Design Principle of Massively Parallel Distributed-Memory Multiprocessor Architecture

Makoto Amamiya and Tetsuo Kawano
Department of Information Systems
Kyushu University
Kasuga Fukuoka 816 Japan

In this paper, we discuss the design principles of massively parallel distributed-memory multiprocessor architecture and propose the Datarol-II architecture. We present the architecture of the massively parallel Datarol-II machine and show a Datarol-II processor design, including communication protocol and handling mechanisms of remote memory access and remote process/procedure invocation. Last, we show several evaluations of the Datarol-II processor from the viewpoint of effect on thread execution, effect of implicit register loading, and swapping cost by software simulation.

1. Introduction

Distributed-memory multiprocessor architecture is essential in developing massively parallel machines. One of the most important design issues in such a distributed-memory multiprocessor architecture is a latency problem, which is caused by remote procedure invocation and remote memory access. These conditions occur often in massively parallel execution, thus causing processes to suspend their executions until the response to the remote memory accesses or remote process/procedure invocations is received. Therefore, a new processor architecture should be developed to perform efficient context switching among fine-grain concurrent processes in order to solve this latency problem.

We propose a massively parallel distributed-memory multiprocessor architecture, called *Datarol-II*, which is tolerable to the latency caused by remote memory access and remote process invocation. Datarol-II is an advanced version of the *Datarol* architecture [3], which was designed to implement the distributed-memory massively parallel multiprocessor system. The original Datarol machine architecture *Datarol-I* [2, 3], was designed to support ultra-multiprocessing, in which a number of fine-grain processes can be executed in a highly concurrent fashion. The primary idea behind the Datarol-I architecture is to implement an efficient multithread execution control by introducing a continuation-based computation mechanism, by-reference execution control mechanism, and multiple register-sets, all of which enable us to eliminate redundant execution control in conventional dataflow architecture.

The two major architectural differences between Datarol-II and Datarol-I are: (a) the Datarol-II processor is designed to support efficient execution for both fine-grain and coarse-grain parallelisms; and (b) the Datarol-II processor integrates hardware modules for communication control and memory access control, which handle remote memory access and remote process communication in parallel. In order to control concurrent executions among various grain sizes of processes, the execution control mechanism of the Datarol-II processor is designed in such a manner that if the thread is a little bit long and has locality, it can execute a thread exclusively, while, if the

thread has no locality, it can switch the thread or process with very small overhead. Due to this mechanism, Datarol-II has features which are tolerable to remote memory access and remote process/procedure invocation latencies.

Although the Datarol-II processor is designed to execute each thread exclusively, the design is also based on the conjecture that the thread length is not so long in massively parallel computations. The thread length may be less than twenty or thirty instructions. It may be assumed that, in a distributed-memory multiprocessor environment, it will be very hard for programmers to construct such massively parallel programs with long threads. Therefore, implementing high-efficient thread switching is more important than implementing high-speed sequential execution in each thread.

The massively parallel Datarol-II machine is constructed with several thousand Datarol-II processors, in which each has local memory and is connected in a two- or three-dimensional mesh. It is constructed with three modules: Communication Control Module (CM), Memory Module (MM), and Processor Module (PM).

The Datarol-II processor executes a multithread control-flow program called Datarol. The execution of Datarol consists of program-count-based sequential execution and continuation-based thread initiation, i.e., thread synchronization and split-phase operations.

The execution of fine-grain sequential threads by a program-counter-based mechanism is referred to as *short-cycle* execution. In order to reduce the overhead of context switching, an implicit register-loading mechanism is embedded in the execution pipeline. In this mechanism, data is automatically loaded into registers from the local memory in the background of the instruction execution. A two-level hierarchical memory system is implemented in order to reduce memory access latency. A load control mechanism is introduced to improve the memory access locality. This mechanism makes the hierarchical memory system work effectively and prevents the hardware resources from suffering overload.

The Datarol-II processor also performs the continuation-based execution in a circular pipeline known as *long-cycle* execution. Since a token in the circular pipeline triggers a "thread" execution, which contains several instructions, the number of tokens in a circular pipeline of Datarol-II is much smaller than the number of tokens found in a dataflow machine. This decreases the overhead of a synchronization unit.

The Datarol-II processor achieves high-speed context switching and effective multiprocess execution by implementing these hierarchical memory systems and thread-level circular pipeline execution schemes.

2. Motivation and Overview of Datarol Architecture

One of the important issues in designing a parallel computer is to make it scalable, that is, to achieve higher performance when more processors are added to the computer. In order to achieve this goal, a processing element should be designed to exploit program parallelism as much as possible. For this application there are several fundamental problems that must be solved: multiprocessing [4], long memory access latency, remote process/procedure call latency, and waiting time for events synchronization. These problems are strongly related to each other and conventional control-flow multiprocessor architectures attempt to compromise between them. Processor idling time, which is caused during the long-latency remote memory access and remote process call, could be avoided by a split transaction or multiphase operation. However, these operations require a highly efficient synchronization mechanism for switching the processor execution.

Dataflow multiprocessors avoid this contradiction by performing the context switching at instruction level via hardware. They are more tolerable to memory-access, process-call latencies, and synchronization overhead. Furthermore, the parallelism inherent in programs is maximally exploited without the need to be explicitly specified in programs. Dynamic dataflow architecture supports parallel execution in both function instance level and instruction level.

However, the following drawbacks of dataflow architecture have to be eliminated in order to develop massively parallel computers:

- The dataflow model requires a great deal of low-level communications because each instruction has to transfer its context by means of a token. This large communication overhead should be reduced by eliminating redundant dataflow synchronizations.
- The pure fine-grain dataflow execution scheme employs a by-value data access mechanism and this causes explicit data copying. This explicit data copying is redundant and should be eliminated by introducing by-reference data access in which data are accessed only when they are needed [1].
- Matching the memory mechanism requires complex and expensive hardware. This also makes it difficult to use caches.

