1

Introduction*

The word "embedded" literally means "within," so embedded systems are information processing systems *within* (embedded into) other systems. In other words, an embedded system is a system that uses a computer to perform a specific task but are neither used nor perceived as a computer. Essentially, an embedded system is virtually any computing system other than a desktop or a server computer. Embedded systems have links to physical components/systems, which distinguishes them from traditional desktop and server computing [1]. Embedded systems possess a large number of common characteristics such as real-time constraints, dependability, and power/energy efficiency.

Embedded systems can be classified based on functionality as transformational, reactive, or interactive [2]. *Transformational* embedded systems take input data and transform the data into output data. *Reactive* embedded systems react continuously to their environment at the speed of the environment, whereas *interactive* embedded systems react with their environment at their own speed.

Embedded systems can be classified based on orchestration/architecture as *single-unit* or *multi-unit/distributed* and/or *parallel* embedded systems. Single-unit embedded systems refer to embedded systems that possess computational capabilities and interact with the physical world via sensors and actuators, but are fabricated on a single chip and are enclosed in a single package. Multi-unit embedded systems, also referred to as *distributed embedded systems*, consist of a large number of physically distributed nodes that possess computation capabilities, interact with the physical world via a set of sensors and actuators, and communicate with each other via a wired or wireless network. An emerging trend is to connect these distributed embedded systems via a wireless network instead of a bulky, wired networking infrastructure. Cyber-physical systems (CPSs) and embedded wireless sensor networks (EWSNs) are typical examples of distributed embedded systems.

To meet the continuously increasing performance demands of many application domains (e.g., medical imaging, mobile signal processing), many embedded systems leverage multicore (manycore) architectures. Since processing is done in parallel in multicore-based embedded

*A portion of this chapter appeared in: Arslan Munir, Sanjay Ranka, and Ann Gordon-Ross, Modeling of Scalable Embedded Systems, CH 29 in *Scalable Computing and Communications: Theory and Practice*, Samee U. Khan, Lizhe Wang, and Albert Y. Zomaya (Eds.), ISBN: 978-1-1181-6265-1, John Wiley & Sons, pp. 629–657, January 2013.

Modeling and Optimization of Parallel and Distributed Embedded Systems, First Edition.
Arslan Munir, Ann Gordon-Ross and Sanjay Ranka.

systems, these embedded systems are often referred to as *parallel embedded systems*. Often parallel embedded systems are networked together to form *parallel and distributed embedded systems*. The burgeoning multicore revolution in computing industry is the main thrust behind the emergence of these parallel and distributed embedded systems. The multicore innovation in computer industry has induced parallel computing in embedded domain, which was previously used predominantly in supercomputing domain only. Parallel and distributed embedded systems have proliferated in a wide variety of application domains. These application domains include military, health, ecology, environment, industrial automation, transportation, control, and medical, to name a few.

Embedded systems often require specific quantifiable design goals, such as real-time constraints, performance, power/energy consumption, and cost. In order to design embedded computer systems to meet these quantifiable goals, designers realize that no one system is best for all embedded applications. Different requirements lead to different trade-offs between performance and power, hardware and software, and so on. Different implementations need to be created to meet the requirements of a family of applications. Solutions should be programmable enough to make the embedded design flexible and long-lived, but not provide unnecessary flexibility that would impede meeting the application requirements [3]. To meet various design goals, embedded systems design requires optimization of hardware and software. In particular, performance and power optimizations are required for many embedded applications. Embedded systems leverage various techniques to optimize and manage power in embedded systems. These techniques include, but are not limited to, power-aware high-level language compilers, dynamic power management policies, memory management schemes, and bus encoding techniques [4].

Embedded systems design is traditionally power-centric, but there has been a recent shift toward high-performance embedded computing (HPEC) due to the proliferation of compute-intensive embedded applications. For example, the signal processing for a 3G mobile handset requires 35–40 giga operations per second (GOPS) for a 14.4 Mbps channel and 210–290 GOPS for a 100 Mbps orthogonal frequency-division multiplexing (OFDM) channel. Considering the limited energy of a mobile handset battery, these performance levels must be met with a power dissipation budget of approximately 1 W, which translates to a performance efficiency of 25 mW/GOPS or 25 pJ/operation for the 3G receiver and 3–5 pJ/operation for the OFDM receiver [5, 6]. These demanding and competing power–performance requirements make modern embedded systems design challenging.

The high-performance energy-efficient embedded computing (HPEEC) domain addresses the unique design challenges of high-performance and low-power/energy embedded computing. The HPEEC domain can be termed as high-performance *green* computing; however, green may refer to a bigger notion of environmental impact. The high-performance and low-power design challenges are competing because high performance typically requires maximum processor speeds with enormous energy consumption, whereas low power typically requires nominal or low processor speeds that offer modest performance. HPEEC requires thorough consideration of the thermal design power (TDP) and processor frequency relationship while selecting an appropriate processor for an embedded application. For example, decreasing the processor frequency by a fraction of the maximum operating frequency (e.g., reducing from 3.16 to 3.0 GHz) can cause 10% performance degradation but can decrease power consumption by 30–40% [7]. To meet HPEEC power–performance requirements, embedded systems design has transitioned from a single-core paradigm to a multicore paradigm that favors multiple

low-power cores running at low processor speeds rather than a single high-speed power-hungry core. The multicore embedded systems have integrated HPEEC and parallel computing into high-performance energy-efficient parallel embedded computing (HPEPEC) domain.

HPEPEC domain encompasses both single-unit and multi-unit distributed embedded systems. Chip multiprocessors (CMPs) provide a scalable HPEPEC platform as performance can be increased by increasing the number of cores as long as the increase in the number of cores offsets the clock frequency reduction by maintaining a given performance level with less power [8]. Multiprocessor systems-on-chip (MPSoCs), which are multiprocessor version of systems-on-chip (SoCs), are another alternative HPEPEC platform, which provide an unlimited combination of homogeneous and heterogeneous cores. Though both CMPs and MPSoCs are HPEPEC platforms, MPSoCs differ from CMPs in that MPSoCs provide custom architectures (including specialized instruction sets) tailored for meeting peculiar requirements of specific embedded applications (e.g., real-time, throughput-intensive, reliability-constrained). Both CMPs and MPSoCs rely on HPEPEC hardware/software techniques for delivering high performance per watt and meeting diverse application requirements.

Although HPEPEC enables more sophisticated embedded applications and meets better competing performance and energy requirements, HPEPEC further complicates embedded systems design. Embedded systems design is highly challenging as the interaction with the environment, timing of the operations, communication network, and peculiar application requirements that may need integration of on-chip hardwired and/or reconfigurable units have to be considered. Both hardware and software designs of embedded systems are complex, for example, current automotive embedded systems contain more than 100 million lines of code. Multicore—a crucial enabler for HPEPEC—while providing performance and energy benefits further aggravates design challenges of embedded systems. While industry focuses on increasing the number of on-chip processor cores to meet customer performance demands, this increasing number of cores has led to an exponential increase in design complexity. Embedded system designers face the new challenge of optimal layout of these processor cores along with the memory subsystem (caches and main memory) to satisfy power, area, and stringent real-time constraints. The short *time-to-market* (time from product conception to market release) of embedded systems further exacerbates design challenges.

