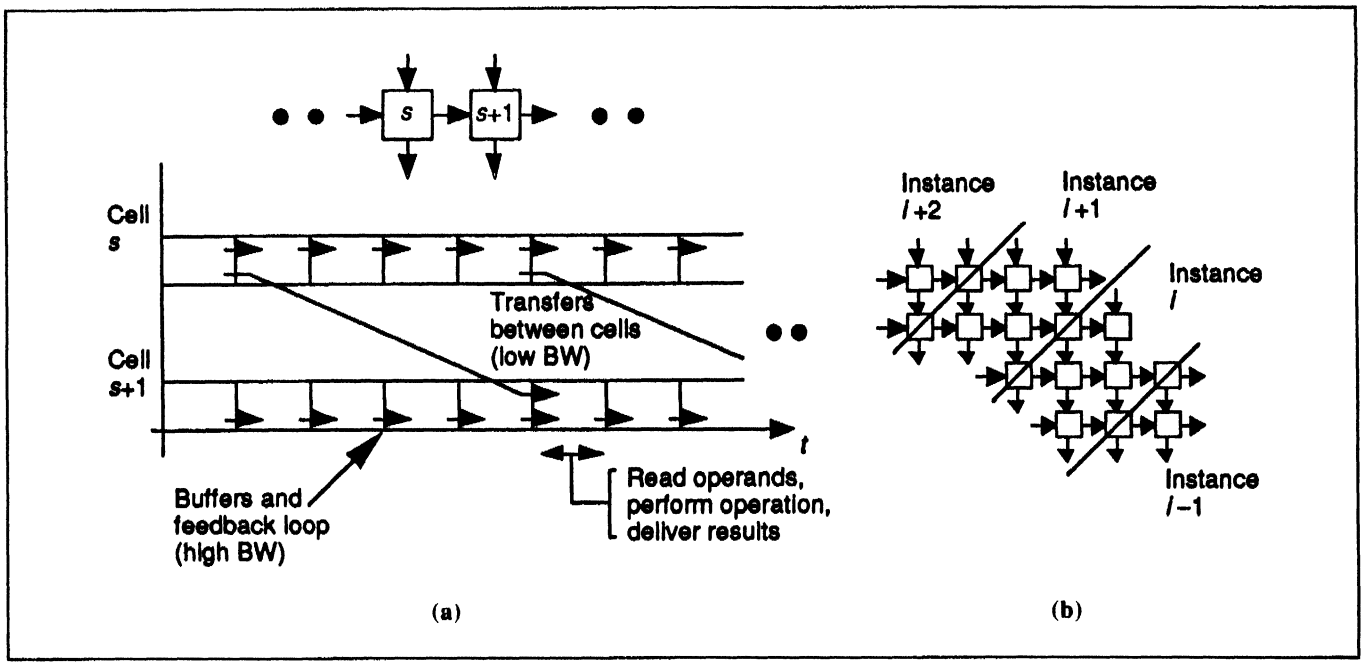


Figure 6. Computational model of mesh arrays: (a) flow of data per cell, (b) flow of data in an array.

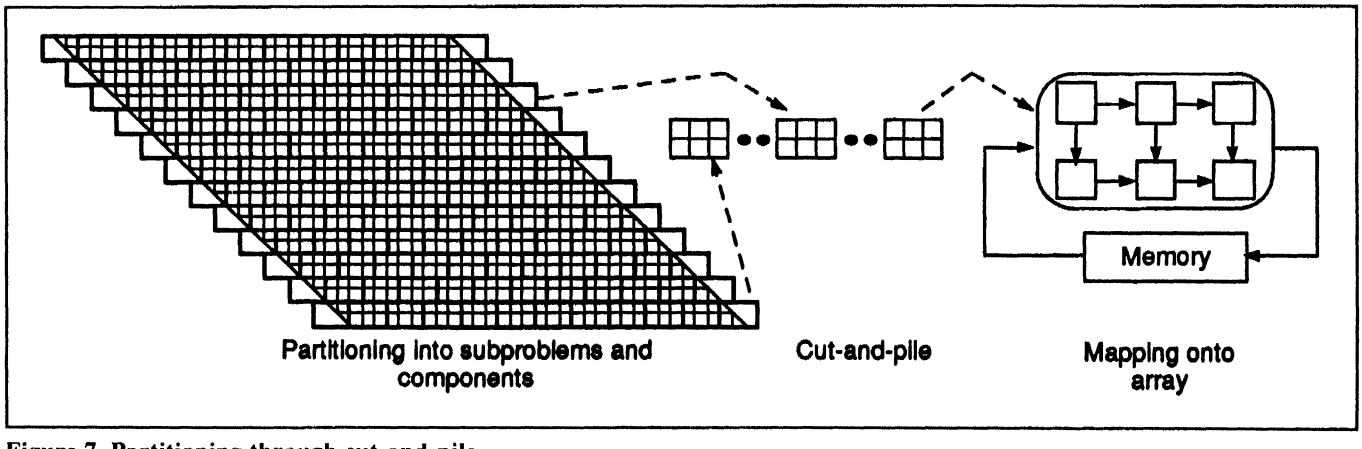


Figure 7. Partitioning through cut-and-pile.

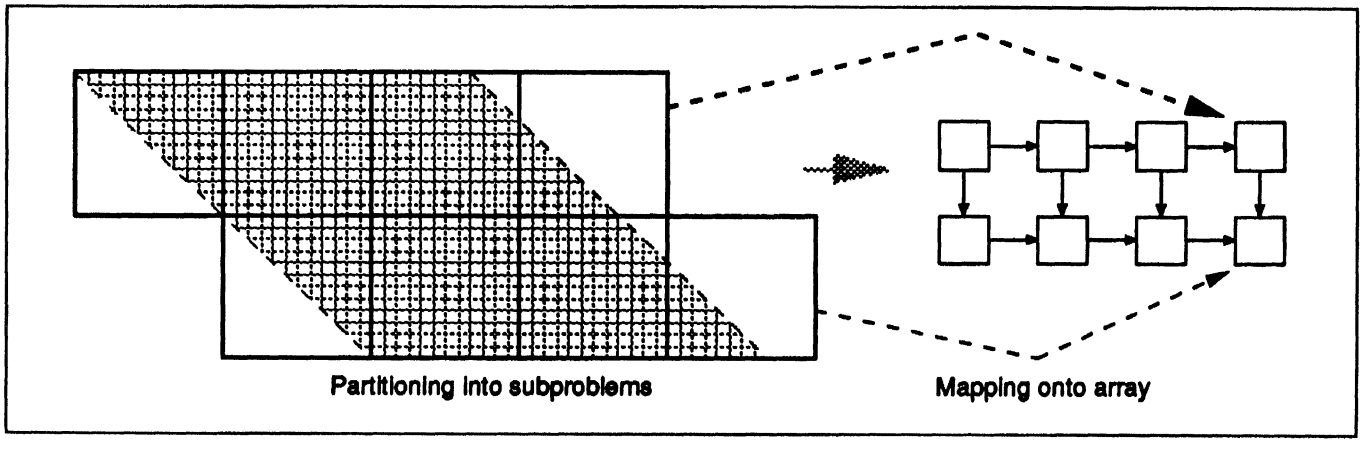


Figure 8. Partitioning through coalescing.

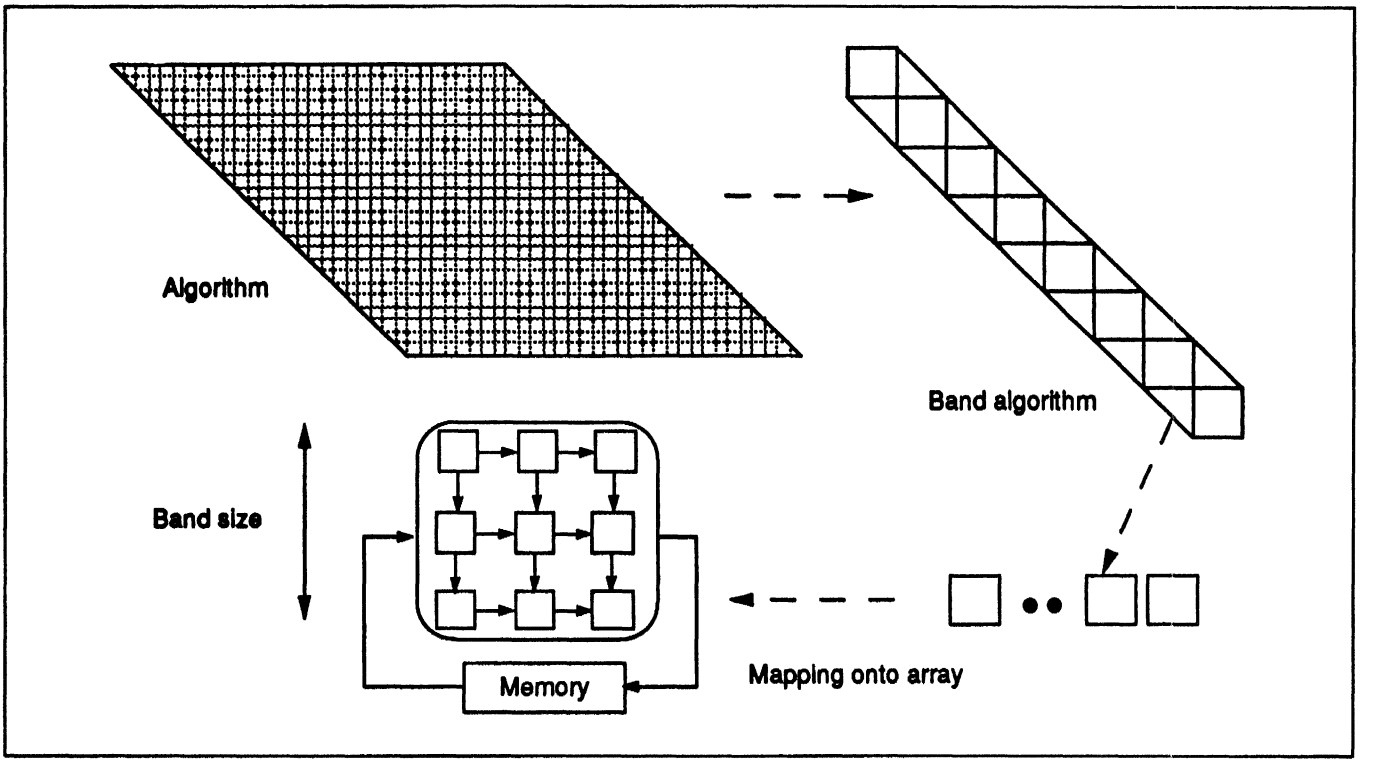


Figure 9. Partitioning through decomposition into subalgorithms.

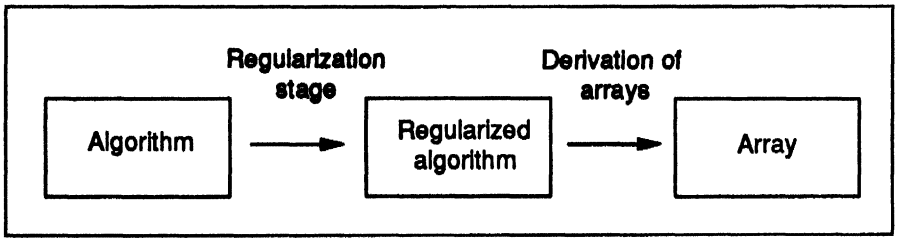


Figure 10. The stages in a design method.

tioning is known as *coalescing*.[8] It has also been referred to as *locally sequential globally parallel partitioning*.[7]

A different partitioning strategy, shown in Figure 9, was proposed by Navarro et al.[8] In this case an algorithm with large, dense matrices is transformed into an algorithm with band matrices and computed in an array tailored to the band size. This approach has the potential for high performance when applicable but is less general than the schemes discussed above, because the decomposition depends on the algorithm.

Characteristics of array design methods

Several techniques have been proposed for the design of arrays, as reviewed by Fortes et al.[9] The most successful approach has been a transformational paradigm, wherein the description of an algorithm is successively transformed and made suitable for implementation. Let's examine some of the important issues regarding transformational methods for the design of application-specific arrays.

Stages in a transformational method. We identify two stages in the application of a transformational design technique: regularization, and derivation of arrays (see Figure 10).

Regularization is the derivation of a regularized representation of an algorithm from an initial admissible form. This regularized form must provide an implicit or explicit description of parallelism in a manner suitable for implementation in arrays. Moreover, the regularized representation must be in a form suitable for manipulation in the remaining steps of the method.

The second stage, derivation of arrays, uses the regular description obtained above and determines the topology and structure of the array, as well as the characteristics of cells, the flow of data, and the I/O.

The requirements and characteristics associated with each stage allow us to precisely define and compare the capabilities of different techniques. Within this framework the features of a design method can be stated in terms of specific factors. In the regularization stage these factors are

- the class of algorithms to which the

method can be applied, that is, the generality of the initial admissible form of the algorithms;
- the completeness and simplicity of the transformations used to produce a regular description; and
- the effectiveness of the regular description in conveying the properties of an algorithm in a form suitable for implementation in arrays.

In the derivation-of-arrays stage these factors correspond to the capabilities and simplicity of the transformations used to derive an array from the regularized form. In particular, they include the capabilities of the transformations to

- incorporate implementation constraints and restrictions, such as limited local storage and limited bandwidth, into the design;
- incorporate different attributes of the processing elements, such as pipelining, nonconventional arithmetic, and specialized functional units;
- perform optimization of specific performance measures as part of the design process;
- design arrays for fixed-size data and partitioned problems; and
- realize algorithm-specific arrays and map algorithms onto class-specific arrays.

An additional factor, applicable to both stages, is suitability for automation.

Representation of regularized algorithms. Transformational approaches differ. Among these differences are the way a regularized description is obtained and represented and the capabilities associated with the description. In other words, methods differ in the notation used to represent a regularized form and in the suitability of the notation to perform transformations to derive an array. This notation determines the simplicity of the methods as well as the guidance provided to select suitable transformations, as discussed below.

Abstract notations and/or lack of simple systematic transformations produce techniques that hide important properties of algorithms and implementations, in many cases leading to inadequate conclusions regarding an algorithm's features and their suitability for a particular array. An example is the use of algebraic expressions in H.T. Kung's pioneering work on systolic arrays.[6] Kung concluded that "LU-decomposition, transitive closure, and matrix multiplication are all defined by recurrences of the same 'type.' Thus, it is not coincidental that they are solved by similar algorithms using hexagonal arrays." Moldovan[10] made a similar statement when describing an algebraic design approach. However, it has been shown that the algorithms for these computations have quite different dependency structures, so that they are mapped efficiently only onto different arrays.[5]

The two most popular types of representation are algebraic expressions and graphical descriptions.[9] In algebraic-based methods the regularized description is given as a set of algebraic expressions, and transformations are applied to these expressions to obtain an implementation. Research by Rao and Kailath[11] provided a unifying framework for many of the algebraic-based approaches, which basically are all techniques derived from Karp, Miller, and Winograd's work.[12]

A different line of research uses graphical notations to describe an algorithm. Examples are the signal flow graph method[7] and our multimesh graph method.[5] Graph-based methods start by representing an algorithm as a graph. Transformations are applied on the graph, as part of the regularization stage, to render it more suitable for later design steps. The regularized graph is then mapped (projected) onto an array either directly or through other intermediate representations.

Many design techniques — algebraic-based approaches in particular — have not provided specific tools to obtain the corresponding regularized representation. Instead they have assumed that this representation is already available. In cases where some attention has been given to the regularizing process, the proposed techniques are either ad hoc or heuristic, and the results obtained are not satisfactory. In other words, proposed methods have addressed only the second design stage and have largely ignored the first.

For some simple algorithms, such as matrix multiplication, finding the regularized version (for example, a regular iterative algorithm[11] or a uniform recurrence equation[13]) is straightforward, so that the lack of a systematic procedure is not an issue. However, simple algorithms are relatively few; most matrix algorithms of importance (LU-decomposition, QR-decomposition, transitive closure, Gaussian elimination, and the Faddeev algorithm, for example) are not easily described in those regularized forms. Moreover, the regularized form is often more complex than the original algorithm, perhaps having additional operations. Specific examples of these issues are given elsewhere.[5]

Limitations of methods in the derivation of arrays. Each of the previously proposed methods has certain limitations regarding the derivation of arrays. A method may

- be oriented toward the design of systolic arrays only, that is, arrays of cells with no local storage and high bandwidth;
- assume that all cells in an array are identical;
- have predefined characteristics of cells and therefore be unable to incorporate other implementation constraints or restrictions, such as type of cells and I/O bandwidth, into the design;
- be unable to analyze trade-offs among implementation parameters such as amount of local storage and communication bandwidth of processing elements; or
- produce arrays with suboptimal cost and/or performance.

The limitations in regularization and realization discussed above motivated development of the multimesh graph method.

The multimesh graph method

The class of matrix algorithms suitable for the multimesh graph (MMG) method is described recursively by an outermost loop and a loop body that contains scalar, vector, and matrix operators, as well as other matrix algorithms (see Figure 11). A sequence of these algorithms is also a matrix algorithm. The operators have the following characteristics:

(1) Scalar, or primitive, operators (such as add, multiply, rotation, and sine) are basic unary, binary, or ternary operations that can produce up to two outputs and whose computation time is data independent. In practice, scalar operators produce a single result, except in cases such as rotation of a pair of elements, which produce two outputs.

(2) Vector operators have up to two vector operands and produce up to two

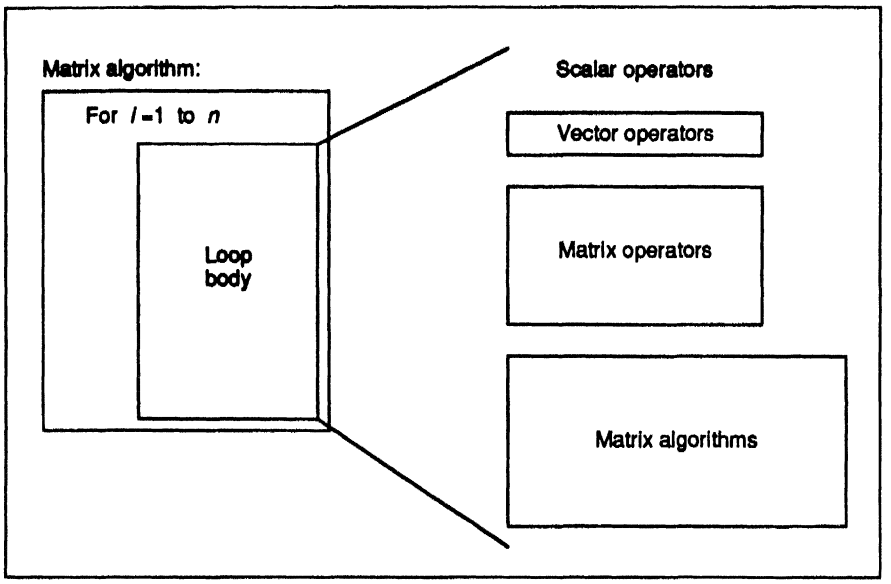


Figure 11. Canonical form of a matrix algorithm.

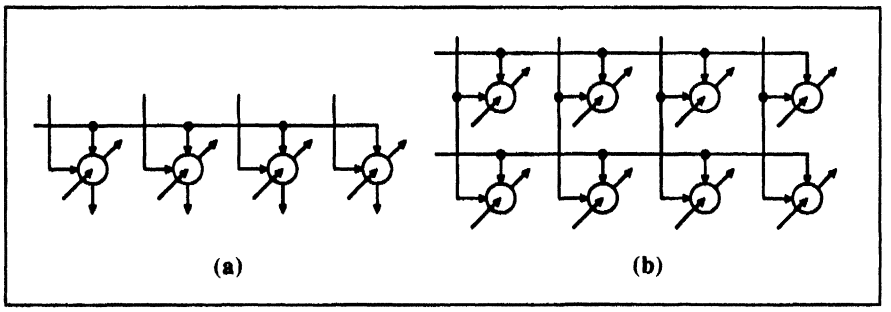


Figure 12. Dependency graphs of vector operator (a) and matrix operator (b).

vector results. The same primitive operator is applied to each element of the vector operands to produce the vector results. An additional scalar operand is common (broadcast) to all instances of the primitive operator. Figure 12a shows the dependency graph of a vector operator.

(3) Matrix operators have up to one matrix operand, one vector operand common to rows of the matrix operator, and a second vector operand common to columns of the matrix operator. A matrix operator produces a matrix result by applying the same primitive operator to each element of the matrix operand (and associated elements from the vector operands). Figure 12b shows the dependency graph of a matrix operator.

The discussion above shows that vector and matrix operators consist of primitive operations tied together by the common operand(s). Such an operand(s) corresponds to the broadcasting of data throughout the elements of the vector/matrix. To implement these operators in mesh arrays, we must eliminate the broadcasted data, which is accomplished by applying a transformation (replacing data broadcasting with transmitted data).

Since primitive operators can be unary, binary, or ternary, matrix and vector operators need not have all the operands indicated above. For example, the addition of a constant to each element of a vector does not use a second vector operand. On the other hand, the properties of matrix and vector operators rule out performing operations with two matrices or with three vectors (that is, adding two matrices or rotating elements of two vectors by corresponding angles contained in a third vector). These operators are not suitable for implementation in arrays, because they require external data input to inner cells (there is no common [broadcasted] data that is transferred through cells). For our purposes, such cases correspond to sets of scalar operations.

Limitations in the number of inputs and outputs to/from the operators in a matrix algorithm, as described above, arise from the objective of realizing those algorithms in mesh arrays. Since cells of arrays have only two input and two output ports, plus an internal feedback loop, an operator cannot have more than three operands. Moreover, since the arrays have only nearest-neighbor connections and external I/O only at the boundaries, it is necessary to transfer broadcasted data through the cells (as transmitted data). Consequently, the maximum number of input operands is three, and the combined number of computed results and transmitted data output from a cell is also limited to three.

The form of a matrix algorithm, as described above, does not place any requirements on the way loop indices are used to reference the elements of matrices and vectors. Two types of references have been considered: *uniform* and *affine*. With uniform references, each loop index used to address a variable appears in the form $(i-i_0)$ (the index plus/minus a constant). Affine references use the more general form $(i+j+...+k_0)$ (a linear combination of indices and a constant). Uniform references are the more common type and appear in most matrix algorithms — for example, LU-decomposition, QR-decomposition, SVD, and transitive closure, to name just a few.

Using uniform or affine references to access variables does not imply that the algorithm must be a uniform or an affine system of equations, as required by other methods (particularly algebraic-based techniques), such as those described by Rao and Kailath[11] and by Quinton.[13] The form of admissible algorithms given here is more general than in those cases. For example, the MMG method can use the transitive closure algorithm as originally expressed in Warshall's algorithm, while Rao's method requires that the algorithm be represented as an RIA (regular iterative algorithm). Similar situations arise with the other methods and algorithms, such as LU-decomposition, Gaussian elimination, and convolution.

The canonical form of matrix algorithms

shown in Figure 11 and described in the previous paragraphs is quite general; for instance, it directly accepts the important class of matrix algorithms appearing in areas such as real-time signal processing.[1] Moreover, algorithms in this class match well with implementations as mesh arrays. Other representations are also allowed with the MMG method, as long as they have the type of operators listed earlier and are amenable to the transformational process.

Transformational process. The MMG method uses a transformational paradigm. First, a fully parallel data-dependency graph (FPG), in which nodes represent operations and edges correspond to data dependencies, is derived from the algorithm. We use an FPG because this notation exhibits the intrinsic features of an algorithm. This graph could be used to directly derive an implementation by assigning each node to a different processing element (PE) and by adding delay registers to synchronize the arrival of data to PEs. The resulting structure (a pipelined realization of the graph) exhibits minimum delay (determined by the longest path in the graph) and optimal throughput (for multiple-instance computations), but it might require nonneighbor and varying-distance connections, large I/O bandwidth, and many units. The MMG method deals with these problems while preserving the features inherent in the data-dependency graph.

The two design stages, regularization and derivation of arrays, and the steps within them are depicted in the high-level description of the MMG method (Figure 13). The method is described below, where we also indicate the suitability for automation of the different steps involved. More details regarding this method and its theoretical backing are available.[5]

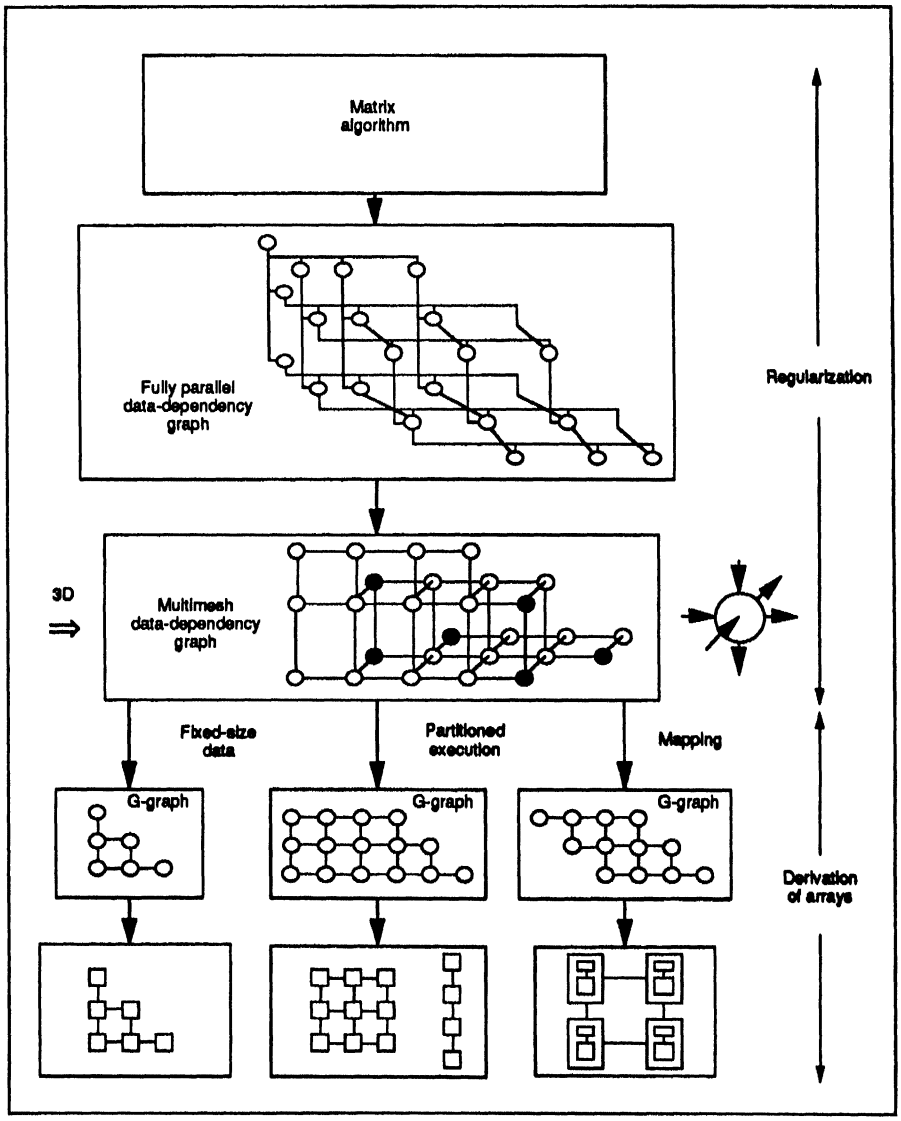


Figure 13. The MMG design method.

The regularization stage. The regularization stage produces a three-dimensional graph, which we call a multimesh dependency graph. This graph has the following characteristics:

- unidirectional and nearest-neighbor dependencies,
- edges parallel to axes of the three-dimensional space, and
- nodes at integer values of the axes.

This regularized form is suitable for the *derivation of arrays* because in addition to preserving all the information present in the FPG, it has the regular properties that characterize mesh arrays. Moreover, as we will show, the multimesh dependency graph makes it easy to obtain the characteristics of an array, the schedule of operations and I/O, and the data transfers.

Some researchers have regarded the dependency graph of a matrix algorithm as multidimensional instead of three dimensional. Such a conclusion was reached by representing in a graph the index dependencies in the algorithm. In other words, the dimensionality of the graph was defined by the number of indices appearing in an algorithm. In those approaches every variable is required to have all indices, so that each instance of a variable is associated with a point in the multidimensional index space.[10,11,13] In contrast, the dimensionality of the data-dependency graph in the MMG method is determined by the maximum of three inputs and three outputs per primitive node. A comparison of data dependencies and index dependencies, and their suitability for a design method, is available.[5]

The regularization stage in the MMG method is performed in two steps. First, we obtain the fully parallel data-dependency graph (FPG) of the algorithm. This graph is obtained by tracing the execution of the algorithm (the outer loop and loop body). This means that we symbolically execute the algorithm, tracking which variables are used and when, and we allocate operations to nodes of a graph and data references to its edges. In other words, the FPG corresponds to an unfolded dataflow graph. As

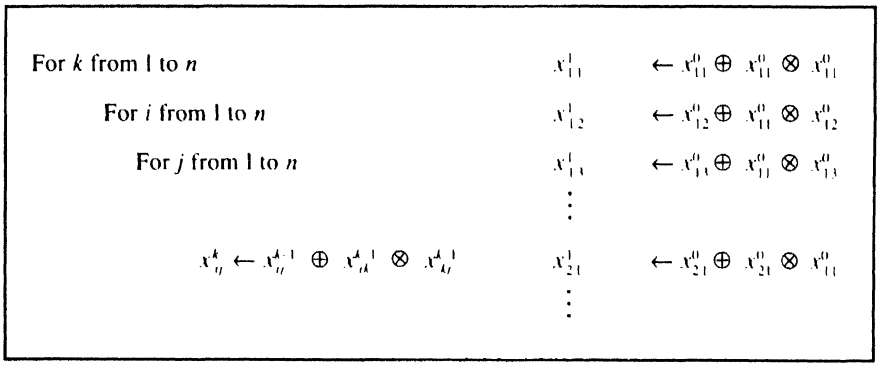


Figure 14. Symbolic execution of Warshall's transitive closure algorithm.

an example, Figure 14 depicts the symbolic execution of the transitive closure algorithm; the resulting FPG is shown in Figure 20. The FPG of a small problem (for example, n = 4,5) is sufficient to capture the features of an algorithm. In addition, the FPG consists of several subgraphs with the same dependency structure but perhaps different in size.

Second, we transform the fully parallel data-dependency graph into a three-dimensional multimesh graph, like those in Figure 15. To do this, we perform transformations on the FPG to remove properties not allowed in MMGs. The complete set of such properties consists only of

- data broadcasting, which is replaced by transmitted data;
- bidirectional dataflow, which is eliminated by moving dependent operations to one side of the data source; and
- nonregular dependencies, which are removed by adding delay nodes in the nonregular part.

A detailed discussion of these transformations is available.[5] They are illustrated in the next section through their application to the transitive closure algorithm.

Derivation of arrays. To perform the second stage in the MMG method, we collapse the MMG onto a two-dimensional G-graph. We do this by grouping prisms of primitive nodes — of base size p by q — onto G-nodes (see Figure 16). These prisms extend along one complete axis of the MMG. Grouping prisms parallel to axes of the three-dimensional space leads to simpler and more efficient implementations, so that the direction of collapse can be limited to three alternatives, one along each axis.

Selecting prisms of base size 1 by 1 leads to systolic arrays. This particular grouping corresponds to projecting the MMG onto a two-dimensional G-graph.

Next we schedule the order of execution for primitive operations that compose a G-node.

The size and schedule of prisms determine such cell properties as size of local storage, communication bandwidth, and cell pipelining. Scheduling operations by following the flow of transmitted data (see Figure 17) permits efficient use of pipelined cells. Scheduling by meshes of primitive nodes of size p by q, also depicted in Figure 17, minimizes the local storage required. (With prisms of base size 1 by 1, the only schedule possible is determined by the dependencies.) The cell bandwidth is determined by the number of edges that intersect a prism's side (the difference between cell bandwidth and functional unit bandwidth is adjusted by the queues attached to ports). Consequently, trade-offs make it possible to select prism size according to specific implementation requirements. Moreover, grouping is driven by the target implementation, which can be an algorithm-specific array for fixed-size data, a partitioned implementation, or a mapping onto a class-specific array. Consequently, the two steps above make it possible to optimize specific measures on the basis of implementation constraints.

For problems with fixed-size data on

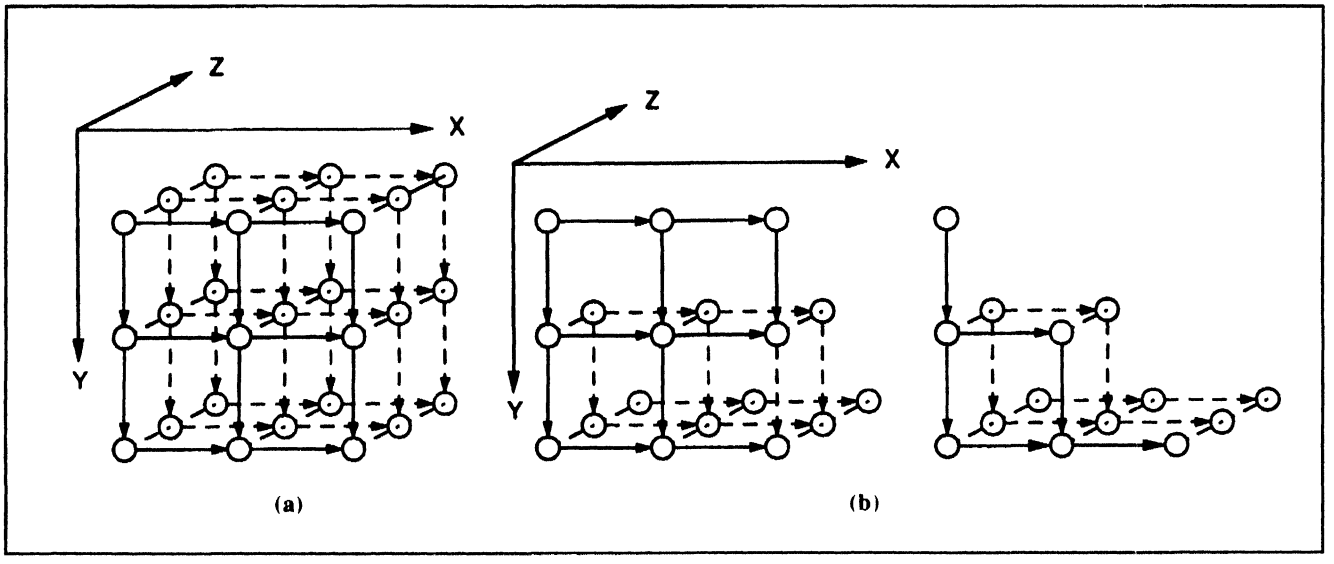


Figure 15. Examples of multimesh graphs: (a) complete multimesh data-dependency graph; (b) incomplete multimesh data-dependency graphs.