Furthermore, since dataflow machines are implemented as a circular pipeline, they have the following problems:

- Dataflow machines are ineffective when they execute sequential programs because the circular pipeline is fully loaded only when a program provides sufficient parallelism. When a program is inherently sequential, a lot of bubbles occur in the instruction-execution stream.
- Latency in a thread execution is relatively long. This latency is caused by multi-staging the circular pipeline and queuing process.
- The synchronization unit, or matching-store mechanism, causes pipeline bottleneck. Since each instruction requires synchronization, i.e., one or two packets trigger one instruction, the number of input packets in the synchronization unit is more than that of executing instructions. Moreover, it is difficult to speed up the synchronization unit because it uses memory.

2.1 Concept of Datarol

The Datarol architecture model [3] was proposed to solve the problems stated previously. The idea of Datarol is to remove redundant dataflow by introducing hardware registers and a by-reference data access mechanism. The Datarol program is a multi-thread control-flow program which reflects the dataflow inherent in its source program. Datarol instructions are similar to conventional three-address instructions. Their execution is performed in four phases: instruction fetch, operand fetch, execute, and result write. The difference in this model is that each instruction explicitly specifies its succeeding instructions by a set of continuation points. The set of continuation points exploits parallelism inherent in programs.

The general Datarol instruction is expressed as

l:[*] *op rs1 rs2 rd*-> *D* [1]

[1] [] means this is optional.

where l is a label of this instruction and D is a set of labels of instruction which should be executed after this instruction. D is called a set of continuation points, and an instruction $e \in D$ is called a dependent of l. This expression means that the instruction *op* operates on two operands *rs1*, *rs2*, and stores its result to *rd*. *rs1*, *rs2*, and *rd* denote the respective register names.

Other instructions are switch and link instructions. Switch instructions, $l : sw\ b \rightarrow D_t, D_f$, transfer control to continuations D_t or D_f depending on the Boolean value b. Link and rlink instructions, which are used for function application, are represented as $l : [*]\ link\ r_{s1}\ r_{s2}\ n$ and $l : [*]\ rlink\ r_{s1}\ r_d\ n$. The meaning of these instructions are discussed later.

The tag "*" indicates that the operand of an instruction triggered by an op-token is not assured to exist in the specified register. The tag "*" is optional. When the tag is shown, the arrival of the partner operand is verified. If the partner operand has not been written in the register, the execution of this instruction will suspend until the partner operand arrives.

It should be noted that Datarol instructions are position independent, i.e., any instruction can be placed in any position in an instruction memory because its succeeding instructions are explicitly specified as continuation points.

This Datarol model offers the following features:

- Even if the result value produced by an instruction is referred to by several instructions, it is not necessary to make copies of the result value since the by-reference mechanism permits those instructions to share the result value. The result value is accessed only when it is needed.

- The address of a register, which is shared between consecutive instructions, can be determined and allocated at compile time, and no run time memory allocation is necessary. This significantly reduces the hardware cost needed for matching memory operations.

- The by-reference mechanism ensures that the operand data is written into a register only when the data is produced by an instruction, and is made available for the succeeding instructions. Therefore, if an operand is a dependent of its partner operand, the availability of the operand need not be checked. This reduces the overhead of matching memory operation.

The important advantage of the Datarol model is that the execution of the program is performed in a circular pipeline and execution of a thread is not exclusive, but can be interleaved with executions of other threads. Thus, multiple threads, whether they are in the same process instance or not, can be interleaved, and, therefore, they can be executed without context switching overhead. Circular pipeline stages will be set in full if sufficient parallelism resides in the execution of concurrent processes. This is quite different from other multithreading architectures such as Monsoon [13] and EM-4 [14].

2.2 Optimization of Datarol Architecture

The motivation for designing an optimized Datarol processor, which we call Datarol-II, is to improve the efficiency in the execution of sequential program code while preserving high efficiency in the execution of parallel threads and/or processes.

A previous version of Datarol, called Datarol-I, has the same circular pipeline problems as mentioned in Section 2.1. In order to solve these problems, the instructions

of a sequential thread are arranged into consecutive memory addresses. Datarol-II executes these instructions in serial order until it encounters the termination point of the thread. This serial order execution reduces latency in sequential execution of a thread, and construction of instructions into one thread enables the use of high-speed registers within an execution unit. In addition to this thread construction, the compiler interleaves several threads of the same process into one combined thread without changing the order of instructions in each interleaved thread. Instructions of several threads can be interleaved in such a way that independent instructions are interleaved in order to avoid bubbles in the execution pipeline, i.e., instruction fetch, operand fetch, execution pipe, and result write. This is an easy task for compilers since every instruction is position independent and can be moved anywhere.

Moreover, thread-level continuation-based execution reduces the number of input packets to a synchronization unit, since a packet triggers a "thread"-level execution. This prevents the synchronization unit from being bottlenecked in a circular pipeline.

A Datarol-II machine instruction, which is a slight modification of Datarol-I instruction, is described as:

[*l*:] [!] *op rs1 rs2 rd* [-> (*dest*, *mc*, *join*)]

Here, (*dest*, *mc*, *join*) means the continuation. Notice that the format of continuation is different from that of Datarol-I. The continuation consists of a destination address (*dest*), a matching counter address (*mc*), and the number of join (*join*). In Datarol-II machine instructions, label and continuation are also optional. Tag "!" indicates that the thread execution terminates at this instruction.

The execution of Datarol-II code is triggered by a packet which contains a thread-ID. The thread-ID consists of a process instance name and an entry address of Instruction Memory. The execution proceeds in serial order of the instruction address until it reaches a termination point. This serial execution cycle is called a short-cycle execution. When an instruction specifies a continuation explicitly during the short-cycle execution, the continuation is set aside from the short-cycle execution. Instructions which cause remote memory access or a remote process call will specify a continuation. These instructions do not necessarily terminate the thread execution, but the thread execution continues from the instruction of the next address. When a remote access is completed, the number of join *join* is compared with the matching counter *mc*. If the synchronization operation succeeds, a thread-ID of the continuation thread, which is specified by *dest* is put into a queue. When an execution reaches a termination point, the processor takes another thread-ID from the queue and switches to this thread.