Modeling of embedded systems helps the designers to cope with increasingly complex design challenges. Modeling of embedded systems helps in reducing the time-to-market by enabling fast application-to-device mapping, early proof of concept (POC), and system verification. Original equipment manufacturers (OEMs) increasingly adopt model-based design methodologies for improving the quality and reuse of hardware/software components. A model-based design allows development of control and dataflow applications in a graphical language familiar to control engineers and domain experts. Moreover, a model-based design enables components' definition at a higher level of abstraction that permits modularity and reusability. Furthermore, a model-based design allows verification of system behavior using simulation. However, different models provide different levels of abstraction for the system under design (SUD). To ensure timely completion of embedded systems design with sufficient confidence in the product's market release, design engineers have to make trade-offs between the abstraction level of a model and the accuracy a model can attain.

The remainder of this chapter is organized as follows. Section 1.1 elaborates on several embedded system application domains. Various characteristics of embedded system applications are discussed in Section 1.2. Section 1.3 discusses the main components of

a typical embedded system's hardware and software. Section 1.4 elaborates modeling, modeling objectives, and various modeling paradigms. Section 1.5 provides an overview of optimization in embedded systems. Finally, Section 1.6 concludes this chapter.

1.1 Embedded Systems Applications

Embedded systems have applications in virtually all computing domains (except desktop computing) such as automobiles, medical, industry automation, home appliances (e.g., microwave ovens, toasters, washers/dryers), offices (e.g., printers, scanners), aircraft, space, military, and consumer electronics (e.g., smartphones, feature phones, portable media players, video games). In this section, we discuss some of these applications in detail.

1.1.1 Cyber-Physical Systems

A CPS is an emerging application domain of multi-unit/networked embedded systems. The CPS term emphasizes the link to physical quantities such as time, energy, and space. Although CPSs are embedded systems, the new terminology has been proposed by researchers to distinguish CPSs from simple microcontroller-based embedded systems. CPSs have become a hot topic for research in recent years and the difference between CPSs and embedded systems is not elucidated in many texts. We aim at making the distinction clear: *every CPS is an embedded system but not every embedded system is a CPS*. For instance, different distributed control functions in automobiles are examples of CPSs, and hence embedded systems (automotive CPSs are often also referred as automotive embedded systems). However, most of consumer electronic devices, such as mobile phones, personal digital assistants (PDAs), digital cameras, printers, and smart cards, are embedded systems but not CPSs.

CPSs enable monitoring and control of physical systems via a network (e.g., Internet, intranet, or wireless cellular network). CPSs are hybrid systems that include both continuous and discrete dynamics. Modeling of CPSs must use hybrid models that represent both continuous and discrete dynamics and should incorporate timing and concurrency. Communication between single-unit embedded devices/subsystems performing distributed computation in CPSs presents challenges due to uncertainty in temporal behavior (e.g., jitter in latency), message ordering because of dynamic routing of data, and data error rates. CPS applications include process control, networked building control systems (e.g., lighting, air-conditioning), telemedicine, and smart structures.

1.1.2 Space

Embedded systems are prevalent in space and aerospace systems where safety, reliability, and real-time requirements are critical. For example, a fly-by-wire aircraft with a 50-year production cycle requires an aircraft manufacturer to purchase, all at once, a 50-year supply of the microprocessors that will run the embedded software. All of these microprocessors must be manufactured from the same production line from the same mask to ensure that the validated real-time performance is maintained. Consequently, aerospace systems are unable to benefit from the technological improvements in this 50-year period without repeating

the software validation and certification, which is very expensive. Hence, for aerospace applications, efficiency is of less relative importance as compared to predictability and safety, which is difficult to ensure without freezing the design at the physical level [9].

Embedded systems are used in satellites and space shuttles. For example, small-scale satellites in low earth orbit (LEO) use embedded systems for earth imaging and detection of ionospheric phenomenon that influences radio wave propagation (the ionosphere is produced by the ionization of atmospheric neutrals by ultraviolet radiation from the Sun and resides above the surface of earth stretching from a height of 50 km to more than 1000 km) [10]. Embedded systems enable unmanned and autonomous satellite platforms for space missions. For example, the dependable multiprocessor (DM), commissioned by NASA's New Millennium Program for future space missions, is an embedded system leveraging multicore processors and field-programmable gate array (FPGA)-based coprocessors [11].

1.1.3 Medical

Embedded systems are widely used in medical equipment where a product life cycle of 7 years is a prerequisite (i.e., processors used in medical equipment must be available for at least 7 years of operation) [12]. High-performance embedded systems are used in medical imaging devices (e.g., magnetic resonance imaging (MRI), computed tomography (CT), digital X-ray, and ultrasound) to provide high-quality images, which can accurately diagnose and determine treatment for a variety of patients' conditions. Filtering noisy input data and producing high-resolution images at high data processing rates require tremendous computing power (e.g., video imaging applications often require data processing at rates of 30 images/s or more). Using multicore embedded systems helps in efficient processing of these high-resolution medical images, whereas hardware coprocessors such as graphics processing units (GPUs) and FPGAs take parallel computing on these images to the next step. These coprocessors offload and accelerate some of the processing tasks that the processor would normally handle.

Some medical applications require real-time imaging to provide feedback while performing procedures such as positioning a stent or other devices inside a patient's heart. Some imaging applications require multiple modalities (e.g., CT, MRI, ultrasound) to provide optimal images as no single technique is optimal for imaging all types of tissues. In these applications, embedded systems combine images from each modality into a composite image that provides more information than the images from each individual modality separately [13].

Embedded systems are used in cardiovascular monitoring applications to treat high-risk patients while undergoing major surgery or cardiology procedures. Hemodynamic monitors in cardiovascular embedded systems measure a range of data related to a patient's heart and blood circulation on a beat-by-beat basis. These systems monitor the arterial blood pressure waveform along with the corresponding beat durations, which determines the amount of blood pumped out with each individual beat and heart rate.

Embedded systems have made telemedicine a reality enabling remote patient examination. Telemedicine virtually eliminates the distance between remote patients and urban practitioners by using real-time audio and video with one camera at the patient's location and another with the treatment specialist. Telemedicine requires standard-based platforms capable of integrating a myriad of medical devices via a standard input/output (I/O) connection such as Ethernet, Universal Serial Bus (USB), or video port. Vendors (e.g., Intel) supply embedded equipment for telemedicine that support real-time transmission of high-definition audio and video while

simultaneously gathering data from the attached peripheral devices (e.g., heart monitor, CT scanner, thermometer, X-ray, and ultrasound machine) [14].

1.1.4 Automotive

Embedded systems are heavily used in the automotive industry for measurement and control. Since these embedded systems are commonly known as electronic control units (ECUs), we use the term ECU to refer to any automotive embedded system. A state-of-the-art luxury car contains more than 70 ECUs for safety and comfort functions [15]. Typically, ECUs in automotive systems communicate with each other over controller area network (CAN) buses.

ECUs in automotives are partitioned into two major categories: (1) ECUs for controlling mechanical parts and (2) ECUs for handling information systems and/ entertainment. The first category includes chassis control, automotive body control (interior air-conditioning, dashboard, power windows, etc.), power-train control (engine, transmission, emissions, etc.), and active safety control. The second category includes office computing, information management, navigation, external communication, and entertainment [16]. Each category has unique requirements for computation speed, scalability, and reliability.

ECUs responsible for power-train control, motor management, gear control, suspension control, airbag release, and antilocking brakes implement closed-loop control functions as well as reactive functions with hard real-time constraints and communicate over a class C CAN-bus (typically 1 Mbps). ECUs responsible for power-train have stringent real-time and computing power constraints requiring an activation period of a few milliseconds at high engine speeds. Typical power-train ECUs use 32-bit microcontrollers running at a few hundreds of megahertzs, whereas the remainder of the real-time subsystems use 16-bit microcontrollers running at less than 1 MHz. Multicore ECUs are envisioned as the next-generation solution for automotive applications with intense computing and high reliability requirements.