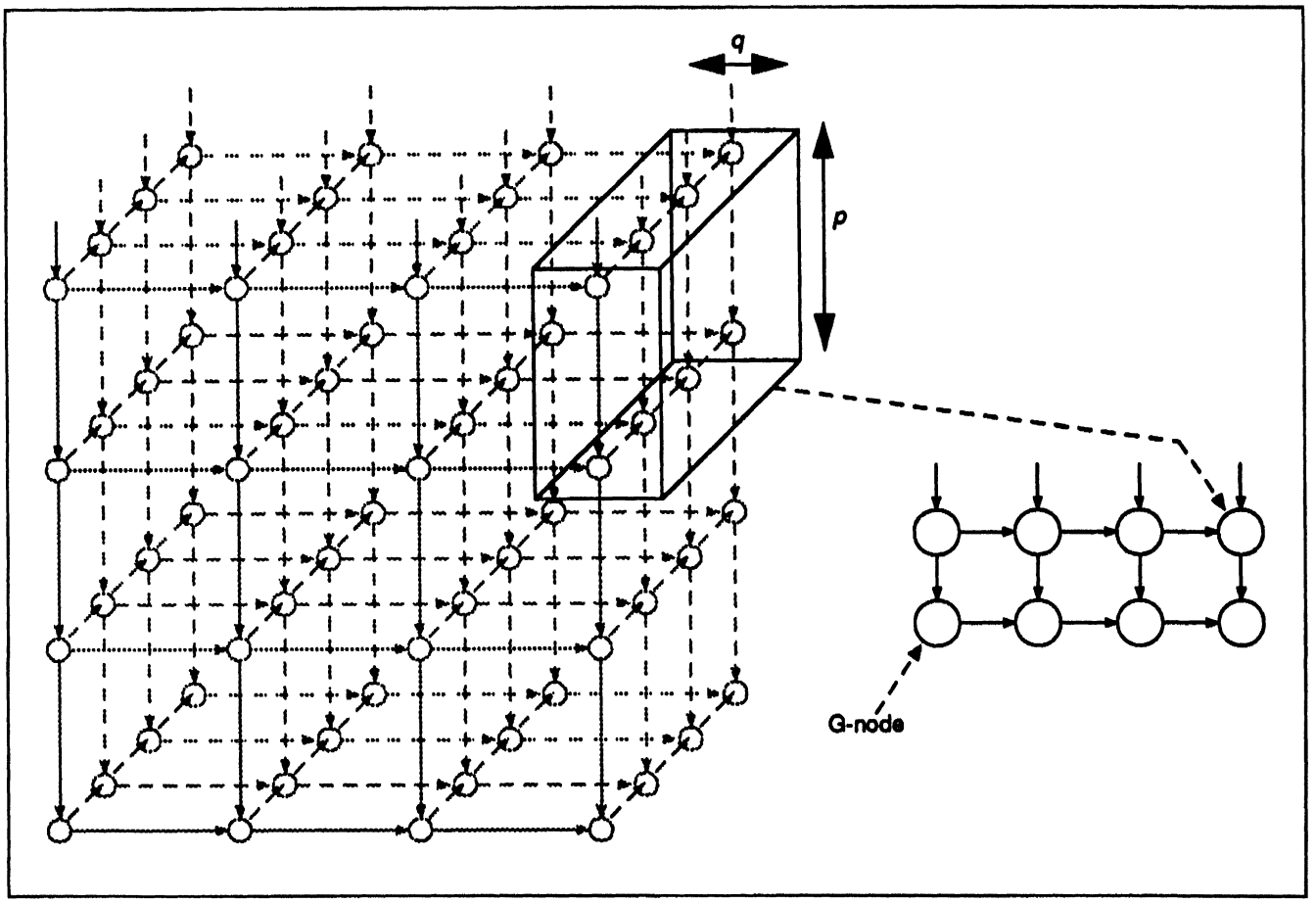


Figure 16. Collapsing an MMG onto a G-graph.

two-dimensional structures, we realize the G-graph obtained above as an array by allocating each G-node to a different processing element and each edge to a different communication link.

For problems with fixed-size data on linear arrays, we apply cut-and-pile to the G-graph. Each partition corresponds to a complete horizontal or vertical path of the G-graph. Nodes in a partition, or a *cut*, are executed concurrently, while different partitions are scheduled (piled) for pipelined execution in the array. Each G-node in a cut is allocated to a different processing element and each edge to a different communication link.

For partitioned problems we first divide (cut) the G-graph into sets of neighbor G-nodes (G-sets), with each G-set having as many nodes as there are cells in the array. Nodes in a G-set are structured in a linear or two-dimensional manner, depending on the desired array topology, and they are executed concurrently in the array. Figure 18 illustrates this process and the corresponding arrays. Data flowing between G-sets is stored in and retrieved from memories external to the array, as Figure 18 shows.

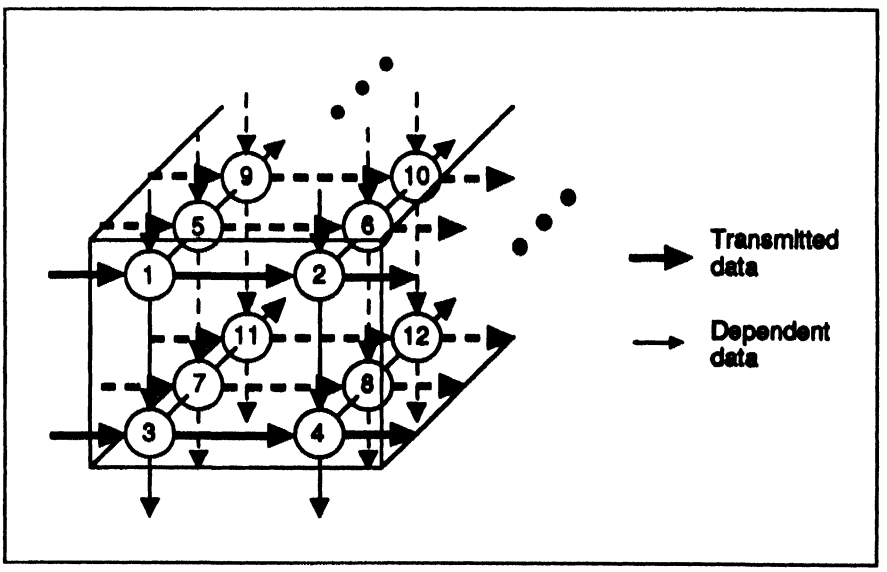


Figure 17. Scheduling primitive nodes within a prism, or G-node.

Next we schedule (pile) G-sets for execution. G-sets are executed in an overlapped (pipelined) manner, as Figure 19

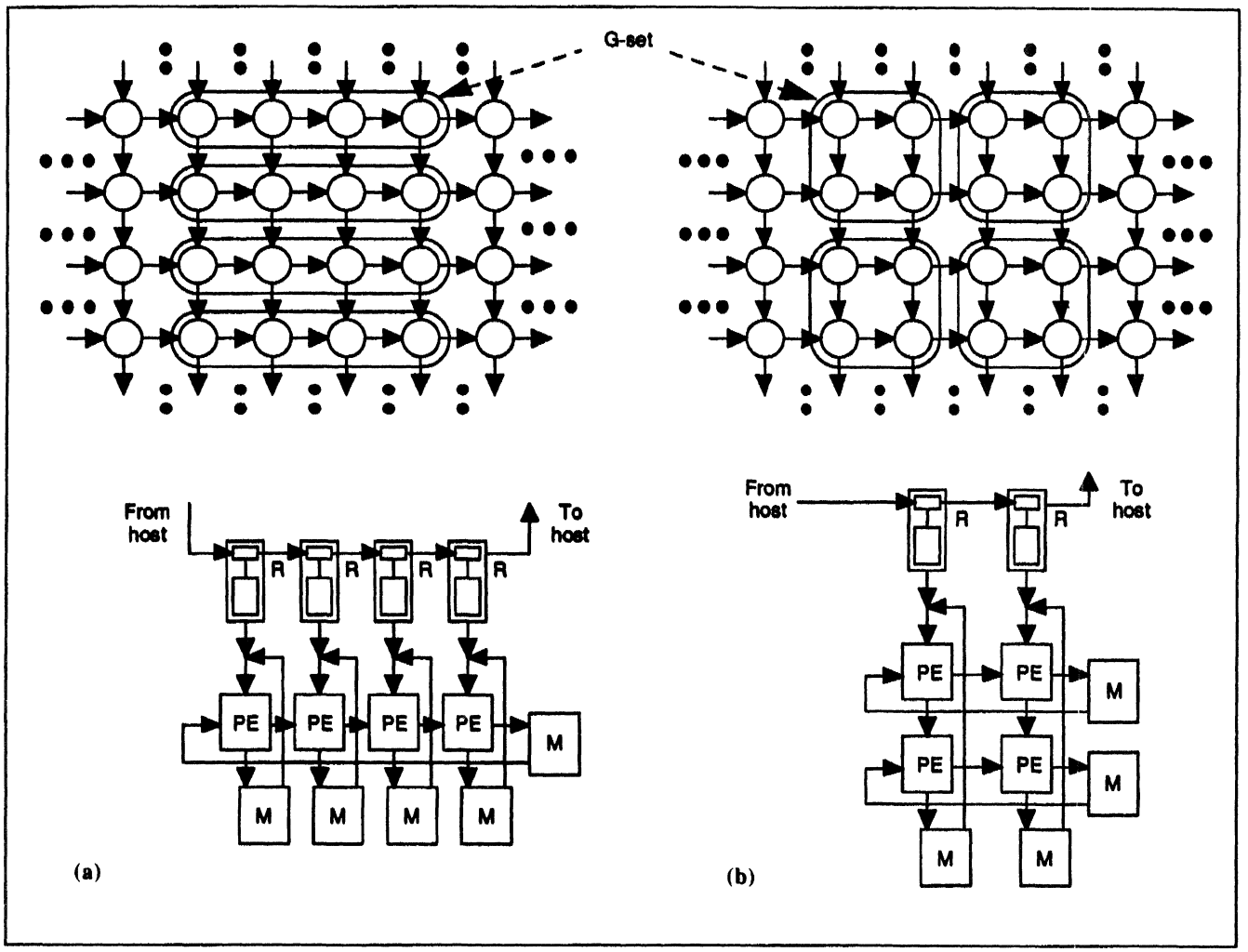


Figure 18. Applying cut-and-pile to the G-graph and corresponding arrays: (a) a linear array; (b) a two-dimensional array.

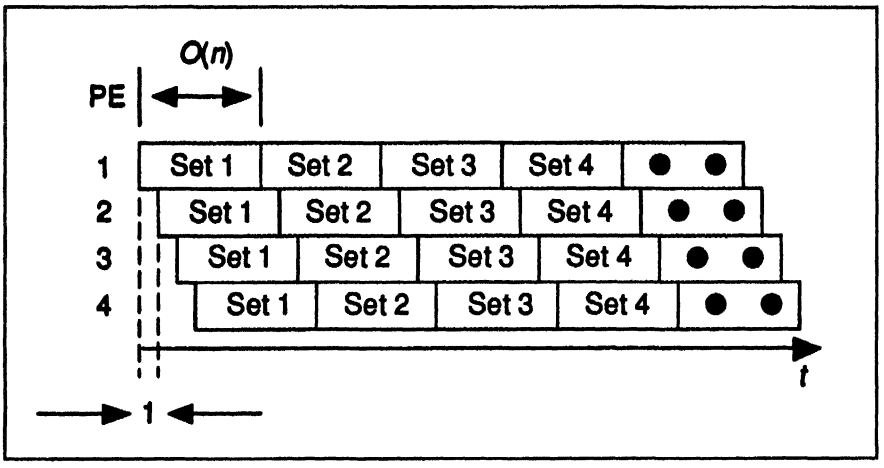


Figure 19. Overlapped execution of G-sets in a linear array.

shows for a linear array. While one set is executing, the input data for the next set is transferred from the host through the I/O structure shown at the top of the array in Figure 18. At the same time, the results from the previous G-set are returned to the host through the same structure.

For mapping onto class-specific arrays, we first divide (cut) the G-graph into sets of neighbor G-nodes (G-sets) whose characteristics (number and topology of nodes) are determined by characteristics of the specific array. Then we schedule (pile) G-sets for execution.

In all of the above cases, the array I/O schedule is determined by the schedule of the G-sets (if applicable) and by the arrival/departure of data to/from G-nodes. In particular, large partitioned problems lead to scheduling of G-sets in the same manner, so that the arrays depicted in Figure 18 correspond to canonical architectures for partitioned execution of algorithms.

The steps in the regularization stage and in the derivation of arrays can be performed automatically. However, implementation of certain steps might be simplified by capitalizing on the graphic characteristics of the method in an interactive

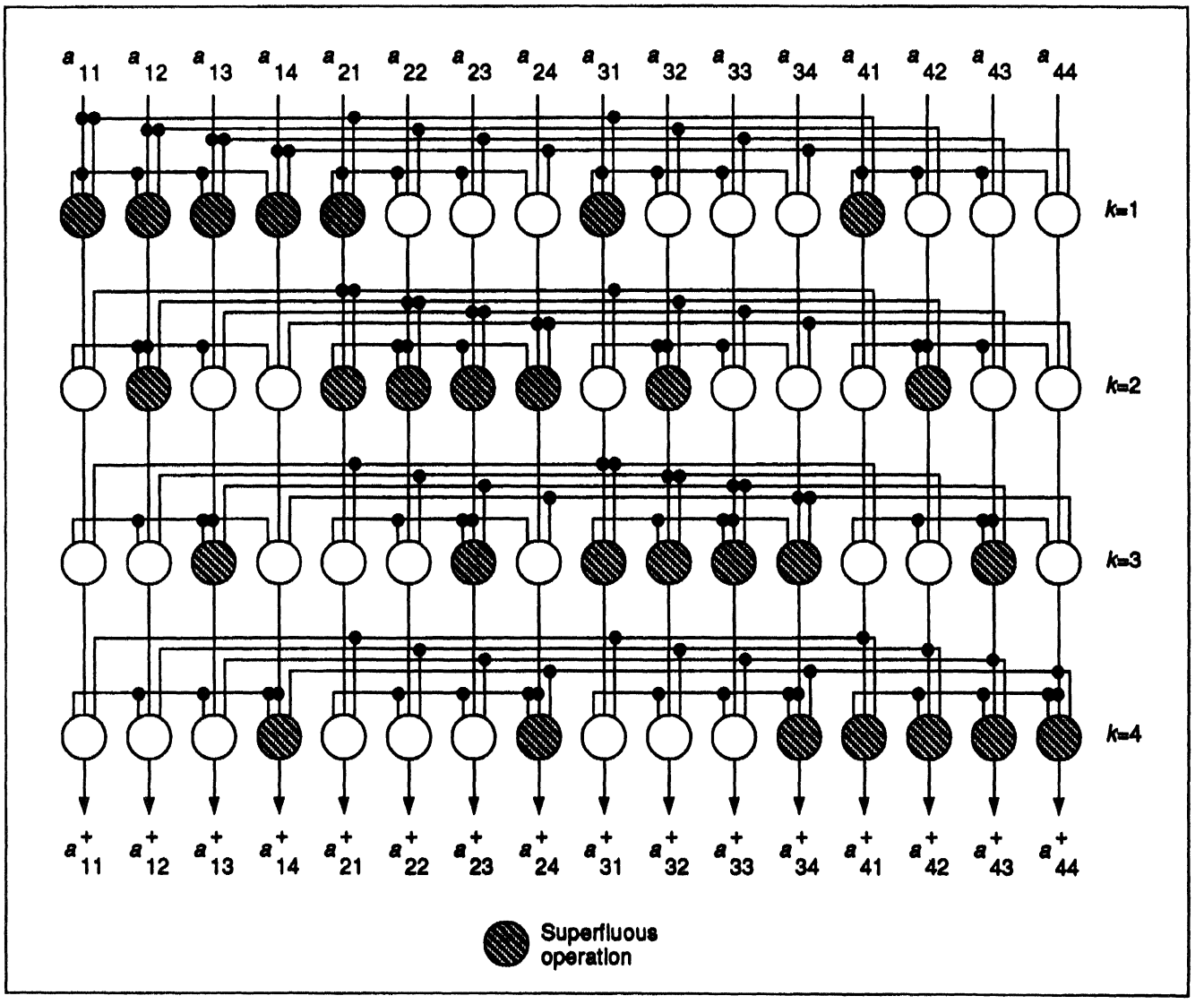


Figure 20. The fully parallel graph for the transitive closure algorithm.

CAD tool (for example, it is easier to visualize broadcasted data in a graph representation than to determine it automatically).

Performance and cost measures. There is no single suitable measure of performance and cost for application-specific implementations. In some cases the number of cells may be important, while in others the stress may be on throughput or utilization of cells. Therefore, to determine the performance of arrays derived with the MMG method, we use the following set of measures (where N is the number of operations in the algorithm):

- T Throughput
- K Number of cells
- U Utilization ($U = N/KT^{-1}$)
- $A_{I/O}$ Input/output bandwidth
- C_w Storage per cell
- C_{BW} Cell bandwidth

Other measures can also be defined by a designer, depending on the requirements of particular applications. The measures are computed with information obtained from the dependency graphs, both the original fully parallel graph and the transformed graphs. Moreover, transformations used in the method affect such measures, so that one can study the impact of a particular transformation on the resulting array's cost and performance while carrying out the transformation. The next section provides examples of these issues as the MMG method is applied to a specific algorithm.

Example application of the MMG method

We can observe the MMG method through the derivation of mesh arrays for the transitive closure algorithm.

The regularization process. Figure 14 shows Warshall's algorithm to compute the transitive closure. It consists of three nested loops and a single-assignment statement. Although it looks very similar to matrix multiplication, the order of the loop indices is reversed, so that the dependencies are different.

Figure 20 depicts the fully parallel graph for a problem of size $n = 4$, obtained from the symbolic execution of Warshall's algo-

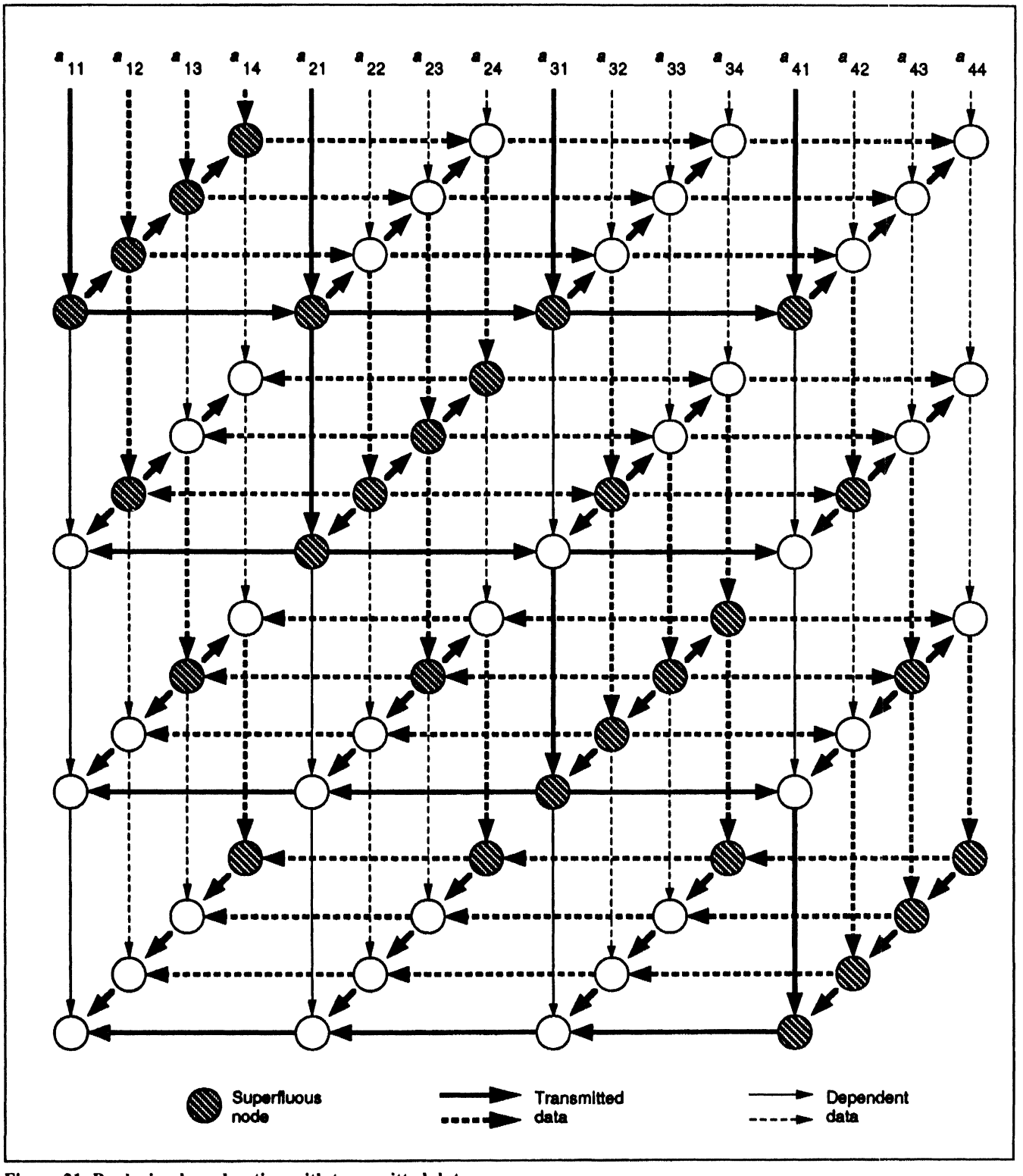


Figure 21. Replacing broadcasting with transmitted data.

rithm. This FPG is characterized by many broadcasted elements and many superfluous operations (highlighted in the figure) where the result is equal to one of the input operands. This property, a consequence of primitive operations AND and OR, is dependent on the specific algorithm, but it illustrates the capabilities of an explicit description. Superfluous operations can be removed if it is advantageous for an implementation (that is, if it simplifies the resulting array).

The FPG shown in Figure 20 consists of n levels, with each level corresponding to one iteration of the outermost loop in

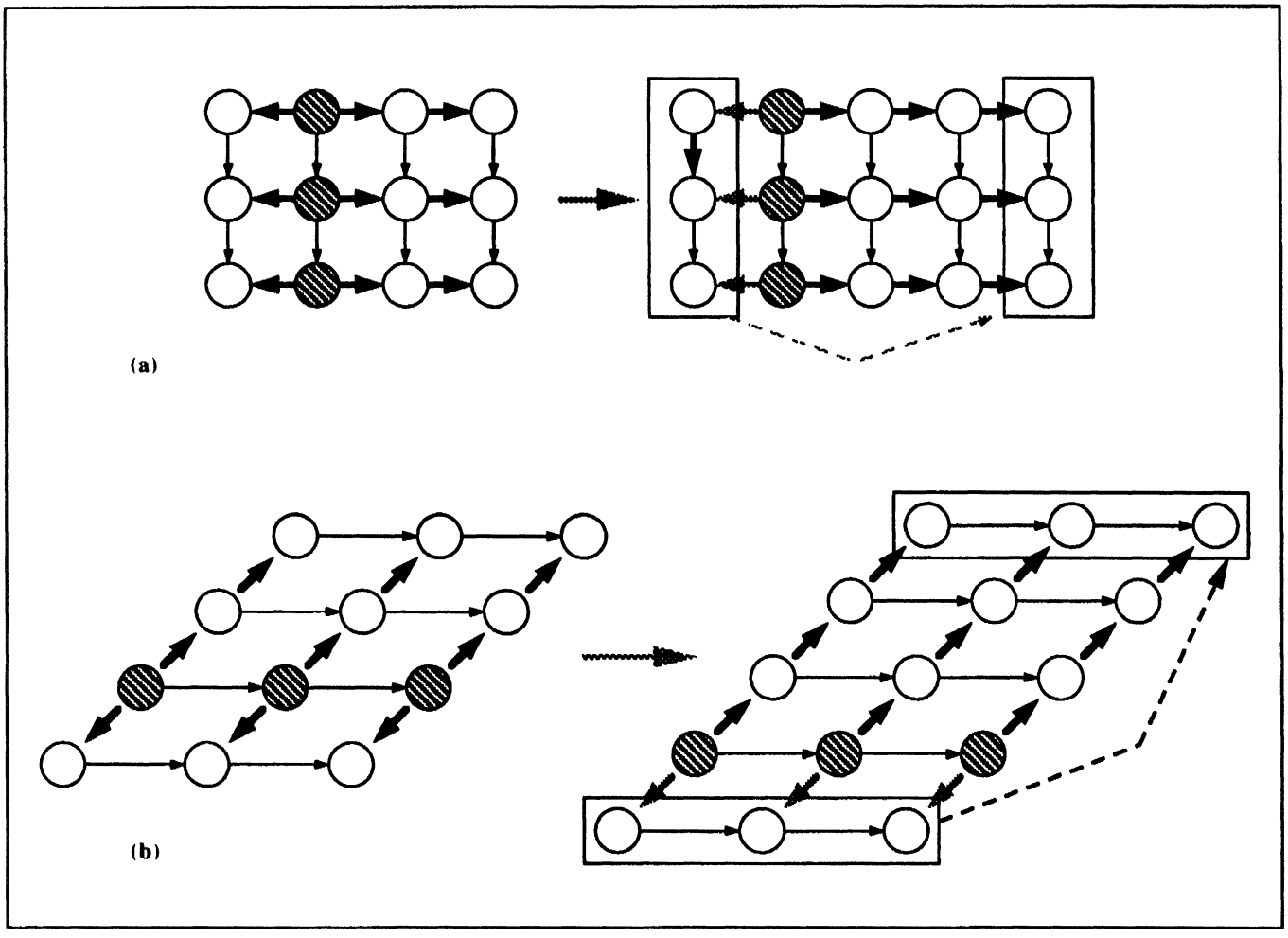


Figure 22. Removing bidirectional transmitted data: (a) removal along x axis; (b) removal along z axis.

Warshall's algorithm. At each level there is global and local broadcasting. Global broadcasting corresponds to data that is broadcast throughout the entire level, while locally broadcast data reaches only a portion of the level. Sources of broadcasting change from level to level; at the kth level of the graph the kth row of matrix data, as well as the kth element of each row, is broadcast.

As indicated earlier, regularizing the graph in Figure 20 consists of replacing data broadcasting with transmitted data, drawing the graph as a three-dimensional structure, removing bidirectional flow of transmitted data, and adding delay nodes to make sure that all dependencies are among nearest-neighbor nodes (in other words, regular). Let's examine this process.

First we transform the FPG by replacing data broadcasting with transmitted data. Globally broadcast data is drawn orthogonal to the flow of locally broadcast elements, in a three-dimensional structure. The resulting three-dimensional graph, shown in Figure 21, does not fulfill the requirements of an MMG, because it exhibits bidirectional flow of transmitted data. This graph has the same dependency structure as the one used in the signal flow graph (SFG) method.[8] However, the graph derived with the SFG method requires rewriting the algorithm in a single-assignment form, which is not necessary with the MMG method. Moreover, nodes in the graph derived with the MMG method compute only one variable, whereas the single-assignment form produces nodes that compute three variables.

Bidirectional transmitted data can be eliminated if all nodes at one side of the data source are part of a movable subgraph.[5] The graph in Figure 21 fulfills this requirement, so that nodes are moved to one side of the transmitted data sources in two steps:

(1) Nodes to the left of sources of horizontal transmitted data are moved to the right end of each level of the graph, as Figure 22a shows; application of this transformation leads to the graph in Figure 23.

(2) Nodes in front of sources of transmitted data along the z axis are moved to the end of this axis, as Figure 22b shows, resulting in the graph of Figure 24.

Figure 24 still exhibits one characteristic not allowed in a multimesh dependency graph: Dependencies between nodes at the boundaries of the three-dimensional structure are not between nearest neighbors. This irregularity is eliminated by adding delay nodes in the vacant posi-

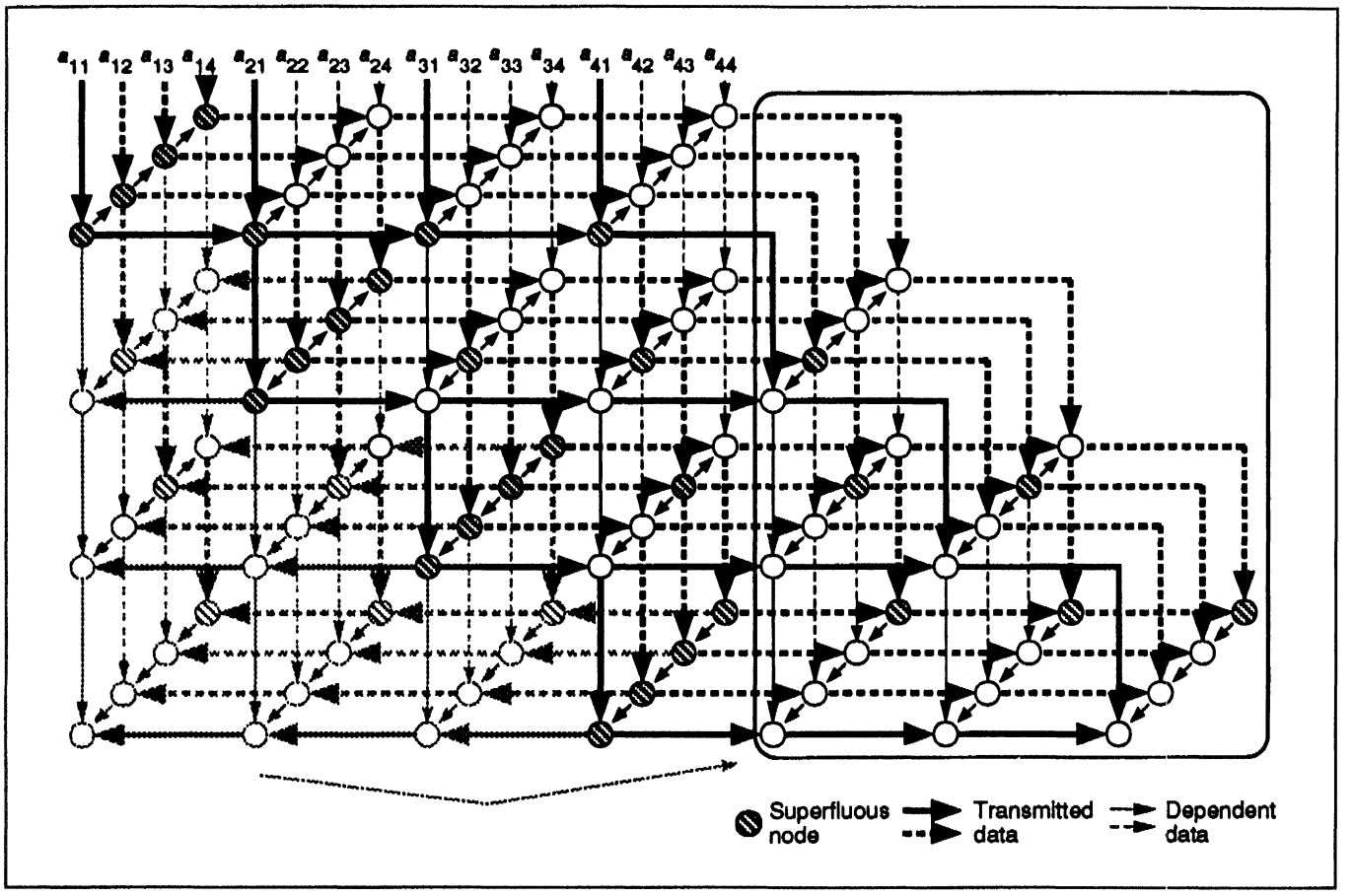


Figure 23. Bidirectional transmitted data along x axis removed.

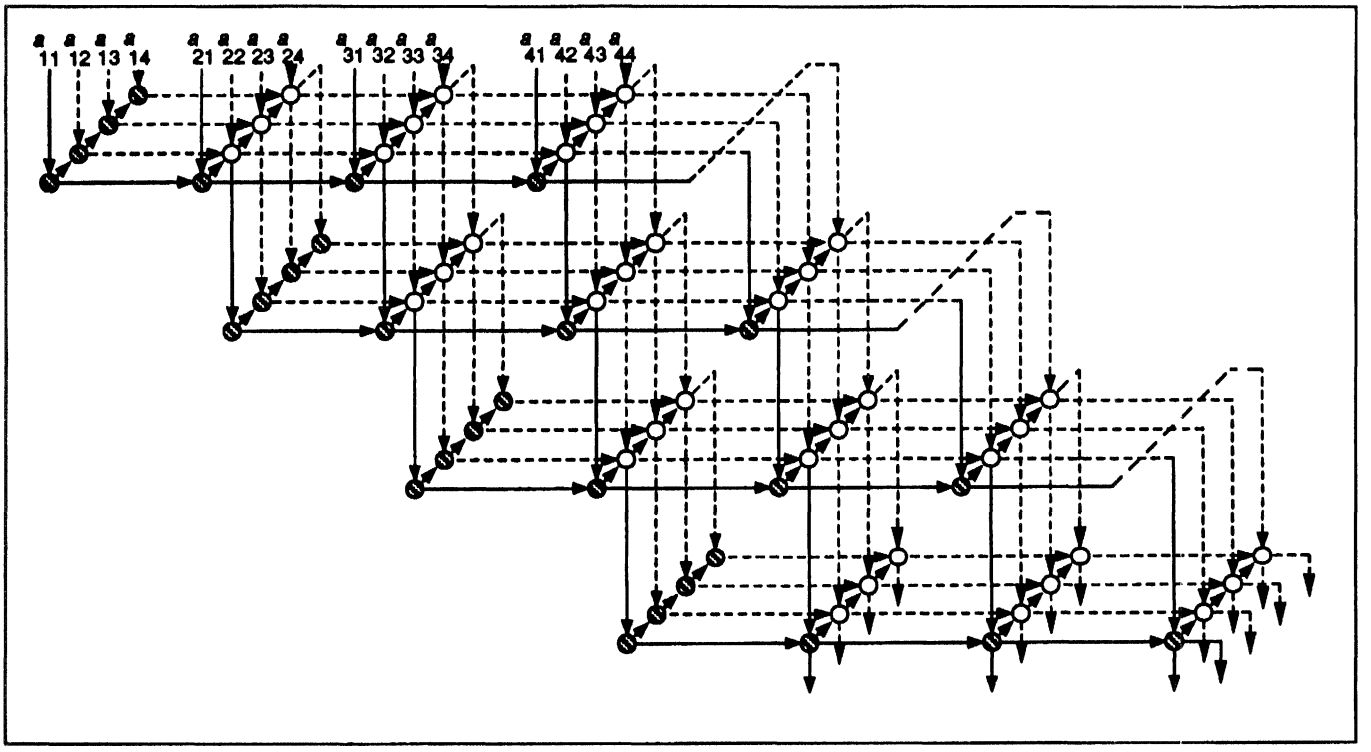


Figure 24. A unidirectional dependency graph.