A load control mechanism is introduced in order to prevent hardware resources from being overloaded. When a Datarol-II processor receives a call packet, it allocates a new instance and sends a reply packet to the caller. After sending the reply packet, the processor checks its load level. If the load level is lower than the threshold, the initial thread of the new instance is invoked. Otherwise, invocation of the initial thread is delayed until the load level reaches lower than the threshold. Thus, Datarol-II processor controls its load and limits the number of active threads.

The memory unit of the Datarol-II processor has a two-level hierarchical memory system, operand memory, and register buffer in order to provide high-speed memory access from the execution unit. The thread-based execution and load control mechanism improve the memory access locality, allowing the hierarchical memory system to perform more effectively.

In the Datarol architecture, each instance has its own logical registers. Since the data values of the logical registers are stored in the memory unit, the register

Instruction	*Semantics*
op rs1 rs2 rd	arithmetic and logical instructions
op rs1 # rd	arithmetic and logical instructions with immediate data
start (*dest, mc, join*)	start new thread
brz *rs1 dest*	branch if *rs1* is zero
brnz *rs1 dest*	branch if *rs1* is not zero
call *rs1 rd-> (dest,mc,join)*	activate new instance
link *rs1 rs2 slot*	link *rs2* value to *rs1*
rlink *rs1 rd slot-> (dest,mc,join)*	link *rd* address to *rs1*
receive *rs1 -> (dest,mc,join)*	receive parameter, *rs1*
return *rs1 rs2*	return *rs2* value to *rs1*
rins	release current instance
alloc *rs1 rd-> (dest,mc,join)*	allocate structure memory
free *rs1*	free structure memory
read *rs1 rd-> (dest,mc,join)*	read structure memory
iread *rs1 rd-> (dest,mc,join)*	
ifread *rs1 rd-> (dest,mc,join)*	
write *rs1 rs2*	write structure memory
iwrite *rs1 rs2*	
ifwrite *rs1 rs2*	

Table 1: Datarol-II machine instructions.

values have to be loaded from the memory unit to the registers, which are accessed by the execution unit. These register-loading processes are overhead to the context switching. In order to overcome this overhead, an implicit register-loading mechanism is embedded in the execution pipeline. Due to this mechanism, register values are loaded from the memory unit, resulting in small overhead when the registers are accessed. Thus, the Datarol-II processor performs very fast context switching. This implicit register-loading mechanism is described in Section 4.

3. Datarol-II Program

3.1 Instruction Set

Table 1 shows the Datarol-II machine instruction set.

Remote memory access and remote call instructions specify a continuation, (*dest, mc, join*), and they are operated as split-phase transactions. The continuation consists of a thread-entry address (*dest*), matching counter address (*mc*), and the number of join (*join*). Each instance has its own matching counters, and they are used to control the thread activation. When a reply packet comes back, the number of join, *join* is compared with the corresponding matching counter (*mc*). If join synchronization succeeds, the continuation thread which starts from *dest* is invoked.

A start instruction initiates the new-thread execution whose entry is specified by continuation. Branch instructions, brz and brnz, also initiate a new thread. Termination flags of every branch of instructions are on. When the branch condition is true, the thread execution starting from *dest* is initiated. Otherwise the thread execution continues to the next address of the branch instruction.

The call instruction activates an instance of the function specified by *rs1* and sets the instance name to *rd*. The call operation is a split-phase operation, and a continuation thread is invoked when the *rd* is available. The link instruction sends the value of *rs2* to the *slot*-th parameter of another instance that is specified by *rs1*. The rlink instruction sends its instance name and *rd* address. The return instruction returns *rs2* value to the register of its caller instance. The register name is specified by *rs1*, whose value is passed by rlink. The receive instruction receives a thread parameter, which is passed through register *rs1*. The rins instruction releases the instance.

Arithmetic and logical instructions are the same as the conventional three-address type instructions.

3.2 Sample Program

An example of a program for function fib(n) which calculates Fibonacci numbers. This function is defined as follows:

fib(n) = if n < 2 then n else fib(n-1)+fib(n-2);

Figure 1 shows a Datarol-II program of fib(n).

```
; *** Initial thread ***
Fib:   receive r1 -> (Th1, 0, 0) ; receive n
     ! receive r2 -> (Th7, 1, 2) ; receive return register name
; *** thread 1 ***
Th1:   lti r1 2 t1
     ! brnz t1 Th6
; *** thread 2 ***
Th2:   set Fib t1
       call t1 r3 -> (Th3, 0, 0)     ; get new ins for fib(n-1)
     ! call t1 r4 -> (Th4, 0, 0)     ; get new ins for fib(n-2)
; *** thread 3 ***
Th3:   addi r1 -1 t1                ; link for fib(n-1)
       link r3 t1 1
     ! rlink r3 r5 2 -> (Th5, 2, 2)
; *** thread 4 ***
Th4:   addi r1 -2 t1                ; link for fib(n-2)
       link r4 t1 1
     ! rlink r4 r6 2 -> (Th5, 2, 2)
; *** thread 5 ***
Th5:   add r5 r6 r7                 ; add fib(n-1) and fib(n-2)
     ! start (Th7, 1, 2)
; *** thread 6 ***
Th6:   mov r1 r7                    ; case of (n < 2)
     ! start (Th7, 1, 2)
; *** thread 7 ***
Th7:   return r2 r7                 ; return result
     ! rins                         ; release this instance
```

Figure 1: Sample program fib(n).