The body electronics ECUs, which serve the comfort functions (e.g., air-conditioning, power window, seat control, and parking assistance), are mainly reactive systems with only a few closed-loop control functions and have soft real-time requirements. For example, driver and passengers issue supervisory commands to initiate power window movement by pressing the appropriate buttons. These buttons are connected to a microprocessor that translates the voltages corresponding to button up and down actions into messages that traverse over a network to the power window controller. The body electronics ECUs communicate via a class B CAN-bus typically operating at 100 kbps.

ECUs responsible for entertainment and office applications (e.g., video, sound, phone, and global positioning system (GPS)) are software-intensive with millions of lines of code and communicate via an optical data bus typically operating at 100 Mbps, which is the fastest bus in automotive applications. Various CAN buses and optical buses that connect different types of ECUs in automotive applications are in turn connected through a central gateway, which enables the communication of all ECUs.

For high-speed communication of large volumes of data traffic generated by 360° sensors positioned around the vehicles, the automotive industry is moving toward the FlexRay communication standard (a consortium that includes BMW, DaimlerChrysler, General Motors, Freescale, NXP, Bosch, and Volkswagen/Audi as core members) [16]. The current

CAN standard limits the communication speed to 500 kbps and imposes a protocol overhead of more than 40%, whereas FlexRay defines the communication speed at 10 Mbps with comparatively less overhead than the CAN. FlexRay offers enhanced reliability using a dual-channel bus specification. The dual-channel bus configuration can exploit physical redundancy and replicate safety-critical messages on both bus channels. The FlexRay standard affords better scalability for distributed ECUs as compared to CAN because of a time-triggered communication channel specification such that each node only needs to know the time slots for its outgoing and incoming communications. To promote high scalability, the node-assigned time slot schedule is distributed across the ECU nodes where each node stores its own time slot schedule in a local scheduling table.

1.2 Characteristics of Embedded Systems Applications

Different embedded applications have different characteristics. Although a complete characterization of embedded applications with respect to applications' characteristics is outside the scope of this chapter, following are some of the embedded application characteristics that are discussed in context of their associated embedded domains.

1.2.1 Throughput-Intensive

Throughput-intensive embedded applications are applications that require high processing throughput. Networking and multimedia applications, which constitute a large fraction of embedded applications [17], are typically throughput-intensive due to ever increasing quality of service (QoS) demands. An embedded system containing an embedded processor requires a network stack and network protocols to connect with other devices. Connecting an embedded device or a widget to a network enables remote device management including automatic application upgrades. On a large scale, networked embedded systems can enable HPEC for solving complex large problems traditionally handled only by supercomputers (e.g., climate research, weather forecasting, molecular modeling, physical simulations, and data mining). However, connecting hundreds to thousands of embedded systems for high-performance computing (HPC) requires sophisticated and scalable interconnection technologies (e.g., packet-switched, wireless interconnects). Examples of networking applications include server I/O devices, network infrastructure equipment, consumer electronics (mobile phones, media players), and various home appliances (e.g., home automation including networked TVs, VCRs, stereos, refrigerators). Multimedia applications, such as video streaming, require very high throughput of the order of several GOPS. A broadcast video with a specification of 30 frames/s with 720×480 pixels/frame requires approximately 400,000 blocks (group of pixels) to be processed per second. A telemedicine application requires processing of 5 million blocks/s [18].

1.2.2 Thermal-Constrained

An embedded application is *thermal-constrained* if an increase in temperature above a threshold could lead to incorrect results or even the embedded system failure. Depending on the

target market, embedded applications typically operate above 45 °C (e.g., telecommunication embedded equipment temperature exceeds 55 °C) in contrast to traditional computer systems, which normally operate below 38 °C [19]. Meeting embedded application thermal constraints is challenging due to typically harsh and high-temperature operating environments. Limited space and energy budgets exacerbate these thermal challenges since active cooling systems (fans-based) are typically infeasible in most embedded systems, resulting in only passive and fanless thermal solutions.

1.2.3 Reliability-Constrained

Embedded systems with high *reliability* constraints are typically required to operate for many years without errors and/or must recover from errors since many reliability-constrained embedded systems are deployed in harsh environments where postdeployment removal and maintenance are infeasible. Hence, hardware and software for reliability-constrained embedded systems must be developed and tested more carefully than traditional computer systems. Safety-critical embedded systems (e.g., automotive airbags, space missions, aircraft flight controllers) have very high reliability requirements (e.g., the reliability requirement for a flight-control embedded system on a commercial airliner is 10^{-10} failures/h where a failure could lead to aircraft loss [20]).

1.2.4 Real-Time

In addition to correct functional operation, *real-time* embedded applications have additional stringent timing constraints, which impose real-time operational deadlines on the embedded system's response time. Although real-time operation does not strictly imply high performance, real-time embedded systems require high performance only to the point that the deadline is met, at which time high performance is no longer needed. Hence, real-time embedded systems require *predictable* high performance. Real-time operating systems (RTOSs) provide guarantees for meeting the stringent deadline requirements for embedded applications.

1.2.5 Parallel and Distributed

Parallel and *distributed* embedded applications leverage distributed embedded devices to cooperate and aggregate their functionalities or resources. Wireless sensor network (WSN) applications use sensor nodes to gather sensed information (statistics and data) and use distributed fault-detection algorithms. Mobile agent (autonomous software agent)-based distributed embedded applications allow the process state to be saved and transported to another new embedded system where the process resumes execution from the suspended point (e.g., virtual migration). Many embedded applications exhibit varying degrees (low to high levels) of *parallelism*, such as instruction-level parallelism (ILP) and thread-level parallelism (TLP). Innovative architectural and software HPEEC techniques are required to exploit an embedded application's available parallelism to achieve high performance with low power consumption.

1.3 Embedded Systems—Hardware and Software

An interesting characteristic of embedded systems design is *hardware/software codesign* (i.e., both hardware and software need to be considered together to find the right combination of hardware and software that would result in the most-efficient product meeting the requirement specifications). The mapping of application software to hardware must adhere to the design constraints (e.g., real-time deadlines) and objective functions (e.g., cost, energy consumption) (objective functions are discussed in detail in Section 1.4). In this section, we give an overview of embedded systems hardware and software as depicted in Fig. 1.1.

1.3.1 Embedded Systems Hardware

Embedded systems hardware is less standardized as compared to that for desktop computers. However, in many embedded systems, hardware is used within a loop where sensors gather information about the physical environment and generate continuous sequences of analog signals/values. Sample-and-hold circuits and analog-to-digital (A/D) converters digitize the analog signals. The digital signals are processed, and the results are displayed and/or used to control the physical environment via actuators. At the output, a digital-to-analog (D/A) conversion is generally required because many actuators are analog. In the following sections, we describe briefly the hardware components of a typical embedded system [1].

1.3.1.1 Sensors

Embedded systems contain a variety of sensors since there are sensors for virtually every physical quantity (e.g., weight, electric current, voltage, temperature, velocity, and acceleration). A sensor's construction can exploit a variety of physical effects including the law of induction (voltage generation in an electric field) and photoelectric effects. Recent advances in smart embedded systems design (e.g., WSNs, CPSs) can be attributed to the large variety of available sensors.