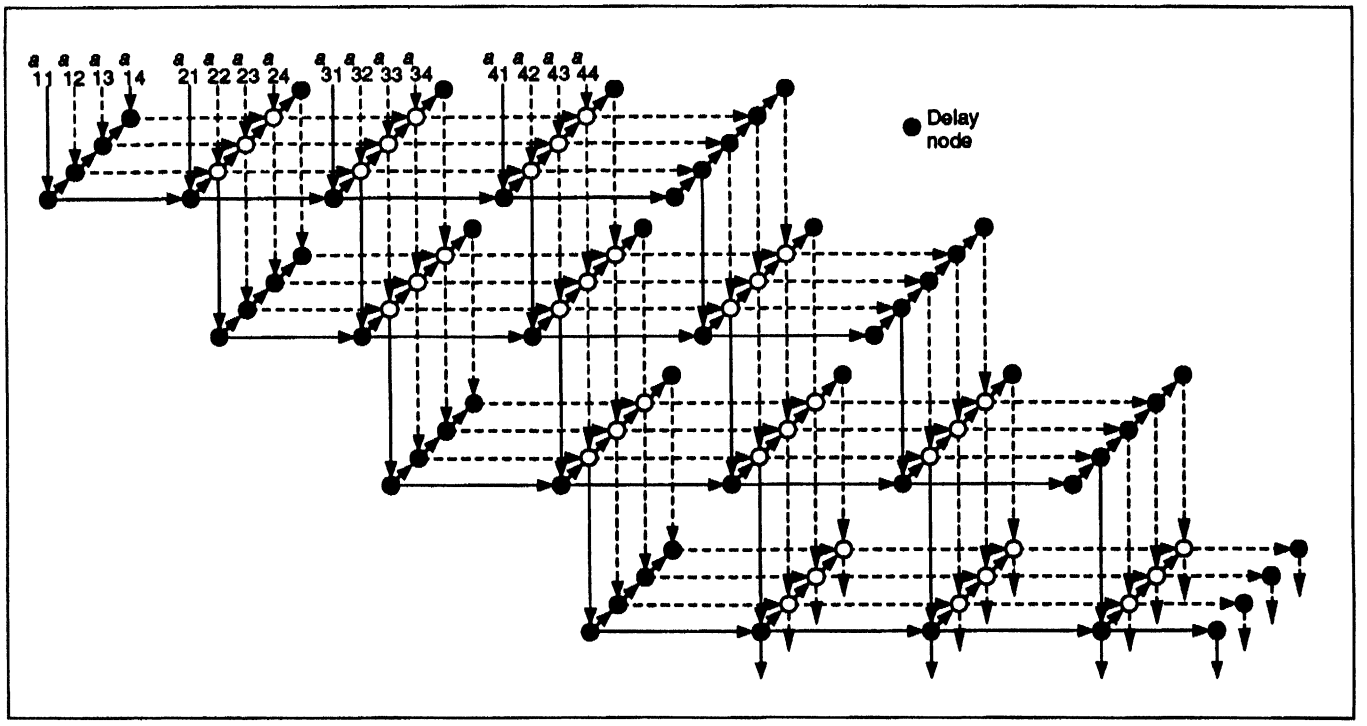


Figure 25. A multimesh dependency graph.

tions, leading to the multimesh dependency graph shown in Figure 25. In this figure we have also replaced superfluous nodes with delay nodes.

From this example we can infer that the MMG is advantageous in describing an algorithm because it provides information on all operations and dependencies without imposing constraints on the algorithm's form. Moreover, an algorithm described by an FPG is transformed into a regular MMG in a simple and systematic manner by using a set of predefined transformations and taking advantage of the graphics capabilities offered by the data dependencies.

Derivation of arrays. To simplify the discussion, we consider grouping by prisms of base size 1 by 1 (that is, grouping for systolic arrays). In Figure 26 we project the MMG of transitive closure along the three axes, producing three G-graphs. (Literature is available describing the more general case of grouping by prisms.[5])

For problems with fixed-size data, the G-graphs are directly realized as arrays. Three structures are obtained; Table 1 summarizes their performance and cost characteristics. These measures are obtained from the MMG and the corresponding G-graph. For example, the number of cells is equal to the number of G-nodes, throughput is given by the largest number of operation nodes grouped into a single G-node, and utilization of cells is related to the length of paths in the MMG (or the number of primitive nodes per prism).

The graphs also allow us to compare the performance and cost of the different resulting structures. For example, grouping prisms parallel to the y axis is not convenient, because paths have different lengths,

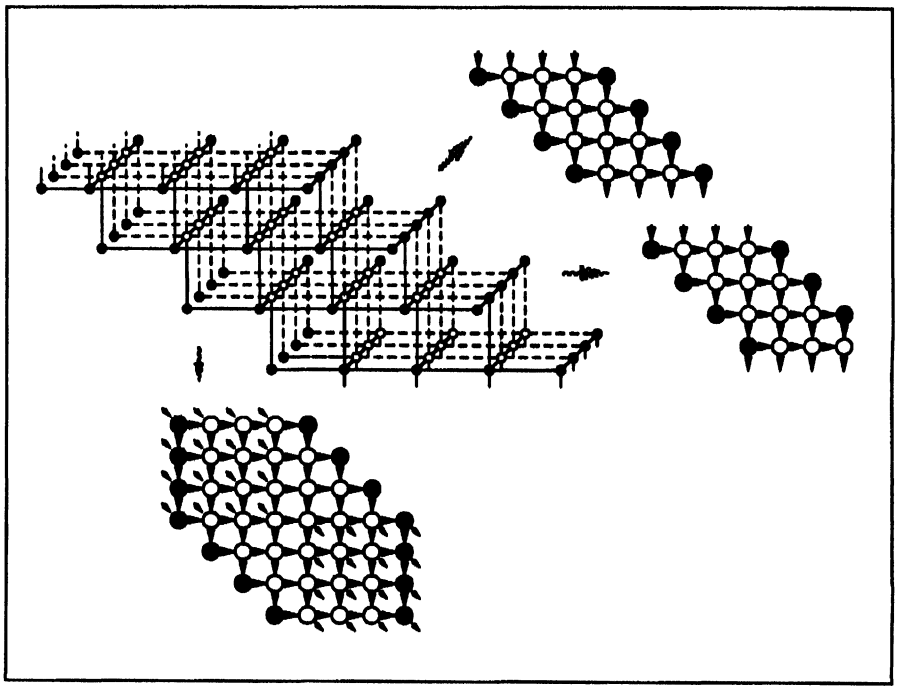


Figure 26. Projecting the multimesh graph onto G-graphs.

Table 1. Cost and performance measures of arrays for transitive closure.

Array	Throughput	Opers. per cell	Number of cells	Number of regs	Links per cell
X-grouping	$1/n$	1	$n(n-1)$	$(n-1)$	4
Y-grouping	$1/n$	1	$3n^2-5n+2$	$(4n-2)$	4
Z-grouping	$1/n$	1	n^2	$2n$	4
Rao's method	$1/n$	3	$\approx 3n^2$		4
SFG method	$1/n$	3	n^2		8

leading to nonoptimal utilization of cells. On the other hand, grouping parallel to axis x or z collapses paths of the same length, with the associated benefits in utilization. Complete details on the performance of these arrays are reported elsewhere, as is a comparison with other arrays previously proposed for the same algorithm.[5] Table 1 also summarizes the data corresponding to arrays derived with two other methods.

As indicated earlier, partitioned problems lead to the canonical structures depicted in Figure 18. For partitioning, the G-graphs derived by grouping prisms parallel to axis x or z are more convenient because both resulting graphs have G-nodes with the same computation time.

The MMG method incorporates features not available in other previously proposed techniques. These features include a general class of admissible algorithms; transformations to regularize algorithms; and transformations to derive arrays, given implementation constraints. This method is applicable to a large class of frequently used matrix computations, including the algorithms described by Speiser and Whitehouse.[1]

Derivation of arrays in the MMG method is systematic, flexible, and suited for obtaining mesh structures, taking into account architectural features, implementation constraints, and performance and cost measures. Moreover, the method has resulted in application-specific arrays with fewer, simpler, and better utilized cells than arrays previously proposed for the same algorithms. ■

Acknowledgments

We thank the reviewers for their constructive comments and their suggestions for improving the article. This work was partially supported by the National Science Foundation under grant MIP-8813340, "Composite Operations Using On-Line Arithmetic for Application-Specific Parallel Architectures." In addition, Jaime Moreno was supported by an IBM Computer Science Graduate Fellowship

References

1. J.M. Speiser and H. Whitehouse, "A Review of Signal Processing with Systolic Arrays," *SPIE Real-Time Signal Processing VI*, Bellingham, Wash., Aug. 1983, pp. 2-6.

2. J.M. Speiser, "An Overview of Matrix-Based Signal Processing," *Proc. Asilomar Conf. Signals, Systems, and Computations*, CS Press, Los Alamitos, Calif., Order No. 871, 1988, pp. 284-289.

3. J.J. Dongarra, J.L. Martin, and J. Worlton, "Computer Benchmarking: Paths and Pitfalls," *IEEE Spectrum*, July 1987, pp. 38-43.

4. H.T. Kung, "Why Systolic Architectures?" *Computer*, Vol. 15, No. 1, Jan. 1982, pp. 37-46.

5. J.H. Moreno, *Matrix Computations on Mesh Arrays*, doctoral dissertation, Computer Science Dept., Univ. of California, Los Angeles, Calif., Aug. 1989.

6. H.T. Kung, "Let's Design Algorithms for VLSI Systems," Caltech Conference on VLSI, Jan. 1979, pp. 65-90.

7. S.Y. Kung, *VLSI Array Processors*, Prentice-Hall, Englewood Cliffs, N.J., 1988.

8. J.J. Navarro, J.M. Llaberia, and M. Valero, "Partitioning: An Essential Step in Mapping Algorithms into Systolic Array Processors," *Computer*, Vol. 20, No. 7, July 1987, pp. 77-89.

9. J.A.B. Fortes, K. Fu, and B.W. Wah, "Systematic Approaches to the Design of Algorithmically Specified Systolic Arrays," in *Computer Architecture*, V. Milutinovic, ed., Elsevier/North Holland, New York, 1988, pp. 454-494.

10. D.I. Moldovan, "On the Design of Algorithms for VLSI Systolic Arrays," *Proc. IEEE*, Vol. 71, No. 1, Jan. 1983, pp. 113-120.

11. S.K. Rao and T. Kailath, "Regular Iterative Algorithms and Their Implementation on Processor Arrays," *Proc. IEEE*, Vol. 76, No. 3, Mar. 1988, pp. 259-269.

12. R.M. Karp, R.E. Miller, and S. Winograd, "The Organization of Computations for Uniform Recurrence Equations," *J. ACM*, Vol. 14, July 1967, pp. 563-590.

13. P. Quinton, "Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations," *Proc. 11th Ann. Symp. Computer Architecture*, CS Press, Los Alamitos, Calif., Order No. 538, June 1984, pp. 208-214.

Further reading

A large body of literature exists for this field. The most important initial work on methods for the design of arrays is reviewed in Reference 9 above, which provides a good summary of 19 different approaches.

Additional references on the design of arrays

Delosme, J.M., and I.C.F. Ipsen, "Design Methodology for Systolic Arrays," *SPIE Advanced Algorithms and Architectures for Signal Processing*, Vol. 696, Bellingham, Wash., Aug. 1986, pp. 245-259.

Guerra, C., and R. Melhem, "Synthesizing Non-Uniform Systolic Designs," *Proc. Int'l Conf. Parallel Processing*, CS Press, Los Alamitos, Calif., Order No. 724, Aug. 1986, pp. 765-771.

Heller, D.E., and I.C.F. Ipsen, "Systolic Networks for Orthogonal Decompositions," *SIAM J. Scientific and Statistical Computing*, Vol. 4, No. 2, June 1983, pp. 261-269.

Rajopadhye, S.V., S. Purushothaman, and R.M. Fujimoto, "On Synthesizing Systolic Arrays from Recurrence Equations with Linear Dependencies," *Proc. Sixth Conf. Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Computer Science, Springer-Verlag, New York, Dec. 1986, pp. 488-503.

Rao, S.K., *Regular Iterative Algorithms and Their Implementation on Processor Arrays*, doctoral dissertation, Information Systems Laboratory, Stanford Univ., Stanford, Calif., Oct. 1985.

Yaacoby, Y., and P. Cappello, "Converting Affine Recurrence Equations to Quasi-Uniform Recurrence Equations," Tech. Report TRCS87-18, Computer Science Dept., Univ. of California, Santa Barbara, Calif., Feb. 1988.

Matrix algorithms for real-time signal processing and other applications

Ahmed, H., J. Delosme, and M. Morf, "Highly Concurrent Computing Structures for Matrix Arithmetic and Signal Processing," *Computer*, Vol. 15, No. 1, Jan. 1982, pp. 65-82.

Andrews, H.C., and C.L. Patterson, "Singular Value Decomposition and Digital Image Processing," *IEEE Trans. Acoustics, Speech, and Signal Processing*, Vol. ASSP-24, Feb. 1976, pp. 26-53.

Bromley, K., and H.J. Whitehouse, "Signal Processing Technology Overview," *SPIE Real-Time Signal Processing IV*, Vol. 298, Bellingham, Wash., Aug. 1981, pp. 102-106.

Klema, V.C., and A.J. Laub, "The Singular Value Decomposition: Its Computation and Some Applications," *IEEE Trans. Automatic Control*, Vol. AC-25, Apr. 1980, pp. 164-176.

Luk, F.T., ed., *Advanced Algorithms and Architectures for Real-Time Signal Processing II*, Vol. 826, SPIE — The Int'l Soc. for Optical Eng., Bellingham, Wash., Aug. 1987.

Luk, F.T., ed., *Advanced Algorithms and Architectures for Real-Time Signal Processing III*, Vol. 975, SPIE — The Int'l Soc. for Optical Eng., Bellingham, Wash., Aug. 1988.

Execution of matrix algorithms in topologies other than mesh arrays

Fox, G., et al., *Solving Problems on Concurrent Processors*, Vol. 1, Prentice-Hall, Englewood Cliffs, N.J., 1987, pp. 167-186 and 363-424.

Fox, G., ed., *Proc. Third Conf. Hypercube Concurrent Computers and Applications — Banded and Full Matrix Algorithms*, ACM Press, Jan. 1988.

Meier, U., and A. Sameh, "Numerical Linear Algebra on the CEDAR Multiprocessor," *SPIE Advanced Algorithms and Architectures for Real-Time Signal Processing II*, Vol. 826, Bellingham, Wash., Aug. 1987, pp. 1-9.

Sameh, A., "Algorithms and Experiments for Parallel Linear Systems Solvers," *Proc. Second SIAM Conf. Parallel Processing for Scientific Computing*, Philadelphia, Pa., Nov. 1985.

SIAM, *Proc. SIAM Conf. Parallel Processing for Scientific Computing*, Philadelphia, Pa., 1987, 1988, 1989.

Specialized arithmetic techniques

The impact of application-specific cells on arrays can be inferred from research on the use of specialized arithmetic techniques such as CORDIC and on-line arithmetic.

Cavallaro, J.R., and F.T. Luk, "CORDIC Arithmetic for an SVD Processor," *Proc. Eighth Symp. Computer Arithmetic*, CS Press, Los Alamitos, Calif., Order No. 774, May 1987, pp. 215-222.

Ercegovac, M., and T. Lang, "On-Line Scheme for Computing Rotation Factors," *J. Parallel and Distributed Computing*, Vol. 5, No. 3, June 1988, pp. 209-227.

Ercegovac, M., and T. Lang, "Redundant and On-Line CORDIC: Application to Matrix Triangularization and SVD," Tech. Report CSD-870046, Computer Science Dept., Univ. of California, Los Angeles, 1987 (to be published in *IEEE Trans. Computers*).

Transitive closure algorithm

The transitive closure algorithm has been considered an important target for the derivation of application-specific arrays. Several authors discuss systolic arrays for this algorithm.

Kung, S.Y., S.C. Lo, and P.S. Lewis, "Optimal Systolic Design for the Transitive Closure and the Shortest Path Problems," *IEEE Trans. Computers*, Vol. C-36, May 1987, pp. 603-614.

Lin, F.C., and I.C. Wu, "Systolic Arrays for Transitive Closure Algorithms," *Proc. Int'l Symp. VLSI Systems Design*, Taipei, Taiwan, May 1985.

Nunez, F.J., and N. Torralba, "Transitive Closure Partitioning and Its Mapping to a Systolic Array," *Proc. Int'l Conf. Parallel Processing*, CS Press, Los Alamitos, Calif., Order No. 783, Aug. 1987, pp. 564-566.

Robert, I., *Algorithmes et Architectures Systoliques*, class notes, Institut National Polytechnique de Grenoble, Grenoble, France, 1986.

Application of the MMG method to specific algorithms

Moreno, J.H., and T. Lang, "On Partitioning the Faddeev Algorithm," *Proc. Int'l Conf. Systolic Arrays*, CS Press, Los Alamitos, Calif., Order No. 860, May 1988, pp. 125-134.

Moreno, J.H., and T. Lang, "Graph-Based Partitioning of Matrix Algorithms for Systolic Arrays: Application to Transitive Closure," *Proc. Int'l Conf. Parallel Processing*, Pennsylvania State Univ. Press, University Park, Pa., Aug. 1988, pp. 28-31.

Moreno, J.H., and T. Lang, "Comments on 'A Systolic Array for Computing BA^{-1}'," *IEEE Trans. Acoustics, Speech, and Signal Processing*, Nov. 1989, pp. 1,768-1,789.

Other aspects of the MMG method

The MMG method has also been used to analyze trade-offs between properties of cells and has been compared with other proposed design approaches. Moreover, the MMG technique has shown that for partitioned execution, linear arrays offer advantages over two-dimensional structures with the same number of cells. This fact led to the derivation of a linear array with high cell utilization.

Moreno, J.H., and T. Lang, "Arrays for Partitioned Matrix Algorithms: Trade-offs Between Cell Storage and Cell Bandwidth," *SPIE Real-Time Signal Processing XI*, Vol. 977, Bellingham, Wash., Aug. 1988, pp. 156-169.

Moreno, J.H., and T. Lang, "Comparing Design Methods Based on Index Dependencies and on Data Dependencies," *Proc. Int'l Conf. Systolic Arrays*, May 1989, pp. 599-608. Published in *Systolic Array Processors*, Prentice-Hall, Englewood Cliffs, N.J.

Moreno, J.H., and T. Lang, "Linear Array for Partitioned Execution of Matrix Algorithms with High Utilization," *SPIE Real-Time Signal Processing XII*, Bellingham, Wash., Aug. 1989, pp. 102-117.

Optimization of Computation Time for Systolic Arrays

Yiwan Wong, *Member, IEEE*, and Jean-Marc Delosme

Abstract— **The time performance of a systolic array implementation of an algorithm is measured by the product of two quantities: the number of systolic cycles required to complete the computations, n_{sys}, and the cycle time, t_{sys}. Yet earlier works on systolic array synthesis have exclusively sought schedules that minimize n_{sys}, an approach justified only when the processors have a single functional unit. This paper deals with the general case of processors with multiple, possibly pipelined, functional units that operate concurrently and presents a method for the minimization of the actual computation time $n_{\text{sys}} \times t_{\text{sys}}$. The selection of a linear scheduling function which minimizes $n_{\text{sys}} \times t_{sys}$ is formulated as a combinatorial optimization problem, which is shown to have a bounded search space. An efficient branch-and-bound method for the solution of that problem is proposed and applied to several examples.**

Index Terms—**Branch-and-bound method, combinatorial optimization, computation time, cycle time, linear scheduling, retiming, systolic array synthesis, uniform recurrence equations.**

I. Introduction

THE insatiable demand for higher computational speed in scientific computing and information processing has led to an extensive search for parallel architectures and programming techniques that can effectively exploit the parallelism inherent in these applications. For problems in these areas it is common to find among the algorithms with lower operation counts algorithms that can be effectively implemented using only local communications on parallel systems with distributed memory. When, in addition, the distribution of the operations and the data transfers are highly structured throughout the algorithm the implementations exhibit a high degree of regularity and only call for simple control. The cost to design and build dedicated hardware that executes these algorithms with a performance comparable to that achieved on general purpose parallel architectures is then relatively low. If the algorithm must be executed routinely the design and use of specialized hardware may actually become cost effective. The economic incentive for such a solution is enhanced if a computer-aided design technique is developed that optimizes performance for a given hardware cost. Such a technique is presented here.

The *systolic array* model, introduced by Kung and Leiserson [12], is a powerful model for special purpose architectures, well-suited to the implementation of numerous algorithms for matrix computations, real-time signal and image processing, string manipulations, etc. [13]. A systolic array is a collection of globally synchronized processing elements (PE's) of a few types, interconnected locally and in a regular pattern. Each PE is connected to a small number of PE's in its vicinity with which it can communicate directly. According to the operations to be performed and the target performance, a PE can be as simple as a bit-serial adder and some shift registers, or as complex as a modern microprocessor data path, with multiple pipelined functional units. The operation of a systolic array resembles that of an assembly line—data travels rhythmically in a *pipelined* fashion and the results are built up incrementally from these data values.

The time performance of a systolic implementation of an algorithm is measured by the product of the number of *systolic cycles* required to complete the computations, n_{sys}, and the maximum duration of a systolic cycle, the cycle time t_{sys}. Both quantities depend on the schedule chosen to order the computations in the algorithm and are thus related. For a given (valid) schedule, n_{sys} is a function of the size of the problem the algorithm solves, and t_{sys} depends on the complexity of the computations performed within a systolic cycle, the precedence among these computations, and the amount of hardware resources available on the PE's. There is a tradeoff between these two quantities as the following example demonstrates.

Example 1: Given a nonsingular $N \times N$ lower triangular band matrix A of bandwidth p, and a column N-vector y, the solution x of the system of equations $Ax = y$ can be found by forward-substitution

$$\text{for } i = 1 \text{ to } N \text{ do}$$
$$x_i = (y_i - \sum_{k<j<i} a_{ij}x_j)/a_{ii}$$

where $k = \text{MAX}(0, i - p)$. To find an efficient parallel implementation, the algorithm is first rewritten in a functionally equivalent form in which the order of computation is defined implicitly by the data dependences in the algorithm. The form adopted in this paper is the Regular Iterative Algorithm (RIA) [26], which is basically a set of recurrence equations with regular dependence structure (see Section II-A for more details). With proper transformations [29] the

Manuscript received May 11, 1989; revised December 26, 1990. This work was supported by the Defense Advanced Research Project Agency (DARPA) under Contract N00014-88-K-0573, by the Army Research Office (ARO) under Contract DAAL03-86-K-0158, and by the Office of Naval Research (ONR) under Contract N00014-85-K-0461.

Y. Wong was with the Department of Computer Science, Yale University, New Haven, CT 06250. He is now with the Speech & Image Understanding Laboratory, Texas Instruments Inc., Dallas, TX 75265.

J.-M. Delosme is with the Department of Electrical Engineering, Yale University, New Haven, CT 06520.

IEEE Log Number 9103098.

Reprinted from *IEEE Transaction on Computers*, pp. 159-177, February 1992.

forward-substitution algorithm can be expressed as a RIA of five recurrences

$$
\begin{aligned}
\boldsymbol{u}_{ij} &= \begin{cases} y_{ij} & \text{if } j = k \\ \boldsymbol{s}_{ij} & \text{if } j \neq k \text{ and } i > j \\ \boldsymbol{q}_{ij} & \text{if } j \neq k \text{ and } i = j \end{cases} \\
\boldsymbol{s}_{ij} &= \boldsymbol{u}_{i,j-1} - \boldsymbol{p}_{ij} \\
\boldsymbol{p}_{ij} &= a_{ij}\boldsymbol{u}^1_{i-1,j} \\
\boldsymbol{q}_{ij} &= \begin{cases} \boldsymbol{u}_{i,j-1}/a_{ij} & \text{if } i = j \\ \text{undefined} & \text{otherwise} \end{cases} \\
\boldsymbol{u}^1_{ij} &= \begin{cases} \boldsymbol{u}_{i-1,j} & \text{if } i > j \\ \boldsymbol{u}_{ij} & \text{otherwise} \end{cases}
\end{aligned} \tag{1}
$$

where $y_{ik} \equiv y_i$, defined upon the set of index points $\{(i,j) \mid 1 \leq i \leq N, \mathrm{MAX}(0, i-p) < j \leq i\}$. The final outputs are $x_i \equiv \boldsymbol{u}_{ii}$, $i = 1 \cdots n$.

Now, assume that each PE on the array has a multiplier/divider and an adder/subtractor which can operate in parallel, and some registers for storage. Furthermore, let $T_\times$ and T_+ be, respectively, the computation delay of the multiplier/divider and of the adder/subtractor. Two systolic array implementations of this algorithm (for $p = 3$) are shown in Fig. 1. Array (I) is the design that requires the minimum number of systolic cycles to compute the RIA; the overall computation time is

$$n_{\mathrm{sys}} \times t_{\mathrm{sys}} = (2N-1) \times (T_+ + T_\times).$$

Array *(II)* requires more systolic cycles but its cycle time is shorter,

$$n_{\mathrm{sys}} \times t_{\mathrm{sys}} = (3N-1) \times \mathrm{MAX}(T_+, T_\times).$$

For example, when $N = 20$, $T_+ = 6$, and $T_\times = 9$, array *(I)* requires 585 time units to compute the results while array *(II)* requires only 531 time units, a close to 10% improvement in computation time. Thus, the implementation with best overall time performance does not necessarily minimize n_{sys}. It depends on the relative magnitudes of the computation delays of the functional units. □

Previous works in systolic array synthesis, e.g., [3], [9], [20], [22], [25], [26], focus exclusively on the minimization of n_{sys}, instead of the actual computation time. Such a model, which does not take t_{sys} into account, is valid when each PE in the array has a single functional unit. Then the cycle time is equal to the time needed to perform *sequentially* the computations assigned to a PE within one systolic cycle. However, the above example indicates that this simple model is not satisfactory even for the case of two functional units per processor.

When implementing a RIA on a systolic array the computations are sequenced according to a schedule typically constrained to be a linear function with integer coefficients of the indexes of the RIA variables (a *linear schedule*, see Section II-C). This constraint leads to simple control and, often, the minimal n_{sys} under that constraint is equal to the absolute minimum or exceeds it by just a small number of cycles. Rao [26] and Delosme and Ipsen [6] show that the integral linear schedules which minimize the total number of systolic cycles n_{sys} can be determined by solving an integer linear programming problem (ILP). In order to solve these ILP's, Lisper [21] and Van Dongen [28] consider the use of enumerative procedures while Li *et al.* [20] propose a combination of linear programming and heuristics that finds feasible but not necessarily optimal solutions. Li and Wah [19] minimize n_{sys} by means of an incremental enumerative search procedure based on the speeds of the data flows on a systolic array implementation. Fortes [9] formulates an optimization problem in which the assignment of computations to processors is taken into account and develops an enumerative procedure for selecting an integral linear schedule that minimizes n_{sys} under constraints on the array interconnections.

Instead of linear schedules, with integer coefficients, Quinton [25] considers schedules derived from linear functions with rational coefficients. Because the systolic cycles are discrete quantities, the systolic cycle values are obtained by rounding the values of these functions down (or up) to the nearest integer. The resulting schedules are not linear and, often, they assign more than one set of computations to a PE during a single systolic cycle. If each PE possesses only one set of functional units, the cycle time of the array is increased by a factor equal to the maximum number of sets of computations assigned by the schedule to a PE in one systolic cycle. On the other hand, if each PE contains multiple sets of functional units, the control and communication requirements are greatly complicated. Hence, unless these implementation issues are resolved, it is not practical to consider such schedules. Karp *et al.* [10] use linear programming to find optimal rational, piecewise linear, functions from which schedules may be obtained by rounding; they show under which conditions these schedules are close to the fastest possible schedules. Unfortunately these theoretically important schedules lead to even more delicate implementation issues than Quinton's.

Leiserson *et al.* [18] and Fettweis [8] present a procedure, called *retiming* in [18], that reduces the clock period of general synchronous circuits by just rearranging their clocked register elements. Furthermore, Leiserson *et al.* develop a polynomial time algorithm to find a rearrangement that minimizes the clock period of a given circuit. It will be shown in Section II-D that once an integral linear schedule has been selected, the minimum value of t_{sys} (with respect to the selected schedule) can be determined with the retiming procedure.

In this paper, the linear schedule which minimizes the product $n_{\mathrm{sys}} \times t_{\mathrm{sys}}$ is shown to be the solution of a combinatorial optimization problem whose solution space is bounded. An efficient branch-and-bound method is developed to find that solution. To avoid the task of resource allocation, which is NP-hard [24], the PE's on the array are assumed to contain a "sufficient" number of functional units to achieve the required level of parallelism. This assumption is valid for Very Large Scale Integration (VLSI) and Wafer Scale Integration (WSI) implementations of systolic arrays. In the final section, the difficulties that arise in trying to adopt a similar method when the PE's on the array have a fixed number of functional units are discussed.

The paper is organized as follows. The next section introduces the necessary background for the formulation of the optimization problem. In Section III that discrete optimization problem is shown to have a bounded solution space and hence to be solvable by exhaustive enumeration. A branch-

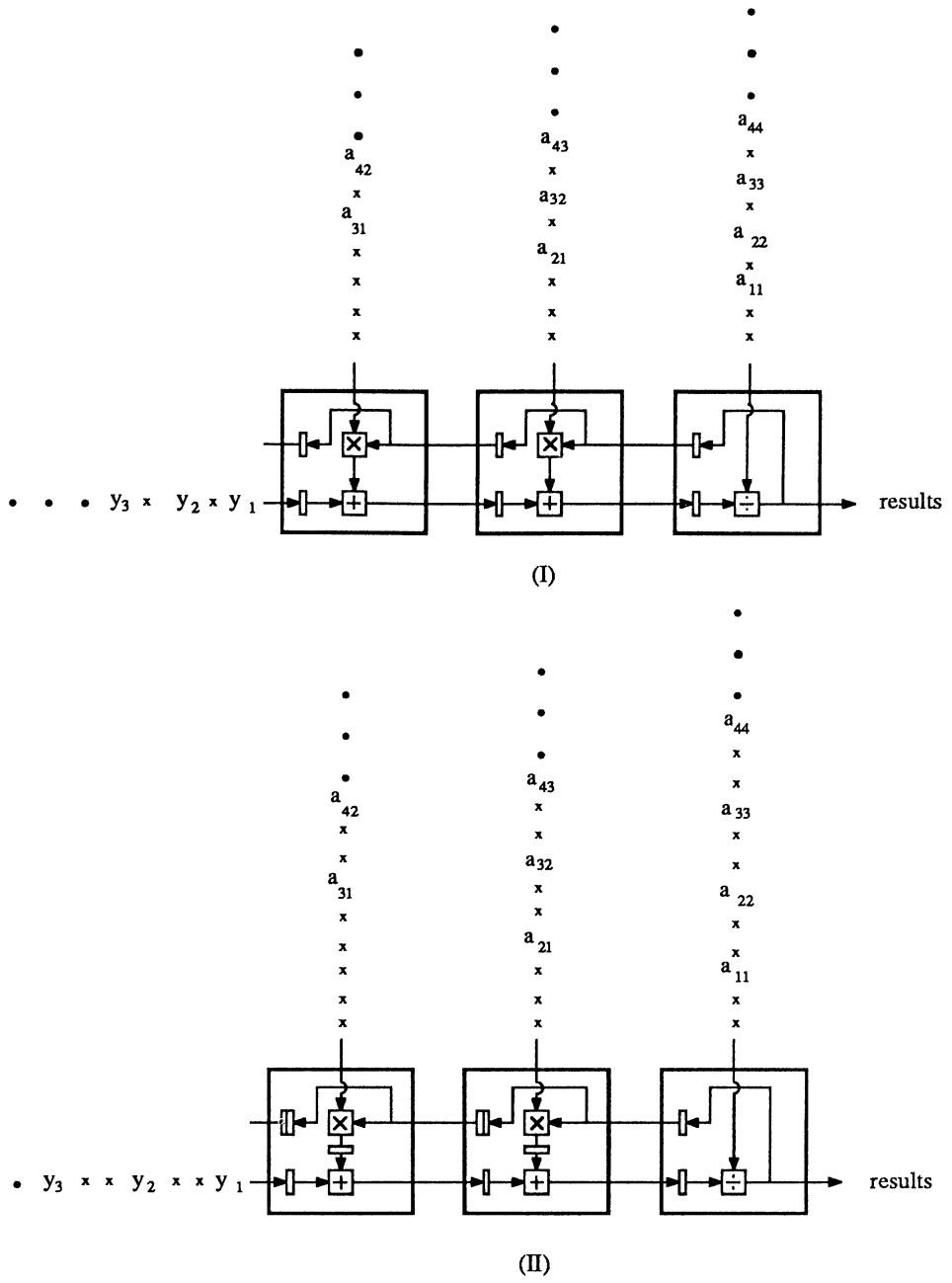


Fig. 1. Two systolic array implementations of the forward-substitution algorithm. The thin rectangular boxes in the PE's denote clocked register elements.

and-bound method, which locates the optimal solution more efficiently, is devised in Section IV. That method is extended in Section V, to cover the case when some of the functional units on the PE's are pipelined. Furthermore, the optimization problem is shown there to be simplified if all the functional units in a PE are pipelines with at least two stages; it reduces then to two ILP's. Performance results of the branch-and-bound method are reported in Section VI. The concluding section discusses other extensions of the computation time optimization problem.

II. Preamble

To facilitate the presentation, the forward-substitution algorithm presented in Example 1 will be used throughout this section as an illustration.

Some terminology from graph theory [2] is needed for the rest of the discussion. Let $G = (V, E)$ be a directed graph. A *path* from node v_1 to node v_p in G is a nonnull sequence of directed edges v_1v_2, v_2v_3, $\cdots$, $v_{p-1}v_p$. A *circuit* is a *closed* path whose end points coincide. A circuit is *elementary* (an elementary circuit, or EC for short) if no two of its nodes, except for the end points, coincide. Let $ec_1 \cdots ec_m$ denote the elementary circuits in G. An edge which does not belong to any EC is called an *exterior* edge, otherwise, it is called *interior*. For convenience, the interior edges are numbered as e_1 to e_k. Define the EC-*edge matrix* for G as the Boolean matrix C of size $m \times k$ where $c_{ij} = 1$ if and only if edge e_j belongs to ec_i. An edge is called a *zero-edge* if its weight

is zero (or is null, if the weight is a vector quantity). A *zero-path* (respectively, a *zero-circuit*) is a path (a circuit) which contains only zero-edges. The vertices in V can be partitioned into equivalence classes V_i, $1 \leq i \leq r$, such that vertices v_i and v_j are equivalent if and only if there is a path from v_i to v_j and a path from v_j to v_i. The *strongly connected components* of G are the graphs $G_i = (V_i, E_i)$, where E_i is the set of edges connecting pairs of vertices in V_i.

A. Algorithm Model

Rao has shown [26], [27] that an algorithm amenable to systolic array implementation can be expressed in a form called Regular Iterative Algorithm (RIA), which is a generalization of the Uniform Recurrence Equation (URE) model introduced by Karp *et al.* [10]. Informally, an n-dimensional RIA is a set of recurrence equations

$$\begin{aligned} \boldsymbol{u}_{\boldsymbol{Q}} &= f_{\boldsymbol{u}}(\boldsymbol{u}_{\boldsymbol{Q}-\boldsymbol{d}_{\boldsymbol{uu}}}, \cdots, \boldsymbol{w}_{\boldsymbol{Q}-\boldsymbol{d}_{\boldsymbol{uw}}}, \boldsymbol{x}_{\boldsymbol{Q}-\boldsymbol{d}_{\boldsymbol{ux}}}, \cdots, \boldsymbol{z}_{\boldsymbol{Q}-\boldsymbol{d}_{\boldsymbol{uz}}}) \\ &\vdots \qquad\qquad\qquad\qquad \vdots \\ \boldsymbol{w}_{\boldsymbol{Q}} &= f_{\boldsymbol{w}}(\boldsymbol{u}_{\boldsymbol{Q}-\boldsymbol{d}_{\boldsymbol{wu}}}, \cdots, \boldsymbol{w}_{\boldsymbol{Q}-\boldsymbol{d}_{\boldsymbol{ww}}}, \boldsymbol{x}_{\boldsymbol{Q}-\boldsymbol{d}_{\boldsymbol{wx}}}, \cdots, \boldsymbol{z}_{\boldsymbol{Q}-\boldsymbol{d}_{\boldsymbol{wz}}}) \end{aligned}$$

defined on a set of index points $\{\boldsymbol{Q}\}$, called the *range* of computation and denoted by Ω; Ω is a subset of the n-dimensional index space. In general, the range is assumed to be an n-dimensional polytope with integral vertices (an integral polytope). This set of recurrences defines the evaluation rules for the *computed* variables, $\{\boldsymbol{u} \cdots \boldsymbol{w}\}$, of the RIA. The rest of the variables, $\{\boldsymbol{x} \cdots \boldsymbol{z}\}$, which appear only on the right-hand side of the recurrences, are *input* variables; they represent data values supplied from external sources. Each recurrence specifies a k-ary, single-valued function ($f_{\boldsymbol{u}}$, for example) and the set of k data values (parameters), $\boldsymbol{u}_{\boldsymbol{Q}-\boldsymbol{d}_{\boldsymbol{uu}}} \cdots \boldsymbol{z}_{\boldsymbol{Q}-\boldsymbol{d}_{\boldsymbol{uz}}}$, required for computing the value of $\boldsymbol{u}_{\boldsymbol{Q}}$. The value of k, which is a constant independent of the size of the problem solved by the RIA, depends on the variable ($\boldsymbol{u}$ in the example) computed by the recurrence. Note that a particular variable may appear more than once, or may not appear at all, on the right-hand side of a recurrence.