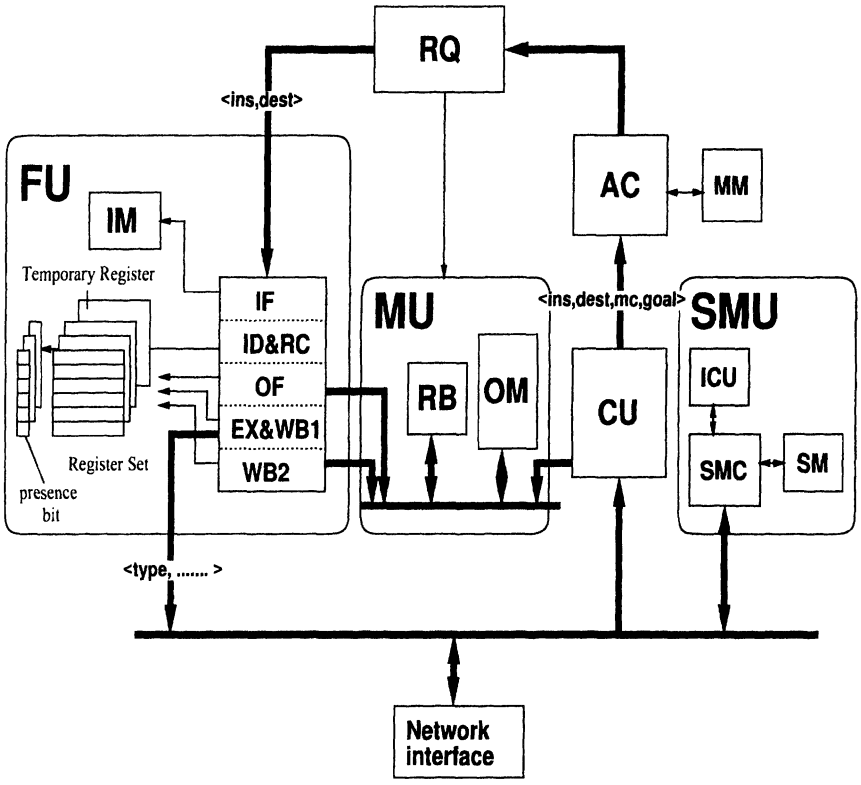


Figure 2: Datarol-II processor element.

4. Architecture of Datarol-II Processor

4.1 Overview of Datarol-II Processor

The Datarol-II machine is a massively parallel distributed-memory multiprocessor system in which thousands of processing elements (PE) are connected through a mesh-connection network. The structure of Datarol-II PE is shown in Figure 2. Datarol-II PE consists of: Function Unit (FU), Communication Unit (CU), Activation Controller (AC), Ready Queue (RQ), Memory Unit (MU), and Structure Memory Unit (SMU).

CU handles input packets sent to the PE. When CU gets a packet, it checks the type of the packet and writes the packet data into MU. Then CU issues the thread-invocation request to AC. The thread-invocation request consists of: thread entry address (*dest*), matching counter address (*mc*), and the number of join (*join*).

AC controls thread activation and has Matching Memory (MM). MM holds the value of the matching counter, which is used for join. AC, when it gets a thread-invocation request, accesses the MM and checks whether the requested thread is ready to run. If the thread is ready to run, AC puts the thread-ID into RQ. The thread-ID consists of instance number and thread entry address (*dest*).

FU has several register sets, dequeues a thread-ID from RQ, and sets an entry address of the thread to the Instruction Pointer (IP). At the same time, FU allocates a free register set and triggers execution of the thread. The thread execution starts from the entry address pointed to by IP, and proceeds in sequential order until it

encounters an instruction whose termination bit is "on." This short-cycle execution mechanism is similar to conventional sequential execution. When the short-cycle execution encounters an instruction whose termination bit is on, FU dequeues the next available thread from RQ and switches the execution context to the new thread.

When a remote memory access instruction or a procedure call instruction is issued during the short-cycle execution, its request packet is delivered to another PE, and the continuation point is set into the destination register. An execution starts from the continuation point when a return packet has arrived. The return packet has an instance number and a register name in which the return value is stored. When a return packet has arrived, CU gets this continuation by reading the destination register, and sends the continuation to AC in order to invoke the continuation thread. Then, CU writes the return value to the destination register.

4.2 Memory Unit

In Datarol architecture, each instance has its own logical register. MU holds the name of this register in Operand Memory (OM). In fine-grain program execution, memory accesses are so frequently issued that the memory access latency has to be kept as short as possible. A high-speed memory called "Register Buffer" (RB) is introduced in order to achieve faster memory access from the FU. RB is accessed by a page number and an offset address which corresponds to the logical register name. Each page holds the register values of each instance. Before FU starts a new thread execution, register values for the new instance have been loaded into RB by the RB preloading mechanism, which is triggered by RQ. This RB preloading mechanism enables FU to read register values very fast during the thread execution.

4.3 Function Unit

FU has several register sets,[2] and each register has a presence bit which indicates whether the register value is valid or not (see Figure 2). FU also has temporary registers which are used to pass the value from an instruction to other instructions in the same thread. When FU starts execution of a new thread, one of the free register sets is chosen and allocated to the new thread. If a register set has already been allocated to that instance, FU reuses it. Otherwise, one of the register sets is selected and all of its presence bits are cleared. Each instance has its own logical registers. In this case, register values are stored in MU and the register values in MU are loaded into FU-registers before registers are accessed. This register loading is performed automatically by hardware when a register is accessed, and the register-loading operation is performed in the FU register in cooperation with the MU.

FU is constructed as a five-stage pipeline consisting of: Instruction Fetch (IF), Instruction Decode and Register Check (ID&RC), Operand Fetch (OF), Execution and Write Back 1 (EX&WB1), and Write Back 2 (WB2). IF stage fetches the IM cell whose address is specified by IP, and passes the fetched instruction to ID&RC stage. If the termination bit of the fetched instruction is on, FU dequeues a new thread-ID from RQ and sets the thread-ID in IP. Otherwise, IP is incremented. ID&RC stage decodes the instruction, then determines the source and destination register addresses and checks the presence bit of the source registers. In the OF stage, source register values are read from the register set or MU according to the result of the RC stage. One MU access is performed within one clock. If two source registers are needed to be read from MU, pipeline interlock occurs. The EX&WB1 stage executes the instruction. When the OF stage reads the source register value from MU, the read

[2]Four register sets is considered in current implementation.

data is stored in the corresponding register. The result data is stored in both the register set and MU, in WB2 stage. Since only one MU access is permitted in single cycle, if MU access occurs at the OF stage and WB2 stage at the same time, pipeline interlock occurs, and access in the OF stage is delayed.

