78

Embedded Computing Systems and Hardware/Software Co-Design

Wayne Wolf
Princeton University

CONTENTS

78.1 Introduction

This chapter describes embedded computing systems that make use of microprocessors to implement part of the system's function. It also describes hardware/software co-design, which is the process of designing embedded systems while simultaneously considering the design of its hardware and software elements.

78.2 Uses of Microprocessors

An embedded computing system (or more simply an embedded system) is any system which uses a programmable processor but itself is not a general purpose computer. Thus, a personal computer is not an embedded computing system (though PCs are often used as platforms for building embedded systems), but a telephone or automobile which includes a CPU is an embedded system. Embedded systems may offer some amount of user programmability—3Com's PalmPilot, for example, allows users to write and download programs even though it is not a general-purpose computer—but embedded systems generally run limited sets of programs. The fact that we know the software that we will run on the hardware allows us to optimize both the software and hardware in ways that are not possible in general-purpose computing systems.

Microprocessors are generally categorized by their word size, since word size is associated both with maximum program size and data resolution. Commercial microprocessors come in many sizes; the term microcontroller is used to denote a microprocessor which comes with some basic on-chip peripheral

devices, such as serial input/output (I/O) ports. Four-bit microcontrollers are extremely simple but capable of some basic functions. Eight-bit microcontrollers are workhorse low-end microprocessors. Sixteen- and 32-bit microprocessors provide significantly more functionality. A 16/32-bit microprocessor may be in the same architectural family as the CPUs used in computer workstations, but microprocessors destined for embedded computing often do not provide memory management hardware. A digital signal processor (DSP) is a microprocessor tuned for signal processing applications. DSPs are often Harvard architectures, meaning that they provide separate data and program memories; Harvard architectures provide higher performance for DSP applications. DSPs may provide integer or floating-point arithmetic.

Microprocessors are used in an incredible variety of products. Furthermore, many products contain multiple microprocessors. Four- and eight-bit microprocessors are often used in appliances: for example, a thermostat may use a microcontroller to provide timed control of room temperature. Automatic cameras often use several eight-bit microprocessors, each responsible for a different aspect of the camera's functionality: exposure, shutter control, etc. High-end microprocessors are used in laser and ink-jet printers to control the rendering of the page. Many printers use two or three microprocessors to handle generation of pixels, control of the print engine, and so forth. Modern automobiles may use close to 100 microprocessors, and even inexpensive automobiles generally contain several. High-end microprocessors are used to control the engine's ignition system—automobiles use sophisticated control algorithms to simultaneously achieve low emissions, high fuel economy, and good performance. Low-end microcontrollers are used in a number of places in the automobile to increase functionality: for example, four-bit microcontrollers are often used to sense whether seat belts are fastened and turn on the seat belt light when necessary.

Microprocessors may replace analog components to provide similar functions, or they may add totally new functionality to a system. They are used in several different ways in embedded systems. One broad application category is signal conditioning, in which the microprocessor or DSP performs some filtering or control function on a digitized input. The conditioned signal may be sent to some other microprocessor for final use. Signal conditioning allows systems to use less-expensive sensors with the application of a relatively inexpensive microprocessor. Beyond signal conditioning, microprocessors may be used for more sophisticated control applications. For example, microprocessors are often used in telephone systems to control signaling functions, such as determining what action to take based on the reception of dial tones, etc. Microprocessors may implement user interfaces; this requires sensing when buttons, knobs, etc. are used, taking appropriate actions, and updating displays. Finally, microprocessors may perform data processing, such as managing the calendar in a personal digital assistant.

There are several reasons why microprocessors make good design components in such a wide variety of application areas. First, digital systems often provide more complex functionality than can be created using analog components. A good example is the user interface of a home audio/video system, which provides more information and is easier use than older, non-microprocessor-controlled systems. Microprocessors also allow related products much more cost-effectively. An entire product family, including models at various price and feature points, can be built around a single microprocessor-based platform. The platform includes both hardware components common to all the family members and software running on the microprocessor to provide functionality. Software elements can easily be turned on or off in various family members. Economies of scale often mean that it is cheaper to put the same hardware in both expensive and cheap models and to turn off features in the inexpensive models rather than to try to optimize the hardware and software configurations of each model separately. Microprocessors also allow design changes to be made much more quickly. Many changes may be possible simply by reprogramming; other features may be made possible by adding memory or other simple hardware changes along with some additional programming. Finally, microprocessors aid in concurrent engineering. After some initial design decisions have been made, hardware and software can be designed in parallel, reducing total design time.