The constant integral vector $\boldsymbol{d}_{\boldsymbol{uv}}$, called *dependence vector*, represents the dependence of the computation of $\boldsymbol{u}$ at an index point $\boldsymbol{Q}$ in Ω on the value of $\boldsymbol{v}$ computed at index point $\boldsymbol{Q} - \boldsymbol{d}_{\boldsymbol{uv}}$. This data dependence is also denoted by

$$\boldsymbol{u}_{\boldsymbol{Q}} \rightarrow \boldsymbol{v}_{\boldsymbol{Q}-\boldsymbol{d}_{\boldsymbol{uv}}}.$$

The data dependence vectors in the forward-substitution algorithm are

$$\begin{aligned} &\boldsymbol{d}_{\boldsymbol{u}_y} = \boldsymbol{d}_{\boldsymbol{us}} = \boldsymbol{d}_{\boldsymbol{uq}} = (0\ 0), \boldsymbol{d}_{\boldsymbol{su}} = (0\ 1), \boldsymbol{d}_{\boldsymbol{sp}} = (0\ 0), \\ &\boldsymbol{d}_{\boldsymbol{p}_a} = (0\ 0), \boldsymbol{d}_{\boldsymbol{pu}^1} = (1\ 0), \boldsymbol{d}_{\boldsymbol{qu}} = (0\ 1), \boldsymbol{d}_{\boldsymbol{q}_a} = (0\ 0) \\ &\boldsymbol{d}_{\boldsymbol{u}^1\boldsymbol{u}^1} = (1\ 0), \boldsymbol{d}_{\boldsymbol{u}^1\boldsymbol{u}} = (0\ 0). \end{aligned}$$

The dependence structure in a RIA can be depicted by a Full Dependence Graph (FDG) in which the nodes form an integral lattice representing the index points in Ω, and the directed edges represent the data dependences. The Full Dependence Graph (FDG) of the recurrences in (1) ($N = 8$, $p = 6$) is shown in Fig. 2, with the dependences due to the null dependence vectors left out.

The computation of a RIA involves the evaluation of the set of recurrence equations at every index point $\boldsymbol{Q}$ in Ω; each index point in Ω can be treated as a unit of computation of the RIA. The final results of the computation are given by the values of one or more of the computed variables on a subset of index points in Ω. In the forward-substitution algorithm, the range of computation is given by

$$\Omega = \{(i, j) \mid 1 \leq i \leq N, \mathrm{MAX}(0, i - p) < j \leq i\}$$

and the final results are the values of $\boldsymbol{u}$ found on the subset of index points

$$\{(i, i) \mid i = 1 \cdots N\}.$$

The uniform dependence structure in a FDG can be captured by a compact representation called a *dependence graph* (DG) [5], [10], [26] in which there is a node for each of the variables defined or used in the RIA, and the directed edges correspond to the dependences among the variables (Fig. 3). The weight of any of the directed edges in the DG is equal to the dependence vector between the two variables the edge connects. There may be multiple edges (with different weights) between a pair of nodes.

B. Reindexing

The dependence vectors in a DG can be modified, while preserving the functionality of the computations the DG represents, by a process called *reindexing* [6]. Let $\{\delta_{\cdot}\}$ be a set of integral n-vectors, one $\delta_{\boldsymbol{u}}$ for each variable $\boldsymbol{u}$ in the RIA. To reindex is to offset by $\delta_{\boldsymbol{u}}$ the set of index points in Ω upon which the variable $\boldsymbol{u}$ is computed. Hence, an instance of a variable $\boldsymbol{u}$ which is originally computed at index point $\boldsymbol{Q}$ will be computed at index point $(\boldsymbol{Q} - \delta_{\boldsymbol{u}})$ after reindexing. If variable $\boldsymbol{u}$ depends on variable $\boldsymbol{v}$ with dependence vector $\boldsymbol{d}_{\boldsymbol{uv}}$ then, after reindexing, the dependence becomes

$$\boldsymbol{u}_{(\boldsymbol{Q}-\delta_{\boldsymbol{u}})} \rightarrow \boldsymbol{v}_{(\boldsymbol{Q}-\boldsymbol{d}_{\boldsymbol{uv}}-\delta_{\boldsymbol{v}})}.$$

Thus, through reindexing, the dependence vector $\boldsymbol{d}_{\boldsymbol{uv}}$ becomes $\bar{\boldsymbol{d}}_{\boldsymbol{uv}}$ where

$$\bar{\boldsymbol{d}}_{\boldsymbol{uv}} = \boldsymbol{d}_{\boldsymbol{uv}} + \delta_{\boldsymbol{v}} - \delta_{\boldsymbol{u}}.$$

For example, picking $\delta_{\boldsymbol{p}} = (1\ 0)$ and null vectors for the other δ's, the DG in Fig. 3 is transformed after reindexing into the one depicted in Fig. 4. The new DG corresponds to rewritting the recurrences for $\boldsymbol{s}$ and $\boldsymbol{p}$ in (1) in the following way:

$$\begin{aligned} \boldsymbol{s}_{ij} &= \boldsymbol{u}_{i,j-1} - \boldsymbol{p}_{i-1,j} \\ \boldsymbol{p}_{ij} &= a_{i+1,j}\boldsymbol{u}^1_{ij}. \end{aligned}$$

It can be verified that these new definitions do not affect the functionality of the RIA.

Let $ec_1 \cdots ec_m$ denote the elementary circuits (EC's) in the DG and S_j be the *sum* of the dependence vectors in ec_j, $j = 1 \cdots m$. Then, obviously, the values of $S_1 \cdots S_m$ are independent of reindexing.

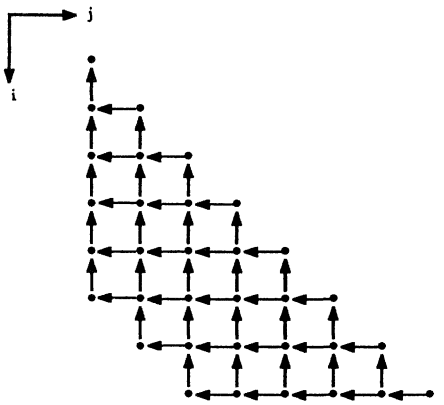


Fig. 2. The Full Dependence Graph (FDG) of the forward-substitution algorithm. The dependences due to the null dependence vectors are not displayed.

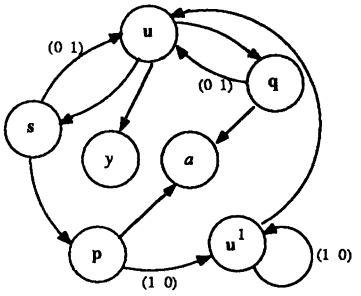


Fig. 3. The Dependence Graph (DG) of the forward-substitution algorithm. All unlabeled edges have null weights.

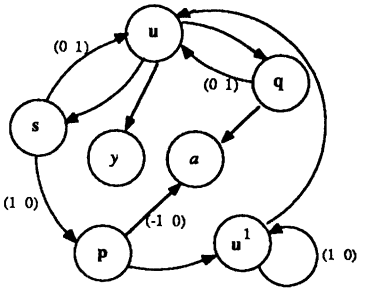


Fig. 4. The DG of the forward-substitution algorithm after reindexing. The reindexing process is equivalent to subtracting the vector (1 0) from the outgoing edges of node $\boldsymbol{p}$ and adding it to the incoming edge of that node.

Reindexing modifies the shape and the size of the computation range; the overall range is the *union* of the shifted computation ranges of the variables. Since the magnitudes of the offset vectors $\{\delta_i\}$ are independent of the size of Ω (N and p in the example), the increase in size of Ω due to reindexing can often be considered as insignificant. The notation $\bar{\Omega}$ is used to denote the range of computation after reindexing. For example, the number of index points in Ω is increased by N (Fig. 5) through the reindexing transformation presented above.

C. Linear Scheduling

A *valid* schedule for a reindexed RIA with dependence vectors $\{\bar{\boldsymbol{d}}_{\boldsymbol{uv}}\}$ is an ordering of the index points in $\bar{\Omega}$ such that the precedence relationships induced by these dependence

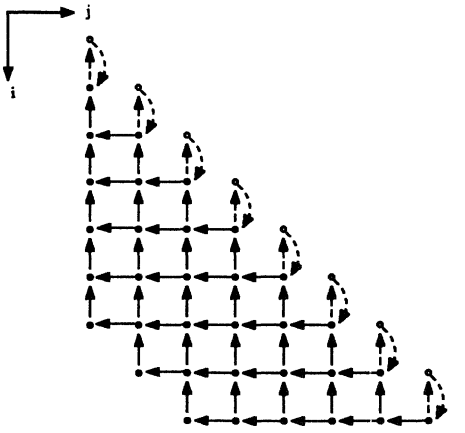


Fig. 5. The FDG of the reindexed forward-substitution algorithm. Empty circles are the extra index points and broken edges are the data dependences introduced by the reindexing. In particular, the curved broken edges denote the dependences due to $\boldsymbol{dpa}$.

vectors are observed. Due to the uniform dependence structure among the index points linear schedules are easily derived. More importantly, linear schedules, accompanied by linear PE allocation schemes, preserve the regularity of the data dependences and, hence, the resultant parallel implementations only need regular interconnections and homogeneous control—thus making the design and construction of hardware more tractable. However, time performance is sometimes traded for that simplicity: the fastest schedule possible is not always a linear schedule. For a given FDG with uniform dependences (the computations in a set of URE's) Karp *et al.* essentially show that, in general, to be able to approximate the time performance (total number of computation cycles) of the fastest schedule, linear schedules are inadequate and piecewise linear functions with rational coefficients must be considered [10], [7]. Fortes explored that issue experimentally [9]; for all of 25 algorithms tested the linear schedule minimizing n_{sys} did not exceed the fastest schedule by more than a cycle.

A linear schedule is specified by an integral column n-vector τ whereby the computations at an index point $\boldsymbol{Q}$ in $\bar{\Omega}$ are scheduled for execution at systolic cycle $\boldsymbol{Q}\tau$. Geometrically, τ is normal to a set of $(n-1)$-dimensional parallel hyperplanes which "slice" $\bar{\Omega}$ so that index points on the same hyperplane will be scheduled for execution at the same systolic cycle (see Fig. 6); that is, each of these hyperplanes represents one systolic cycle. The value of n_{sys}, which is the number of cycles between the first and last systolic cycle at which computation occurs, is therefore given by the total number of such parallel hyperplanes necessary to cover all the index points in $\bar{\Omega}$. Mathematically, the value of n_{sys} is related to τ by

$$n_{\text{sys}} = 1 + \text{MAX}_{\boldsymbol{Q}_1,\boldsymbol{Q}_2 \in \bar{\Omega}}(\boldsymbol{Q}_1 - \boldsymbol{Q}_2)\tau. \tag{2}$$

Since the range is an integral polytope, the above expression can be simplified to

$$n_{\text{sys}} = 1 + \text{MAX}_{\boldsymbol{V}_1,\boldsymbol{V}_2 \in V(\bar{\Omega})}(\boldsymbol{V}_1 - \boldsymbol{V}_2) \tag{3}$$

where $V(\bar{\Omega})$ is the set of vertices of $\bar{\Omega}$.

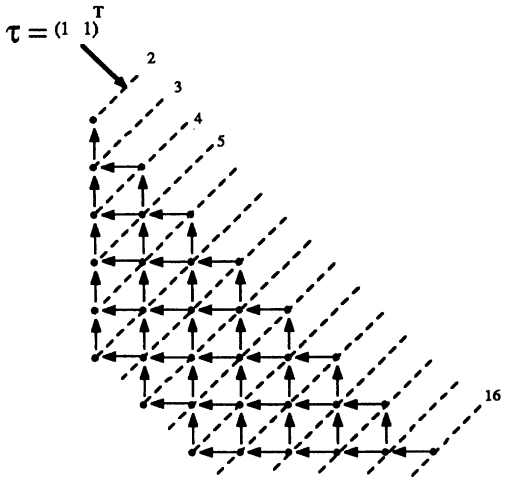


Fig. 6. The "slicing" hyperplanes defined by the linear schedule $\tau = (1\ 1)^T$. Each hyperplane represents one systolic cycle. Note that all the directed edges point from a hyperplane of larger systolic cycle value to a hyperplane of lower systolic cycle value.

The computations at a particular index point can only be executed when all the required data are available. Thus, τ has to be chosen such that the dependence vectors among the index points in $\bar{\Omega}$ do not point from a hyperplane of lower cycle value to one of a higher cycle value. Formally, a valid linear schedule τ has to satisfy the constraints

$$\boldsymbol{Q}\tau \geq (\boldsymbol{Q} - \bar{\boldsymbol{d}}_{\boldsymbol{uv}})\tau \qquad \forall \boldsymbol{Q} \in \bar{\Omega},$$

if variable $\boldsymbol{u}$ depends on variable $\boldsymbol{v}$ with dependence vector $\bar{\boldsymbol{d}}_{\boldsymbol{uv}}$. In other words, τ is a valid linear schedule for the RIA if

$$\bar{\boldsymbol{d}}_{\boldsymbol{uv}}\tau \geq 0 \qquad \forall \bar{\boldsymbol{d}}_{\boldsymbol{uv}}. \tag{4}$$

The value of the inner product $\bar{\boldsymbol{d}}_{\boldsymbol{uv}}\tau$ can be interpreted as the number of systolic cycles that elapse from the time the value of $\boldsymbol{v}$ at index point $(\boldsymbol{Q} - \bar{\boldsymbol{d}}_{\boldsymbol{uv}})$ is computed to the time it is used for computing $\boldsymbol{u}$ at index point $\boldsymbol{Q}$. In particular, a zero value for the inner product implies that the computations of $\boldsymbol{v}$ at $(\boldsymbol{Q} - \bar{\boldsymbol{d}}_{\boldsymbol{uv}})$ and of $\boldsymbol{u}$ at $\boldsymbol{Q}$ occur *sequentially* within the *same* systolic cycle (but not necessarily within the same PE) and hence, the cycle time must be increased.

The relationship between a chosen linear schedule τ and the resultant cycle time of the implementation can be observed from the associated *Register Graph* (RG), which is derived from the DG and τ with the following procedure:

1) associate to each node of the DG an integer value equal to the computation delay of the functional unit that computes the variable the node represents, and
2) *reverse* the direction of the edges and, on each edge, replace the weight, $\bar{\boldsymbol{d}}_{\boldsymbol{uv}}$ say, with the integer value $w_{\boldsymbol{vu}} = \boldsymbol{d}_{\boldsymbol{uv}}\tau$.

Each node in the RG represents a functional unit and the directed edges denote the direction of data flow among the functional units. The edge weights in a RG stand for the number of synchronization delays among the inputs and outputs of the functional units. Hereafter, for ease of discussion, the edge weights in the RG are called "registers" and the nodes in the RG are addressed by the names of the variables the nodes represent in the DG.

For the DG shown in Fig. 3, let $\tau = (1\ 1)^T$; the associated RG is shown in Fig. 7. The symbols T_Δ and $T_=$ denote respectively the computation delay of a register and of an assignment of the form $\boldsymbol{u} = \boldsymbol{v}$. In most cases, T_Δ and $T_=$ are assumed to be zero. There is a one-to-one correspondence between the connectivity structure of this RG and the interconnections among the functional units in array (I) (depicted in Fig. 1), which is a physical realization of the RG. Clearly, a register-free path in the array is represented by a zero-path in the RG. Since the cycle time t_{sys} of the array is given by the maximum sum of computation delays along the register-free paths, if a functional unit by itself is also treated as a "register-free" path (correspondingly, each node in the RG is taken to be a zero-path of zero length), then the cycle time associated with a physical realization of a RG is given by

$$t_{\text{sys}} = \text{MAX}_{\text{every zero-path in RG}}(\text{sum of computation delays on the zero-path}). \tag{5}$$

However, if τ is chosen such that the RG contains a zero-circuit, the corresponding realization exhibits *computation rippling*: a sequence of computations, whose length depends on the size of $\bar{\Omega}$, is scheduled to be computed sequentially (due to data dependences) within the same systolic cycle. Hence, the cycle time of the resultant schedule is proportional to the size of the range, which can be arbitrarily large.

For example, if the schedule $\tau = (1\ 0)^T$ is chosen for the DG shown in Fig. 3 the resultant RG will contain a zero-circuit between the nodes which compute variables $\boldsymbol{s}$ and $\boldsymbol{u}$. Consider the computation of variable $\boldsymbol{s}$ at a particular index point (i, j). Tracing the data dependences between variables $\boldsymbol{s}$ and $\boldsymbol{u}$ from the DG in Fig. 3 gives

$$\boldsymbol{s}_{ij} \rightarrow \boldsymbol{u}_{i,j-1} \rightarrow \boldsymbol{s}_{i,j-1} \rightarrow \boldsymbol{u}_{i,j-2} \rightarrow \boldsymbol{s}_{i,j-2} \rightarrow \cdots \rightarrow \boldsymbol{u}_{ik} \rightarrow y_i,$$

but, according to the schedule chosen, the computations of all these instances of $\boldsymbol{u}$ and $\boldsymbol{s}$ are to occur within the same systolic cycle, cycle i. Thus, the cycle time associated with the given τ must be proportional to p (the bandwidth of the matrix), a size parameter of $\bar{\Omega}$, and is not bounded as p increases. The definition of t_{sys} in (5) is consistent with this observation since a zero-circuit in the RG results in $t_{\text{sys}} = \infty$.

Assuming one of the nodes in ec_j represents the variable $\boldsymbol{u}$, then the computation of $\boldsymbol{u}_{\boldsymbol{Q}}$ has the data dependence

$$\boldsymbol{u}_{\boldsymbol{Q}} \rightarrow \cdots \rightarrow \boldsymbol{u}_{\boldsymbol{Q}-S_j} \rightarrow \cdots \rightarrow \boldsymbol{u}_{\boldsymbol{Q}-2S_j} \rightarrow \cdots.$$

The inner product $S_j\tau$ is the total number of registers on ec_j in the RG and therefore must be nonnegative. If $S_j\tau = 0$, then the computations of all these instances of variables on the dependence path have to occur within the same systolic cycle, $\boldsymbol{Q}\tau$. Equivalently, when $S_j\tau = 0$, there is a register-free feedback path around the functional unit for computing variable $\boldsymbol{u}$. Therefore, in order to avoid computation rippling,

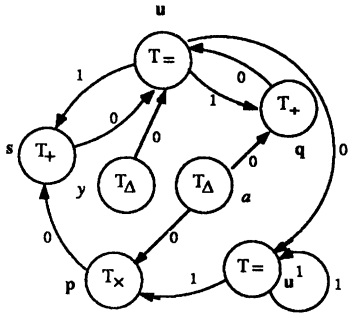


Fig. 7. The Register Graph (RG) derived from the DG in Fig. 3 with $\tau = (1\ 1)^T$. This RG is a graphical representation of the interconnections among the functional units in array (*I*) depicted in Fig. 1. For example, the output from the multiplier is fed directly into the input of the adder without buffering. Correspondingly, there is a directed edge with zero weight from the multiplier node to the adder node in the RG.

a valid linear schedule $\tau \in Z^n$ has to satisfy the additional constraint

$$S_j\tau = l_j \geq 1, \qquad j = 1 \cdots m \tag{6}$$

where S_j is the sum of the dependence vectors (or the edge weights) on the jth elementary circuit ec_j in the DG and l_j is an integer.

As mentioned in Section II-B, the values $S_1 \cdots S_m$ are independent of reindexing. Moreover, it is shown in [6] that (6) alone is a necessary and sufficient condition for τ to be a valid (rippling-free) linear schedule; that is, for each τ which satisfies (6), there is a reindexing such that (4) holds, and vice versa. Hence, it is not required that a suitable reindexing be known *a priori* in order to find a valid linear schedule for a RIA. Suppose the set of vectors $\{\delta_.\}$ is the appropriate reindexing for the given schedule τ, then the expression for the total number of systolic cycles in (2) is equivalent to

$$n_{\text{sys}} = 1 + \text{MAX}_{\boldsymbol{Q}_1,\boldsymbol{Q}_2 \in \Omega}(\boldsymbol{Q}_1 - \boldsymbol{Q}_2)\tau + \text{MAX}_{\boldsymbol{u},\boldsymbol{v}}(\delta_{\boldsymbol{u}} - \delta_{\boldsymbol{v}})\tau \tag{7}$$

where the last term reflects the increase in number of systolic cycles due to reindexing. As will be shown next, the appropriate reindexing, as well as the increase in number of cycles, are determined implicitly by the cycle time minimization procedure.

D. Retiming

Given a register graph RG, the cycle time t_{sys} determined with (5) is not necessarily the minimum achievable for the given τ. Leiserson *et al.* [18] show that with proper rearrangement of the registers in the RG, the cycle time can be minimized. A particular reallocation (following some rules) of the registers in the RG is called a *retiming*. The retiming procedure is in fact the one-dimensional version of the reindexing transformation discussed in Section II-B. To each node $\boldsymbol{u}$ in the RG is assigned an integer value, $r_{\boldsymbol{u}}$, called a label. An edge which extends from node $\boldsymbol{v}$ to node $\boldsymbol{u}$ with weight $w_{\boldsymbol{vu}}$ in the RG will have weight

$$\bar{w}_{\boldsymbol{vu}} = w_{\boldsymbol{vu}} + r_{\boldsymbol{v}} - r_{\boldsymbol{u}}$$

after retiming. Originally, the computation of variable $\boldsymbol{v}$ leads (algebraically) that of variable $\boldsymbol{u}$ by $w_{\boldsymbol{vu}}$ cycles. After retiming, the lead is $\bar{w}_{\boldsymbol{vu}}$ cycles. Since the values of the labels always occur in pairwise differences, it is assumed without loss of generality from now on that $\{r_.\}$ is a set of nonnegative integers and that the smallest element in the set equals zero. Equivalently, the retiming process can be viewed as shifting the registers along the interconnections among the functional units with the following rule [8]:

> "Shifting" r registers from the input stage to the output stage of the functional unit that computes variable $\boldsymbol{u}$ is equivalent to subtracting r from the weight of every incoming edge of node $\boldsymbol{u}$ in the RG and adding r to the weight of every outgoing edge of that node.

It is shown in [8] and [18] that such shifting of registers preserves the functionality of the computations represented by the RG.

A *legal* retiming is a set of labels $\{r_.\}$, one label for each node in the RG, such that all the edges in the retimed RG have nonnegative weights; a legal retiming is *optimal* if the cycle time of the retimed RG is the minimum among all the legal retimings. A necessary and sufficient condition for a RG to have a legal retiming is that the sum of edge weights in any EC in the RG (i.e., $S_j\tau$ or l_j) is strictly positive [18]. Hence, from (6), a RG derived from a valid schedule τ always possesses a legal retiming. Since retiming is a special case of reindexing, the total number of registers [l_j in (6)] in any EC in a RG is independent of retiming.

The relationship between retiming and reindexing is simple. Suppose τ is the valid linear schedule from which the RG is derived, then a retiming of the RG with labels $\{r_.\}$ is equivalent to a reindexing of the DG with offsets $\{\delta_.\}$ where

$$r_{\boldsymbol{u}} = \delta_{\boldsymbol{u}}\tau \qquad \forall \delta_{\boldsymbol{u}}.$$

Since, from the definition of reindexing,

$$\bar{\boldsymbol{d}}_{\boldsymbol{uv}} = \boldsymbol{d}_{\boldsymbol{uv}} + \delta_{\boldsymbol{v}} - \delta_{\boldsymbol{u}},$$

with this set of offsets,

$$\bar{\boldsymbol{d}}_{\boldsymbol{uv}}\tau = w_{\boldsymbol{vu}} + r_{\boldsymbol{v}} - r_{\boldsymbol{u}} = \bar{w}_{\boldsymbol{vu}} \geq 0,$$

hence the reindexing and τ satisfy (4). Knowing τ and an optimal retiming, an appropriate reindexing can be determined accordingly. Expression (7) can be rewritten as

$$n_{\text{sys}} = 1 + \text{MAX}_{\boldsymbol{Q}_1,\boldsymbol{Q}_2 \in \Omega}(\boldsymbol{Q}_1 - \boldsymbol{Q}_2)\tau + r_{\max} \tag{8}$$

which is equivalent to

$$n_{\text{sys}} = 1 + \text{MAX}_{\boldsymbol{V}_1,\boldsymbol{V}_2 \in V(\Omega)}(\boldsymbol{V}_1 - \boldsymbol{V}_2)\tau + r_{\max} \tag{9}$$

where $V(\Omega)$ is the set of vertices of Ω and $r_{\max}$ is the maximum among the labels in $\{r_.\}$,

$$r_{\max} \equiv \text{MAX}_{\boldsymbol{u}}\, r_{\boldsymbol{u}}. \tag{10}$$

Given a RG, the minimum cycle time achievable with retiming is given by

$$t_{\text{sys}} = \text{op\_retime(RG)}. \tag{11}$$

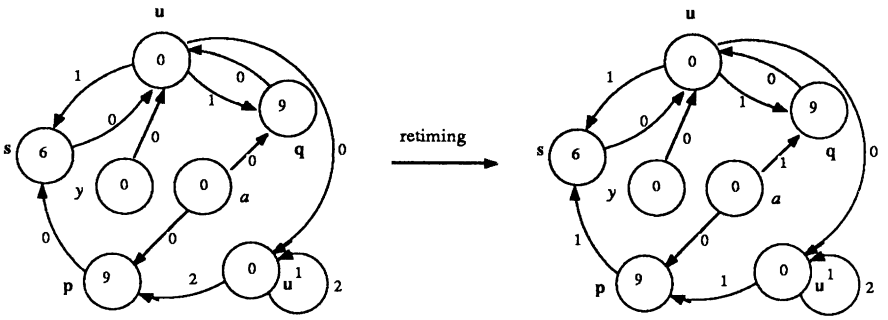


Fig. 8. With the retiming $r_a = r_{\boldsymbol{p}} = 1$ and $r = 0$ for the other variables the cycle time of the RG, which is derived from the DG with $\tau = (2\ 1)^T$, is reduced from 15 to 9 time units.

To implement the op_retime function, Leiserson *et al.* [18] formulate the optimization problem as a shortest-path problem whose solution successively employs the Floyd–Warshall algorithm and the Bellman–Ford algorithm [17].

For example, if $\tau = (2\ 1)^T$ is the schedule chosen for the forward-substitution algorithm, the corresponding RG is as shown in Fig. 8. With the retiming where r_a and $r_{\boldsymbol{p}}$ equal 1 and the rest of the labels equal 0 the cycle time is reduced from 15 to 9 time units.

Two different retimings which result in different register assignments on the edges of the RG may give the same cycle time; such retimings are called *equivalent*. The op_retime function returns only one of the (possibly many) equivalent optimal retimings. Additional criteria may be used to select a particular optimal retiming. For example, the optimal retiming which minimizes the value of $r_{\max}$ defined in (10) can be found by solving a linear program [29], which may be done in polynomial time; the same applies to finding the optimal retiming which minimizes the total sum of edge weights (the total number of registers) [18].

E. Problem Formulation

Combining the results from the previous discussion, the determination of a computation-time-optimal linear schedule for a RIA can be formulated as a discrete optimization problem

$$\begin{cases} \text{minimize} & n_{\text{sys}} \times t_{\text{sys}} \\ \text{subject to} & S_j\tau \geq 1, \qquad j = 1 \cdots m \\ & n_{\text{sys}} = 1 + \text{MAX}_{\boldsymbol{V}_1, \boldsymbol{V}_2 \in V(\Omega)}(\boldsymbol{V}_1 - \boldsymbol{V}_2)\tau \\ & \qquad +r_{\max} \\ & t_{\text{sys}} = \text{op\_retime(RG)} \\ & \tau \in \boldsymbol{Z}^n. \end{cases} \tag{12}$$

Since the quantities t_{sys} and $r_{\max}$ do not have a simple closed-form expression (in terms of τ), "smart" enumeration, i.e., branch-and-bound, is the only viable solution method for the optimization problem. However, a direct enumeration on τ may not terminate since the constraints on τ in

$$S_j\tau \geq 1, \qquad j = 1 \cdots m$$

form an affine cone [23], which admits infinitely many feasible solutions. An alternative formulation of the optimization problem must be sought.

Consider making $l_1 \cdots l_m$, the number of registers in the EC's of the RG, the decision variables of the optimization problem. From (6), the value of l_j constrains the choice of τ such that there are *exactly* l_j registers in ec_j in the resultant RG. The choice of the set of values $\{l_.\}$ affects the values of t_{sys} and n_{sys} indirectly. A set with large values could decrease the value of t_{sys} since the existence of a long zero-path in the RG is then less likely. On the other hand, a set with small values could lead to a τ of smaller magnitude and may, therefore, reduce the value of n_{sys}.

For a given set of values $\{l_.\}$ the valid choices of τ, if any exists, are the integral solutions of the set of equalities

$$S_j\tau = l_j, \qquad j = 1 \cdots m. \tag{13}$$

The following theorem establishes that the minimum cycle time of a RG achievable with retiming is totally determined by the set of values $\{l_.\}$.

Theorem 1: Given a RG, there always exists an optimal retiming such that all the exterior edges have nonzero weights.

Proof: Suppose there are exterior edges with zero weights after the RG has been optimally retimed. Collapse each strongly connected component of the RG into a single node. By definition, the collapsed graph is acyclic and the edges in this graph are the exterior edges in the original RG. Now merge all the nodes which have zero indegree in the collapsed graph into a single node, N_0. Finally, apply the single source longest path algorithm to this graph with N_0 as the source node and length (distance from N_0) being defined as the number of edges on the path. Subtracting these distances from the labels of the original optimal retiming (nodes that were collapsed together have the same distance from N_0), then offsetting these label values such that the minimum label value equals 0, gives a new retiming. The new set of labels guarantees that all exterior edges will have nonzero weights. Furthermore, since the labels of all the nodes in a strongly connected component receive the same adjustment (with respect to the original set of labels), the register assignments in the EC's are preserved. The new set of labels thus preserves the cycle time of the RG and hence it represents an equivalent optimal retiming. □

Since the cycle time of a RG depends on the zero-paths, the above theorem implies that the exterior edges do not affect the minimum achievable cycle time. In other words, the minimum

cycle time of the RG under retiming depends only on the number of registers in the EC's of the RG, $\{l_i\}$.

Let $L = (l_1 \cdots l_m)$ be an integral m-vector with positive components, called a *register assignment*. A register assignment L is *consistent* if the system of diophantine equations

$$Cw = L^T, \qquad w \in \boldsymbol{Z}^m$$

where C is the EC-edge matrix of the DG, has a solution. Suppose L is consistent and let w be a particular solution of the system of diophantine equations. Let $\underline{\mathrm{RG}}$ be the directed graph that results from deleting all the exterior edges in the DG, replacing the node labels with the computation delays of the appropriate functional units, reversing the direction of the edges, and setting the edge weights equal to the corresponding component values of vector w. Then, from Theorem 1, the minimum cycle time of the RG derived from the DG with any τ which satisfies

$$S_j\tau = l_j, \qquad j = 1 \cdots m \tag{14}$$

must be equal to the minimum cycle time of $\underline{\mathrm{RG}}$. Hence, for a given L, the minimum cycle time t_{sys} is fixed,

$$cycle\_time(L) = \begin{cases} \text{op\_retime}(\underline{\mathrm{RG}}) & \text{if } L \text{ is consistent} \\ \infty & \text{otherwise.} \end{cases} \tag{15}$$

Therefore, for a given consistent register assignment L, the computation-time-optimal linear schedule τ can be determined by solving the integer programming problem

$$\begin{cases} \text{minimize} & n_{\mathrm{sys}} = 1 + \mathrm{MAX}_{\boldsymbol{V}_1,\boldsymbol{V}_2 \in V(\Omega)}(\boldsymbol{V}_1 - \boldsymbol{V}_2)\tau \\ & \quad + r_{\max} \\ \text{subject to} & S_j\tau = l_j \qquad j = 1 \cdots m \\ & \tau \in \boldsymbol{Z}^n. \end{cases} \tag{16}$$

However, the objective function still contains $r_{\max}$, which does not have a simple closed-form expression in terms of τ. By observing that $\tau_{\max}$ is independent of the size of Ω and recalling from Section II-D that its value can be minimized easily, it is justified to assume that

$$\mathrm{MAX}_{\boldsymbol{V}_1,\boldsymbol{V}_2 \in V(\Omega)}(\boldsymbol{V}_1 - \boldsymbol{V}_2)\tau \gg r_{\max}.$$

This assumption may not hold only if Ω is small, but then a systolic array implementation is probably not needed. Examples will be provided in Section IV to show that the assumption holds in practical cases. Therefore, a feasible solution of (16) which minimizes

$$\mathrm{MAX}_{\boldsymbol{V}_1,\boldsymbol{V}_2 \in V(\Omega)}(\boldsymbol{V}_1 - \boldsymbol{V}_2)\tau$$

is also an optimal solution of (16). Hence, for a given assignment L, the computation-time-optimal linear schedule τ is given by the solution of the ILP:

$$\begin{cases} \text{minimize} & \mathrm{MAX}_{\boldsymbol{V}_1,\boldsymbol{V}_2 \in V(\Omega)}(\boldsymbol{V}_1 - \boldsymbol{V}_2)\tau \\ \text{subject to} & S_j\tau = l_j \qquad j = 1 \cdots m \\ & \tau \in \boldsymbol{Z}^n. \end{cases} \tag{17}$$

The number of systolic cycles associated with a particular register assignment L is

$$ncycle(L) = \begin{cases} 1 + \mathrm{MAX}_{\boldsymbol{V}_1,\boldsymbol{V}_2 \in V(\Omega)}(\boldsymbol{V}_1 - \boldsymbol{V}_2)\tau + r_{\max} \\ \qquad \text{if (17) has an optimal solution} \\ \infty \quad \text{otherwise.} \end{cases}$$

where $r_{\max}$ is the maximum of the label values found by retiming the RG, which is derived from the DG with the schedule τ obtained from ILP (17). Consequently, if the total number of consistent register assignments L that need to be considered is bounded, an optimal solution to the discrete optimization problem (12) can be found in a finite number of iterations.

The number of different L's that need to be considered is shown in the next section to be indeed finite. An efficient branch-and bound method is then devised to locate the assignment L which minimizes the value of $n_{\mathrm{sys}} \times t_{\mathrm{sys}}$.

III. Bounded Search Space

Let λ_j be the number of edges in ec_j, $j = 1 \cdots m$. An elementary circuit ec_j in the RG associated to a valid linear schedule is said to be *register-sufficient* (a register-sufficient EC) if there are at least λ_j registers in the EC, else it is said to be *register-deficient* (a register-deficient EC). An edge is said to be *free* if it is shared only by register-sufficient EC's, otherwise, it is said to be *bound*.

The following theorem establishes that the minimum cycle time of a RG under retiming is independent of the actual number of registers in the register-sufficient EC's.

Theorem 2: Let RG_1 and RG_2 be two register graphs where RG_2 is obtained from deleting all the free edges in RG_1. Then, RG_1 and RG_2 have the same minimum cycle time under retiming.

Proof: Let t^1_{sys} and t^2_{sys} be the respective minimum cycle times of RG_1 and RG_2, and $\{r^1_\cdot\}$ and $\{r^2_\cdot\}$ be the respective optimal retimings chosen. Moreover, from Theorem 1, all the exterior edges in the two graphs are assumed to have nonzero weights after retiming.