In this pipeline, no explicit load/store instructions are necessary since MU access is performed implicitly in a background operation. Data accesses to MU is performed in parallel with the foreground execution so that the load/store overhead is hidden.

Branch instruction causes the creation of a new thread. Branch address is determined at the EX stage, and the new thread invocation packet is sent to CU. Since the termination bit of the branch instruction is set "on," the thread execution terminates at the branch instruction.

4.4 Structure Memory Unit

The Structure Memory Unit (SMU) supports synchronized memory access by using the I-structure mechanism. SMU also controls the PE load and book keeping of an instance.

Each memory cell has a two-bit tag: presence bit (pr) and pending bit (pe). This tag represents:

00: *Empty*
Neither read nor write access has arrived yet.

01: *Suspended*
Read access has already arrived but write access has not arrived yet.

10: *Full*
Write access has already arrived.

When a synchronized read access is issued to an *Empty* cell, its tag is changed to *Suspended* and a read access packet is stored into the cell. The read access packet contains a register name to which the result of the memory access is written. Table 2 shows memory access protocols. A load control mechanism is introduced in SMU. When SMU gets a call packet, it allocates a new instance and sends a reply packet which contains a new instance number. Then, SMU invokes an initial thread of the new instance. If the PE load is too heavy, SMU delays the invocation of the initial thread and stores the delayed threads in SM. When PE load gets light, SMU invokes the delayed initial threads.

4.5 Function Call Mechanism

The function call in the Datarol-II machine is performed as follows:

1. In a caller instance, say ins_p, a call instruction is executed in FU and FU issues a call packet to network.
2. Since network has a load distribution mechanism, the call packet is delivered to the lightest loaded processor, say PE_c.
3. In the callee processor, PE_c, the call packet is received at SMU. In the SMU, a new instance is allocated, and its name, say ins_c, is returned to the caller instance.
4. The callee processor initiates an initial thread of ins_c. In the initial thread, receive instructions are executed. A receive instruction sets a thread entry address (*dest*) and its synchronization condition (*mc*, *join*).

Protocol	*Before*		*After*		*Action*
	pr	*pe*	*pr*	*pe*	
write	—	—	1	0	Write data
iwrite	0	0	1	0	Write data
	0	1	1	0	Resume read access and write data
	1	0	—	—	Error
ifwrite	0	0	1	0	Write data
	0	1	0	0	Resume read access
	1	0	—	—	Error
read	0	—	0	—	Return empty value
	1	0	1	0	Read data
iread	0	0	0	1	Pending read access
	0	1	—	—	Error
	1	0	1	0	Read data
ifread	0	0	0	1	Pending read access
	0	1	—	—	Error
	1	0	0	0	Read data

Table 2: Structure memory access protocols.

5. When the caller processor gets a reply packet, the processor initiates a continuation thread, which contains link and rlink operations. The link instruction sends a *rs2* to ins_c (*rs1*) as a *slot*-th parameter. The rlink instruction stores in a register a continuation thread which will be initiated when return data arrives, and then sends the register name to which the return data is written.

6. When the callee processor receives a link packet, the *slot*-th register value of ins_c is read, and its values are sent to AC to request a thread initiation. Then, CU writes the link data to MU.

7. When AC gets a thread initiation request, (*ins*, *dest*, *mc*, *join*), if *join* is not zero, AC performs a join operation. AC increments the matching counter which is specified by *ins* and *mc*. If the value of the counter reaches *join*, the join is established and a thread-ID, which consists of *ins* and *dest* is put into RQ. If *join* is zero, AC puts the thread-ID into RQ without a synchronizing operation.

8. The callee thread is then initiated by link packets. When the callee thread meets a return instruction, a return packet is sent to ins_p. When the execution of ins_c reaches a rins instruction, that instance is released.

9. The caller instance gets a return packet and initiates a continuation thread which corresponds to the result data.

5. Evaluation

In this section, several evaluations of Datarol-II architecture are shown by software simulation. A software simulator of Datarol-II machine has been developed for this purpose. The simulation of the Datarol-II machine system occurs at the register transfer level. Two benchmark programs are used: fib(n) and queen(n). The program

Type	*(A) No. of Instructions*	*(B) No. of AC Input Packets*	*(C)(B)//(A)*	*(D) No. of Matching*
Dataflow	17752	24656	1.39	6904
Datarol-I	17752	22706	1.28	5917
Datarol-II	17752	9863	0.56	2959

Table 3: Load balance of FU and AC.

fib(n) calculates Fibonacci numbers as described in Section 3.2. Since this program consists of short threads, it requires fast context switching for efficient execution. The program queen(n) finds all solutions for the N-queen problem. This program creates a large number of threads, and a multitude of instances running concurrently because many instances are executed in random, and their memory accesses are less local.

5.1 Threading Effect

Dataflow-based processors, which are implemented as a circular pipeline, experience bottleneck in this pipeline due to the synchronization mechanism by AC in a Datarol-II processor as well as matching-store mechanism in dataflow. Suppose N instructions execute a thread, and the ratio k of the instructions require the operand check, then the synchronization mechanism has to handle $(1+k)N$ packets. Since the synchronization mechanism is, in general, implemented by using random access memory, speed-up of the synchronization is difficult. Thus, the packet handling speed is slower than that of the instruction execution.