While embedded computing systems traditionally have been fabricated at the board level out of multiple chips, embedded computing systems will play an increasing role in integrated circuit design as well. As VLSI technology moves toward the ability to fabricate chips with billions of transistors,

...rated circuits will increasingly incorporate one or several microprocessors executing embedded ...are. Using microprocessors as components in integrated circuits increases design productivity, ... CPUs can be used as large components which implement a significant part of the system function-... Single-chip embedded systems can provide much higher performance than board-level equivalents, ... chip-to-chip delays are eliminated.

78.3 Embedded System Architectures

Although embedded computing spans a wide range of application areas, from automotive to medical, there are some common principles of design for embedded systems. The application-specific embedded software runs on a hardware platform. An example hardware platform is shown in Figure 78.1. It contains a microprocessor, memory, and I/O devices. When designing on a general-purpose system such as a PC, the hardware platform would be predetermined, but in hardware/software co-design the software and hardware can be designed together to better meet cost and performance requirements.

Depending on the application, various combinations of criteria may be important goals for the system design. Two typical criteria are speed and manufacturing cost. The speed at which computations are made often contributes to the general usability of the system, just as in general-purpose computing. However, performance is also often associated with the satisfaction of deadlines—times at which computations must be completed to ensure the proper operation of the system. If failure to meet a deadline causes a major error, it is termed a hard deadline. And missed deadlines, which result in tolerable but unsatisfactory degradations are called soft deadlines. Hard deadlines are often (though not always) associated with safety-critical systems. Designing for deadlines is one of the most challenging tasks in embedded system design. Manufacturing cost is often an important criteria for embedded systems. Although the hardware components ultimately determine manufacturing cost, software plays an important role as well. First, the size of the program determines the amount of memory required, and memory is often a significant component of the total component cost. Furthermore, the improper design of software can cause one to require higher-performance, more-expensive hardware components than are really necessary. Efficient utilization of hardware resources requires careful software design. Power consumption is becoming an increasingly important design metric. Power is certainly important in battery-operated devices, but it can be important in wall socket-powered systems as well—lower power consumption means smaller, less-expensive power supplies and cooling and may result in environmental ratings that are advantageous in the marketplace. Once again, power consumption is ultimately determined by the hardware, but software plays a significant role in power characteristics. For example, more efficient use of on-chip caches can reduce the need for off-chip memory access, which consumes much more power than on-chip cache references.

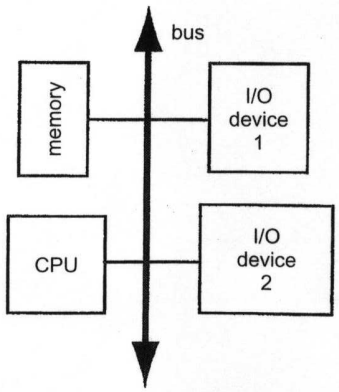

FIGURE 78.1 Hardware structure of a microprocessor system.

Figure 78.1 shows the hardware architecture of a basic microprocessor system. The system includes the CPU, memory, and some I/O devices, all connected by a bus. This system may consist of multiple chips for high-end microprocessors or a single-chip microcontroller. Typical I/O devices include analog/digital (ADC) and digital/analog (DAC) converters, serial and parallel communication devices, network and bus interfaces, buttons and switches, and various types of display devices. This configuration is a complete, basic, embedded computing hardware platform on which application software can execute.

The embedded application software includes components for managing I/O devices and for performing the core computational tasks. The basic software techniques for communicating with I/O devices are polling and interrupt-driven. In a polled system, the program checks each device's status register to determine if it is ready to perform I/O. Polling allows the CPU to determine the order in which I/O operations are completed, which may be important for ensuring that certain device requests are satisfied at the proper rate. However, polling also means that a device may not be serviced in time if the CPU program does not check it frequently enough. Interrupt-driven I/O allows a device to change the flow of control on the CPU and call a device driver to handle the pending I/O operation. An interrupt system may provide both prioritized interrupts to allow some devices to take precedence over others and vectored interrupts to allow devices to specify which driver should handle their request.

Device drivers, whether polled or interrupt-driven, will typically perform basic device-specific functions and hand-off data to the core routines for processing. Those routines may perform relatively simple tasks, such as transducing data from one device to another, or may perform more sophisticated algorithms such as control. Those core routines often will initiate output operations based on their computations on the input operations.