Since RG_2 is a subgraph of RG_1, an optimal retiming for RG_1 must also be a *legal* retiming for RG_2. Hence,

$$t^1_{\mathrm{sys}} \geq t^2_{\mathrm{sys}}. \tag{18}$$

Now, when applied to RG_1, the retiming $\{r^2_\cdot\}$ is not necessarily legal. If it is legal, then

$$t^1_{\mathrm{sys}} \leq t^2_{\mathrm{sys}} \tag{19}$$

and the desired result follows.

If it is illegal, then retiming RG_1 with the labels $\{r^2_\cdot\}$ must result in negative edge weights. Call this retimed graph RG_3. If a legal retiming for RG_3, $\{r^3_\cdot\}$, such that the cycle time of the retimed RG_3 does not exceed t^2_{sys} could be constructed, then, by adding the labels $\{r^3_\cdot\}$ to $\{r^2_\cdot\}$ and offsetting the resultant labels such that the minimum label value equals 0, a legal retiming for RG_1 such that the cycle time of the retimed RG_1 does not exceed t^2_{sys} would ensue, (19) would hold, and the proof would be completed. It remains to construct the set of labels $\{r^3_\cdot\}$.

Since RG_2 is a subgraph of RG_3 and $\{r^2_\cdot\}$ is a legal retiming for RG_2, the edges in RG_3 whose weights are negative *must* belong to the set of free edges. Given a node $\boldsymbol{s}$ in RG_3 with an *incoming* edge that is free and has nonpositive weight, the procedure RELABEL($\boldsymbol{s}$) defined in Fig. 9 returns

```
Procedure RELABEL(s)
Begin
 for each outgoing edge e of s do
  let q ← e.head.node ;
  if q.label = 1 and e.weight ≤ 1 then
   Begin
    if q ≡ v then
     begin
      Error: e 0 is not free!
      STOP;
     end;
    q.label ← 0;                    (*)
    RELABEL(q);
   end;
end;
```

Fig. 9. Procedure RELABEL.

a retiming of RG_3 such that the weight of this edge increases by 1.

Let e_0 be a free edge with nonpositive weight. Suppose e_0 extends from node $\boldsymbol{v}$ to node $\boldsymbol{u}$ in RG_3. Collapse each strongly connected component in RG_3 which does not contain e_0 into a single node. Call the new graph $\overline{RG}_3$. Define the variable *label* for each node in $\overline{RG}_3$. For each edge e in $\overline{RG}_3$, let $e.weight$ be the weight of e and let $e.head.node$ and $e.tail.node$ be, respectively, the nodes associated with the head and tail of e. If two or more edges go from one node to another then only the edge with the minimum weight is used; the other edges are deleted from the graph. Set $\boldsymbol{u}.label$ to 0 and the *labels* of the remaining nodes in $\overline{RG}_3$ to 1. Then the values of *label* set by the call

$$\text{RELABEL}(\boldsymbol{u})$$

make up a retiming for RG_3 if nodes that are collapsed into a single node in $\overline{RG}_3$ are assigned label values equal to the label value of the corresponding collapsed node. With this retiming, the weight of e_0 is increased by 1,

$$e_0.weight \leftarrow e_0.weight + \boldsymbol{v}.label - \boldsymbol{u}.label = e_0.weight + 1 - 0.$$

Furthermore, as will be proved later on, the retiming does not increase the number of edges with nonpositive weights nor the number of bound zero-edges and it does not decrease the values of the nonpositive edge weights in RG_3. Hence, by updating the edge weights in RG_3 with this set of labels and repeating the procedure an appropriate number of times, the weight of e_0 eventually becomes 1. If that procedure is applied to all the free edges with nonpositive weights in RG_3, eventually all the free edges in RG_3 have positive weights. Since no bound zero-edge is introduced by the procedure and all the free edges have nonzero weights at the end, the zero paths in the final retimed RG_3 must be a subset of those in RG_2 and hence, the cycle time of the final RG_3 cannot be greater than t_{sys}^2.

To prove that procedure RELABEL indeed returns a retiming with the above mentioned properties, it is only necessary to show that, for any pair of nodes $\boldsymbol{s}$ and $\boldsymbol{q}$ in RG_3 connected by an edge e directed from $\boldsymbol{s}$ to $\boldsymbol{q}$, the labels returned by procedure RELABEL satisfy the condition

$$\begin{cases} e.weight + \boldsymbol{s}.label - \boldsymbol{q}.label \geq e.weight & \text{if } e.weight \leq 1 \\ e.weight + \boldsymbol{s}.label - \boldsymbol{q}.label \geq 1 & \text{if } e.weight > 1. \end{cases} \tag{20}$$

The condition ensures that the values of the nonpositive edge weights in RG_3 are not decreased by the labels (first case with $e.weight \leq 0$) and that no extra bound zero-edge is introduced (first and second case with $e.weight \geq 1$).

Termination: Clearly, the procedure RELABEL terminates after a finite number of iterations since the number of nodes in $\overline{RG}_3$ is finite and once a label has been set to 0 in one iteration it is left unaltered.

Correctness: For the case $e.weight > 1$, condition (20) is automatically satisfied by the labels returned from procedure RELABEL since these labels can only have the values 0 or 1. For $e.weight \leq 1$, the labels fail to satisfy (20) only if $\boldsymbol{s}.label$ is 0 and $\boldsymbol{q}.label$ is 1. This is, however, impossible because if $\boldsymbol{s}.label$ was set to 0 at line (*) in the procedure then, since $\boldsymbol{q}$ is the head node of an outgoing edge (with weight ≤ 1) of $\boldsymbol{s}$, the next recursive call of RELABEL on node $\boldsymbol{s}$ would have set the value of $\boldsymbol{q}.label$ to 0 at line (*). Once set to 0, this label cannot change any more. Therefore, if the final value of $\boldsymbol{s}.label$ is 0 the final value of $\boldsymbol{q}.label$ must also be 0. Thus, the labels returned by procedure RELABEL satisfy condition (20).

Abnormal termination: Suppose abnormal termination occurs at node $\boldsymbol{q}$, then there is an edge e from $\boldsymbol{q}$ to $\boldsymbol{v}$ with $e.weight \leq 1$, and $\boldsymbol{q}.label$ is equal to 0. Now, $\boldsymbol{q}.label$ is set to 0 only if there is an edge with weight ≤ 1 *incoming* at $\boldsymbol{q}$ and originating from another node, $\boldsymbol{s}$ say, with $\boldsymbol{s}.label$ already 0. Repeating this argument and noting that the RELABEL procedure visits the nodes in $\overline{RG}_3$ in a depth-first ordering starting from node $\boldsymbol{u}$ (see Fig. 10), there must exist a path from $\boldsymbol{u}$ to $\boldsymbol{v}$, which contains nodes $\boldsymbol{s}$ and $\boldsymbol{q}$, whose edges all have weights ≤ 1. With the edge e_0, which has nonpositive weight, this path forms a register-deficient elementary circuit and therefore e_0 cannot be free. □

With this result, all the free edges in a RG can be deleted without affecting the minimum cycle time. The rest of the graph is totally represented by the register-deficient EC's in the original RG. Hence, the minimum cycle time of a RG under retiming is determined *solely* by the number of registers in the register-deficient EC's, that is, by the set of values $\{l_j \mid l_j < \lambda_j\}$. Consequently, it suffices to consider the set of register assignments $\{L \mid 1 \leq l_j \leq \lambda_j,\ j = 1 \cdots m\}$, where $l_j = \lambda_j$ is to be construed as meaning that there are *at least* λ_j registers in ec_j, and the functions $cycle\_time$ and $ncycle$ are redefined in the following manner.

Without loss of generality, suppose L is such that the first i elementary circuits, $ec_1 \cdots ec_i$ are register-sufficient and the remaining ones are register-deficient, that is, $L = (l_1 \cdots l_i l_{i+1} \ldots l_m)$ where $l_j = \lambda_j$, $j = 1 \cdots i$, and $l_j < \lambda_j$, $j = i+1 \cdots m$. Register assignment L is said to be consistent if the system of diophantine equations

$$\bar{C}\bar{w} = (l_{i+1} \cdots l_m)^T$$

where $\bar{C}$ denotes the matrix formed by the last $m - i$ rows of the EC-edge matrix of the DG, has a solution. The value of

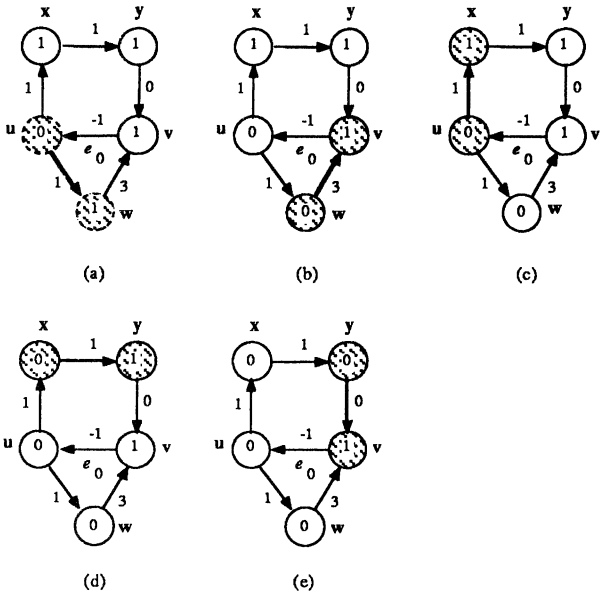


Fig. 10. Snapshots of the procedure call RELABEL($\boldsymbol{u}$) with the nodes and edges being operated on (nodes $\boldsymbol{s}$, $\boldsymbol{q}$, and the edge e in Fig. 9) highlighted: (a) Since $e.weight \leq 1$, the value of $\boldsymbol{w}.label$ is set to 0 and the recursive procedure call RELABEL($\boldsymbol{w}$) is invoked. (b) Now $e.weight > 1$ and the label of the node $\boldsymbol{v}$ is unaltered. (c) Backtrack to the next outgoing edge of node $\boldsymbol{u}$. Since $e.weight \leq 1$, the value of $\boldsymbol{x}.label$ is set to 0. (d) Similarly, $\boldsymbol{y}.label$ is set to 0. (e) An error condition arises because $e.weight \leq 1$ and the $e.head.node$ is $\boldsymbol{v}$. Indeed, edge e_0 is part of the register-deficient elementary circuit containing the nodes $\{\boldsymbol{v}, \boldsymbol{u}, \boldsymbol{x}, \boldsymbol{y}\}$.

$cycle\_time(L)$ is ∞ if L is inconsistent, else it is given by

$cycle\_time(L) =$

- delete all the edges which are not part of any of $ec_{i+1} \cdots ec_m$ in the DG
- form RG : set edge weights according to $\bar{w}$
- return op_time(RG)

and the value of $ncycle(L)$ is $(1 + Y + r_{\max})$ where Y is obtained from solving the ILP

$$\begin{cases} \text{minimize} & Y \\ \text{subject to} & Y - (\boldsymbol{V}_i - \boldsymbol{V}_j)\tau \geq 0, \quad \forall \boldsymbol{V}_i, \boldsymbol{V}_j \in V(\Omega) \\ & S_j\tau = \lambda_j, \quad j = 1 \cdots i \\ & S_j\tau = l_j, \quad j = i+1 \cdots m \\ & \tau \in \boldsymbol{Z}^n \end{cases} \tag{21}$$

and $r_{\max}$ is the maximum among the labels returned from retiming the RG, which is derived from the DG with the solution τ of the ILP. The constraints $S_j\tau \geq \lambda_j$, $j = 1 \cdots i$, in the above ILP ensure that the solution τ is chosen such that the elementary circuits $ec_1 \cdots ec_i$ in the RG are indeed register-sufficient. The *actual* numbers of registers in the register-sufficient EC's are determined by the chosen schedule τ.

The set of values

$$\{cycle\_time(L) | 1 \leq l_j \leq \lambda_j, j = 1 \cdots m\}$$

represents all possible minimum cycle time values of the RG's derived from the given DG with any valid schedule. With the functions $ncycle$ and $cycle\_time$ defined above, the register assignment L (and hence the τ) which minimizes the product $n_{\text{sys}} \times t_{\text{sys}}$ can be determined with exhaustive enumeration on all the possible L's. Such enumeration scheme requires a maximum of $\prod_{j=1}^{m} \lambda_j$ iterations.

IV. Branch-and-Bound Method

To find the time-optimal schedule τ with exhaustive enumeration is computationally expensive because, apart from the large number of L vectors that need to be considered, the function $ncycle(L)$ calls for the solution of an ILP (21) which is, in general, hard to solve. In this section, it is shown that a decision tree of m levels can be used to construct the optimal L vector *incrementally*. An efficient branch-and-bound method is devised to prune the decision tree to speed up the process. The performance of this approach depends on the quality of the initial solution and on the lower bounding function used.

Each level of the decision tree represents one component of L: the first level contains λ_1 nodes with labels $1, \cdots, \lambda_1$; to each node with label l_1 at the first level correspond λ_2 nodes at the second level with labels $l_1 1$ to $l_1\lambda_2$, and so on (Fig. 11). Nodes at the ith level of the decision tree have labels of length i, $l_1 l_2 \cdots l_i$, and therefore, the labels of the leaf nodes on the tree correspond to the full L vectors.

Let L_i, called a *partial* register assignment, be the label of a node, $node_i$, at the ith level. Define the two functions $lb\_cycle\_time(L_i)$ and $lb\_ncycle(L_i)$ as

$lb\_cycle\_time(L_i) =$

- delete all edges which do not belong to any of $ec_1 \cdots ec_i$
- delete all edges shared only by $\{ec_j | l_j = \lambda_j \ , \ j = 1 \cdots i\}$
- form RG if L_i is consistent
- return op_time(RG) if L_i is consistent else return ∞

and $lb\_ncycle(L_i) = 1 + Y$ if the linear program (LP)

$$\begin{cases} \text{minimize } Y \\ \text{subject to } Y - (\boldsymbol{V}_i - \boldsymbol{V}_j)\tau \geq 0, \quad \forall \boldsymbol{V}_i, \boldsymbol{V}_j \in V(\Omega) \\ \quad S_j\tau = l_j, \text{ if } l_j < \lambda_j, \ j = 1 \cdots i \text{ (register\_deficient EC)} \\ \quad S_j\tau \geq \lambda_j, \text{ if } l_j = \lambda_j, \ j = 1 \cdots i \text{(register\_sufficient EC)} \\ \quad S_j\tau \geq 1, \ j = i+1 \cdots m \ (l_j \text{ not yet assigned}) \\ \quad \tau \in \boldsymbol{Q}^n \text{ (rational } n\text{-vector)} \end{cases} \tag{22}$$

is feasible, else $lb\_ncycle(L_i) = \infty$.

Clearly, the label of any leaf node in the subtree that roots at $node_i$ is of the form $L = (L_i \mid l_{i+1} \cdots l_m)$ for some $(l_{i+1} \cdots l_m)$, and

$$\begin{cases} lb\_cycle\_time(L_i) \leq cycle\_time(L) \\ lb\_ncycle(L_i) \leq ncycle(L). \end{cases}$$

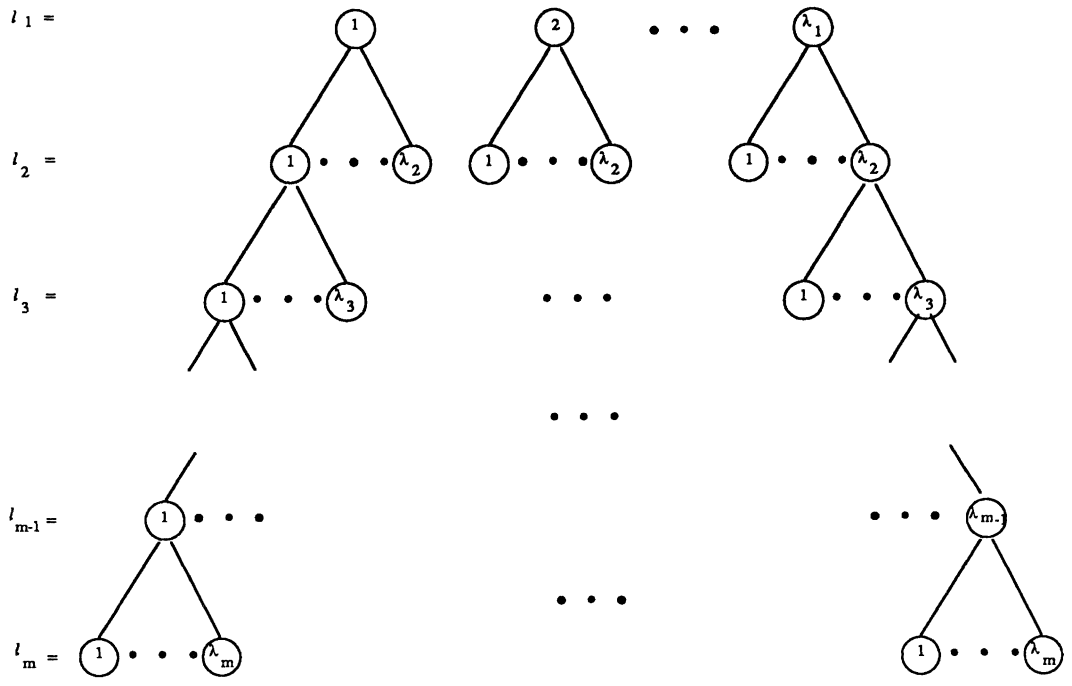


Fig. 11. The structure of the decision tree used by the branch-and-bound method.

Thus, if the value of $lb\_ncycle(L_i) \times lb\_cycle\_time(L_i)$ is already larger than that of the incumbent solution then the subtree at $node_i$ should be pruned. This provides a bounding function for the branch-and-bound method:

$$lb(L_i) = lb\_ncycle(L_i) \times lb\_cycle\_time(L_i).$$

The computation cost of the bounding function is low (polynomial time) : $lb\_ncycle(L_i)$ involves a linear program which can be solved in polynomial time, and $lb\_cycle\_time(L_i)$ can be computed with the procedure detailed in [18] which has polynomial time complexity. At the leaf level, however, the lb_ncycle function has to be replaced by the $ncycle$ function to produce an integral solution τ. Due to the interaction among elementary circuits which share part of their edges, many partial register assignments L_i are either inconsistent or the constraint set in the LP (22) is infeasible. The subtree that roots at a node with such a label L_i can be pruned. To take advantage of this property, elementary circuits which share edges should, if possible, be numbered consecutively. An ordering heuristic similar to the one proposed in [4] could be used for that purpose.

An initial incumbent solution for the branch-and-bound method is chosen between the schedules that minimize one of the two objectives: n_{sys} and t_{sys}. The schedule that minimizes n_{sys} is found by solving the ILP:

$$\begin{cases} \text{minimize} & Y \\ \text{subject to} & Y - (\boldsymbol{V}_i - \boldsymbol{V}_j)\tau \geq 0, \qquad \forall \boldsymbol{V}_i, \boldsymbol{V}_j \in V(\Omega) \\ & S_j\tau \geq 1, \qquad j = 1 \cdots m \\ & \tau \in \boldsymbol{Z}^n. \end{cases}$$

Having found the schedule τ, retiming is performed on the associated RG to determine the minimum cycle time t_{sys} and the value of $r_{\max}$. For the second objective, the minimum t_{sys} is given by the maximum computation delay of the functional units (or node labels) in the RG. A schedule which achieves this cycle time is found by solving the ILP:

$$\begin{cases} \text{minimize} & Y \\ \text{subject to} & Y - (\boldsymbol{V}_i - \boldsymbol{V}_j)\tau \geq 0, \qquad \forall \boldsymbol{V}_i, \boldsymbol{V}_j \in V(\Omega) \\ & S_j\tau \geq \lambda_j, \qquad j = 1 \cdots m \\ & \tau \in \boldsymbol{Z}^n. \end{cases}$$

Of the above two schedules, the one with smaller $n_{\text{sys}} \times t_{\text{sys}}$ value should be used as the initial solution of the branch-and-bound method.

Example 2: Consider one of the steps in the Toeplitz Hyperbolic Cholesky Solver (THCS) [6], which can be expressed as a RIA of six recurrences:

$$\rho_{j+1,j} = \boldsymbol{s}_{j+1,j-1}/\boldsymbol{r}_{j,j-1} \qquad \boldsymbol{s}_{ij} = \boldsymbol{s}_{i,j-1} - \boldsymbol{q}_{ij}$$
$$\boldsymbol{r}_{ij} = \boldsymbol{r}_{i-1,j-1} - \boldsymbol{p}_{ij} \qquad \boldsymbol{q}_{ij} = \bar{\rho}_{ij} \times \boldsymbol{r}_{i-1,j-1}$$
$$\boldsymbol{p}_{ij} = \bar{\rho}_{ij} \times \boldsymbol{s}_{i,j-1} \qquad \bar{\rho}_{ij} = \begin{cases} \bar{\rho}_{i-1,j} & \text{if } i \neq j+1 \\ \rho_{ij} & \text{otherwise.} \end{cases}$$

The range of computation is a triangle, defined by the inequalities $1 \leq j < i \leq N$, where N is the size of the Toeplitz matrix. The DG of this RIA is shown in Fig. 12(a). Let the computation delays of the adder, multiplier, and divider be 1, 5, and 5 time units, respectively.

The n_{sys}-optimal design has $\tau = (1\ 1)^T$ and $n_{\text{sys}} = 2N - 2$. The corresponding RG for this choice of τ is depicted in Fig. 12(b). The minimum cycle time is $t_{\text{sys}} = 11$ ($r_{\max} = 0$). Hence, the overall time performance of the systolic implementation derived with $\tau = (1\ 1)^T$ is

(a) (b)

Fig. 12. (a) The DG of one of the steps in THCS and (b) the corresponding RG derived with $\tau = (1\ 1)^T$.

$n_{\text{sys}} \times t_{\text{sys}} = (2N-2) \times 11$ time units. For $N = 10$, say, the value equals 198 time units.

On the other hand, a t_{sys}-optimal design can be obtained from picking, say, $\tau = (1\ 3)^T$. This choice increases the value of n_{sys} to $4N-1$ but, with retiming, the value of t_{sys} is reduced to 5 time units [see Fig. 13(a) and (b)], which is the minimum achievable. The value of $r_{\max}$ is 3 and the overall time performance is $(4N-1) \times 5$ time units. For $N = 10$, the total time required equals 195 time units which is marginally better than the time of the n_{sys}-optimal design.

Now, consider the linear schedule, $\tau = (1\ 2)^T$. The corresponding RG is shown in Fig. 14(a). The value of n_{sys} is $3N$, and the value of t_{sys} obtained from retiming ($r_{\max} = 3$) is 6 [Fig. 14(b)], neither of which is the minimum value achievable. However, the overall time performance of the schedule is $3N \times 6$ time units, which is better than the previous two schedules for $N > 5$. Implementation using this (optimal) linear schedule requires roughly 10% less time than the previous two designs to compute the same RIA. This improvement in performance is essentially independent of the actual value of N, provided that N is larger than 10. □

Example 3: The N point Discrete Fourier Transform (DFT) of a data sequence x_q, $q = 0 \cdots N-1$, is computed from the formula

$$y_k = \sum_{q=0}^{N-1} x_q w_N^{kq} \qquad k = 0 \cdots N-1$$

where $w_N = \exp(-j2\pi/N)$. Decomposing the complex multiply into real multiplies and addition/subtractions and using Horner's rule, the formula can be rewritten as the RIA shown at the bottom of this page.

The DG is shown in Fig. 15(a); unlabeled edges have (0 0) as their weights. The range of computation is a rectangle $0 \le k \le N-1$, $1 \le q \le N$. Let the computation delays of the multiplier and the adder/subtractor be 10 and 6 time units, respectively. Furthermore, let N be 256.

The n_{sys}-optimal schedule for the algorithm is found to be $\tau = (1\ 1)^T$ and $n_{\text{sys}} = 2N-1$; the corresponding RG is shown in Fig. 15(b). The cycle time is $t_{\text{sys}} = 22$ time units ($r_{\max} = 0$). Hence, the overall computation time of the DFT algorithm with this schedule is $(2N-1) \times 22$ which equals 11242 time units for the given N.

The t_{sys}-optimal schedule for the algorithm is given by $\tau = (1\ 3)^T$. The RG's before and after retiming are shown in Fig. 16(a) and Fig. 16(b), respectively. The cycle time is $t_{\text{sys}} = 10$ ($r_{\max} = 2$), which is the minimum value achievable for any feasible τ. The total number of cycles required is increased to $4N-1$ and hence, the total computation time of the algorithm is $(4N-1) \times 10$. For $N = 256$, this value equals 10230 time units.

Finally, using the branch-and-bound method, the optimal schedule is determined to be $\tau = (1\ 2)^T$. The RG's before and after retiming for this choice of τ are shown in Fig. 17(a) and Fig. 17(b), respectively. The cycle time t_{sys} is 12 ($r_{\max} = 2$) and the number of cycles required is $3N$. Hence, the overall computation time is $36N$ which equals 9216 time units, a more than 10% improvement in computation time compared to the other two schedules. □

$$\boldsymbol{yr}_{kq} = \boldsymbol{a}_{kq} - \boldsymbol{b}_{kq} \qquad \boldsymbol{yi}_{kq} = \boldsymbol{c}_{kq} + \boldsymbol{d}_{kq}$$

$$\boldsymbol{a}_{kq} = \boldsymbol{wr}_{kq} \times \boldsymbol{e}_{kq} \qquad \boldsymbol{b}_{kq} = \boldsymbol{wi}_{kq} \times \boldsymbol{f}_{kq}$$

$$\boldsymbol{c}_{kq} = \boldsymbol{wr}_{kq} \times \boldsymbol{f}_{kq} \qquad \boldsymbol{d}_{kq} = \boldsymbol{wi}_{kq} \times \boldsymbol{e}_{kq}$$

$$\boldsymbol{e}_{kq} = \boldsymbol{yr}_{k,q-1} + \boldsymbol{xr}_{kq} \qquad \boldsymbol{f}_{kq} = \boldsymbol{yi}_{k,q-1} + \boldsymbol{xi}_{kq}$$

$$\boldsymbol{xr}_{kq} = \begin{cases} \boldsymbol{xr}_{k-1,q} & \text{if } k \neq 1 \\ \text{Real}(x_{N-q}) & \text{otherwise} \end{cases} \qquad \boldsymbol{xi}_{kq} = \begin{cases} \boldsymbol{xi}_{k-1,q} & \text{if } k \neq 1 \\ \text{Im}(x_{N-q}) & \text{otherwise} \end{cases}$$

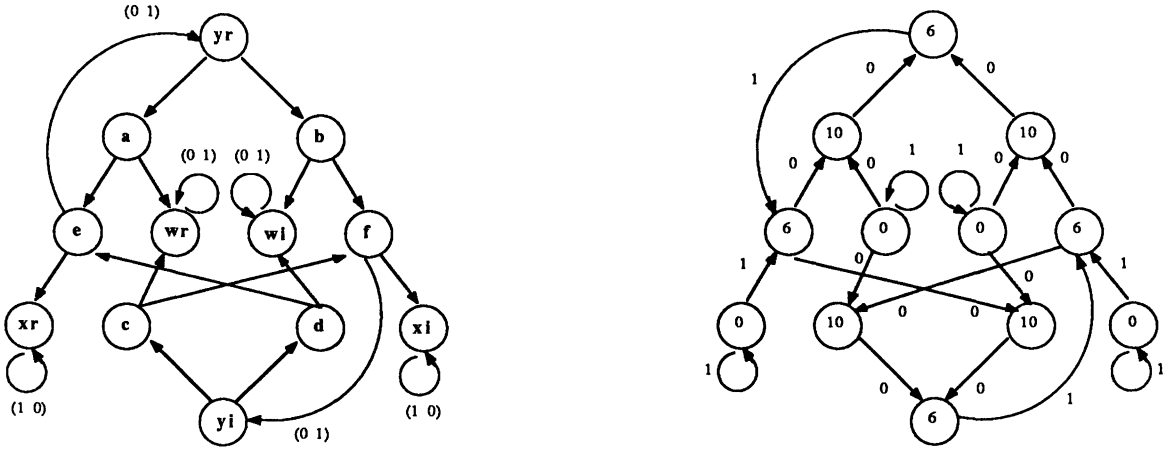
$$\boldsymbol{wr}_{kq} = \begin{cases} \boldsymbol{wr}_{k,q-1} & \text{if } q \neq 1 \\ \text{Real}(w_N^k) & \text{otherwise} \end{cases} \qquad \boldsymbol{wi}_{kq} = \begin{cases} \boldsymbol{wi}_{k,q-1} & \text{if } q \neq 1 \\ \text{Im}(w_N^k) & \text{otherwise} \end{cases}$$

(a) (b)

Fig. 13. (a) The RG of THCS derived with $\tau = (1\ 3)^T$ before and (b) after retiming. The cycle time is reduced from 11 to 5 time units.

(a) (b)

Fig. 14. (a) The RG of THCS derived with $\tau = (1\ 2)^T$ before and (b) after retiming. The cycle time is reduced from 11 to 6 time units.



(a) (b)

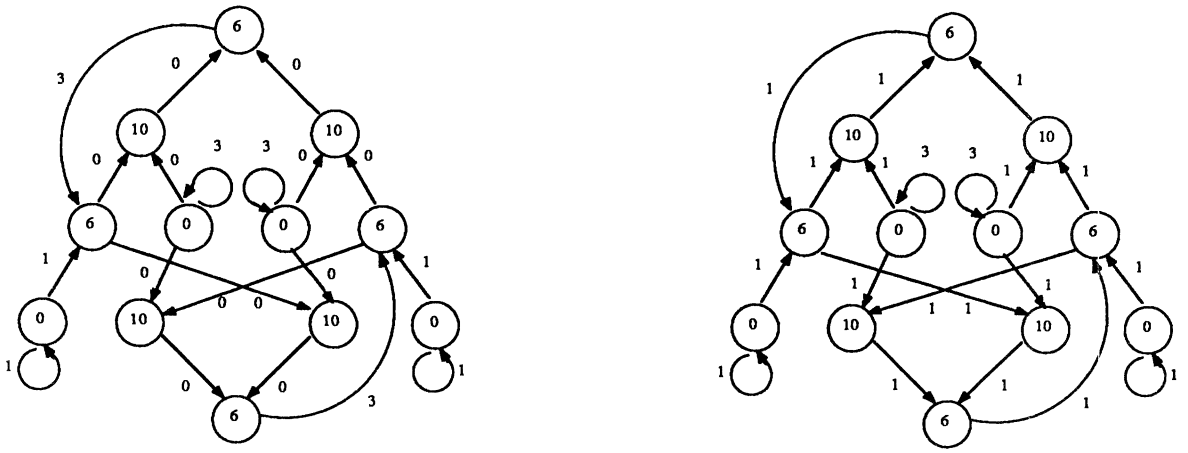
Fig. 15. (a) The DG of the DFT algorithm and (b) the corresponding RG derived with $\tau = (1\ 1)^T$. The cycle time is 22 time units.

Obviously, the choice of schedule which minimizes the overall computation time depends on the relative magnitudes of the computation delays of the functional units. For instance, if the computation delays in Example 3 are such that $T_\times > 4T_+$ then the n_{sys}-optimal schedule optimizes the overall computation time. Note also that the values of $r_{\max}$ in all the cases shown in the examples are small compared to the actual values of n_{sys}—an assumption made in formulating the discrete optimization problem in Section II-E.

V. Two-Level Pipeline Implementation

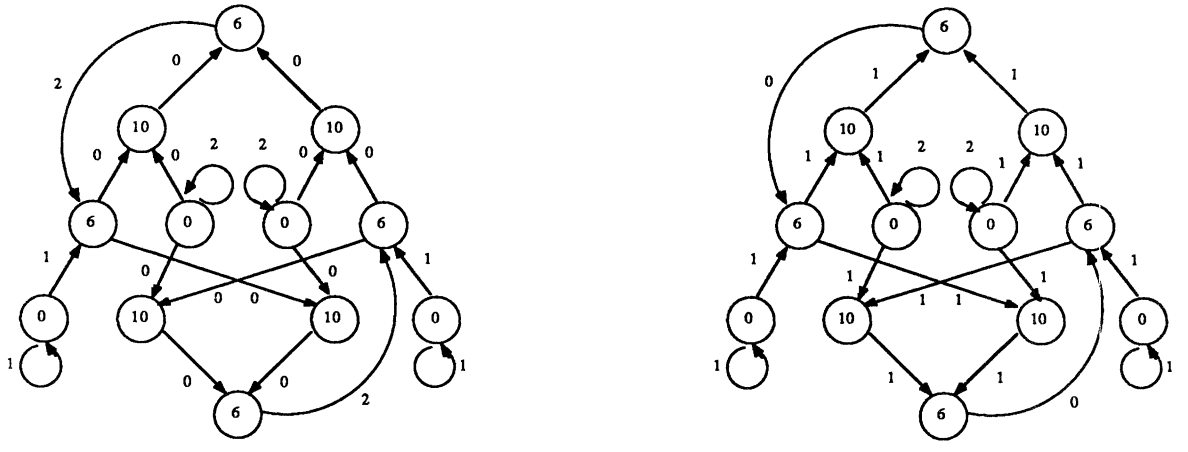
The throughput of a systolic array can be greatly enhanced by utilizing pipelined functional units in the PE's. The corresponding implementation is called a two-level pipeline [14] because pipelining is supported at both the inter-PE (array) and intra-PE (functional unit) levels. A well known example of such two-level pipelined systolic architecture is the CMU WARP computer [15] which contains a linear array of ten PE's where each PE has a pipelined floating point adder and a pipelined floating point multiplier.

In this section, the objective is to find the linear schedule which minimizes the value of $n_{\text{sys}} \times t_{\text{sys}}$ when all or some of the functional units on the PE's are pipelined. Two cases are considered. First, it is assumed that all the functional units



(a) (b)

Fig. 16. (a) The RG of DFT derived with $\tau = (1\ 3)^T$ before and (b) after retiming. The cycle time is reduced from 22 to 10 time units.



(a) (b)

Fig. 17. (a) The RG of DFT derived with $\tau = (1\ 2)^T$ before and (b) after retiming. The cycle time is reduced from 22 to 12 time units.

are pipelined (with at least two stages) and the propagation delay of each stage, the *stage-delay*, is the same, equal to t_{stage}, for all the functional units. This corresponds to the case where the PE's are built from off-the-shelf pipelined chip sets whose stage-delays have been carefully *equalized*. Second, the more general VLSI/WSI implementation of two-level pipelines is considered — the stage-delays of the functional units can differ. Both pipelined and combinational functional units are allowed in this case.

A. Case 1

Let $\sigma_{\boldsymbol{v}}$ be the number of pipeline stages in the functional unit that computes variable $\boldsymbol{v}$. This quantity is also commonly known as the *latency* of a pipelined functional unit. Suppose the computation of variable $\boldsymbol{u}$ depends on the value of variable $\boldsymbol{v}$, with dependence vector $\bar{\boldsymbol{d}}_{\boldsymbol{uv}}$. Since the functional units are pipelined, the computation of $\bar{\boldsymbol{v}}$ at index point $(\boldsymbol{Q} - \boldsymbol{d}_{\boldsymbol{uv}})$ will now take $\sigma_{\boldsymbol{v}}$ systolic cycles to complete. Thus, a feasible τ must be such that

$$\boldsymbol{Q}\tau \geq (\boldsymbol{Q} - \bar{\boldsymbol{d}}_{\boldsymbol{uv}})\tau + (\sigma_{\boldsymbol{v}} - 1) \text{ or } \bar{\boldsymbol{d}}_{\boldsymbol{uv}}\tau \geq \sigma_{\boldsymbol{v}} - 1 \qquad \forall \bar{\boldsymbol{d}}_{\boldsymbol{uv}}.$$

This is simply a generalization of condition (4) stated in Section II-C. Define σ_j as

$$\sigma_j = \sum_{\text{nodes } \boldsymbol{u} \text{ in } ec_j} (\sigma_{\boldsymbol{u}} - 1).$$

Then, the necessary and sufficient condition for τ to be a feasible linear schedule becomes [cf. (6)]

$$S_j\tau = \sigma_j + l_j \qquad \forall j$$

where l_j is a nonnegative integer. Note that since all the functional units are assumed to have at least two pipeline stages, computation rippling cannot occur (because there is at least one register within each of the functional units) and hence, $l_j = 0$ is acceptable.