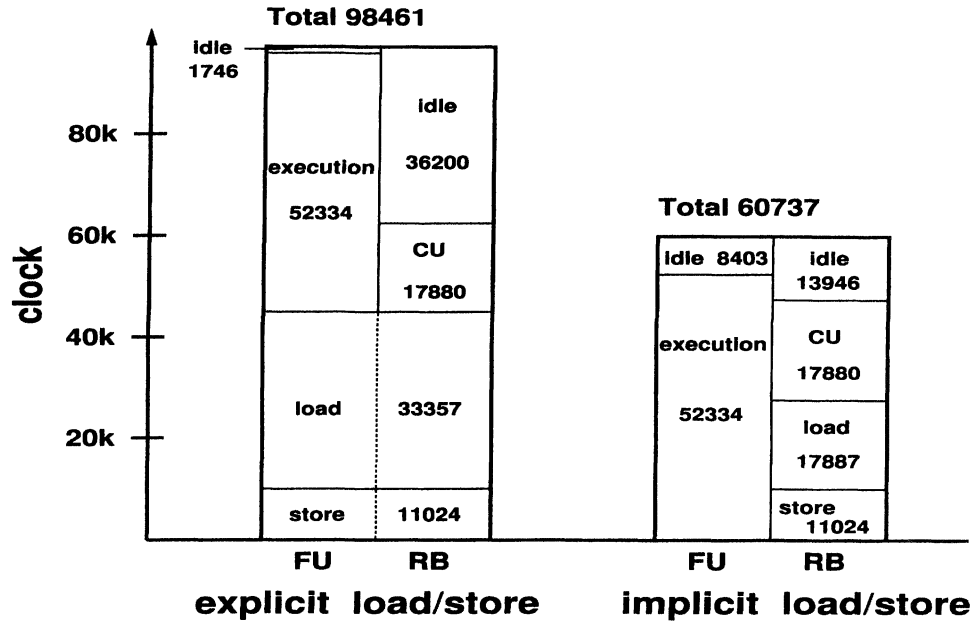
Table 3 shows a balance between FU operations and AC matching operations for the program fib(15). In this table, column (A) shows the total number of executed instructions, and column (B) shows the total number of input packets to the matching-store in conventional dataflow machines and AC in the Datarol-I machine. The ratio (B)/(A) is important since it means load balance between the execution unit and synchronization unit. As the ratio (B)/(A) is more than one in these machines, the speed of matching-store limits the pipeline and consequently the speed of these machines. On the other hand, as the ratio (B)/(A) is 0.56 and less than one in Datarol-II, Datarol-II can use the execution unit at higher speeds and achieves higher performance.

5.2 Effect of Implicit Register Loading

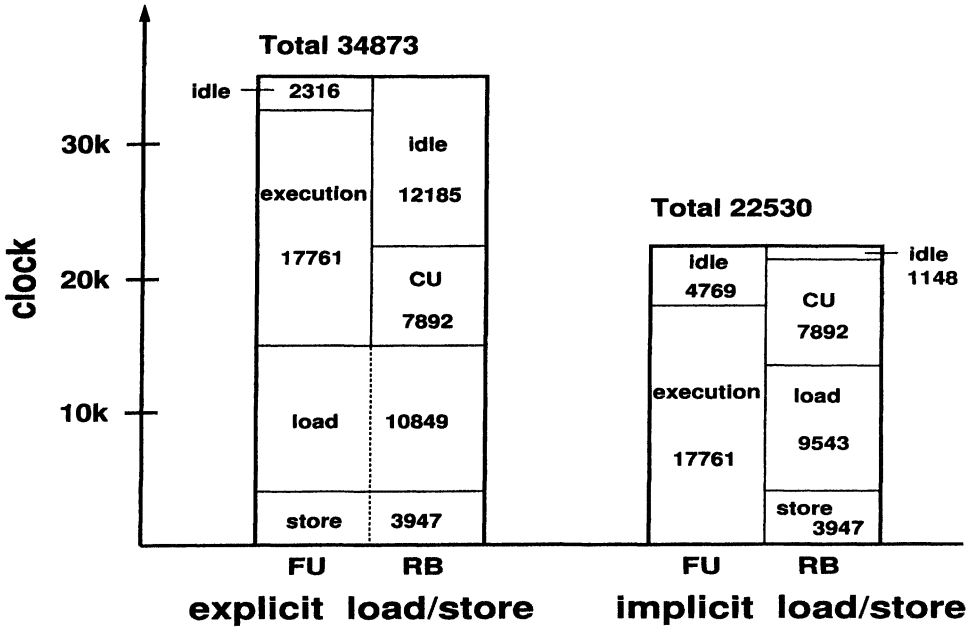
In the Datarol-II processor, register values are loaded from MU during the thread execution automatically. The register loading is implicitly performed without explicit load instruction. The register-loading mechanism is embedded in the execution pipeline in order to hide register loading overhead.

Figure 3 shows the effect of implicit register loading in comparison to explicit loading by load instructions embedded in Datarol codes. In the explicit loading case, load instructions are inserted in the Datarol code. In this program code, every thread execution loads register values from MU explicitly when the thread execution initiates. On the other hand, in the implicit loading case, since the register-loading operations are performed in background of the thread execution, the total thread execution is achieved in a significantly shorter amount of time.

Figure 4 shows the FU and RB utilization ratio for the execution of queen(6). In this case, FU and RB keep a high utilization ratio during the entire period of execution. This data shows the Datarol-II machine makes good use of the RB bandwidth.



(a) fib(15)



(b) queen(6)

Figure 3: Explicit load vs. implicit load.

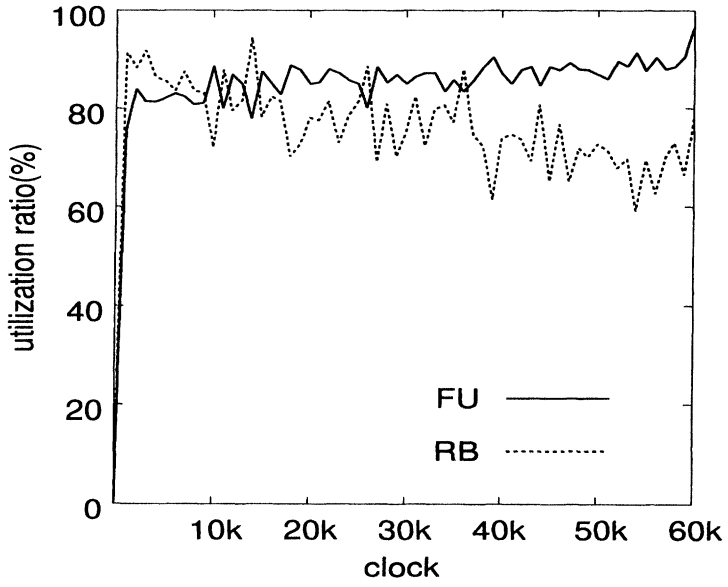


Figure 4: Utilization ratio of FU and RB.