1.3.1.2 Sample-and-Hold Circuits and A/D Converters

Sample-and-hold circuits and A/D converters work in tandem to convert incoming analog signals from sensors into digital signals. Sample-and-hold circuits convert an analog signal from the continuous time domain to the discrete time domain. The circuit consists of a clocked transistor and a capacitor. The transistor functions similar to a switch where each time the switch is closed by the clocked signal, the capacitor is charged to the voltage $v(t)$ of the incoming voltage $e(t)$. The voltage $v(t)$ essentially remains the same even after opening the switch because of the charge stored in the capacitor until the switch closes again. Each of the voltage values stored in the capacitor are considered as an element of a discrete sequence of values generated from the continuous signal $e(t)$. The A/D converters map these voltage values to a discrete set of possible values afforded by the quantization process that converts these values to digits. There exists a variety of A/D converters with varying speed and precision characteristics.

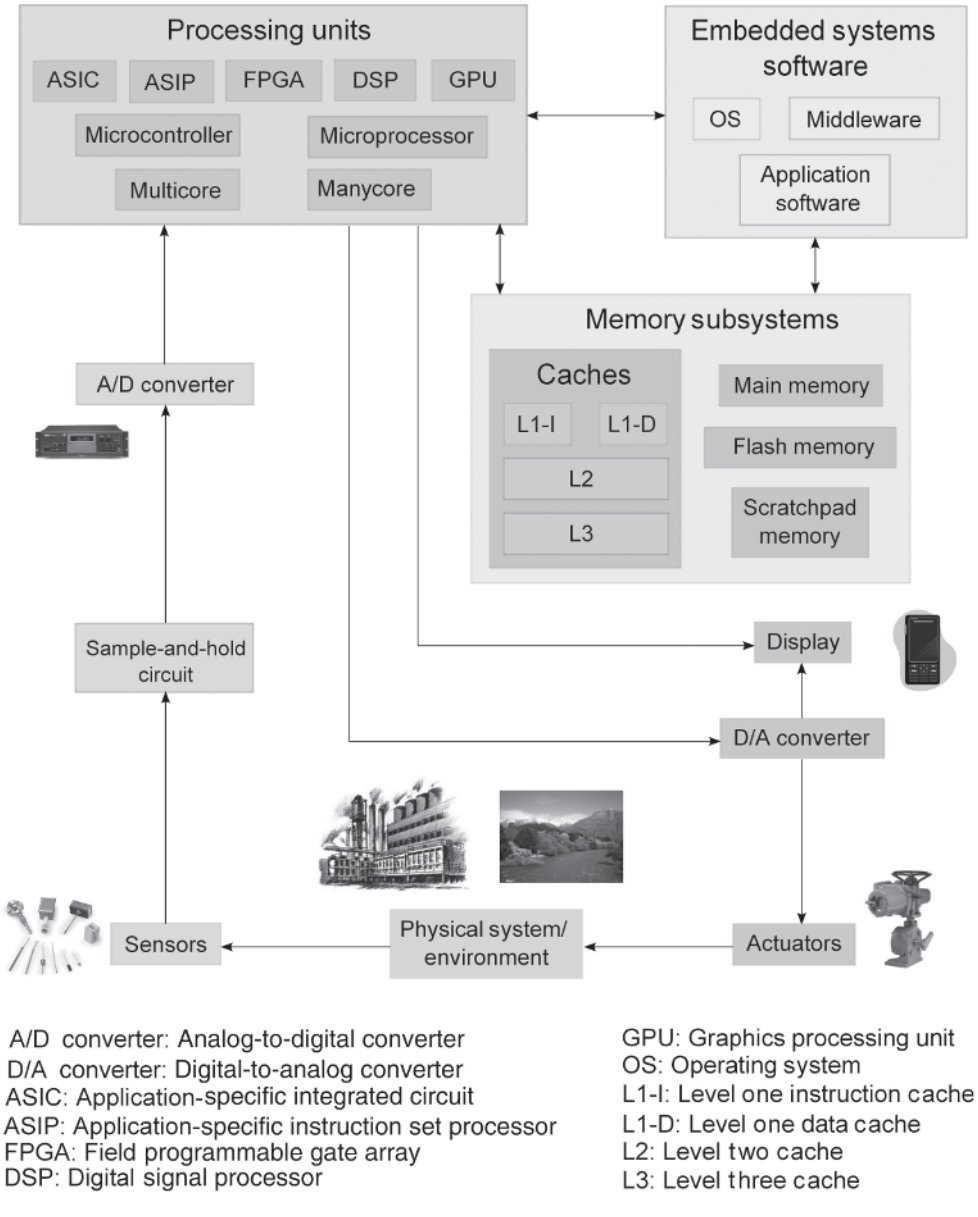

Figure 1.1 Embedded systems hardware and software overview

1.3.1.3 Processing Units

The processing units in embedded systems process the digital signals output from the A/D converters. Energy efficiency is an important factor in the selection of processing units for embedded systems. We categorize processing units into three main types:

(1) *Application-Specific Integrated Circuit*s (*ASICs*): ASICs implement an embedded application's algorithm in hardware. For a fixed process technology, ASICs provide the highest energy efficiency among available processing units at the cost of no flexibility (Fig. 1.1).
(2) *Processors:* Many embedded systems contain a general-purpose microprocessor and/or a microcontroller. These processors enable flexible programming but are much less energy efficient than ASICs. High-performance embedded applications leverage multicore/manycore processors, application domain-specific processors (e.g., digital signal processors (DSPs)), and application-specific instruction set processors (ASIPs) that can provide the required energy efficiency. GPUs are often used as coprocessors in imaging applications to accelerate and offload work from the general-purpose processors (Fig. 1.1).
(3) *Field-Programmable Gate Arrays (FPGAs):* Since ASICs are too expensive for low-volume applications and software-based processors can be too slow or energy inefficient, reconfigurable logic (of which FPGAs are the most prominent) can provide an energy-efficient solution. FPGAs can potentially deliver performance comparable to ASICs but offer reconfigurability using different, specialized configuration data that can be used to reconfigure the device's hardware functionality. FPGAs are mainly used for hardware acceleration of low-volume applications and rapid prototyping. FPGAs can be used for rapid system prototyping that emulates the same behavior as the final system and thus can be used for experimentation purposes.

1.3.1.4 Memory Subsystems

Embedded systems require memory subsystems to store code and data. Memory subsystems in embedded systems typically consist of on-chip caches and an off-chip main memory. Caches in embedded systems have different hierarchy: level one instruction cache (L1-I) for holding instructions, level one data cache for holding data (L1-D), level two unified (instruction/data) cache (L2), and recently level three cache (L3). Caches provide much faster access to code and data as compared to the main memory. However, caches are not suitable for real-time embedded systems because of limited predictability of hit rates and therefore access time. To offer better timing predictability for memory subsystems, many embedded systems especially real-time embedded systems use scratchpad memories. Scratchpad memories enable software-based control for temporary storage of calculations, data, and other work in progress instead of hardware-based control as in caches. For nonvolatile storage of code and data, embedded systems use Flash memory that can be electrically erased and reprogrammed. Examples of embedded systems using Flash memory include PDAs, digital audio and media players, digital cameras, mobile phones, video games, and medical equipment, and so on.

1.3.1.5 D/A Converters

As many of the output devices are analog, embedded systems leverage D/A converters to convert digital signals to analog signals. D/A converters typically use weighted resistors to

generate a current proportional to the digital number. This current is transformed into a proportional voltage by using an operational amplifier.

1.3.1.6 Output Devices

Embedded systems' output devices include displays and electro-mechanical devices known as actuators. Actuators can directly impact the environment based on the processed and/or control information from embedded systems. Actuators are key elements in reactive and interactive embedded systems, especially CPSs.