With the assumption that the stage-delay is the same for all the functional units, the cycle time, t_{sys}, of a two-level pipeline

systolic implementation can only be either t_{stage} or $2t_{\text{stage}}$, where t_{stage} is the propagation delay of a stage of a pipelined functional unit. This is because if *all* the edges on the retimed RG are nonzero then the cycle time equals t_{stage}, otherwise, the output from one functional unit is used as the input of a second functional unit within the same systolic cycle and hence the cycle time must be $2t_{\text{stage}}$. The cycle time value cannot be larger because the internal registers within the pipelined functional units guarantee that there is no accumulation of stage-delay within the functional units.

To minimize the product of $n_{\text{sys}} \times t_{\text{sys}}$, only two alternatives must be considered. First, let $t_{\text{sys}} = t_{\text{stage}}$ ($l_j \geq \lambda_j$), the corresponding schedule τ that minimizes n_{sys} $(= 1 + Y)$ is found by solving the ILP:

$$\begin{cases} \text{minimize} & Y \\ \text{subject to} & Y - (\boldsymbol{V}_i - \boldsymbol{V}_j)\tau \geq 0, \qquad \forall \boldsymbol{V}_i, \boldsymbol{V}_j \in V(\Omega) \\ & S_j\tau \geq \sigma_j + \lambda_j, \qquad j = 1 \cdots m \\ & \tau \in \boldsymbol{Z}^n. \end{cases}$$

Then, let $t_{\text{sys}} = 2t_{\text{stage}}$ ($l_j = 0$), τ is obtained by solving

$$\begin{cases} \text{minimize} & Y \\ \text{subject to} & Y - (\boldsymbol{V}_i - \boldsymbol{V}_j)\tau \geq 0, \qquad \forall \boldsymbol{V}_i, \boldsymbol{V}_j \in V(\Omega) \\ & S_j\tau \geq \sigma_j, \qquad j = 1 \cdots m \\ & \tau \in \boldsymbol{Z}^n. \end{cases}$$

The τ which results in the smallest $n_{\text{sys}} \times t_{\text{sys}}$ value is the appropriate choice.

B. Case 2

The assumptions that all the functional units have the same stage-delay and that each functional unit has at least two pipeline stages can be relaxed. In that case, each node $\boldsymbol{u}$ in the RG can be decomposed into a *chain* of $\sigma_{\boldsymbol{u}}$ subnodes linked by edges of unit weight (see Fig. 18); each of the subnodes represents one stage of the pipeline. Since the registers within a pipelined functional unit are fixed, the weights on the edges linking consecutive subnodes are not to be altered by retiming. This is accomplished by imposing an additional constraint

$$r_{\boldsymbol{u}_i} = r_{\boldsymbol{u}_{i+1}}$$

when determining the optimal retiming for the RG [29], where $\boldsymbol{u}_i$ and $\boldsymbol{u}_{i+1}$ denote two consecutive pipeline stages of the functional unit which computes variable $\boldsymbol{u}$.

With such modification to the RG, the branch-and-bound method presented in the previous section can be applied to determine the time-optimal schedule. The search range of each l_j, $j = 1 \cdots m$, is

$$\text{MAX}(1, \sigma_j) \leq l_j \leq \text{MAX}(1, \sigma_j) + \lambda_j - 1$$

where λ_j is the total number of edges in elementary circuit ec_j *prior* to the decomposition of nodes into subnodes. Hence,

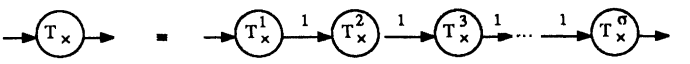


Fig. 18. A multiplier node with σ pipeline stages is decomposed into a chain of σ subnodes linked by directed edges of unit weight.

the introduction of pipelined functional units does not increase the size of the decision tree nor the complexity of the branch-and-bound method.

Example 4: Consider the matrix–matrix multiplication algorithm for computing the product of two square matrices, A and B, of order N. The algorithm can be expressed as a RIA shown at the bottom of this page, where a_{ij} and b_{ij} are the (i, j) elements of A and B, respectively. The range of computation is a cube of size N, with lower left-hand corner located at index point (1,1,1). The DG of the algorithm is shown in Fig. 19. Suppose the adder has three pipeline stages and the multiplier has six pipeline stages. Furthermore, assume that the stage delay, t_{stage}, is one time unit.

For $t_{\text{sys}} = t_{\text{stage}}$, the optimal τ is determined by solving the ILP:

$$\begin{cases} \text{minimize} & n_{\text{sys}} \\ \text{subject to} & (0\ 0\ -1)\tau \geq 3 \\ & (0\ 1\ 0)\tau \geq 1 \\ & (1\ 0\ 0)\tau \geq 1 \\ & \tau \in \boldsymbol{Z}^n \end{cases}$$

which yields the solution $\tau = (1\ 1\ -3)^T$. The reindexing transformation is selected such that

$$\begin{cases} [(0\ 0\ 0) + \delta_{\boldsymbol{p}} - \delta_{\boldsymbol{c}}]\tau \geq 6 \\ [(0\ 1\ 0) + \delta_{\boldsymbol{a}} - \delta_{\boldsymbol{p}}]\tau \geq 1 \\ [(0\ 0\ -1) + \delta_{\boldsymbol{b}} - \delta_{\boldsymbol{p}}]\tau \geq 1. \end{cases}$$

The first inequality ensures proper synchronization of data $\boldsymbol{p}$, which is computed by a multiplier of six pipeline stages, and $\boldsymbol{c}$. A feasible reindexing transformation is $\delta_{\boldsymbol{c}} = (0\ 0\ 0)$, $\delta_{\boldsymbol{p}} = \delta_{\boldsymbol{a}} = \delta_{\boldsymbol{b}} = (1\ 2\ -1)$. This is equivalent to a retiming of the RG with

$$r_{\boldsymbol{c}_1} = r_{\boldsymbol{c}_2} = r_{\boldsymbol{c}_3} = 0$$
$$r_{\boldsymbol{p}_1} = r_{\boldsymbol{p}_2} = \cdots = r_{\boldsymbol{p}_6} = 1$$

and therefore,

$$\delta_{\boldsymbol{c}}\tau = \sum_{i=1}^{3} r_{\boldsymbol{c}_i} = r_{\boldsymbol{c}} = 0$$
$$\delta_{\boldsymbol{p}}\tau = \sum_{i=1}^{6} r_{\boldsymbol{p}_i} = r_{\boldsymbol{p}} = 6$$
$$\delta_{\boldsymbol{a}}\tau = r_{\boldsymbol{a}} = 6$$
$$\delta_{\boldsymbol{b}}\tau = r_{\boldsymbol{b}} = 6.$$

$$\boldsymbol{c}_{ijk} = \boldsymbol{c}_{i,j,k+1} + \boldsymbol{p}_{ijk} \qquad \boldsymbol{p}_{ijk} = \boldsymbol{a}_{i,j-1,k} \times \boldsymbol{b}_{i-1,j,k}$$

$$\boldsymbol{a}_{ijk} = \begin{cases} \boldsymbol{a}_{i,j-1,k} & \text{if } j-1 > 0 \\ a_{ik} & \text{otherwise} \end{cases} \qquad \boldsymbol{b}_{ijk} = \begin{cases} \boldsymbol{b}_{i-1,j,k} & \text{if } i-1 > 0 \\ b_{kj} & \text{otherwise} \end{cases}$$

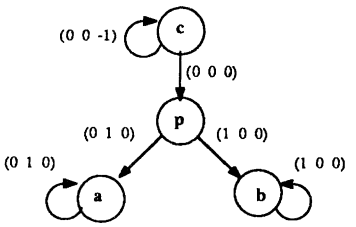


Fig. 19. The DG of the matrix-matrix multiplication algorithm.

The total number of cycles required equals $5N + 2$ and hence the total time needed is $5N + 2$ time units.

Similarly, for $t_{\text{sys}} = 2t_{\text{stage}}$, the optimal τ is determined from the ILP:

$$\begin{cases} \text{minimize} & n_{\text{sys}} \\ \text{subject to} & (0\ 0\ -1)\tau \geq 2 \\ & (0\ 1\ 0)\tau \geq 1 \\ & (1\ 0\ 0)\tau \geq 1 \\ & \tau \in \boldsymbol{Z}^n \end{cases}$$

which yields the solution $\tau = (1\ 1 - 2)^T$. A feasible reindexing transformation is $\delta_{\boldsymbol{c}} = (0\ 0\ 0)$, $\delta_{\boldsymbol{p}} = \delta_{\boldsymbol{a}} = \delta_{\boldsymbol{b}} = (1\ 1 - 2)$. The total number of cycles required equals $4N + 3$ and hence the total time needed is $2 \times (4N + 3) = 8N + 6$ time units.

Thus, for the given pipelined functional units, the first schedule minimizes the overall computation time of the matrix–matrix multiplication algorithm. □

VI. Performance Evaluation

The effectiveness of the branch-and-bound method can be assessed by the percentage or the number of nodes on the decision tree it traverses. As mentioned in Section IV, at a nonleaf node of the decision tree the bounding function can be evaluated in polynomial time. At a leaf node, however, one needs to solve an ILP (the *ncycle* function) which is computationally expensive. Hence, an alternative measure of performance for the branch-and-bound method is the number of leaf nodes that survive the pruning.

The branch-and-bound method has been implemented in the programming language C on a SUN/3 workstation under the BSD 4.3 Unix operating system. Shown in Table I are performance measures obtained when applying the branch-and-bound method to three test cases: the Toeplitz Hyperbolic Cholesky Solver (THCS), the Discrete Fourier Transform algorithm (DFT), and a version of the dynamic programming algorithm (DP). Only the characteristics of the algorithms which are relevant to the discussion are provided. In the table, the number of elementary circuits quoted does not include the self-loops on the DG, and the CPU time quoted does not include the time for solving the integer linear programs at the leaf nodes.

As can be observed, the branch-and-bound scheme is extremely efficient in pruning the decision trees. For decision trees of small size (e.g., DFT), however, the percentage of nodes traversed may be large although the *actual* number of nodes traversed will remain small. Thus, the method is efficient enough for use in an interactive environment to assist the design of time critical VLSI/WSI systems (e.g., with standard cells) for applications such as real-time signal/image processing.

VII. Concluding Remarks

In this paper, the derivation of time-optimal linear schedules for algorithms that can be expressed in the RIA model is formulated as a discrete optimization problem. It is shown that the solution space of the problem is bounded and an efficient branch-and-bound method is introduced for determining the optimal linear schedule. The optimality criterion is based on two assumptions. First, the number of functional units on the PE's is assumed unconstrained. Second, uniformity and regularity in control and clocking is considered to be a key factor for design cost reduction, thus ensuring the cost effectiveness of VLSI systolic arrays.

While the first assumption is valid for VLSI/WSI systolic arrays, it does not necessarily apply to general purpose, fixed architecture systolic machines. Moreover, for some RIA's, the number of functional units required to support the time optimal schedule may be too large to be practical, even for VLSI implementations. It is therefore of practical value to consider an extension of the problem in which the number of functional units on the PE's is fixed. However, a major complication in this case is that the optimal retiming for a RG can no longer be determined in polynomial time. In fact, to find the cycle time of a given RG for a chosen legal retiming under hardware constraint is equivalent to solving a multiprocessor scheduling problem—remove all the nonzero edges from the retimed RG and treat the resultant graph as a task precedence graph where the computation delay associated to a node is interpreted as the task processing time—which is known to be NP-hard. One possibility is to rely on heuristics for the multiprocessor scheduling problem [11] to determine a close-to-optimal cycle time of a given (legally) retimed RG. But even so, to find the close-to-optimal retiming for the RG may still be intractable because the number of legal retimings that needs to be considered is exponential.

The second assumption is upheld at more than one place in the course of development of the discrete optimization problem. First, the full set of dependence vectors in the RIA is instantiated at all the index points to make the dependence structure shift-invariant (i.e., ignoring the conditionals in the RIA). Second, for a constant global clock period, the cycle time of the systolic array is taken to be the maximum duration of a systolic cycle. Obviously, such an approach may degrade the time performance of the implementation. As an example, many (systolic) algorithms possess a small set of *expensive* computations which are distributed evenly among all the other computations—such as the division operations in the LU decomposition algorithm and in Example 1 and 2, and the square-root operation in the hyperbolic Cholesky factorization algorithm [1]. Since the maximum duration of a systolic cycle is taken as the global clock period of the systolic array, the time performance is severely degraded by this small set of *expensive* operations.

TABLE I
SOME PERFORMANCE RESULTS OF THE BRANCH-AND-BOUND METHOD; m IS THE TOTAL NUMBER OF ELEMENTARY CIRCUITS (EXCLUDING THE SELF-LOOPS) IN THE DG AND λ_i IS THE NUMBER OF EDGES IN ec_i.

Algorithm	Dependence Graph (DG)			Decision Tree			Run-time
	No. of vertices	m	$\{\lambda_i\}$	Tree size	No. of nodes traversed	No. of surviving leaves	CPU seconds
THCS	6	5	4,6,6,4,4	3052	84 (2.75%)	3	14.7
DP	9	4	4,3,4,3	208	17 (8.17%)	1	7.4
DFT	12	3	6,3,3	78	21 (26.92%)	2	1.9

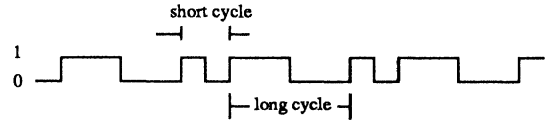


Fig. 20. A global clock with alternatively long and short clock period.

Now, consider the possibility of using a global clock whose period can vary on alternate cycles (for regularity) to drive the target systolic array. Then, by choosing the schedule carefully, it may be possible to arrange for the execution of the expensive computations at regular clock cycle intervals to match the pattern of the global clock. Hence, the expensive computations are only computed during the "long" clock cycles and the other computations are executed in the "short" cycles at a much higher rate. For an illustration, consider the forward-substitution algorithm in Example 1 and suppose the computation delays of a divider, a multiplier, and an adder/subtractor are, respectively, 6, 2, and 1 time units. With the schedule $\tau = (1\ 1)^T$, the division operations (which take 6 time units to complete) occur only at cycle $2i$, $i = 1 \cdots N$, i.e., on the even cycles (see Fig. 6). During the odd cycles, a multiply-and-add operation is needed, that only costs 3 time units. Using a global clock with clock period which varies between long and short cycle (Fig. 20), the overall computation time of the algorithm on the array is

$$N \times 6 + (N-1) \times 3 \approx 9N \text{ time units.}$$

This time performance surpasses that of any other systolic array implementation which relies on a global clock of fixed clock period by at least 25%, since the minimum value of n_{sys} for any valid linear schedule is $2N-1$ and the minimum t_{sys} is 6 time units. Thus, a slight relaxation of the regularity and uniformity requirement could pay off handsomely.

The applicability of such a clocking scheme depends on whether a valid schedule could be found so that the expensive computations occur at a regular interval. Suppose $\{\boldsymbol{R}\}$ is the subset of index points of Ω at which these expensive computations occur. The additional constraint on τ is that there exist integer constants t_0 and k, $k > 1$, such that

$$\boldsymbol{R}\tau = \alpha k + t_0, \qquad \forall \boldsymbol{R}$$

where α is an integer, whose value depends on $\boldsymbol{R}$.

Pushing to the extreme, instead of relying on global synchronization, a self-timed variant of the systolic array may be of interest. This is the so-called *wavefront* processor array proposed by S. Y. Kung [16]. However, the hardware and control overhead necessary for supervising the asynchronous communication may outweigh the gain in time performance.

REFERENCES

[1] H. M. Ahmed, J.-M. Delosme, and M. Morf, "Highly concurrent computing structures for matrix arithmetic and signal processing," *IEEE Comput. Mag.*, vol. 15, pp. 65–81, Jan. 1982.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Design and Analysis of Computer Algorithms*. Reading, MA: Addison-Wesley, 1974.

[3] P. R. Cappello and K. Steiglitz, "Unifying VLSI array designs with geometric transformations," in *Proc. Int. Conf. Parallel Processing*, 1983, pp. 448–457.

[4] K.-C. Chen and S. Muroga, "Input assignment algorithm for decoded-PLA's with multi-input decoders," in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 1988, pp. 474–477.

[5] J.-M. Delosme and I. C. F. Ipsen, "An illustration of a methodology for the construction of efficient systolic architectures in VLSI," in *Proc. 2nd Int. Symp VLSI Technology, Syst., and Appl.*, May 1985, pp. 268–273.

[6] ——, "Systolic array synthesis: Computability and time cones," in *Parallel Algorithms & Architectures*, M. Cosnard *et al.*, Eds. New York: Elsevier Science, 1986, pp. 295–312.

[7] ——, "Parallel computation of algorithms with uniform dependences," in *Proc. 4th SIAM Conf. Parallel Processing for Scientif. Comput.*, Dec. 1989, pp. 319–325.

[8] A. Fettweis, "Digital circuits and systems," *IEEE Trans. Circuits Syst.*, vol. CAS-31, no. 1, pp. 31–48, 1984.

[9] J. A. B. Fortes, "Algorithm transformations for parallel processing and VLSI architecture design," Ph.D. dissertation, Univ. of Southern California, Los Angeles, Dec. 1983.

[10] R. M. Karp, R. E. Miller, and S. Winograd, "The organization of computations for uniform recurrence equations," *J. ACM*, vol. 14, pp. 563–590, 1967.

[11] H. Kasahara and S. Narita, "Practical multiprocessor scheduling algorithms for efficient parallel processing," *IEEE Trans. Comput.*, vol. C-33, no. 11, pp. 1023–1029, 1984.

[12] H. T. Kung and C. E. Leiserson, "Systolic arrays for VLSI," *Sparse Matrix Proceedings*, SIAM, pp. 245–282, 1978.

[13] H. T. Kung, "The structure of parallel algorithms," in *Advances in Computers, Vol. 19*. New York: Academic, 1980.

[14] H. T. Kung and M. S. Lam, "Wafer-scale integration and two-level pipelined implementations of systolic arrays," *J. Parallel Distributed Comput.*, vol. 1, pp. 32–63, 1984.

[15] H. T. Kung, "Systolic algorithms for the CMU warp processor," in *Systolic Signal Processing*, E. Swartzlander, Ed. New York: Marcel Dekker, 1987, pp. 73–96.

[16] S. Y. Kung, "On supercomputing with systolic/wavefront array processors," *Proc. IEEE*, vol. 72, pp. 867–884, July 1984.

[17] E. L. Lawler, *Combinatorial Optimization: Networks and Matroids*. New York: Holt, Rinehart and Winston, 1976.

[18] C. E. Leiserson, F. M. Rose, and J. B. Saxe, "Optimizing synchronous circuitry by retiming," in *Third Caltech Conference on VLSI*, R. Bryant, Ed. Rockville, MD: Computer Science Press, 1983, pp. 87–116.

[19] G.-J. Li and B. W. Wah, "The design of optimal systolic arrays," *IEEE Trans. Comput.*, vol. C-34, pp. 66–77, 1985.

[20] J. Li, M. Chen, and M. Young, "Design of systolic algorithms for large scale multiprocessors," Res. Rep. YALEU/DCS/RR-513, Dep. Comput. Sci., Yale Univ., Oct. 1988.

[21] B. Lisper, "Time-optimal synthesis of systolic arrays with pipelined cells," Res. Rep. YALEU/DCS/RR-560, Dep. Comput. Sci., Yale Univ., Sept. 1987.

[22] D. I. Moldovan, "ADVIS: A software package for the design of systolic arrays," in *Proc. Int. Conf. Comput. Design*, 1984, pp. 158–164.
[23] G. L. Nemhauser and L. A. Wolsey, *Integer and Combinatorial Optimization*. New York: Wiley, 1988.
[24] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization Algorithms and Complexity*. Englewood Cliffs, NJ: Prentice-Hall, 1982.
[25] P. Quinton, "Automatic synthesis of systolic array from uniform recurrent equations," in *Proc. 11th Int. Symp. Comput. Architecture*, June 1984, pp. 208–214.
[26] S. K. Rao, "Regular iterative algorithms and their implementations on processor arrays," Ph.D. dissertation, Stanford Univ., Oct. 1985.
[27] S. K. Rao and T. Kailath, "Architecture design for regular iterative algorithms," in *Systolic Signal Processing Systems*.E. E. Swartzlander, Ed. New York: Marcel Dekker, 1987, pp. 209–298.
[28] V. van Dongen, "PRESAGE, A tool for the design of low-cost systolic circuits," in *Proc. Int. Symp. Circuits and Syst.*, June 1988, pp. 2765–2768.
[29] Y. Wong, "Algorithms for systolic array synthesis," Ph.D. dissertation, Dep. Comput. Sci., Yale Univ., Dec. 1989.

MSSM—A Design Aid for Multi-Stage Systolic Mapping

YIN-TSUNG HWANG AND YU HEN HU

Department of Electrical and Computer Engineering, University of Wisconsin, Madison, 1415 Johnson Drive, Madison, WI 53706

Received January 20, 1991; Revised November 5, 1991.

Abstract. A multi-stage algorithm is a computing algorithm consisting of a sequence of nested loop constructs to be executed sequentially. In this paper, a systematic approach to address the multi-stage systolic mapping problem is proposed. To reduce the inter-stage data communication overhead, we argue that the adjacent stages should have matched I/O interface. For this, the conditions of I/O matching between two stage's mappings are established. A systematic method to derive the I/O matched mapping is also presented. To improve the performance degradation due to the *initiation* and *conclusion* phases of computation in systolic array, a technique called *chaining* which tries to overlap part of the computations in successive stages and thus effectively reduces the computation latency is employed. With these results, the multi-stage mapping problem is formulated as an optimization problem and a heuristic search based multi-stage systolic mapping (MSSM) tool is developed. Several design examples are presented to illustrate the potential use of MSSM.

1. Introduction

Many research works focused on the mapping of digital signal processing (DSP) computing algorithms onto VLSI systolic architectures. Kuhn [1] and Moldovan [2]–[4] proposed methods to map a recursive algorithm with n-level nested loops to a $(n - t)$-dimensional processor arrays. Methods for $t = 1$ have also been proposed by Miranker and Winkler [5], and Quinton [6], [7]. Li and Wah [8] derived the optimal mapping based on vector constraint equations. Chen [9] used a formal language to describe the recursive algorithms which enables the proof of the correctness of the systolic mapping. S.Y. Kung [10] proposed a dependence graph (DG) based approach and a cut set retiming procedure which can map an n-dimensional DG into a $(n - 1)$-dimensional processor array. H.T. Kung [11] derived a canonical matrix representation first for the algorithm and then applied algebraic transformations to determine the delay distribution and the I/O periods. Capello [12] used the concept of geometrical transformation and transformed the n-tuple indices into $(n - 1)$-tuple processor indices and 1-tuple time index. Lee [13] developed a method of mapping algorithms consisting of an n-level nested for-loop onto a linear array. Lam and Mostow [14] adopted the transformational paradigm which consists of a *bag of tricks* algorithm transformations. The design process is proceeded in a bottom up manner and contains the hardware allocation, schedule computation and optimization phases. A comprehensive survey of these early works can be found in [15].

Most of these existing methods assume the DSP algorithm is a Regular Iterative Algorithm (RIA) [16]. An RIA is a nested do loop consisting of a set of uniform recurrence equations. Many existing DSP algorithms can be formulated as an RIA. However, there are also many sophisticated DSP computing algorithms such as the Kalman filter, which cannot be described by a single nested loop. Instead, many of them consist of a sequence of nested loops to be executed sequentially. In this paper, we call such an algorithm a *multi-stage algorithm*. The number of the stages is the number of nested loops. Since successive nested loops are to be executed sequentially, it is more economical to map these nested loops onto a single processor array. If each individual nested loop is itself an RIA, existing mapping methods can readily be applied to map them to the same processor array. In this case, the design issue is to better interface the computations of successive stages. Inadequate interface may cause serious performance degradation and offset the potential parallelism. In the past, the multi-stage mapping was usually achieved manually by knowledgeable designers. Despite the early work by JáJá [17], no systematic mapping procedure has been reported. Nor are any computer assisted design tools available.

In this paper, we present a systematic approach to address the multi-stage mapping problem. To reduce

Reprinted with permission from *Journal of VLSI Signal Processing*, Y. T. Hwang and Y. H. Hu, "MSSM - A Design Aid for Multi-stage Systolic Mapping," Vol. 4, No. 2/3, pp. 125-146, May 1992.

data communication overhead, we argue that the adjacent stages should have matched I/O interface. For this, we establish conditions for matching I/O operations between successive stages and propose a method to derive the I/O matched mapping. Due to the nature of wavefront computation, not all the processors in the systolic array are activated during the *initiation* and *conclusion* phases of computation. We thus employ a technique called *chaining* to overlap part of the computations in successive stages and reduce the performance degradation. With these results, we then develop a multi-stage systolic mapping (MSSM) algorithm to search for the best multi-stage mapping result. We develop the multi-stage systolic mapping procedure into a software package, MSSM. Several design examples using MSSM have shown very encouraging results.

The remainder of this paper will be arranged as follows: In Section 2, the multi-stage mapping problem will be formulated and basic terminologies will be defined. Two interface design issues, I/O matching and computation overlapping, will be discussed in Sections 3 and 4, respectively. In Section 5, a *parameterized dependence graph* is briefly introduced as a unified representation of the input algorithm for the multi-stage mapping problem. A best-first search method for solving the multi-stage mapping problems will also be described. Several design examples compiled by our multi-stage systolic mapping tool (MSSM) will be illustrated in Section 6.

2. The Multi-stage Systolic Mapping Problem

2.1. Basic Terminologies and Notations

In the conventional systolic mapping problem, the array design can be derived systematically by applying a standard space-time transformation method to the computing algorithm. The algorithm is usually expressed as a nested loop containing a system of uniform recurrence equations. It can be described by an index space of the nested loop and a set of dependence vectors associated with the recurrence equations. Graphically, it is equivalent to the data dependence graph (DG) proposed in [18], where each node incorporates one loop iteration of the computation in the recurrence equation and an arc connecting two nodes corresponds to a dependence vector. Each node can be uniquely addressed by its coordinate (an n-dimensional vector with integral elements) in the graph. The space-time transformation consists of two parts: (1) scheduling function and (2) processor allocation function. Scheduling function is an n-dimensional integral vector (called scheduling vector $\underline{s}$) which assigns the scheduled time for each node to be executed. Processor allocation function (or processor basis [18]) P is an $(n-1) \times n$ integral matrix of full row rank, that relates each node in the DG to an $(n-1)$-dimensional vector representing the coordinate of the processor in the processor domain defined by P. The processor allocation function can be characterized by an n-dimensional co-prime projection vector $\underline{\lambda}$ which spans the one-dimensional null space of P. Any two processor allocation functions corresponding to the same projection vector are considered as *equivalent*. The readers are referred to [3] and [18] for more details.

In this paper, we adopt the following notation conventions (superscript t denotes vector transpose):

1. $\underline{s}$: a column vector
2. $\underline{h}^t$: a row vector
3. A: a matrix
4. $n(\underline{j})$: a DG node with coordinate vector $\underline{j}$
5. $x[\underline{i}]$: a variable with l-tuple array index $\underline{i}$
6. $\underline{a}$ (or $\underline{a}^t$) $\in A$: $\underline{a}$ (or $\underline{a}^t$) is a column (row) of matrix A
7. $\underline{e}_i = [0 \ \ldots \ 0 \ 1 \ 0 \ \ldots \ 0]^t$, where "1" is the i-th entry.

We also adopt the following conventions in our discussion: (1) Without loss of generality, we will use stage 1 and stage 2 to represent two adjacent stages. (2) To avoid confusion, x and x' are used to denote the same variable appeared in two adjacent stages. (3) The processor array obtained from each stage's mapping will be called a *logical processor array* to differentiate it from the *physical processor array* (target machine) where the logical array processors will be realized. (4) To distinguish the index vectors in different domains, we use the following terms:

1. *DG index* ($\underline{j}$): the coordinate vector of a node in the DG domain
2. *array index* ($\underline{i}$): the index of an array variable,
3. *processor index* ($\underline{r}$): the index for a processor in processor array.

The DG index $\underline{j}$ thus represents a node $n(\underline{j})$ in DG, the array index $\underline{i}$ represents an element $x[\underline{i}]$ of array variable x and the processor index $\underline{r}$ represents a processor $p(\underline{r})$ in the systolic array.

2.2. Problem Statement

We define *multi-stage algorithm* as a numerical computing algorithm consisting of a sequence of nested do

loop constructs. (For convenience, we shall use *stage* and *nested loop* interchangeably in the rest of this paper.) Each of these nested loops is an RIA, and therefore can be unfolded and described by a shift invariant DG. In many cases, the sequence of these nested loops needs to be executed sequentially: the data generated in the current nested loop will be the input to the following loop. It is therefore more efficient, with respect to hardware utilization, to map these loops onto a single processor array. This, however, does not preclude the possible overlapping of the computations between successive stages as long as the dependence constraint is satisfied. The problem now is to handle the interface problem between successive stage's mappings carefully such that the overheads can be reduced.

Several assumptions about MSSM will be made here:

(1) We assume that the DG of each stage has the same dimension.
(2) We assume only a single systolic processor array is available. The size and the shape of the array are to be determined.
(3) We assume the size of the DG can be fitted into the processor array without partitioning.

Below is an example of a three-stage algorithm which consists of two consecutive matrix multiplications, followed by an LU decomposition:

Example 1. A three-stage computing algorithm.

$$A \cdot B = C \ \ldots\ldots\ldots \text{ first stage}$$

$$C \cdot D = E \ \ldots\ldots\ldots \text{ second stage}$$

$$E = L \cdot U \ \ldots\ldots\ldots \text{ third stage}$$

A variable is an *I/O variable* if it is computed in one stage and then used in the successive stage. In this example, matrices C and E are I/O (array) variables and will require inter-stage data communication between adjacent stage's computations. When I/O variables are passed from one stage to the next, interface overheads may arise from two causes: (1) The mismatch of the data flow sequence or the I/O locations of adjacent stages. The result is that extra time or storage buffers will be needed to redirect the data. (2) The lack of computation overlap between adjacent stages due to inadequate computing schedules. This leads to longer computation time and low processor utilization. These overheads may severely deteriorate the overall performance despite the use of systolic pipelined computation in individual stage. This can be illustrated with a two-stage consecutive matrix vector multiplication example:

$$\underline{c}_{n\times 1} = A_{n\times n} \cdot \underline{b}_{n\times 1}$$

$$\underline{y}_{n\times 1} = D_{n\times n} \cdot \underline{c}_{n\times 1}$$

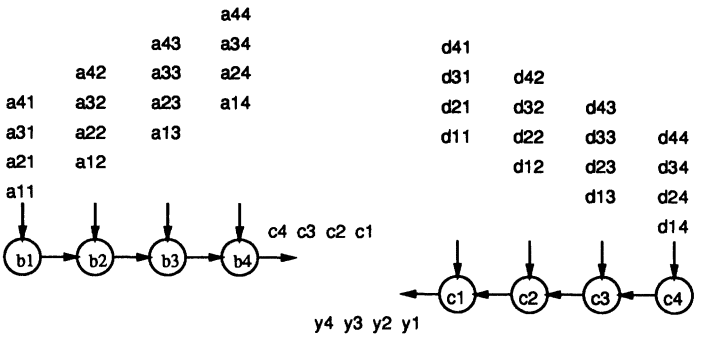
Two possible multi-stage mapping designs (1) and (2) are depicted in figure 1 and figure 2 respectively. In design (1), the systolic mappings for each individual stage are optimal. However, the final design after combining two stages is very poor. In particular, an interstage storage buffer is needed to transform the data flow pattern from the first stage to the second one. There is no computation overlap between these stages, either. In design (2), the I/O interface between successive stages are perfectly matched. No interstage storage buffer is used, and there is a 50% computation overlap between the two stages. The total computation time is also 25% less than that of design (1). Clearly, the second design is better.

In summary,

> the goal of the multi-stage mapping problem is to efficiently map a multi-stage computing algorithm onto a fix-size, mesh-connected systolic processor array so as to minimize the interface overhead (additional delay and storage space) between successive stages.

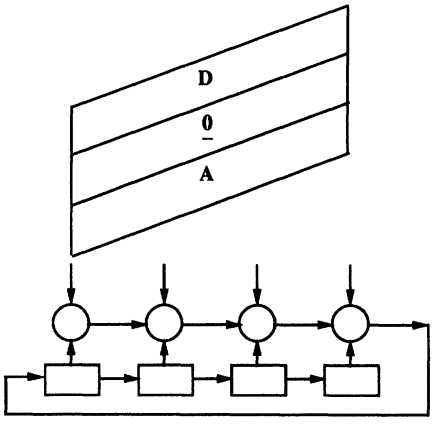
3. The I/O Matching between Successive Stages

The goal of the I/O matching is to best match the I/O locations and the data flow between adjacent stages so that the overhead due to data flow mismatch is minimized. In this first phase design of a multi-stage systolic mapping tool, several assumptions about I/O matching are made to simplify the problem. (1) We assume that the processor allocation function for each stage is properly selected such that all the communication links derived after mapping are permissible and hence will not affect the conditions of I/O matching. The matching issues thus focus on the data location and data flow only but not on the communication links. (2) We assume that only the final value, not the intermediate result, of the I/O variable x is interesting. The I/O activities will therefore occur only at the boundary of the DG where either the initial value of the I/O variable x is imported, or the final value of x is generated. (3) The recursion of each variable is no more than one dimension. Although these assumptions seem to be restrictive, many multi-stage DSP algorithms do satisfy all of them.



(a) The mapping for the first stage

(b) The mapping for the second stage



(c) The I/O mismatched array design for two stages

Fig. 1. An I/O mismatched lousy design.