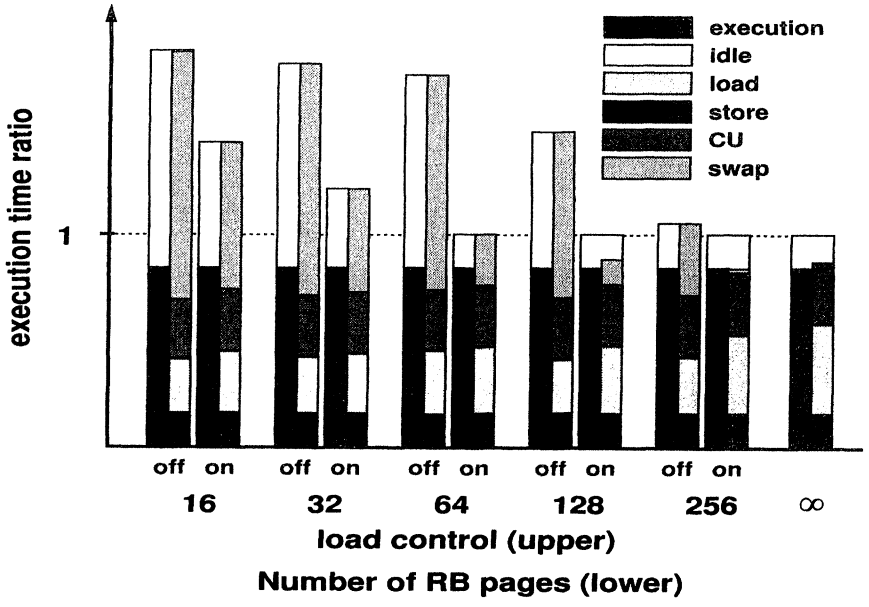


Figure 5: Effect of the number of RB pages.

5.3 Swapping Cost

In the evaluation of Sections 5.1 and 5.2, sufficient RB pages are assumed to exist and no data transfer between the RB and OM. In this section, we evaluate the data transfer cost between RB and OM, in the case of a limited number of RB pages. This cost is referred to as "swapping cost."

Little locality of data access will be expected in dataflow-based execution. When a dataflow machine uses a hierarchical memory system, like a cache, it causes a great increase in execution time because of thrashing.

A two-level hierarchical memory system, which consists of RB and OM, is introduced in Datarol-II. A load control mechanism is also introduced in order to avoid the thrashing between RB and OM. Figure 5 shows the execution times of queen(6) in the case of a various number of RB pages and whether the load control is on or off. When the load control is off, the execution time increases when there is a decrease in RB pages. When the number of RB pages is less than 128, the swapping occurs frequently and execution time increases greatly. When the load control is on, swapping is suppressed. In this case, if the number of RB is more than 64, the swapping time is hidden in the execution time. Even if the RB would require high-speed random access memory, 64 pages will be an acceptable size.

6. Related Work

Datarol-I was designed to achieve efficient context switching and high throughput for highly parallel programs. Datarol-II performs efficient execution both in intra-thread serial execution and in inter-thread concurrent execution. The intra-thread serial execution has a high-speed performance due to a RISC-type execution mechanism in the short-cycle. The inter-thread concurrent execution also achieves high throughput due to the ultra-multiprocessing mechanism inherited from the Datarol-I architecture.

Similar architectures are Monsoon [13], *T [12], EM-4 [14], J-Machine [6], and Epsilon-2 [8].

Monsoon is also designed to support multithread execution by introducing a synchronizing join operator. Monsoon uses ordinary random access memory for join counter management and performs its join operation as one of its general instructions. The join operation in Datarol-II machines is done in the Activation Controller (AC), which uses a special synchronization bit table called Matching Memory (MM), and AC performs the synchronization control in parallel with FU operations in a circular pipeline.

*T has a data processor (dP) and synchronization co-processor (sP), which are connected with a thread queue. *T is very similar to Datarol-II in this way. However, *T has the same join mechanism as Monsoon. Although *T performs join operations in sP, it still uses ordinary random access memory for the join counter, and this slows down the join operation since one join operation takes a few instruction steps, i.e., load, increment, test and store. Even though the join operation is done in sP, in parallel with dP, it should be done in one cycle in order to create a balance with high-speed thread switching in dP. Furthermore, *T requires longer threads for efficient execution than that of Datarol-II, since *T uses a conventional processor for dP.

Activation Controller in Datarol-II, on the other hand, is performed in one cycle since the test-and-set operation is done only for a one bit field in MM, and this mechanism is easily implemented by hardware logic.

EM-4 implements similar synchronization operations by a direct matching mechanism, which uses a direct address assignment of matching-store to the operand address

of instruction which requires synchronization between operands. This causes the problem of inefficient memory usage, since memory cells cannot be assigned a continuous address and also one memory cell is used exclusively by only one instruction. AC-memory bit address assignment in Datarol, on the other hand, is independent of the instruction address and operand address, and, therefore, bit address assignment is continuous and the same bit is shared by several instructions.