1.3.2 Embedded Systems Software

Embedded systems software consists of an operating system (OS), a middleware, and an application software (Fig. 1.1). Embedded software has more stringent resource constraints (e.g., small memory footprint, small data word sizes) than traditional desktop software. In the following sections, we describe the main software components of embedded systems.

1.3.2.1 Operating System

Except for very simple embedded systems, most embedded systems require an operating system (OS) for scheduling, task switching, and I/O. Embedded operating systems (EOSs) differ from traditional desktop OSs because EOSs provide limited functionality but a high-level configurability in order to accommodate a wide variety of application requirements and hardware platform features. Many embedded system's applications (e.g., CPSs) are real-time and require support from a RTOS. An RTOS leverages deterministic scheduling policies and provides predictable timing behavior with guarantees on the upper bound of a task's execution time.

1.3.2.2 Middleware

Middleware is a software layer between the application software and the EOS. Middleware typically includes communication libraries (e.g., message passing interface (MPI), ilib API for Tilera [21]). We point out that real-time embedded systems require a real-time middleware.

1.3.2.3 Application Software

Embedded systems contain application software specific to an embedded application (e.g., a portable media player, a phone framework, a healthcare application, and an ambient condition monitoring application). Embedded applications leverage communication libraries provided by the middleware and EOS features. Application software development for embedded systems requires knowledge of the target hardware architecture as assembly language fragments are often embedded within the software code for hardware control or performance purposes. The software code is typically written in a high-level language, such as C, which promotes application software conformity to stringent resource constraints (e.g., limited memory footprint and small data word sizes).

Application software development for real-time applications must consider real-time issues, especially the worst-case execution time (WCET). The WCET is defined as the largest execution time of a program for any input and any initial execution state. We point out that the exact WCET can only be computed for certain programs and tasks such as programs without recursions, without while loops, and loops with a statically known iteration count [1]. Modern pipelined processor architectures with different types of hazards (e.g., data hazards, control hazards) and modern memory subsystems composed of different cache hierarchies with limited hit rate predictability make WCET determination further challenging. To offer better timing predictability for memory subsystems, many embedded systems (real-time embedded systems in particular) use scratchpad memories. Scratchpad memories enable software-based control for temporary storage of calculations, data, and other work in progress instead of hardware-based control as in caches. Since exact WCET determination is extremely difficult, designers typically specify upper bounds on the WCET.

1.4 Modeling—An Integral Part of the Embedded Systems Design Flow

Modeling stems from the concept of abstraction (i.e., defining a real-world object in a simplified form). Formally, a model is defined as [1]: "A model is a simplification of another entity, which can be a physical thing or another model. The model contains exactly those characteristics and properties of the modeled entity that are relevant for a given task. A model is minimal with respect to a task if it does not contain any other characteristics than those relevant for the task."

The design of embedded systems increasingly depends on a hierarchy of models. Models have been used for decades in computer science to provide abstractions. Since embedded systems have very complex functionality built on top of very sophisticated platforms, designers must use a series of models to successfully accomplish the system design. Early stages of the design process require simple models of reasonable accuracy; later design stages need more sophisticated and accurate models [3].

The key phases in the embedded systems design flow are as follows: requirement specifications, hardware/software (HW/SW) partitioning, preliminary design, detailed design, component implementation, component test/validation, code generation, system integration, system verification/evaluation, and production [15]. The first phase, requirement specifications, outlines the expected/desired behavior of the SUD, and *use cases* describe potential applications of the SUD. Young et al. [22] commented on the importance of requirement specifications: "A design without specifications cannot be right or wrong, it can only be surprising!". HW/SW partitioning partitions an application's functionality into a combination of interacting hardware and software. Efficient and effective HW/SW partitioning can enable a product to more closely meet the requirement specifications. The preliminary design is a high-level design with minimum functionality that enables designers to analyze the key characteristics/functionality of an SUD. The detailed design specifies the details that are absent from the preliminary design such as detailed models or drivers for a component. Since embedded systems are complex and are comprised of many components/subsystems, many embedded systems are designed and implemented component-wise, which adds component implementation and component testing/validation phases to the design flow. Component validation may involve simulation followed by a code generation phase that generates the

appropriate code for the component. System integration is the process of integrating the design of the individual components/subsystem into the complete, functioning embedded system. Verification/evaluation is the process of verifying quantitative information of key objective functions/characteristics (e.g., execution time, reliability) of a certain (possibly partial) design. Once an embedded systems design has been verified, the SUD enters that production phase that produces/fabricates the SUD according to market requirements dictated by supply and demand economic model. Modeling is an integral part of the embedded systems design flow, which abstracts the SUD and is used throughout the design flow, from the requirement specifications' phase to the formal verification/evaluation phase.

Most of the errors encountered during embedded systems design are directly or indirectly related to incomplete, inconsistent, or even incorrect requirement specifications. Currently, the requirement specifications are mostly given in sentences of a natural language (e.g., English), which can be interpreted differently by the OEMs and the suppliers (e.g., Bosch, Siemens that provide embedded subsystems). To minimize the design errors, the embedded industry prefers to receive the requirement specifications in a modeling tool (e.g., graphical or language based). Modeling facilitates designers to deduce errors and quantitative aspects (e.g., reliability, lifetime) early in the design flow.

Once the SUD modeling is complete, the next phase is validation through simulation followed by code generation. Validation is the process of checking whether a design meets all of the constraints and performs as expected. Simulating embedded systems may require modeling the SUD, the operating environment, or both. Three terminologies are used in the literature depending on whether the SUD or the real environment or both are modeled: "Software-in-the-loop" refers to the simulation where both the SUD and the real environment are modeled for early system validation; "Rapid prototyping" refers to the simulation where the SUD is modeled and the real environment exists for early POC; and "Hardware-in-the-loop" refers to the simulation where the physical SUD exists and real environment is modeled for exhaustive characterization of the SUD.

Scalability in modeling/verification means that if a modeling/verification technique can be used to abstract/verify a specific small system/subsystem, the same technique can be used to abstract/verify large systems. In some scenarios, modeling/verification is scalable if the correctness of a large system can be inferred from a small verifiable modeled system. Reduction techniques such as partial order reduction and symmetry reduction address this scalability problem; however, this area requires further research.

1.4.1 Modeling Objectives

Embedded systems design requires characterization of several objectives, or design metrics, such as the average execution time and WCETs, code size, energy/power consumption, safety, reliability, temperature/thermal behavior, electromagnetic compatibility, cost, and weight. We point out that some of these objectives can be taken as design constraints since in many optimization problems, objectives can be replaced by constraints and vice versa. Considering multiple objectives is a unique characteristic of many embedded systems and can be accurately captured using mathematical models. A system or subsystem's mathematical model is a mathematical structure consisting of sets, definitions, functions, relations, logical predicates (true or false statements), formulas, and/or graphs. Many mathematical models for embedded

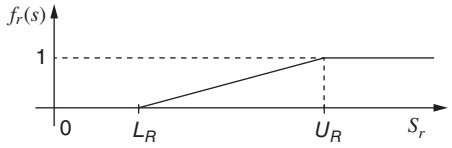

Figure 1.2 A linear objective function for reliability

systems use objective function(s) to characterize some or all of these objectives, which aids in early evaluation of embedded systems design by quantifying information for key objectives.