Input and output may occur either periodically or aperiodically. Sampled data is a common example of periodic I/O, while user interfaces provide a common source of aperiodic I/O events. The nature of the I/O transactions affects both the device drivers and the core computational code. Code which operates on periodic data is generally driven by a timer which initiates the code at the start of the period. Periodic operations are often characterized by their periods and the deadline for each period. Aperiodic I/O may be detected either by an interrupt or by polling the devices. Aperiodic operations may have deadlines which are generally measured from the initiating I/O event. Periodic operations can often be thought of as being executed within an infinite loop. Aperiodic operations tend to use more event-driven code, in which various sections of the program are exercised by different aperiodic events, since there is often more than one aperiodic event which can occur.

Embedded computing systems exhibit a great deal of parallelism which can be used to speed up computation. As a result, they often use multiple microprocessors which communicate with each other to perform the required function. In addition to microprocessors, application-specific ICs (ASICs) may be added to accelerate certain critical functions. CPUs and ASICs in general are called processing elements (PEs). An example multiprocessor system built from several PEs along with I/O devices and memory is shown in Figure 78.2.

The choice of several small microprocessors or ASICs rather than one large CPU is primarily determined by cost. Microprocessor cost is a nonlinear function of performance, even within a microprocessor family. Vendors generally supply several versions of a microprocessor which run at different clock rates; chips which run at varying speeds are a natural consequence of the variations in the VLSI manufacturing process. The slowest microprocessors are significantly less expensive than the fastest ones, and the cost increment is larger at the high end of the speed range than at the low end. As a result, it is often cheaper to use several smaller microprocessors to implement a function.

When several microprocessors work together in a system, they may communicate with each other in several different ways. If slow data rates are sufficient, serial data links are commonly used for their low hardware cost. The I^2C bus is a well-known example of a serial bus used to build multi-microprocessor embedded systems; the CAN bus is widely used in automobiles. High-speed serial links can achieve moderately high performance and are often used to link multiple DSPs in high-speed signal processing

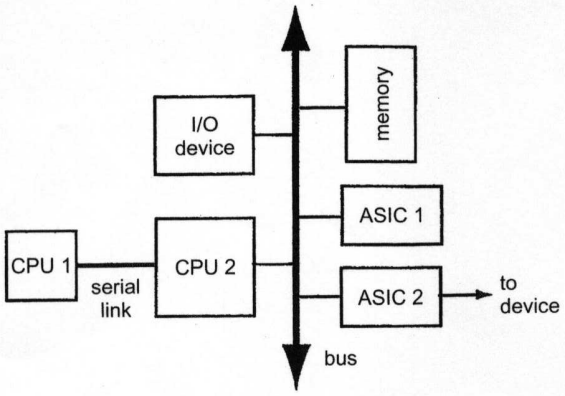

FIGURE 78.2 A heterogeneous embedded multiprocessor.

systems. Parallel data links provide the highest performance thanks to their sheer data width. High-speed buses such as PCI can be used to link several processors.

The software for an embedded multiprocessing system is often built around processes. A process, as in a general-purpose computing system, is an instantiation of a program with its own state. Since problems complex enough to require multiprocessors often run sophisticated algorithms and I/O systems, dividing the system into processes helps manage design complexity. A real-time operating system (RTOS) is an operating system specifically designed for embedded, and specifically real-time applications. The RTOS manages the processes and device drivers in the system, determining when each executes on the CPU. This function is termed scheduling. The partitioning of the software between application code which executes core algorithms and an RTOS which schedules the times to which those core algorithms are executed is a fundamental design principle in computing systems in general and is especially important for real-time operation.

There are a number of techniques which can be used to schedule processes in an embedded system—that is, to determine which process runs next on a particular CPU. Most RTOSs use process priorities in some form to determine the schedule. A process may be in any one of three states: currently executing (there can obviously be only one executing process on each CPU); ready to execute; or waiting. A process may not be able to execute until, for example, its data has arrived. Once its data arrives, it moves from waiting to ready. The scheduler chooses among the ready processes to determine which process runs next. In general, the RTOS's scheduler chooses the highest-priority ready process to run next; variations between scheduling methods depend in large part on the ways in which priorities are determined. Unlike general-purpose operating systems, RTOSs generally allow a process to run until it is preempted by a higher-priority process. General-purpose operating systems often perform time-slicing operations to maintain fair access of all the users on the system, but time-slicing does not allow the control required for meeting deadlines.

A fundamental result in real-time scheduling is known as rate-monotonic scheduling. This technique schedules a set of processes which run independently on a single CPU. Each process has its own period, with the deadline happening at the end of each period. There can be arbitrary relationships between the periods of the processes. It is assumed that data does not in general arrive at the beginning of the period, so there are no assumptions about when a process goes from waiting to ready within a period. This scheduling policy uses static priorities—the priorities for the processes are assigned before execution begins and do not change. It can be shown that the optimal priority assignment is based on period—the shorter the period, the higher the priority. This priority assignment ensures that all processes will meet their deadlines on every period. It can also be shown that at most, 69% of the CPU is used by this scheduling policy. The remaining cycles are spent waiting for activities to happen—since data arrival times are not known, it is not possible to utilize 100% of the CPU cycles.