EM-4 uses high-speed registers only within the strongly connected block which corresponds to the thread in Datarol-II. Since the execution unit is directly connected to the synchronization unit, it causes a pipeline bubble when it fails in operand matching. Moreover, EM-4 has no hierarchical memory system and it requires a large amount of high-speed memory. Datarol-II is more tolerable to the pipeline bubble, since Datarol-II uses registers for both intra-thread and inter-thread within an instance, and RQ works as a buffer between the execution unit and the synchronization unit. Furthermore, Datarol-II offers more flexible memory usage since the hierarchical memory system and the on-demand implicit data swapping control are incorporated into Datarol-II.

J-Machine has a feature for handling messages and low overhead remote procedure calls. However, J-Machine has no special features for efficient synchronization operation like *T, EM-4, and Datarol.

7. Conclusions

A design principle of massively parallel distributed-memory architecture was discussed, and the Datarol-II architecture, which meets this criteria, was proposed and implemented. Datarol-II inherits the highly concurrent execution mechanism from the Datarol-I architecture, which offers high throughput execution of multiple threads due to its continuation-based ultra-multiprocessing mechanism. In addition to this efficient multithread execution, Datarol-II also performs high-speed single-thread execution by introducing the short-cycle RISC-type execution mechanism.

The Datarol-II processor is an optimized version of Datarol-I. The key issues of optimization are: (1) employment of an efficient thread execution mechanism which does not specify the continuation point explicitly, and (2) load control mechanism which improves locality of memory access and enables the configuration of hierarchical memory.

The Datarol-II processor consists of: Function Unit, Memory Unit, Activation Controller, Communication Unit, Ready Queue, and Structure Memory Unit. The Function Unit performs program-counter-based short-cycle execution within a thread as well as continuation-based long-cycle execution among threads. The Memory Unit has a two-level hierarchical memory system, RB and OM, and it provides high-speed memory access. The Structure Memory Unit provides synchronized memory access by an I-structure-based mechanism. This construction of the processor is highly suitable for massively parallel execution since this processor provides high throughput execution of multiple threads due to its continuation-based execution mechanism.

We have evaluated the Datarol-II processor by software simulation for simple program samples. The simulation results showed that (1) thread level continuation-based execution, which reduces the number of input packets to a synchronization unit, prevents the pipeline in the synchronization unit from bottlenecking; (2) the implicit register-loading mechanism hides the memory access overhead, which is caused by context switching; and (3) the load control mechanism, which reduces the traffic between RB and OM, makes the hierarchical memory system work more effectively.

Currently, we are pursuing further simulation experiments using various applications in order to evaluate the performance of the Datarol-II processor in greater detail.

References

[1] M. Amamiya, "Dataflow Computing and Parallel Reduction Machine," *Future Generation Computer Systems*, Vol. 4, No. 1, 1988, pp. 53–67.

[2] M. Amamiya and R. Taniguchi, "Datarol: A Massively Parallel Architecture for Functional Language," *Proc. IEEE 2nd Symp. Parallel and Distributed Processing*, IEEE CS Press, Los Alamitos, Calif., 1990, pp. 726–735.

[3] M. Amamiya, "An Ultra-Multiprocessing Architecture for Functional Languages," in *Advanced Topics in Data-Flow Computing*, J.-L. Gaudiot and L. Bic, Eds. Prentice-Hall, Englewood Cliffs, N.J., 1991.

[4] Arvind and R. Iannucci, "Two Fundamental Issues in Multiprocessing," *Proc. DFVLR Conf. on Parallel Processing in Science and Engineering*, Bon-Bad Godesberg, Germany, 1987.

[5] D.E. Culler et al., "Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Control Threaded Abstract Machine," *Proc. 4th ASPLOS*, ACM Press, New York, N.Y., 1991, pp. 164–175.

[6] W.J. Dally et al., "The J-Machine: A Fine-grain Concurrent Computer," *Proc. 11th IFIP*, 1989, pp. 1147–1153.

[7] T. von Eicken et al., "Active Messages: a Mechanism for Integrated Communication and Computation," *Proc. 19th Ann. Int'l Symp. Computer Architecture*, ACM Press, New York, N.Y., 1992, pp. 256–266.

[8] V.G. Grafe and J.E. Hoch, "The Epsilon-2 Multi-Processor System," *J. Parallel and Distributed Computing*, Vol. 10, 1990, pp. 309–318.

[9] S.W. Keckler and W.J. Dally, "Processor Coupling: Integrating Compile Time and Runtime Scheduling for Parallelism," *Proc. 19th Ann. Int'l Symp. Computer Architecture*, ACM Press, New York, N.Y., 1992, pp. 202–213.

[10] S. Kusakabe et al., "Parallelism Control and Storage Management in Datarol PE," *Proc. IFIP World Congress*, Vol. 1, 1992, pp. 535–541.

[11] R.S. Nikhil and Arvind, "Can dataflow subsume von Neumann computing?," *Proc. 16th Ann. Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1989, pp. 262–272.

[12] R.S. Nikhil, G.M. Papadopoulos, and Arvind, "*T: A Multithread Massively Parallel Architecture," *Proc. 19th Ann. Int'l Symp. Computer Architecture*, ACM Press, New York, N.Y., 1992, pp. 156–167.

[13] G.M. Papadopoulos and D.E. Culler, "Monsoon: an Explicit Token-Store Architecture," *Proc. 17th Ann. Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1990, pp. 82–91.

[14] S. Sakai et al., "An Architecture of a Dataflow Single Chip Processor," *Proc. 16th Ann. Int'l Symp. Computer Architecture*, IEEE CS Press, Los Alamitos, Calif., 1989, pp. 46–53.

[15] T. Tachibana, R. Taniguchi, and M. Amamiya, "Compiling Method of Functional Programming Language *Valid* by Dataflow Analysis—Extraction of Datarol Program," *J. Information Processing* (in Japanese), Vol. 30, No. 12, 1989, pp. 1628–1638.

[16] R. Taniguchi, T. Kawano, and M. Amamiya, "A Distributed-Memory Multi-Thread Multiprocessor Architecture for Computer Vision and Image Processing: Optimized Version of AMP," *Proc. 26th ICSS*, Vol. 1, 1993, pp. 151–160.