The objectives for an embedded system can be captured mathematically using linear, piecewise linear, or nonlinear functions. For example, a linear objective function for the reliability of an embedded system operating in a state s (Fig. 1.2) can be given as [23]

$$f_r(s) = \begin{cases} 1, & r \geq U_R \\ (r - L_R)/(U_R - L_R), & L_R < r < U_R \\ 0, & r \leq L_R \end{cases} \tag{1.1}$$

where r denotes the reliability offered in the current state s (denoted as s_r in Fig. 1.2), and the constant parameters L_R and U_R denote the minimum and maximum allowed/tolerated reliability, respectively. The reliability may be represented as a multiple of a base reliability unit equal to 0.1, which represents a 10% packet reception rate [24].

Embedded systems with multiple objectives can be characterized by using either multiple objective functions, each representing a particular design metric/objective, or a single objective function that uses a weighted average of multiple objectives. A single overall objective function can be formulated as

$$\begin{aligned} & f(s) = \sum_{k=1}^{m} \omega_k f_k(s) \\ \text{s.t.} \quad & s \in S \\ & \omega_k \geq 0, \quad k = 1, 2, \ldots, m \\ & \omega_k \leq 1, \quad k = 1, 2, \ldots, m \\ & \sum_{k=1}^{m} \omega_k = 1 \end{aligned} \tag{1.2}$$

where $f_k(s)$ and ω_k denote the objective function and weight factor for the Kth objective/design metric (weight factors signify the weightage/importance of objectives with respect to each other), respectively, given that there are m objectives. Individual objectives are characterized by their respective objective functions $f_k(s)$ (e.g., a linear objective function for reliability is given in Eq. (1.1) and depicted in Fig. 1.2).

A single objective function allows selection of a single design from the design space (the design space represents the set containing all potential designs); however, the assignments of weights for different objectives in the single objective function can be challenging using informal requirement specifications. Alternatively, the use of multiple, separate objective

functions returns a set of designs from which a designer can select an appropriate design that meets the most critical objectives optimally/suboptimally. Often embedded systems modeling focuses on optimization of an objective function (e.g., power, throughput, reliability) subject to design constraints. Typical design constraints for embedded systems include safety, hard real-time requirements, and tough operating conditions in a harsh environment (e.g., aerospace) though some or all of these constraints can be added as objectives to the objective function in many optimization problems as described earlier.

1.4.2 Modeling Paradigms

Since embedded systems contain a large variety of abstraction levels, components, and aspects (e.g., hardware, software, functional, verification) that cannot be supported by one language or tool, designers rely on various modeling paradigms, each of which target a partial aspect of the complete design flow from requirement specifications to production. Each modeling paradigm describes the system from a different point of view, but none of the paradigms cover all aspects. We discuss some of the modeling paradigms used in embedded systems design in the following sections, each of which may use different tools to assist with modeling.

1.4.2.1 Differential Equations

Differential equations-based modeling can either use ordinary differential equations (ODEs) or partial differential equations (PDEs). ODEs (linear and nonlinear) are used to model systems or components characterized by quantities that are continuous in value and time, such as voltage and current in electrical systems, speed and force in mechanical systems, or temperature and heat flow in thermal systems [15]. ODE-based models typically describe analog electrical networks or the mechanical behavior of the complete system or component. ODEs are especially useful for studying feedback control systems that can make an unstable system into a stable one (feedback systems measure the error (i.e., difference between the actual and desired behavior) and use this error information to correct the behavior). We emphasize that ODEs work for smooth motion where linearity, time invariance, and continuity properties hold. Nonsmooth motion involving collisions requires hybrid models that are a mixture of continuous and discrete time models [25].

PDEs are used for modeling behavior in space and time, such as moving electrodes in electromagnetic fields and thermal behavior. Numerical solutions for PDEs are calculated by finite-element methods (FEMs) [25].

1.4.2.2 State Machines

State machines are used for modeling discrete dynamics and are especially suitable for reactive systems. Finite-state machines (FSMs) and state-charts are some of the popular examples of state machines. Communicating Finite-state machines (CFSMs) represent several FSMs communicating with each other. State-charts extend FSMs with a mechanism for describing hierarchy and concurrency. Hierarchy is incorporated using *super-states* and *sub-states*, where super-states are states that comprise other sub-states [1]. Concurrency in state-charts

is modeled using *AND-states*. If a system containing a super-state *S* is always in all of the sub-states of *S* whenever the system is in *S*, then the super-state *S* is an *AND-super-state*.

1.4.2.3 Dataflow

Dataflow modeling identifies and models data movement in an information system. Dataflow modeling represents processes that transform data from one form to another, external entities that receive data from a system or send data into the system, data stores that hold data, and dataflow that indicates the routes over which the data can flow. A dataflow model is represented by a directed graph where the nodes/vertices, *actors*, represent computation (computation maps input data streams into output data streams) and the arcs represent communication channels. Synchronous dataflow (SDF) and Kahn process networks (KPNs) are common examples of dataflow models. The key characteristics of these dataflow models is that SDFs assume that all actors execute in a single clock cycle, whereas KPNs permit actors to execute with any finite delay [1].

1.4.2.4 Discrete Event-Based Modeling

Discrete event-based modeling is based on the notion of firing or executing a sequence of discrete events, which are stored in a queue and are sorted by the time at which these events should be processed. An event corresponding to the current time is removed from the queue, processed by performing the necessary actions, and new events may be enqueued based on the action's results [1]. If there is no event in the queue for the current time, the time advances. Hardware description languages (e.g., VHDL, Verilog) are typically based on discrete event modeling. SystemC, which is a system-level modeling language, is also based on discrete event modeling paradigm.

1.4.2.5 Stochastic Models

Numerous stochastic models exist, which mainly differ in the assumed distributions of the state residence times, to describe and analyze system performance and dependability. Analyzing the embedded system's performance in an early design phase can significantly reduce late-detected, and therefore cost-intensive, problems. Markov chains and queueing models are popular examples of stochastic models. The state residence times in Markov chains are typically assumed to have exponential distributions because exponential distributions lead to efficient numerical analysis, although other generalizations are also possible. Performance measures are obtained from Markov chains by determining steady-state and transient-state probabilities. Queueing models are used to model systems that can be associated with some notion of queues. Queueing models are stochastic models since these models represent the probability of finding a queueing system in a particular configuration or state.

Stochastic models can capture the complex interactions between an embedded system and its environment. Timeliness, concurrency, and interaction with the environment are primary characteristics of many embedded systems, and *nondeterminism* enables stochastic models to incorporate these characteristics. Specifically, nondeterminism is used for modeling unknown

aspects of the environment or system. Markov decision processes (MDPs) are discrete stochastic dynamic programs, an extension of discrete time Markov chains, that exhibit nondeterminism. MDPs associate a reward with each state in the Markov chain.

1.4.2.6 Petri Nets

A Petri net is a mathematical language for describing distributed systems and is represented by a directed, bipartite graph. The key elements of Petri nets are conditions, events, and a flow relation. Conditions are either satisfied or not satisfied. The flow relation describes the conditions that must be met before an event can fire as well as prescribes the conditions that become true when after an event fires. Activity charts in unified modeling language (UML) are based on Petri nets [1].

1.4.3 Strategies for Integration of Modeling Paradigms

Describing different aspects and views of an entire embedded system, subsystem, or component over different development phases requires different modeling paradigms. However, sometimes partial descriptions of a system need to be integrated for simulation and code generation. Multiparadigm languages integrate different modeling paradigms. There are two types of multiparadigm modeling [15]:

(1) One model describing a system complements another model resulting in a model of the complete system.
(2) Two models give different views of the same system.