3.1. Preliminary Discussion

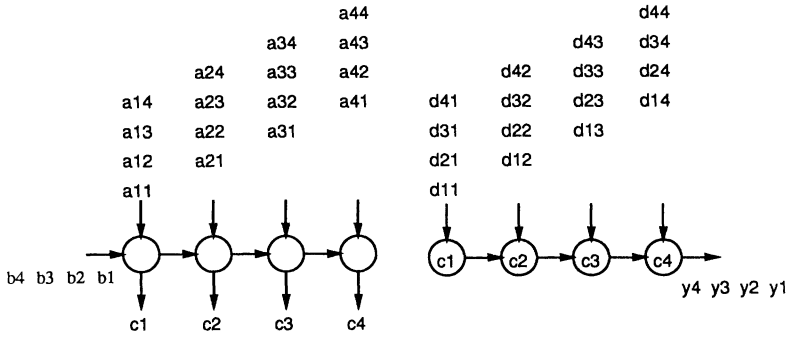
Consider the following pseudo RIA expressed in the form of a nested do loop:

$$\begin{aligned}&\text{DO } j_1 = \ldots\\ &\quad \ldots\\ &\quad\text{DO } j_n = \ldots\\ &\qquad x[g_1(\underline{j}), g_2(\underline{j}), \ldots, g_l(\underline{j})] = \ldots\end{aligned}$$

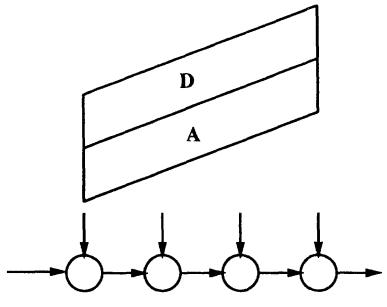
where the vector $\underline{j} = [j_1 \ldots j_n]^t$ denotes an n-tuple loop index and $\underline{g}(\underline{j}) = [g_1(\underline{j}) \ldots g_l(\underline{j})]^t = F_{l\times n} \cdot \underline{j} + \underline{b}$ is an *array index function* mapping the *loop index* $\underline{j}$ into an l-dimensional *array index* $\underline{i} = \underline{g}(\underline{j})$. The range of loop index $\underline{j}$ is called the *index space* of the algorithm. Each integral point $\underline{j}$ within the index space then corresponds to one loop iteration. If matrix F in array index function $\underline{g}(\cdot)$ has a null space vector $\underline{\rho}$ such that $F \cdot \underline{\rho} = \underline{0}$, then we have

$$\underline{g}(\underline{j} + \underline{\rho}) = F \cdot (\underline{j} + \underline{\rho}) + \underline{b} = F \cdot \underline{j} + \underline{b} = \underline{g}(\underline{j}).$$

This means the same array element $x[\underline{i}]$ with $\underline{i} = \underline{g}(\underline{j})$ will be accessed at loop iteration $\underline{j}$ and all iterations $\underline{j} + c \cdot \underline{\rho}$ for any integer c such that $\underline{j} + c \cdot \underline{\rho}$ stays within the index space. Since we assume the recursion is at most one-dimensional, the null space dimension of the array index function $\underline{g}$ is at most one, too. The dimension l of array index $\underline{i}$ is thus at least $n - 1$. Since each loop iteration (a point in the index space) corresponds to a node in DG, variable $x[\underline{i}]$ is propagated from one DG node to another along a straight line defined by the null space vector of array index function. Among those nodes accessing the same array element of $x[\underline{i}]$, two extremal nodes at the boundary of DG will keep the initial and final values of $x[\underline{i}]$, i.e., the value upon entering the DG and the value upon leaving the DG. The DG index of these two nodes will be denoted by $\underline{f}_i(x[\underline{i}])$ and $\underline{f}_f(x[\underline{i}])$ respectively. For I/O matching, the node $n(\underline{f}_f(x[\underline{i}]))$ producing the final value of $x[\underline{i}]$ in the first stage's DG should be mapped onto the processor where the node $n(\underline{f}_i(x'[\underline{i}]))$ receiving the



(a) The mapping for the first stage (b) The mapping for the second stage



(c) The I/O matched array design for two stages

Fig. 2. An I/O matched good design.

initial value of $x'[\underline{i}]$ in the following stage. This is not a trivial task because the coordinates in two adjacent stages may not be the same. The following definitions are given to facilitate the discussion of I/O matching.

DEFINITION 1. *Dependence graph.* The dependence graph can be characterized by a two-tuple [*IS*, *D*]. *IS*, the index space of the DG, is a convex polyhedron $\{\underline{j} \mid \Pi \cdot \underline{j} \leq \underline{b}\}$ for some integral matrix Π and integral vector $\underline{b}$. The set of all DG nodes $\{n(\underline{j})\}$ is then equal to the set of all integral points in *IS*. *D* is the dependence matrix with each column vector $\underline{d}_i \in D$, called dependence vector, describing the read/write dependence between two nodes.

DEFINITION 2. *Boundary plane.* Let *G* be a dependence graph with $IS = \{\underline{j} \mid \Pi \cdot \underline{j} \leq \underline{b}\}$. The boundary plane of *G* can be described by a three-tuple [Γ, $\underline{\pi}_i$, Ω]. Γ is the index space of the boundary plane (a subspace of *IS*) described by $\{\underline{j} \mid \underline{\pi}_i^t \cdot \underline{j} = b_i$ and $\underline{\pi}_k^t \cdot \underline{j} \leq b_k$ $\forall k \neq i\}$, where $\pi_i^t \in \Pi$ is called the normal vector of Γ and points toward the exterior of *IS*. Ω is the lattice matrix of the boundary plane with all its column vectors as the basis of hyperplane $\underline{\pi}_i^t \cdot \underline{j} = 0$.

Note that Γ is also a convex polyhedron with dimension one less than that of *IS*. In multi-stage algorithm, each stage's DG can be partitioned into three regions. An *input region* is where the DG nodes taking the input values of I/O variable from the previous stage. An *output region* is where the DG nodes sending the final values of I/O variable to the following stage. An *internal region* contains all the DG nodes not in either input or output regions and does not interface with adjacent stages. Both input and output regions contain the nodes at boundary planes only.

DEFINITION 3. *Index mapping vector/matrix.* Let $x[\underline{i}]$ be an I/O variable. The *k*-th input (or output) index mapping vector of *x* is defined as $\underline{u}_k = \underline{f}_i(x[\underline{i}_1]) - \underline{f}_i(x[\underline{i}_2])$, (or $\underline{u}_k = \underline{f}_f(x[\underline{i}_1]) - \underline{f}_f(x[\underline{i}_2])$) where $\underline{i}_1 - \underline{i}_2 = \underline{e}_k$. The matrix $U = [\underline{u}_1\ \underline{u}_2\ \ldots\ \underline{u}_{n-1}]$ will be called an *index mapping matrix.*

Since the DG of an RIA is shift invariant, $\underline{u}_k$ is a constant vector as long as $\underline{i}_1 - \underline{i}_2 = \underline{e}_k$. The index mapping matrix can then be used as a mapping from the array index $\underline{i}$ to the DG index $\underline{j}$. Given the DG index $\underline{j}_0 = \underline{f}_i(x[\underline{0}])$ (or $\underline{j}_0 = \underline{f}_f(x[\underline{0}])$), the DG node

keeping the initial (or final) value of variable element $x[\underline{i}]$ will be $\underline{j} = U \cdot \underline{i} + \underline{j}_0$. If we assume each iteration will modify only one element $x[\underline{i}]$ of the array variable, then all the index mapping vectors are linearly independent. This is because if not all index mapping vectors are linearly independent, there exists a nonzero integral vector $\underline{v}$ such that $U \cdot \underline{v} = 0$. In this case, both $x[\underline{i}]$ and $x[\underline{i} + \underline{v}]$ will be modified at the same iteration and violate the assumption. The index mapping matrix U thus has full column rank.

DEFINITION 4. *Input/Output plane.* An input (output) plane Θ for I/O variable x is a boundary plane belongs to the input (output) region of DG with respect to the I/O variable x. Θ can be equivalently characterized by a five-tuple $[x, \Gamma, \underline{\pi}, U, \underline{j}_0]$, where Γ and $\underline{\pi}$ are the index space and normal vector of the boundary plane respectively. U is the index mapping matrix of variable x and $\underline{j}_0 = \underline{f}(x[\underline{0}])$ is a reference node.

The index mapping matrix can also be viewed as the lattice matrix of the boundary plane if all the nodes on the boundary plane are in the input/output region. Since we always consider the I/O matching problem between an output plane of the current stage and an input plane of the following stage, for simplicity, we will use the general terms I/O plane and index mapping vector (matrix) without specifying *input* or *output* explicitly.

DEFINITION 5. *Pattern matrix.* Let U be the index mapping matrix for the I/O variable $x[\underline{i}]$ in the I/O plane. Given the processor allocation function P, the pattern matrix W for x is defined as $W_{(n-1)\times l} = P_{(n-1)\times n} \cdot U_{n\times l}$.

Since a DG node $n(\underline{j})$ will be mapped to processor index $\underline{r} = P \cdot \underline{j}$, the pattern matrix can be used as a mapping from the array index $\underline{i}$ to the processor index $\underline{r}$. Given the reference processor index $\underline{r}_0 = P \cdot \underline{j}_0$, the processor holding the initial/final copy of variable element $x[\underline{i}]$ has processor index $\underline{r} = W \cdot \underline{i} + \underline{r}_0$. Even though the index mapping matrix U always has full column rank, after projection, the pattern matrix W is not necessarily to be full column rank. The following example illustrates these definitions:

Example 2. Consider a two stage consecutive matrix-matrix multiplication algorithm.

$$C_{4\times6} = A_{4\times4} \cdot B_{4\times6}$$
$$E_{4\times4} = C_{4\times6} \cdot D_{6\times4}$$

Each stage's RIA is a 3-level nested do loop with loop index (i, j, k). The array indices for the variables are:

stage 1: $a[i, k]$, $b[k, j]$, $c[i, j]$

stage 2: $c'[i, k]$, $d[k, j]$, $e[i, j]$

Note that these two stages do not have a unified DG index system because the array indices of variable c are different in two stages. The corresponding DGs are given in figure 3.

The index space of the first stage is $4 \times 6 \times 4$ and the index space of the second stage is $4 \times 4 \times 6$. Variable c is an I/O variable between two stages. The array index function of $c[i, j]$ has a null space vector $\underline{\rho}_1 = [0\ 0\ 1]^t$, $c[i, j]$ is thus propagated along the index k direction in the first stage. The array index function of $c'[i, k]$ has a null space vector $\underline{\rho}_2 = [0\ 1\ 0]^t$, $c'[i, k]$ is then propagated along the index j direction in the second stage. The output plane for $c[i, j]$ in the first stage is described by hyperplane $k = 4$ and contains all the DG nodes with DG index $\underline{f}_f(c[i, j]) = [i\ j\ 4]^t$. The input plane for $c'[i, k]$ in the second stage is described by hyperplane $j = 1$ and contains all the DG nodes with DG index $\underline{f}_i(c'[i, k]) = [i\ 1\ k]^t$. The out-

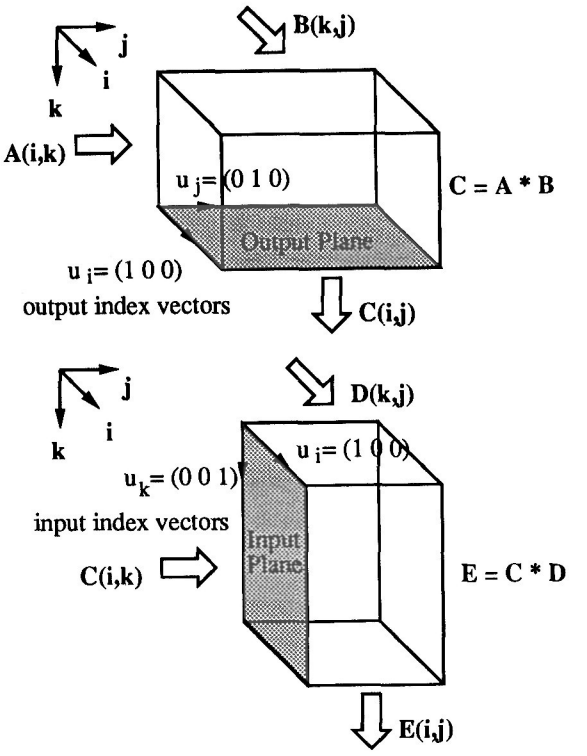


Fig. 3. DGs for 2-stage consecutive matrix-matrix multiplication algorithm.

put index mapping vectors of variable c in stage 1 (also form the basis of the output plane) are $\underline{u}_i = [1\ 0\ 0]^t$ and $\underline{u}_j = [0\ 1\ 0]^t$. Similarly, the input index mapping vectors of variable c' in stage 2 (also form the basis of the input plane) are $\underline{u}_i = [1\ 0\ 0]^t$ and $\underline{u}_k = [0\ 0\ 1]^t$. If the processor allocation function P_1 of stage 1 is

$$P_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix},$$

the pattern matrix

$$W_1 = P_1 \cdot U_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

The I/O matching problem can be divided into three sub-problems.

1. *structure matching*: the logical processor arrays of two adjacent stages should have the same lattice structure such that both stages can share the same physical processor array.
2. *location matching*: the processor allocation for an I/O variable should be the same in both stages such that no extra delay in data movement is needed when passing the value from one stage to the next.
3. *data flow matching*: the order in which an I/O variable is computed by one stage should be the same as the order in which it will be consumed by the following stage such that the processor idle time can be minimized.

3.2. Structure Matching

The logical processor arrays of two adjacent stages may differ in their sizes or shapes because of the discrepancies in their original DGs. The basic lattice structures of two arrays, however, must be the same to share the same physical array configuration. Recall that the processor allocation function P in an $(n-1) \times n$ integral matrix with full row rank, every valid node in the processor domain can be expressed as an *integral linear combination* of the column vectors in P.

DEFINITION 6. *Structural matching of two processor allocation functions.* Two processor allocation functions P_1 and P_2 are structurally matched if and only if the columns of P_1 and P_2 generate the same lattice, that is, for any $\underline{a} \in I^n$, one can find a vector $\underline{b} \in I^n$ such that $P_1 \cdot \underline{a} = P_2 \cdot \underline{b}$ and vice versa.

To test whether two processor allocation functions are structurally matched, we have to transform both P_1 and P_2 matrices into their *Hermite normal form* [19].

A matrix of full row (or column) rank is said to be in column (or row) Hermite normal form if it has the form

$$[B\ 0]\left(\begin{bmatrix} B \\ 0 \end{bmatrix}\right),$$

where B is a nonsingular, lower triangular, nonnegative matrix, in which each row (column) has a unique maximum entry, which is located on the main diagonal of B.

Each matrix of full row (column) rank has a unique column (row) Hermite normal form. It can be determined by a series of elementary column operations using a polynomial time algorithm. [19]

LEMMA 1. Theorem 4.2 in [19]. The columns of two matrices with full row ranks generate the same lattice if and only if they have the same Hermite normal form.

LEMMA 2. Two processor allocation functions P_1 and P_2 are structurally matched if and only if they have the same Hermite normal form.

Proof. From Definition 6 and Lemma 1.

The structural matching test becomes very simple if the *dense array* constraint [20], [21] is imposed on the selection of processor allocation function. That is, both P_1 and P_2 must be *extended unimodular*. (An $m \times n$ matrix A with $m \leq n$ is said to be *extended unimodular* [20] if the greatest common divisor (gcd) of the determinants of all its $m \times m$ submatrices is 1.) If P_1 and P_2 are extended unimodular, then every node in the processor index subspace is a valid node. It is shown in [20] that the Hermite normal form of an extended unimodular $m \times n$ matrix is always $[I_m\ 0]$. According to Lemma 2, two matrices with full row rank are thus structurally matched if they are both extended unimodular.

3.3. Location Matching

If both processor allocation functions are structurally matched, the problem next is to check whether I/O variables in both stages are assigned to the matched processor locations. Since each element of I/O variable $x[\underline{i}]$ will be mapped to processor $W \cdot \underline{i} + \underline{r}_0$, where

W is the pattern matrix and $\underline{r}_0 = P \cdot \underline{j}_0$ is the reference processor index, the locations of the I/O variable $x[\underline{i}]$ on the processor array can be uniquely described by W and $\underline{r}_0$. (Note: $\underline{j}_0$ is defined in Definition 4.) The task of testing two mappings have matched I/O locations can be accomplished by simply checking the equivalence of two pattern matrices $W_1 = W_2$ and the equivalence of two reference processor indices $\underline{r}_{01} = \underline{r}_{02}$. Since $\underline{r}_{01}$ and $\underline{r}_{02}$ are just the reference processor indices of the processor array, they can always be made equal by adding an index offset vector when performing the systolic mapping and hence will no longer appear in the following discussion of I/O matching problem. In addition to merely examining the conditions of location matching between two mappings, a more challenging task is how to derive I/O matched mapping in the second stage under the constraint imposed by the first stage's mapping result. The problem can be formulated as follows:

Location Matching Problem: Given a pattern matrix W_1 of stage 1 and the index mapping matrix U_2 of stage 2, find a processor allocation P_2 such that $W_1 = P_2 \cdot U_2 = W_2$.

The process of finding such a processor allocation function can be decomposed into the following two steps:

1. find the projection vector $\underline{\lambda}$
2. construct a valid processor allocation function P from its null space vector $\underline{\lambda}$.

The projection vector (null space vector) of P, provided one exists, can be found using the procedure listed below:

Procedure 1: To find the null space vector of P

1. Perform the elementary column operation to convert the W_1 matrix into a lower triangular matrix $\tilde{W}_1$. That is, to find an $(n-1) \times (n-1)$ matrix C such that
$$W_1 \cdot C = \tilde{W}_1$$
2. Denote $\tilde{U}_2 = U_2 \cdot C$. Since C is full rank, and U_2 has full column rank, $\tilde{U}_2$ must also have full column rank.
3. If the number of zero columns in $\tilde{W}_1$ is greater than one, multiple projection is needed to achieve such a pattern matrix. Return a FAIL message indicating that a unique single null space vector of the processor allocation function P_2 cannot be found.
4. If there is no zero column in $\tilde{W}_1$, any projection vector which is not spanned by the column vectors of $\tilde{U}_2$ will preserve the rank of $\tilde{U}_2$ and is thus a potentially valid projection vector.
5. If there is exactly one zero column in $\tilde{W}_1$, the corresponding column in $\tilde{U}_2$ will be derived null space vector $\underline{\lambda}_2$ of P_2.

Note that null space vector only determines the projection direction, there are infinite many processor allocation functions P corresponding to this vector.

DEFINITION 7. *Equivalence of two processor allocation functions.* Two processor allocation functions P_1 and P_2 are said to be equivalent if they correspond to the same projection vector (null space vector).

Because P_1 and P_2 are both with full row rank and have the same null space vector, the row vectors of P_1 and P_2 will span the same space. There hence exists a nonsingular matrix T such that $P_1 = T \cdot P_2$. However, if T is not unimodular, then P_1 and P_2 will have different Hermite normal forms and cannot generate the same lattice. Therefore, two equivalent processor allocation functions do not mean they are structurally matched. Similarly, two structurally matched processor allocation functions may have different projection vectors and are not equivalent, either. In the following, we will apply the dense array constraint and confine the processor allocation functions to be the extended unimodular ones where the strucural matching criterion is automatically guaranteed. Since different P's will lead to different pattern matrices, the next step is to construct the P matrix leading to the desired pattern matrix.

LEMMA 3. Equivalence of two processor allocation functions. If P and $\tilde{P}$ are two $(n-1) \times n$ extended unimodular matrices with the same null space vector $\underline{\lambda}$, there exists an $(n-1) \times (n-1)$ unimodular matrix T such that $P = T \cdot P$.

Proof. Because P and $\tilde{P}$ are equivalent, there exists a nonsingular matrix T such that $T \cdot \tilde{P} = P$. Since an $(n-1) \times n$ extended unimodular matrix has a Hermite normal form $[I_{n-1}\ \underline{0}]$, we have

$$T \cdot \tilde{P} \cdot \tilde{C} = P \cdot \tilde{C}$$

$$\Rightarrow T \cdot [I_{n-1}\ \underline{0}] = P \cdot \tilde{C}$$

$$\Rightarrow [T\ \underline{0}] = P \cdot \tilde{C}$$

Because $P \cdot \tilde{C}$ is integral and the elementary column operations will preserve the gcd of all order $n-1$ submatrices, which is 1 for extended unimodular matrix P, T thus must be unimodular.

The lemma states: starting with an arbitrary extended unimodular processor allocation function $\tilde{P}$, we can construct any *equivalent* (with the same projection vector) and *structurally matched* (extended unimodular as well) processor allocation function P by multiplying a *unimodular* matrix T to $\tilde{P}$. In the location matching problem, an arbitrary extended unimodular processor allocation function $\tilde{P}$ for the projection vector $\underline{\lambda}$ is derived first. This will lead to a pattern matrix $W = \tilde{P} \cdot U$. We then try to find if there exists a unimodular matrix T such that $T \cdot W$ is the same pattern matrix as the one determined by the previous stage. The desired processor allocation function of current stage is then equal to $T \cdot \tilde{P}$. If such T cannot be found, according to the lemma, no other equivalent extended unimodular processor allocation functions can be found to yield the matched I/O locations.

Finding unimodular transformation matrix problem: Given two pattern matrices W_1 and W_2, to find if there exists a unimodular matrix T such that $T \cdot W_2 = W_1$.

Again, we may use the (row) Hermite normal form to determine if such T exists or not. Since the (row) Hermite normal form is unique for a matrix with full column rank, if the pattern matrix W_1 does not have full column rank (which can be determined in the procedure 1 of finding the null space vector), we will work on the submatrix $\hat{W}_1$ of W_1 that has a full column rank $k = \text{rank}\ (W_1)$ instead of W_1. Similarly, we will use the corresponding submatrix $\hat{W}_2$ of W_2, too.

LEMMA 4. Given two $(n-1) \times k$ $(k \leq n-1)$ full column rank matrices W_1 and W_2, let $H_1 = R_1 \cdot W_1$ and $H_2 = R_2 \cdot W_2$ be their Hermite normal forms and R_1, R_2 both be $(n-1) \times (n-1)$ unimodular matrices. There exists a unimodular matrix $T = R_1^{-1} \cdot R_2$ such that $T \cdot W_2 = W_1$ if and only if $H_1 = H_2$.

Proof. If part:

If $H_1 = H_2$, since both R_1 and R_2 are unimodular, $T = R_1^{-1} \cdot R_2$ is also unimodular.

only if part:

If $H_1 \neq H_2$ and assume there exists a unimodular matrix T such that $T \cdot W_2 = W_1$, then we have

$$T \cdot R_2^{-1} \cdot H_2 = W_1$$

Since $T \cdot R_2^{-1}$ is unimodular, this means H_2 is also a Hermite normal form of W_1 (because H_2 can be derived from W_1 with elementary row operations) and it contradicts the uniqueness of Hermite normal form.

Note that R_1 and R_2 are not unique, either.

Procedure 2: Construct processor allocation function P

1. Perform Procedure 1 to find out the projection vector $\underline{\lambda}_2$ of P_2. If more than one $\underline{\lambda}_2$'s can be found (the case in step 4 of Procedure 1), for simplicity, choose projection vectors from the index directions only. (Note that the chosen vector cannot be spanned by the column vectors of $\tilde{U}_2$.)
2. For the given projection vector $\underline{\lambda}_2$, construct an arbitrary extended unimodular processor allocation function $\tilde{P}_2$. Derive $W_2 = \tilde{P}_2 \cdot U_2$.
3. Apply elementary row operations to derive the row Hermite normal forms of both W_1 and W_2, i.e., $R_1 \cdot W_1 = H_1$, $R_2 \cdot W_2 = H_2$. If $H_1 \neq H_2$ then return FAIL in constructing the P_2 matrix. Otherwise let $T = R_1^{-1} \cdot R_2$.
4. The processor allocation function $P_2 = T \cdot \tilde{P}_2$.

If the pattern matrices do not have full column rank, the found T, which is based on the submatrices $\hat{W}_1$ and $\hat{W}_2$ only, should be checked with full matrix to see if $T \cdot W_2 = W_2$.

3.4. Data Flow Pattern Matching

To facilitate systolic processing, the input and output data flow patterns between adjacent stages must be matched such that no extra storage buffer or data rerouting will be needed.

PROPOSITION 1. Data flow pattern matching. Let U_1, U_2 be two index mapping matrices for an I/O variable x in two adjacent stages. Both stages have matched data flow pattern if $\underline{s}_1^t \cdot U_1 = \underline{s}_2^t \cdot U_2$, where $\underline{s}_i$, $i = 1, 2$ is the scheduling vector of stage i.

Proof. According to the definition of index mapping vector, the comuptation of $x[\underline{i}_1]$ and $x[\underline{i}_2]$ in the output plane of stage 1 are scheduled by

$$\tau_1 = \underline{s}_1^t \cdot U_1 \cdot (\underline{i}_2 - \underline{i}_1)$$

time units apart. In the input plane of stage 2, the corresponding x's are scheduled by

$$\tau_2 = \underline{s}_2^t \cdot U_2 \cdot (\underline{i}_2 - i_1)$$

time units apart. If $\underline{s}_1^t \cdot U_1 = \underline{s}_2^t \cdot U_2$, we have $\tau_1 = \tau_2$. Since, the relative timing for any pair of data in adjacent stages remains the same, they must have matched pattern of data flow.

To conclude the discussion of I/O matching, the conditions for a matched I/O mapping are summarized as follows:

Conditions for I/O matching: Two adjacent stages are said to have matched I/O mappings with respect to the I/O variable x if and only if

(1) *structure matching:* P_1 and P_2 have the same Hermite normal form.
(2) *location matching:* $W_1 = W_2$
(3) *data flow pattern matching:* $\underline{s}_1^t \cdot U_1 = \underline{s}_2^t \cdot U_2$

Example 3: I/O matching. Consider the algorithm given in example 2 (refer to figure 3 for DG). Assume the mapping in the first stage is as follows:

$$\underline{s}_1^t = [1\ 1\ 1]$$

$$U_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}$$

$$P_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

$$W_1 = P_1 \cdot U_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

στρυθτυρε ανδ λοθατιον ματθηινγς⟨

With

$$U_2 = \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix},$$

procedure 1 is first applied to find the null space vector $\underline{\lambda}_2$ of P_2. Since W_1 is full column rank and lower triangular, C is thus an identity matrix and from step 4 in Procedure 1, $\underline{\lambda}_2$ can be any vector which is not spanned by the column vectors of U_2. If we choose $\underline{\lambda}_2$ along the directions of index axes, we have

$$\underline{\lambda}_2 = [0\ 1\ 0]^t$$

Next, apply Procedure 2 to construct P_2. Choose

$$\tilde{P}_2 = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

arbitrarily. Pattern matrix W_2 is now

$$W_2 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Since W_1 is already in its Hermite normal form, $W_1 = H_1$. W_2 can be transformed into its Hermite normal form H_2 by interchanging the two rows, i.e.,

$$R_2 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

Because $H_1 = H_2$, the transformation matrix T is equal to R_2 and P_2 is

$$P_2 = T \cdot \tilde{P}_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

data flow matching:

Choose $\underline{s}_2^t = [1\ 1\ 1]$:

$$\underline{s}_1^t \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} = \underline{s}_2^t \cdot \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix} = [1\ 1]$$

Both stages have perfect I/O matching if P_2 and $\underline{s}_2$ are so chosen.

To provide a way in evaluating the mapping result, the types of I/O matching can be classified into the following three classes.

1. Class A—all three conditions of I/O matching are satisfied.
2. Class B—all but data flow condition are satisfied.
3. Class C—otherwise.

Each class of I/O matching can then be assigned a value (design cost) in an ascending order from A to C. It will be shown later that by properly choosing the values assigned to each class of I/O matching, those designs with mismatched I/O operation will have a higher design cost and are less likely to be selected as a solution of MSSM.

4. Computation Overlap between Successive Stages

Since not all the processors are activated during the initiation and conclusion phases of computation, the performance degradation is considerable if the length of these two phases is comparable to the length of *steady*

state phase when all the processors are busy. In MSSM, we may effectively reduce such performance degradation by exploring the concurrency of computations between different stages. The technique presented in this paper, called *chaining*, is to overlap the computations between the initiation/conclusion phases of adjacent stages.[1] The computation overlap problem is not independent of I/O matching problem. Since it considers only the computational profiles between successive stages, it is assumed that either there is no inter-stage data dependence or the two stages have matched I/O locations and data flows such that no extra delay needed in redirecting the data across different stages. If it is not the case, we can still try to seek the concurrency between the data rerouting process, called *data migration* or *data alignment*, and the following stage's computation. This is, however, beyond the scope of this paper.

4.1. Definitions and Terminologies

DEFINITION 8. *Pipelining Period* [18]. The pipelining period $\alpha \equiv \underline{s}^t \cdot \underline{\lambda}$ is the time interval between two successive computations of a processor, where $\underline{s}$ and $\underline{\lambda}$ are the scheduling and projection vectors respectively.

DEFINITION 9. *Logical processor array.* The logical processor array Δ can be described by a two-tuple $[\Upsilon, \Psi]$. Υ is the index space of the processor array confined by a convex polyhedron and $\Psi = \{\underline{\psi}_j\}$ is the set of all vertex nodes (processors) of Υ.

DEFINITION 10. *Initial/Final plane.* Let G be an n-dimensional dependence graph, and let $\underline{\lambda}$ and $\underline{s}$ be the projection and scheduling vectors, respectively. The initial (final) plane Φ_i (Φ_f) is a boundary plane of G with normal vector $\underline{\pi}$ satisfying either of the following conditions:

initial plane: $\underline{\lambda}^t \cdot \underline{\pi} \neq 0$ and $\underline{s}^t \cdot \underline{\pi} < 0$,

final plane: $\underline{\lambda}^t \cdot \underline{\pi} \neq 0$ and $\underline{s}^t \cdot \underline{\pi} > 0$.

Φ can be described by a quadruplet $[\Gamma, \underline{\pi}, \Omega, Z]$, where Γ and Ω are the index space and the lattice matrix of the corresponding boundary plane respectively. $Z = \{\underline{z}_i | i = 1 \text{ to } m\}$ is the set of all vertex nodes of Γ with $m > n$.

Each node on the initial plane will be mapped onto a distinct processor (since $\underline{\lambda}^t \cdot \underline{\pi} \neq 0$) and will be executed first among all the DG nodes assigned to the same processor. Similarly, each node on the final plane will be executed last among all those DG nodes mapped onto the same processor. For convenience, we will call a boundary plane an I/F plane if it is either an initial or a final plane.

DEFINITION 11. *Activation vector.* Let $\Phi = [\Gamma, \underline{\pi}, \Omega, Z]$ denote an I/F plane of a DG with four tuples defined as before. Given the scheduling vector $\underline{s}$ and the processor allocation function P, the activation vector $\underline{h}$ for Φ is defined as

$$\underline{h}^t = \underline{s}^t \cdot \Omega \cdot (P \cdot \Omega)^{-1}$$

Because the projection vector $\underline{\lambda}$ satisfies $\underline{\lambda}^t \cdot \underline{\pi} \neq 0$, $\underline{\lambda}$ is not parallel to the I/F plane and hence $P \cdot \Omega$ is a full rank $(n-1) \times (n-1)$ matrix. Consider any two nodes $n(\underline{j}_1)$ and $n(\underline{j}_2)$ on Φ such that $\underline{y} = \underline{j}_2 - \underline{j}_2 \in I^n$. Since $\underline{y}$ is parallel to the I/F plane Φ and Ω is the lattice matrix, $\underline{y}$ can be expressed as

$$\underline{y} = \sum_{i=1}^{n-1} c_i \cdot \underline{\omega}_i = \Omega \cdot \underline{c}$$

where the c_i's are integers and $\underline{\omega}_i \in \Omega$. The time interval δ between the scheduling of these two nodes is

$$\delta = \underline{s}^t \cdot \underline{y}.$$

Let $\underline{x}_1$ and $\underline{x}_2$ be the processor indices of the two processors to which the nodes $n(\underline{j}_1)$ and $n(\underline{j}_2)$ allocated, then

$$\underline{x} \equiv \underline{x}_2 - \underline{x}_1 = P \cdot y.$$

Therefore, we have the following relation:

LEMMA 5. $\underline{h}^t \cdot \underline{x} = \delta$

Proof.

$$\underline{h}^t \cdot \underline{x} = \underline{h}^t \cdot P \cdot \underline{y} = \underline{h}^t \cdot P \cdot \sum_{i=1}^{n-1} c_i \cdot \underline{\omega}_i$$

$$= \underline{h}^t \cdot P \cdot \Omega \cdot \underline{c}$$

$$= \underline{s}^t \cdot \Omega \cdot (P \cdot \Omega)^{-1} \cdot (P \cdot \Omega) \cdot \underline{c}$$

$$= \underline{s}^t \cdot \Omega \cdot \underline{c} = \underline{s}^t \cdot \underline{y} = \delta$$

The inner product of the activation vector $\underline{h}$ and the vector $\underline{x}$ indicates the relative scheduling of the first (or the last) computations for two processors separated by a distance vector $\underline{x}$. Similar to the scheduling vector in describing the *equitemporal* hyperplane (the set of nodes which are scheduled at the same time) in the DG,

the activation vector can be used to describe the first (or the last) computation wavefront [18] (a hyperplane as well) consisting of all the processors starting their first computations (or finishing their last computations) at the same time. The activation vector is then the normal vector to this computation wavefront.

DEFINITION 12. *Computation overlap period* τ. Let $\underline{s}_i$ and ζ_i, $i = 1, 2$, be the scheduling vector and global timing offset of two successive stages such that a DG node $n(\underline{j})$ in stage i will be scheduled at $t_i = \underline{s}_i^t \cdot \underline{j} + \zeta_i$, where t_i is measured with respect to the global time index. Also let $\tilde{t}_1 = max\{\underline{s}_1^t \cdot \underline{j}_1 + \zeta_1 | \underline{j}_1 \in IS_1\}$ and $\tilde{t}_2 = min\{\underline{s}_2^t \cdot \underline{j}_2 + \zeta_2 | \underline{j}_2 \in IS_2\}$ where IS_1 and IS_2 are the index spaces of stage 1 and stage 2 respectively. The computation overlap period τ is defined as

$$\tau = \tilde{t}_1 - \tilde{t}_2 + 1$$

Note that $\tilde{t}_1$ is the scehduled time instance for the last node's computation in stage 1. Similarly, $\tilde{t}_2$ is the scheduled time instance for the first node's computation in stage 2. The computation overlap period τ is then the period when two adjacent stages perform the computations concurrently. In the worst case, $\tilde{t}_2 - \tilde{t}_1 = 1$, i.e., $\tau = 0$, there is no computation overlap at all. The above definitons can be illustrated in the following example.

Example 4. Computation overlap of two matrix-matrix multiplications.

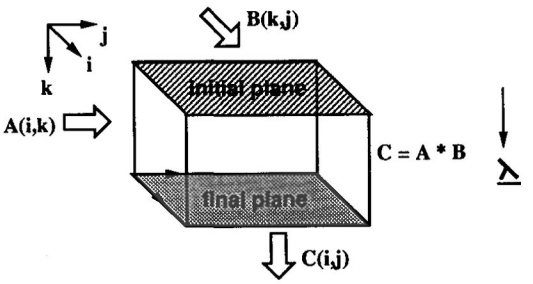
Consider the first stage matrix-matrix multiplication algorithm in example 2

$$C_{4\times 6} = A_{4\times 4} \cdot B_{4\times 6}$$

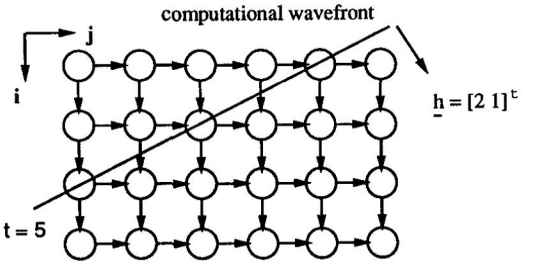
Referring to figure 4, the corresponding DG is a $4 \times 6 \times 4$ cube. There are six boundary planes of DG defined by hyperplanes $i = 0$; $i = 3$; $j = 0$; $j = 5$; $k = 0$ and $k = 3$. Choosing scheduling vector $\underline{s}$ and projection vector $\underline{\lambda}$ to be $[2\ 1\ 1]^t$ and $[0\ 0\ 1]^t$ respectively, the boundary plane $k = 0$ will be an initial plane with normal vector $\underline{\pi}_1 = [0\ 0\ -1]^t$ and the boundary plane $k = 3$ will be a final plane with normal vector $\underline{\pi}_2 = [0\ 0\ 1]^t$. The activation vector $\underline{h} = [2\ 1]^t$.