UML is an example of multiparadigm modeling, which is often used to describe software-intensive system components. UML enables the designer to verify a design before any hardware/software code is written/generated [26] and allows generation of the appropriate code for the embedded system using a set of rules. UML offers a structured and repeatable design: if there is a problem with the behavior of the application, then the model is changed accordingly; and if the problem lies in the performance of the code, then the rules are adjusted. Similarly, MATLAB's Simulink modeling environment integrates a continuous time and discrete time model of computation based on equation solvers, a discrete event model, and an FSM model.

Two strategies for the integration of heterogeneous modeling paradigms are [15] as follows:

(1) Integration of operations (analysis, synthesis) on models
(2) Integration of models themselves via model translation.

We briefly describe several different integration approaches that leverage these strategies in the following sections.

1.4.3.1 Cosimulation

Cosimulation permits simulation of partial models of a system in different tools and integrates the simulation process. Cosimulation depends on a central cosimulation engine, called a

simulation backplane, that mediates between the distributed simulations run by the simulation engines of the participating computer-aided software engineering (CASE) tools. Cosimulation is useful and sufficient for model validation when simulation is the only purpose of model integration. In general, cosimulation is useful for the combination of a system model with a model of the system's environment since the system model is constructed completely in one tool and enters into the code generation phase, whereas the environment model is only used for simulation. Alternatively, cosimulation is insufficient if both of the models (the system and its environment model) are intended for code generation.

1.4.3.2 Code Integration

Many modeling tools have associated code generators, and code integration is the process of integrating the generated codes from multiple modeling tools. Code integration tools expedite the design process because in the absence of a code integration tool, subsystem codes generated by different tools have to be integrated manually.

1.4.3.3 Code Encapsulation

Code encapsulation is a feature offered by many CASE tools that permits code encapsulation of a subsystem model as a block box in the overall system model. Code encapsulation facilitates automated code integration as well as overall system simulation.

1.4.3.4 Model Encapsulation

In model encapsulation, an original subsystem model is encapsulated as an equivalent subsystem model in the modeling language of the enclosing system. Model encapsulation permits *coordinated code generation* in which the code generation for the enclosing system drives the code generator for the subsystem. The enclosing system tool can be regarded as a master tool and the encapsulated subsystem tool as a slave tool; therefore, model encapsulation requires the master tool to have knowledge of the slave tool.

1.4.3.5 Model Translation

In model translation, a subsystem model is translated syntactically and semantically to the language of the enclosing system. This translation results in a homogeneous overall system model so that one tool chain can be used for further processing of the complete system.

1.5 Optimization in Embedded Systems

General-purpose computing systems are designed to work well in a variety of contexts. Although embedded computing systems must be flexible, the embedded systems can often be tuned or optimized to a particular application. Consequently, some of the design precepts that are commonly adhered in the design of general-purpose computers do not hold for embedded computers. Given the huge number of embedded computers sold each year, many application areas make it worthwhile to spend the time to create a customized and optimized architecture.

Optimization techniques at different design levels (e.g., hardware and software, data link layer, routing, OS) assist designers in meeting application requirements. Embedded systems optimization techniques can be generally categorized as *static* or *dynamic*. Static optimizations optimize an embedded system at deployment time and remain fixed for the embedded system's lifetime. Static optimizations are suitable for stable/predictable applications, whereas they are inflexible and do not adapt to changing application requirements and environmental stimuli. Dynamic optimizations provide more flexibility by continuously optimizing an embedded system during runtime, providing better adaptation to changing application requirements and actual environmental stimuli.

Parallel and distributed embedded systems add more facets to optimization problem than traditional embedded systems as a growing number of distributed embedded systems leverage wireless communication to connect with different embedded devices. This wireless connectivity between distributed embedded systems requires optimization of radios (aka data transceivers) and various layers of Open Systems Interconnect (OSI) model of the International Standards Organization (ISO) implemented in these embedded systems. Embedded systems' radios carry digital information and are used to connect to networks. These networks may be specialized, as in cell phones, but increasingly radios are used as the physical layer in the Internet protocol systems [3]. The radios in these distributed wireless embedded systems must perform several tasks: demodulate the signal down to the baseband, detect the baseband signal to identify bits, and correct errors in the raw bit stream. Wireless data radios in these embedded systems may be built from combinations of analog, hardwired digital, configurable, and programmable components. Software-defined radios (SDRs) are also being used in some parallel and distributed embedded systems. A *software radio* is a radio that can be programmed; the term SDR is often used to mean either a purely or a partly programmable radio [3].

Although it may seem that embedded systems would be too simple to require the use of the OSI model, embedded systems increasingly implement multiple layers of the OSI model. Even relatively simple embedded systems provide physical, data link, network, and application services. An increasing number of embedded systems provide Internet service that requires implementing the full range of functions in the OSI model. The OSI model defines a seven-layer model for network services [3]:

(1) *Physical:* The electrical and physical connection
(2) *Data link:* Access and error control across a single link
(3) *Network:* Basic end-to-end service
(4) *Transport:* Connection-oriented services
(5) *Session:* Activity control, such as checkpointing
(6) *Presentation:* Data exchange formats
(7) *Application:* The interface between the application and the network.

The OSI model layers can be implemented in hardware and/or software in embedded systems depending on the embedded application requirements. Embedded systems are optimized for various application requirements and design metrics in almost all design phases. Embedded systems optimization benefits in particular from the following design phases:

- Modeling at different levels of abstraction
- Profiling and analysis revamp system requirements and software models into more specific requirements on the platform hardware architecture

- Design space exploration (whether exhaustive or some heuristic-based) evaluates hardware alternatives.

The design phases in embedded systems optimize various design metrics, such as performance, power, cost, and reliability. The remainder of this section focuses on optimization of these design metrics for embedded systems.

1.5.1 Optimization of Embedded Systems Design Metrics

An embedded application determines the basic *functional requirements*, such as input and output relationship. The embedded system designer must also determine the *nonfunctional requirements*, such as performance, power, cost, some of which are derived directly from the application and some from other factors, such as marketing. Design metrics are generally derived from application requirements. Often design metrics derived from nonfunctional requirements are equally important as those from functional requirements. An embedded systems applications may have many design metrics, such as performance, power, reliability, and quality. Some of these metrics can be accurately measured and predicted while others are less so.

Various optimization techniques at different levels (e.g., architecture, middleware, and software) can be used to enable an embedded platform to optimize various design metrics and meet the embedded application requirements. In the following, we elaborate the optimization of a few design metrics: performance, power, temperature, cost, design time, reliability, and quality.

1.5.1.1 Performance

Performance metric refers to some aspect of speed. Performance can be measured in many different ways: average performance versus worst-case or best-case performance, throughput versus latency, and peak versus sustained performance [3]. Chapter 7 of this book discusses various performance optimization techniques for embedded systems. For example, throughput-intensive applications can leverage architectural innovations (e.g., tiled multicore architectures, high-bandwidth interconnects), hardware-assisted middleware techniques (e.g., speculative approaches, dynamic voltage and frequency scaling (DVFS), hyperthreading), and software techniques (e.g., data forwarding, task scheduling, and task migration). Please refer to Chapter 7 of this book for a comprehensive discussion on performance optimization techniques.

1.5.1.2 Energy/Power

Energy and/or power consumption are critical metrics for many embedded systems. Energy consumption is particularly important for battery-operated embedded systems as reduced energy consumption leads to an increased battery life of the embedded system. Power consumption also affects heat generation. Less power consumption not only engenders less cooling costs but also enables sustainable long-term functioning without damaging the chip due to overheating. Energy/power optimization techniques for embedded systems include

DVFS, power gating, and clock gating. Please refer to Chapter 7 of this book for details on energy optimization techniques.

1.5.1.3 Temperature

Thermal-aware design has become a prominent aspect of microprocessor and SoC design due to the large thermal dissipation of modern chips. In very high-performance systems, heat may be difficult to dissipate even with fans and thermally controlled ambient environments. Many consumer devices avoid the use of cooling fans due to size constraints [3].

Heat transfer in integrated circuits relies on the thermal resistance and thermal capacitance of the chip, its package, and the associated heat sink. *Thermal resistance* is a heat property and a measurement of a temperature difference by which an object or material resists a heat flow. Absolute thermal resistance is the temperature difference across a structure when a unit of heat energy flows through the structure in unit time. *Thermal capacitance* is equal to the ratio of the heat added to (or subtracted from) an object to the resulting temperature change. Thermal models can be solved in the same manner as are electrical resistor–capacitor (RC) circuit models. The activity of the architectural units determines the amount of heat generated in each unit.

There exists various techniques to optimize temperature design metric, such as temperature-aware task scheduling [27], temperature-aware DVFS [28], thermal profile management [29], proactive scheduling for processor temperature [30], and reactive scheduling for processor temperature [31]. To illustrate optimization of temperature design metric, we summarize the approach in reactive scheduling for processor temperature in the following.

Wang and Bettati [31] developed a reactive scheduling algorithm while ensuring that the processor temperature remained below a threshold T_H. Let $S(t)$ represent the processor speed/frequency at time t. The processor power consumption is modeled as:

$$P(t) = KS^{\alpha}(t) \tag{1.3}$$

where K is a constant and $\alpha > 1$ (usually, it is assumed that $\alpha = 3$). The thermal properties of the system can be modeled as

$$a = \frac{K}{C_{\text{th}}} \tag{1.4}$$

$$b = \frac{1}{R_{\text{th}}C_{\text{th}}} \tag{1.5}$$

where b is a positive constant that represents the power dissipation rate. The *equilibrium speed* S_E is defined as the speed at which the processor stays at the threshold temperature T_H and is given as

$$S_E = \left(\frac{b}{a}T_H\right)^{1/\alpha} \tag{1.6}$$

According to reactive scheduling for processor temperature:

- When the processor has useful work to do and is at its threshold temperature, the processor clock speed is set to the S_E.

- When the processor has useful work to do and the temperature is below the threshold temperature, increase the processor speed (the processor speed can be increased up to the maximum available processor frequency/speed).

1.5.1.4 Cost

The monetary cost of a system is an important design metric. Cost can be measured in several ways. *Manufacturing cost* is determined by the cost of components and the manufacturing processes used. *Unit cost* is the monetary cost of manufacturing each copy of the system, excluding nonrecurring engineering (NRE) cost. *NRE cost* is the one-time monetary cost of designing the system. NRE cost is also known as *design cost*. Design cost is determined both by labor and by the equipment used to support the designers. The server farm and computer-aided design (CAD) tools needed to design a large chip cost several million dollars. *Lifetime cost* comprises of software and hardware maintenance and upgrades. The *total cost* of producing a certain number of units of an embedded design can be given as [32]

$$\text{Total cost} = \text{NRE cost} + (\text{unit cost} \times \text{number of units}) \tag{1.7}$$

The *per-product cost* can be given as [32]

$$\begin{aligned} \text{Per-product cost} &= \frac{\text{total cost}}{\text{number of units}} \\ &= \left(\frac{\text{NRE cost}}{\text{number of units}}\right) + \text{unit cost} \end{aligned} \tag{1.8}$$

Let us consider an example for the illustration of different costs. Suppose NRE cost of designing an embedded system is \$3000 and the unit cost is \$100. If 20 units are produced for that embedded system, then total cost from Eq. (1.7) is as follows: total cost = \$3000 + (20 × \$100) = \$3000 + \$2000 = \$5000. Per-product cost for that embedded system from Eq. (1.8) is as follows: per-product cost = \$3000/20 + \$100 = \$150 + \$100 = \$250.

Many optimization problems deal with reducing the cost of an embedded systems design while meeting application requirements. In general, a cost minimization problem takes a form similar to the following equation:

$$\begin{aligned} \min \quad & f_c(x, y, z) \\ \text{s.t.} \quad & x \leq a \\ & y \leq b \\ & z \leq c \end{aligned} \tag{1.9}$$

where $f_c(x, y, z)$ represents the cost function to be minimized and is a function of parameters x, y, and z. The constraints of the optimization problem are also specified in Eq. (1.9) where a, b, and c denote some constants that restrict parameters x, y, and z, respectively.

1.5.1.5 Design Time

The time required to design a system is an important metric for many embedded systems. Design time is often constrained by *time-to-market*. Time-to-market is the time required to

develop a system to the point that it can be released and sold to customers. If an embedded systems design takes too long to complete, the product may miss its intended market. Revenue for an embedded product depends on *market window*, which is the period during which the product would have highest sales. Delays can be costly in the delayed entry of an embedded product in the market window. For example, calculators must be ready for the back-to-school market each fall. Various modeling and CAD tools are used in embedded systems design to reduce the design time and meet the time-to-market constraints.

1.5.1.6 Reliability

Different embedded systems have different reliability requirements. In some consumer markets, customers do not expect to keep the product (e.g., mobile phone) for a long period. Automobiles, in contrast, must be designed to be safe and reliable. Dependability assimilation in safety-critical embedded systems (e.g., automobiles, aircrafts) is paramount because of product liability legislations, ISO standards, and increasing customer expectations. The product liability law holds responsible the manufacturers, distributors, suppliers, and retailers for the injuries caused by those products. According to the law, the manufacturer's product liability is excluded if a failure cannot be detected using the state-of-the-art science and technology at the time of product release.

Reliability techniques that help in designing a reliable embedded system include N-modular redundancy, watchdog timers, coding techniques (e.g., parity codes, checksum, cyclic codes), algorithm-based fault tolerance, acceptance tests, and checkpointing. Various reliability modeling methodologies also assist the designers in meeting reliability requirements of an embedded design.

1.5.1.7 Quality

Quality is a design metric that is often hard to quantify and measure. Quality or QoS is often considered as a performance measure. In some markets (e.g., few consumer devices), factors such as user interface design, availability of the connection, speed of data streaming, and processing may be associated with quality. In safety-critical systems, such as automobiles, QoS must be considered as a dependability or reliability measure that can impact the system's availability and safety. For example, if the end-to-end delay in a cyber-physical automotive system exceeds beyond a certain critical threshold, the driver can totally lose the control of his/her car.

1.5.2 Multiobjective Optimization

Embedded systems design must meet several different design objectives. The traditional operations research approach of defining a single objective function and possibly some minor objective functions, along with design constraints, may not befittingly capture an embedded system's requirements. Economist Vilfredo Pareto proposed a theory for multiobjective analysis known as *Pareto optimality*. The theory also delineates the method by which optimal solutions are assessed: an optimal solution cannot be improved without making some other part of the solution worse [3]. Optimization of embedded systems is often multiobjective, and

the designer has to make trade-offs between different objectives depending on the criticality (weight factor) of an objective (Eq. (1.2)).

1.6 Chapter Summary

This chapter introduced parallel and distributed embedded systems. We elaborated on several embedded systems applications domains including CPSs, space, medical, and automotive. The chapter discussed various characteristics of embedded systems applications, such as throughput-intensive, thermal-constrained, reliability-constrained, real-time, and parallel and distributed. We elucidated the main components of a typical embedded system's hardware and software. We presented an overview of modeling, modeling objectives, and various modeling paradigms. Finally, we elaborated optimization of various design metrics, such as performance, energy/power, temperature, cost, design time, reliability, and quality, for embedded systems.