4.2. Maximum Computation Overlap Period

The maximum computation overlap period τ_{max} is the longest possible period when two adjacent stages can perform their computations concurrently. We will first assume that there is only one I/F plane under a given



(a) initial and final planes of DG



(b) computational wavefront in logical processor array

Fig. 4. I/F planes and activation vector of a matrix-matrix multiplication algorithm.

projection direction $\underline{\lambda}$. The case for more than one I/F planes will be addressed later. Let Γ and Z denote the index space and the set of vertex nodes in the I/F plane respectively. Because Γ is convex and $\underline{\lambda}^t \cdot \underline{\pi} \neq 0$, where $\underline{\pi}$ is the normal vector of I/F plane, the projection of Γ on the logical processor array (denoted as $\tilde{\Upsilon}$) is also convex. In addition, a vertex node $\underline{z}_i \in Z$ in the I/F plane will be mapped onto a vertex processor in $\tilde{\Upsilon}$ as well. If Υ denotes the index space of the logical processor array then $\tilde{\Upsilon} \subset \Upsilon$, but both share the same set of vertex processors $\Psi = \{\underline{\psi}_j | \underline{\psi}_j = P \cdot \underline{z}_j \ \forall\ \underline{z}_j \in Z\}$. The vertex processors of the logical processor array can thus be determined by the vertex nodes in I/F plane. Without loss of generality, we may assume that two adjacent stages share the same logical processor array, i.e., both have the same set of vertex processors Ψ. If this is not the case, we may always construct a subarray which is the intersection of two logical processor arrays. (Note that the intersection of two convex polyhedra is also convex.) The processors not in the intersection imply they are activated in one stage only and can be simply ignored because they will not affect the computation overlap between adjacent stages.

Condition for computation overlap: Let $\underline{h}_i$ and $\underline{s}_i$, $i = 1, 2$ denote the activation vector and scheduling vector in two successive stages respectively. Also let η_i be the global timing offset such that each processor

with index $\underline{y}$ will perform its last computation in the first stage at a global time instance $\underline{h}^t \cdot \underline{y} + \eta_1$, and will start its first computation in the second stage at $\underline{h}^t \cdot \underline{y} + \eta_2$. Then, for any $\underline{y}$ in the processor array, the condition

$$\underline{h}_1^t \cdot \underline{y} + \eta_1 < \underline{h}_2^t \cdot \underline{y} + \eta_2$$

must be satisfied to ensure no processor scheduling conflict.

PROPOSITION 2. *Simplified condition for computation overlap.* Define $\underline{h}_i$, $\underline{s}_i$, η_i, $i = 1, 2$ as above. If for every vertex processor $\underline{\psi}_j \in \Psi$, $j = 1$ to m, we have

$$\underline{h}_1^t \cdot \underline{\psi}_j + \eta_1 < \underline{h}_2^t \cdot \underline{\psi}_j + \eta_2$$

then all the processors $\underline{y}$'s in the logical processor array will also satisfy the above condition.

Proof. Since the index space Υ of processor array is convex, any node $\underline{y} \in \Upsilon$ can be expressed as

$$\underline{y} = \sum_{j=1}^{m} c_j \cdot \underline{\psi}_j$$

where $0 \leq c_j \leq 1$ and $\Sigma_{j=1}^{m} c_j = 1$. Hence,

$$\underline{h}_1^t \cdot \underline{y} + \eta_1 = \underline{h}_1^t \cdot \sum_{j=1}^{m} c_j \cdot \underline{\psi}_j + 1 \cdot \eta_1$$

$$= \sum_{j=1}^{m} c_j \cdot (\underline{h}_1^t \cdot \underline{\psi}_j + \eta_1)$$

$$< \sum_{j=1}^{m} c_j \cdot (\underline{h}_2^t \cdot \underline{\psi}_j + \eta_2) = \underline{h}_2^t \cdot \underline{y} + \eta_2$$

We now present a procedure to find the maximum computation overlap period τ_{max} by minimizing the value $b = \eta_2 - \eta_1$.

PROCEDURE 3. *Maximum computation overlap period.* Let $\Psi = \{\underline{\psi}_j\}$ be the set of vertex processors shared by two logical processor arrays. Also let $\underline{h}_1$ and $\underline{h}_2$ be the activation vectors of two adjacent stages respectively.

Step 1:

find a minimum constant b such that

$$\underline{h}_1^t \cdot \underline{\psi}_j < \underline{h}_2^t \cdot \underline{\psi}_j + b$$

for all $\underline{\psi}_j$'s $\in \Psi$. b is then equal to

$$b = max\{(\underline{h}_1 - \underline{h}_2)^t \cdot \underline{\psi}_j + 1 \,|\, \forall\, \underline{\psi}_j \in \Psi\}$$

Step 2:

find $t_1 = max\{\underline{h}_1^t \cdot \underline{\psi}_j\}$ and $t_2 = min\{\underline{h}_2^t \cdot \underline{\psi}_j\}$ for all j's.

The maximum computation overlap period is then equal to

$$\tau_{max} = t_1 - (t_2 + b) + 1$$

The effective computation latency for the i-th stage by overlapping the computations between the $(i - 1)$-th stage's conclusion phase and i-th stage's initiation phase now becomes $\hat{L}_i = L_i - \tau_{max}$, where L_i is the computation latency of stage i without computation overlap. If τ_{max} is equal to zero, no computation overlap between these two stages is possible.

Example 5. Computation overlap.

Assume two adjacent stages both have the same 4×6 rectangle logical processor array, then $\Psi = \{[0\ 0]^t, [0\ 5]^t, [3\ 0]^t, [3\ 5]^t\}$ contains 4 vertex processors. Let $\underline{h}_1 = [1\ 1]^t$ and $\underline{h}_2 = [1\ -1]^t$ denote the activation vectors in two stages respectively. The maximum computation overlap period can be found as follows:

Step 1:

$$b = max\{(\underline{h}_1^t - \underline{h}_2^t) \cdot \underline{\psi}_j + 1 \,|\, \forall\, \underline{\psi}_j \in \Psi\}$$
$$= max\{1, 11, 1, 11\} = 11$$

Step 2:

$$t_1 = max\{\underline{h}_2^t \cdot \underline{\psi}_j \,|\, \forall\, \underline{\psi}_j\} = 8$$
$$t_2 = min\{\underline{h}_2^t \cdot \underline{\psi}_j \,|\, \forall\, \underline{\psi}_j\} = -5$$

The max computation overlap period τ_{max} then equal to

$$\tau_{max} = t_1 - b - t_2 + 1 = 3$$

The procedure for finding the computation overlap period with more than one I/F planes in each stage is similar to the one described above. The problem is first decomposed into a set of subproblems dealing with the intersection of only one I/F plane from each stage. The constant b in step 1 of Procedure 3 is now the maximum of all b's found in each subproblem. Note that in Procedure 3, the global timing offsets η_2 and η_1 are never calculated explicitly because there is only one I/F plane in each stage and every node will have the same global timing offset. This, however, is not true for the case with more than one I/F planes in each stage. A different approach is adopted here by finding the difference of two global timing offsets ζ_1 and ζ_2 (both

defined in Definition 12) first. The maximum computation overlap period τ_{max} is then equal to

$$\tau_{max} = max\{\underline{s}_1^t \cdot \underline{j}_1 | \underline{j}_1 \in IS_1\} - min\{\underline{s}_2^t \cdot \underline{j}_2 | \underline{j}_2 \in IS_2\} - \zeta + 1$$

where $\zeta = \zeta_2 - \zeta_1$. Because DG is convex, to find either $min\{\underline{s}_2^t \underline{j}_2 | \underline{j}_2 \in IS_2\}$ or $max\{\underline{s}_1^t \cdot \underline{j}_1 | \underline{j}_1 \in IS_1\}$, only the vertex nodes in all the I/F planes need to be checked. The procedure can be summarized as follows:

PROCEDURE 4. Maximum computation overlap period. Let $\{\Phi_{k,1}\}$ and $\{\Phi_{l,2}\}$ denote the set of I/F planes in stages 1 and 2 respectively.

Step 1:
For any pair of $(\Phi_{k,1}, \Phi_{l,2})$, if the intersection of the two logical processor arrays corresponding to the mapping of $\Phi_{k,1}$ and $\Phi_{l,2}$ is not empty, then perform step 1 in Procedure 3 to find constant b.

Step 2:
Choose $\hat{b}$ to be the maximum of all b's

Step 3:
Pick an arbitrary vertex processor $\underline{\psi}$ from the logical processor array. Let $\underline{z}_1 \in \Phi_{k,1}$ and $\underline{z}_2 \in \Phi_{l,2}$ be the corresponding DG vertex nodes in the I/F plane of two stages, respectively. That is, $P_1 \cdot \underline{z}_1 = P_2 \cdot \underline{z}_2 = \underline{\psi}$, where P_i is processor allocation function for stage i. The global timing difference ζ is now equal to

$$\zeta = \zeta_2 - \zeta_1 = \underline{s}_1^t \cdot \underline{z}_1 - \underline{s}_2^t \cdot \underline{z}_2 + (\underline{h}_2^t - \underline{h}_1^t) \cdot \underline{\psi} + \hat{b}$$

The maximum computation overlap period τ_{max} is equal to

$$\tau_{max} = max\{\underline{s}_1^t \cdot \underline{z}_1 | \underline{z}_1 \in Z_{k,1} \ \forall k\} - min\{\underline{s}_2^t \cdot \underline{z}_2 | \underline{z}_2 \in Z_{l,2} \ \forall l\} - \zeta + 1$$

where $\underline{s}_i$ is the scheduling vector for stage i and $Z_{m,i}$ denotes the set of vertex DG nodes for I/F plane $\Phi_{m,i}$.

5. Optimization of Multi-Stage Mapping Problem

With the results obtained in previous sections, we are now able to develop an automated design tool to aid the selection of projection and scheduling directions for each stage of a multi-stage algorithm.

5.1. Parameterized Dependence Graph

The algorithm representation adopted in the multi-stage systolic mapping is called *parameterized dependence graph* (PDG). A regular iterative algorithm can be described by a shift invariant dependence graph. However, due to different algorithm transformation techniques, more than one DGs may be derived from the same computing algorithm. The transformation is usually to convert global broadcasting variables into local transmittal variables propagated from one processor to another. Apparently, the transmittal variable can be propagated in either direction (if the trajectory of broadcasting is a straight line) without affecting the correctness of the computation. One thus usually chooses any one of the two possible transmission directions arbitrarily and yields different dependence graphs. However, not all of these DGs exhibit the same property in parallel computing. For multi-stage systolic mapping, where the mapping problem is even complicated by the inter-stage interface problems, committing to any choice of DG too early may eliminate a better solution prematurely. These options hence should be kept open so that a more sensible decision can be made in later design phase.

In a PDG, the dependence vector corresponding to a propagation direction of a transmittal variable and the dependence vector corresponding to a recursive variable of an associative operation are parameterized. The directions of them are decided by a parameter $m_i = \pm 1$. The actual value of m_i will not be determined until later mapping phases. In doing so, we are able to collapse all possible DGs into a single, compact representation. This representation facilitates systematic exploration of those DGs corresponding to the same computing algorithm. Details on PDGs can be found in [22].

5.2. Optimization Formulation

We model the multi-stage mapping procedure as an optimization problem in which the projection directions and scheduling vectors of every stage will be determined subject to a design cost function. A design cost function is a weighted linear combination of several design measurement variables. Each design measurement variable regards one particular design issue in multi-stage systolic mapping and is assigned a numerical value to quantize the merits of mapping design. The design measurement variables considered in MSSM are

1. N_i: the number of processors needed for the i-th stage's computation.
2. α_i: the pipelining period within the i-th stage.
3. $\hat{L}_i$: the computation latency of the i-th stage subtracting the computation overlap period between state $i-1$ and stage i.
4. β_i: the type of I/O matching between the $(i-1)$-th and i-th stages.
5. δ_i: the number of additional communication links introduced in stage i for each processor.

For a k-stage algorithm, the design cost function is then expressed as

$$g(\cdot) = \sum_{i=1}^{k} (c_1 \cdot N_i + c_2 \cdot \alpha_i + c_3 \cdot \hat{L}_i + c_4 \cdot \beta_i + c_5 \cdot \delta_i)$$

The c_i's are the weighting coefficients associated with each design measurement variable. The user may supply different weightings for different design emphases. Since the assigned values of design measurement variables are proportional to the design complexity (cost), the ultimate goal of MSSM is to minimize the design cost function.

5.3. Best First Search Modeling

Because the design cost function cannot be expressed in a close form of scheduling and projection vectors, this optimization problem is solved by searching all the feasible solutions, i.e., the combinations of mapping and scheduling vectors at each stage. Without imposing any constraints, the number of potential solutions can be very large: Assume a k-stage algorithm with N nodes in each stage's DG. In each stage, there will be $O(N)$ distinct scheduling vectors and $O(N)$ projection vectors. The total distinct choices of scheduling and projection vectors for a k-stage algorithm is $(N)^{2k}$ which becomes impractically large even with modest values of N and k. For example, let $N = 100$ and $k = 3$, the exhaustive search space is now $(100)^{2\cdot3} = 10^{12}$. To avoid the combinatorial explosion of the search space, a *best-first* heuristic search procedure is implemented. We first reduce the potential search space by pruning out least promising solutions with heuristic rules. Next, we conduct an incremental, best first heuristic search to identify a most favorable solution from this reduced search space. Because of PDG, in our search model, we first enumerate all the possible parameter combinations for each stage (usually a small number). For each different parameter setting, which corresponds to a different DG alternative, the projection direction and hence the processor allocation function is determined first. Then the scheduling vector within the same stage is chosen next. We will confine our processor allocation function heuristically to the following 3 classes:

1. Projections giving minimum number of processors.
2. Projections leading to type A or type B I/O matching.
3. Projections along any coordinate axis.

As for the scheduling vector, only those with highest throughput rate or with shortest computation latency will be explored. The algorithm starts from the first stage's DG and concludes with the last stage's DG.

One special feature arising from this search process is that often there are a number of equally good, in terms of the design cost function, partial solutions at certain stages. Each of them may, however, have different impact on later stage's mappings. Our strategy is to keep all of them in an open queue, and proceed the design with an arbitrarily chosen one. Later, when it becomes apparent that this choice is inferior, we backtrack to this current node to make another choice. The detailed search algorithm is given in Appendix A.

5.4. Program Implementaiton (MSSM)

MSSM is the prototype implementation of the multi-stage mapping algorithm described above. It is built on top of the VLSI Array Compiler System (VACS) described by Kung and Jean as an add-in module. VACS is a graphics-oriented computer aided design package which accepts a high level algorithmic input in terms of a dependence graph (DG), and generates (optimal) VLSI array structures. Several optimality criteria and design tradeoffs may be evaluated, design correctness may be verified by simulation, and design errors may be detected at the earliest possible stage [23]. Since our design methodology for individual stage mappings is the same as that used in VACS, it is convenient to use VACS in finding a feasible set of scheduling and projection vectors and then develop our multi-stage CAD package by modifying the VACS program. MSSM will accept parameterized DGs as an input and then conduct a best first search to find out the projection vectors, scheduling vectors as well as the values of the parameters for each stage.

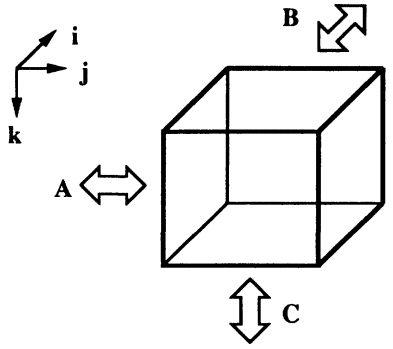
6. Design Examples

We will now give several examples to illustrate the multi-stage mapping algorithm.

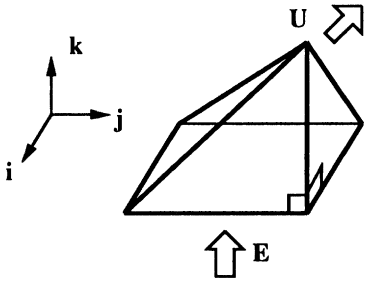
Example 6. 3-stage matrix multiplication and LU decomposition.

The algorithm consists of two successive matrix multiplications followed by an LU decomposition. The parameterized DGs for the matrix multiplication and LU decomposition are shown in figure 5(a) and figure 5(b) respectively. The MSSM mapping result is summarized as follows:

3-Stage Algorithm Design			
Factor	Stage 1	Stage 2	Stage 3
Scheduling vector	(1 1 1)	(1 −1 1)	(1 1 1)
Projection vector	(0 0 1)	(0 1 0)	(0 0 1)
No. of processors	16	16	10
Computation latency	10	10	10
Accumulated cost	160	224	390
Best first search cost	294	294	390



(a) PDG of the matrix-matrix multiplication algorithm



(b) DG of the LU factorization algorithm

Fig. 5. DGs for a 3-stage algorithm.

Note that the *accumulated cost* in the table entry means the design cost up to the current stage. In order not to penalize those nodes with more depth in the search tree, an adjusted cost, i.e., *best first search cost* in the table entry, is used as the cost associated with each node. It is equal to the sum of accumulated cost and the estimate cost from the current stage to the last stage. The estimate cost is the minimum possible cost that can be obtained based on minimum processor counts, minimum pipeline period, best I/O matching and computation overlap. All these terms can be calculated before the search process begins. The corresponding systolic array design is given in figure 6. The pipelined computations between the first and second stages are perfectly matched. We will, however, need an inter-stage buffer between the second and the third stage. It can be easily verified that there are at most two consecutive stages can be matched in this three-stage algorithm. An architectural model with toroidal communication links can be used to solve this problem. According to the MSSM program, there are only 127 nodes visited in the best first search algorithm. This means the heuristic rules that we used in MSSM can greatly reduce the search space while preserving the quality of mapping.

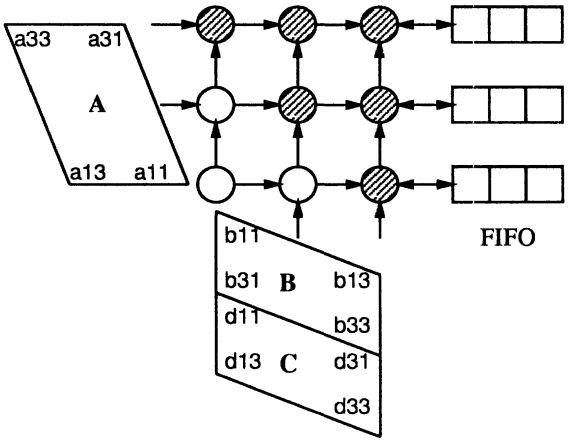


Fig. 6. Array design of a 3-stage algorithm.

As mentioned above, different designs may be obtained by simply changing the heuristic cost functions. This can be shown by the following example in which two consecutive matrix-matrix multiplications are performed.

Example 7. 2-stage consecutive matrix-matrix multiplications.

Consider the 2-stage algorithm in example 2, with two different design criteria, the MSSM mapping result is given below.

Two Designs With Different Design Criteria				
	Design 1		Design 2	
Factor	Stage 1	Stage 2	Stage 1	Stage 2
Scheduling vector	(1 1 1)	(1 1 1)	(1 1 1)	(1 1 1)
Projection vector	(0 0 1)	(0 1 0)	(0 1 0)	(0 0 1)
No. of processors	24	24	16	16
Computation latency	12	12	12	12
Accumulated cost	408	576	352	544
Best first search cost	520	576	464	544

In the first design, the weighting factor for the processor cost is set to zero so that we arrive at a larger array design but the pipelined computations across two stages are perfectly matched. In the second design, the weighting factor for the processor cost is set to a much larger value so that the hardware concern will override the performance concern. In this case, the number of processors needed is minimum (16), however, the total computation time will be increased due to I/O mismatch between two stages.

Example 8. Kalman Filter.

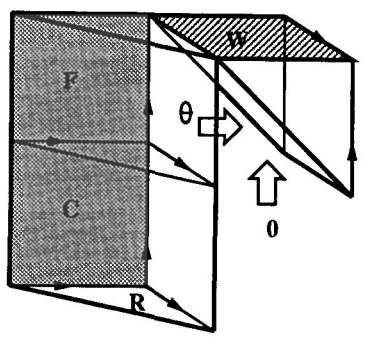
We adopt the recursive formulation by Paige [24] and try to compare the mapping result with the previous work on Kalman Filter by [18]. The recursive formulation is to perform the QR decomposition to a block matrix and make it an upper triangular one.

$$Q_{k+1}^{H} \cdot \begin{bmatrix} R(k) & 0 & b_k \\ \tilde{C}(k) & 0 & \tilde{y}_{k+1} \\ \tilde{F}(k) & W(k+1) & 0 \end{bmatrix} = \begin{bmatrix} R_{k,k} & R_{k,k+1} & b_k' \\ 0 & R(k+1) & b_{k+1} \\ 0 & 0 & r_k \end{bmatrix}$$

The formulation can be considered as a 3-stage algorithm. In the first stage, $\tilde{C}(k)$ is nullified by the Given's rotation. In the second stage, $\tilde{F}(k)$ is nullified subsequently and finally in the last stage, the $R(k+1)$ matrix is obtained. The DGs for 3 stages are given in figure 7. The MSSM mapping result is summarized as follows and the corresponding systolic design is shown in figure 8.

Kalman Filter Design			
Factor	Stage 1	Stage 2	Stage 3
Scheduling vector	(1 1 1)	(1 1 1)	(−1 1 1)
Projection vector	(0 0 1)	(0 0 1)	(0 1 0)
No. of processors	10	10	10
Computation latency	10	10	7
Accumulated cost	100	140	255
Best first search cost	225	195	255

In this example, we have exactly the same tri-array design as the one proposed in [18].



Note : The arrows in the graph stand for dependence vectors
input plane output plane

Fig. 7. DGs for the 3-stage Kalman filter algorithm.

Example 9. Singular Value Decomposition.

In [25], a three phase operation for k-th iteration of the singular value decomposition is proposed as follows:

- *Phase 1:* QR decompositon for $A_k = Q_k \cdot R_k$
- *Phase 2:* computing the Q_k matrix by solving $R_k^t \cdot Q_k^t = A_k^t$
- *Phase 3:* computing $A_{k+1} = R_k \cdot Q_k$

The SVD can be considered as a multi-stage algorithm with each phase's operation equivalent to a single stage. The MSSM mapping result is summarized as follows. The DG and the corresponding systolic design are shown in figure 9, and figure 10 respectively.

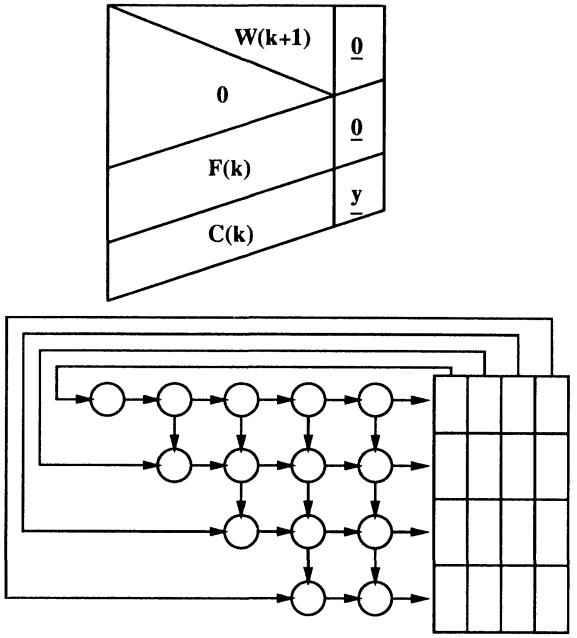


Fig. 8. Tri-array configuration for the Kalman filter.

Singular Value Decomposition			
Factor	Stage 1	Stage 2	Stage 3
Scheduling vector	(1 1 1)	(1 1 −1)	(1 1 1)
Projection vector	(1 0 0)	(0 0 1)	(0 1 0)
No. of processors	10	10	10
Computation latency	10	10	10
Accumulated cost	100	140	225
Best first search cost	210	210	225

The derived array design by MSSM is the same as the tri-array design obtained in [25].

7. Conclusion

In this paper, we introduced a more general mapping problem called multi-stage systolic mapping which considers computing algorithms containing more than one nested loop constructs to be executed sequentially. Since a single systolic array is reused for each stage's computation, the interface problem between successive stages has now become a dominant factor in performing the

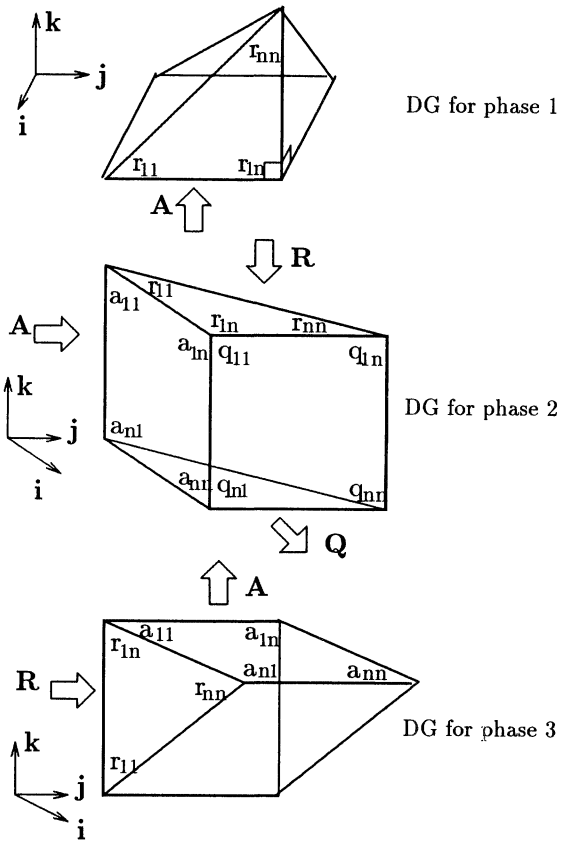


Fig. 9. DGs for 3-stage iterative singular valude decomposition.

multi-stage systolic mapping. Two interface problems, i.e., I/O matching and computation overlap, have been described and formulated in this paper. To provide a framework of systematic approach to these problems, we established the conditions of I/O matching and proposed a procedure to derive an I/O matched mapping. We also presented a method to explore the maximum computation overlap period between two successive stage's computations. By incorporating these new results, we developed an automated design tool, called MSSM, for multi-stage systolic mapping. The MSSM is built on the top of VACS, which is a graphic oriented mapping tool for single stage algorithm, and takes parameterized dependence graphs as the input. In MSSM, the mapping problem is formulated as an optimization problem subject to a design cost function. A heuristic best first search algorithm is then proposed to find the design in each stage. From the several design examples compiled by MSSM, we are confident that this design tool is able to produce a multi-stage design which rivals the existing designs done by human experts.

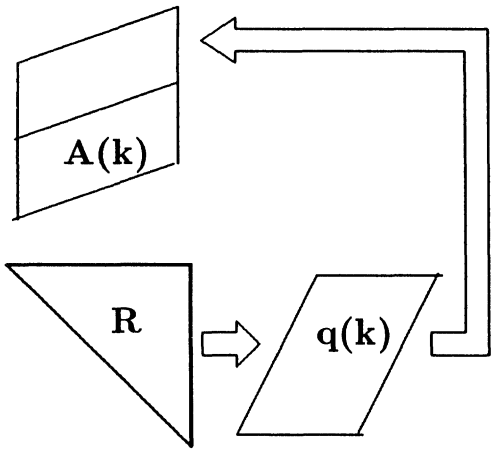


Fig. 10. Tri-array configuration for the iterative singular value decomposition.

Acknowledgment

The authors would like to thank Prof. S. Kung at Princeton University, and Dr. S.N. Jean at the Wright State University for sharing the source code of VACS with us. An earlier version of the MSSM program was written by Mr. Gebre Gessesse as a class project. The authors also gratefully acknowledge the valuable comments of the anonymous reviewers in improving the earlier draft of this paper. This work is supported in part by a grant from National Science Foundation under contract No. MIP-8896111, and in part by a grant from the Graduate School of the University of Wisconsin—Madison. A preliminary version of this paper appeared in *The 4-th Workshop on VLSI Signal Processing* held on November 7–9, 1990, San Diego, CA.

Note

1. Another technique to achieve computation concurrency is called *pipeline interleaving* which performs more than one stage's computations in an interleaving manner.

Appendix

Best-First Search Algorithm for Multi-Stage Systolic Mapping

Given a k-stage algorithm A, let Y_i be the parameter set associated with the i-th stage. Each different setting of Y_i is denoted by Y_{ij}. Let $D(Y_{ij})$ be the set of i-th stage's dependence vectors under the parameter setting Y_{ij}. Let Ψ denote the open node queue with each element ξ containing four fields: (μ, ν, η, ρ), where μ represents the parent node of ξ in the search tree, ν is either a scheduling vector $\underline{s}_i$ or a processor allocation function P_i for the i-th stage, η is the accumulative design cost up to the current node ξ, and ρ is the adjusted design cost for the entire mapping. The notations ξ_μ, ξ_ν, ξ_η, ξ_ρ are used to single out the specific field in the element ξ. Also let $g_i(\cdot)$ be the design cost function for stage i and $\Gamma(i)$ be the minimum accumulative design cost from stage i to the last stage k. The adjusted design cost is thus equal to $\xi\_\rho = \xi\_\eta + \Gamma(i+1)$. $\Gamma(i)$ can be calculated by choosing the minimum values of all design measurement variables.

1. for $i = 2, \ldots, k$, calculate $\Gamma(i)$
2. Initialization: $\Psi \leftarrow \phi$; $i = 1$;
3. for each different parameter setting Y_{1j}
   use VACS to generate a set of projection vectors $\{\underline{\lambda}_{1p}\}$; select only those projection vectors which satisfy at least one of the following criteria:
   1) minimum number of processors N_i
   2) type A or B I/O matching
   3) projection $\underline{\lambda}_i$ along any coordinate axis
   for each different $\underline{\lambda}_{1p}$
   $cost = g_1(N_1, \delta_1)$;
   $\rho = cost + \Gamma(2)$;
   insert $\epsilon = (nil, P_{1p}, \text{cost}, \rho)$ into Ψ;
   sort Ψ with key ρ;
4. repeat
   retrieve the first element ξ from Ψ;
   if $\xi\_\nu = \underline{s}_i$
   $i = i + 1$;
   for each different parameter setting Y_{ij}
   use VACS to generate a set of projection vectors $\{\underline{\lambda}_{ip}\}$;
   select only those projection vectors which satisfy at least one of the following criteria:
   1) minimum number of processors N_i
   2) type A or B I/O matching
   3) projection $\underline{\lambda}_i$ along any coordinate axis
   for each different $\underline{\lambda}_{ip}$
   $cost = g_i(N_i, \delta_i) + \xi\_\eta$;
   $\rho = cost + \Gamma(i+1)$;
   insert $\epsilon = (\xi, P_{ip}, \text{cost}, \rho)$ into Ψ;
   sort Ψ with key ρ;
   else
   for the corresponding Y_{ij} of ξ_ν
   use VACS to generate a set of scheduling vectors $\{\underline{s}_{ip}\}$ which satisfies $\underline{s}_{ip}^t \cdot D(Y_{ij}) > \underline{0}$ and one of the following criteria:
   1) minimum α_{ip};
   2) minimum computation latency;

for each different $\underline{s}_{ip}$
$\alpha_i = \underline{s}_{ip}^t \cdot \underline{\lambda}_{ip}$;
$cost = g_i(\alpha_i, \hat{L}_i, \beta_i) + \xi\_\eta$;
$\rho = cost + \Gamma(i + 1)$;
insert $\epsilon = (\xi, \underline{s}^{ip}, \text{cost}), \rho)$ into Ψ;
sort Ψ with key ρ;
until ($\Psi = \phi$) or (mapping is done);

References

1. R. Kuhn, "Transforming algorithms for single stage and VLSI architectures," in *Workshop on Interconnection Networks for Parallel and Distributed Processing*, 1980. IEEE Press, Piscataway, NJ, pp. 11–19.
2. D. Moldovan, "On the analysis and synthesis of VLSI algorithms," *IEEE Trans. on Computers*, vol. C-31, 1982, pp. 1121–1126.
3. D. Moldovan, "On the design of algorithms for VLSI systolic arrays," *Proc. of IEEE*, vol. 71, 1983, pp. 113–119.
4. D. Moldovan and J. Fortes, "Partitioning and mapping algorithms into fixed size systolic arrays," *IEEE Trans. on Computers*, vol. C-35, 1986, pp. 1–12.
5. W. Miranker and A. Winkler, "Spacetime representations of computational structures," *Computing*, vol. 32, 1984, pp. 93–114.
6. P. Quinton, "Automatic synthesis of systolic arrays from uniform recurrent equations," in *Proc. 11th Annu. Symp. Comput. Architecture*, 1984, pp. 208–214.
7. P. Quinton, "Mapping recurrences on parallel architectures," in *Proc. Third Int. Conf. Supercomputing*, International Supercomputing Institute, Inc. Florda, May 1988, pp. 1–8.
8. G. Li and B. Wah, "The design of optimal systolic arrays," *IEEE Trans. on Computers*, vol. C-34, 1985, pp. 66–77.
9. M. Chen and C. Mead, "Concurrent algorithms as space-time recursion equations," in *Proc. USC Workshop VLSI Modern Signal Processing*, Los Angeles, CA, 1982, pp. 31–52.
10. S. Kung, "On supercomputing with systolic/wavefront array processors," *Proc. of IEEE*, vol. 72, 1984, pp. 867–884.
11. H. Kung and W. Lin, "An algebra for VLSI algorithm design," in *Conf. on Elliptic problem solvers*, 1983.
12. P. Cappello and K. Steiglits, "Unifying VLSI array designs with geometric transformation," in *Int'l. Conf. on Parallel Processing*, IEEE Press, Piscataway, NJ, 1983, pp. 448–457.
13. P. Lee and Z. Kedem, "Synthesizing linear array algorithms from nested for loop algorithms," *IEEE Trans. on Computers*, vol. 37, 1988, pp. 1578–1598.
14. M. Lam and J. Mostow, "A transformational model of VLSI systolic design," *Computer*, 1985, pp. 42–52.
15. J. Fortes, K. Fu, and B. Wah, "Systematic approaches to the design of algorithmically specified systolic arrays," in *Int'l. Conf. on Design Automation*, IEEE, 1985, pp. 300–303.
16. S. Rao and T. Kailath, "What is a systolic algorithm," *SPIE*, vol. 614, 1986, pp. 34–48.
17. A. Athavale, J. JáJá, and J. Rowlett, "Compiling programs for systolic arrays," in *VLSI Signal Processing III*, IEEE Press, Piscataway, NJ, 1988, pp. 509–519.
18. S. Kung, *VLSI Array Processors*, Englewood Cliffs, NJ: Prentice Hall, 1987.
19. A. Schrijver, *Theory of Integer and Linear Programming*, New York: Wiley and Sons, 1988.
20. X. Zhong, I. Wong, and S. Rajopadhye, "Bounds on the number of linear allocation functions," in *VLSI Signal Processing IV*, IEEE Press, 1990, pp. 85–94.
21. Y. Wong and J.-M. Delosme, "Optimization of processor count for systolic arrays," Technical Report YALEU/DCS/RR-697, Yale University, 1989.
22. Y. Hwang and Y. Hu, "Parameterized dependence graph and its applications to multi-stage systolic array mapping," in *Proc. ICASSP*, IEEE Press, Piscataway, NJ, 1989, pp. 1029–1032.
23. S. Kung and S. Jean, "A VLSI array compiler system (VACS) for array design," in *VLSI Signal Processing III*, (E.R. Brodersen and H.S. Moscovitz, eds.), IEEE Press, Piscataway, NJ, 1988, pp. 495–508.
24. C. Paige and M. Saunders, "Least squares estimation of discrete linear dynamic systems using orthogonal transformation," *SIAM J. Numerical Analysis*, 1977, pp. 180–193.
25. K. Liu, S. Hsieh, and K. Yao, "Comparisons of parallel SVD in VLSI array processors," in *VLSI Signal Processing IV*, IEEE Press, Piscataway, NJ, 1990, pp. 135–146.