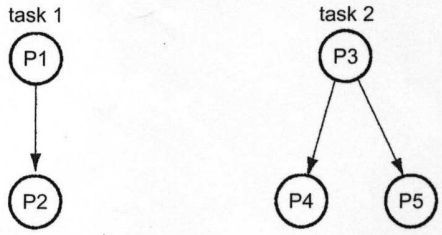

FIGURE 78.3 A task graph with two tasks and data dependencies between processes.

Another well-known, real-time scheduling technique is earliest deadline first (EDF). This is a dynamic priority scheme—process priorities change during execution. EDF sets priorities based on the impending deadlines, with the process whose deadline is closest in the future having the highest priority. Clearly, the rate of change of process priorities depends on the periods and deadlines. EDF can be shown to be able to utilize 100% of the CPU, but it does not guarantee that all deadlines will be met. Since priorities are dynamic, it is not possible in general to analyze whether the system will be overloaded at some point.

Processes may be specified with data dependencies, as shown in Figure 78.3, to create a task graph. An arc in the data dependency graph specifies that one process feeds data to another. The sink process cannot become ready until all the source processes have delivered their data. Processes which have no data dependency path between them are in separate tasks. Each task can run at its own rate. Data dependencies allow schedulers to make more efficient use of CPU resources. Since the source and sink processes of a data dependency cannot execute simultaneously, we can use that information to eliminate some combinations of processes which may want to run at the same time. Narrowing the scope of process conflicts allows us to more accurately predict how the CPU will be used.

A real-time operating system is often designed to have a small memory footprint, since embedded systems are more cost-sensitive than general-purpose computers. RTOSs are also designed to be more responsive in two different ways. First, they allow greater control over the order of execution of processes, which is critical for ensuring that deadlines are met. Second, they are designed to have lower context-switching overhead, since that overhead eats into the time available for meeting deadlines. The kernel of an RTOS is the basic set of functions that is always resident in memory. A basic RTOS may have an extremely small kernel of only a few hundred instructions. Such microkernels often provide only basic context-switching and scheduling facilities. More complex RTOSs may provide high-end operating system functions such as file systems and network support; many high-end RTOSs are POSIX (a Unix standard) compliant. While running such a high-end operating system requires more hardware resources, the extra features are useful in a number of situations. For example, a controller for a machine on a manufacturing line may use a network interface to talk to other machines on the factory floor or the factory coordination unit; it may also use the file system to access a database for the manufacturing process.

78.4 Hardware/Software Co-Design

Hardware/software co-design refers to any methodology which takes into account both hardware and software during the design of an embedded computing system. When the hardware and software are designed together, the designer has more opportunities to optimize the system by making tradeoffs between the hardware and software components. Good system designers intuitively perform co-design, but co-design methods are increasingly being embodied in computer-aided design (CAD) tools. We will discuss several aspects of co-design and co-design tools, including models of the design, co-simulation, performance analysis, and various methods for architectural co-synthesis. We will conclude with a look at design methodologies that make use of these phases of co-design.

P5

cies between processes.

lline first (EDF). This is a dynamic
priorities based on the impending
ıving the highest priority. Clearly,
deadlines. EDF can be shown to
t all deadlines will be met. Since
ıer the system will be overloaded

igure 78.3, to create a task graph
data to another. The sink proces
ıeir data. Processes which have no
sk can run at its own rate. Data
sources. Since the source and sink
ı use that information to eliminate
ne. Narrowing the scope of proces
sed.
emory footprint, since embedded
OSs are also designed to be more
he order of execution of processes
e designed to have lower context-
for meeting deadlines. The kerne
emory. A basic RTOS may have as
rokernels often provide only basic
may provide high-end operating
gh-end RTOSs are POSIX (a Unix
requires more hardware resources
e, a controller for a machine on a
ıchines on the factory floor or the
a database for the manufacturing

s into account both hardware and
ıen the hardware and software are
e the system by making tradeoffs
ners intuitively perform co-design
-aided design (CAD) tools. We will
odels of the design, co-simulation
ıesis. We will conclude with a look

78.4.1 Models

In designing embedded computing systems, we make use of several different types of models at different points in the design process. We need to model basic functionality. We must also capture nonfunctional requirements: speed, weight, power consumption, manufacturing cost, etc.

In the earliest stages of design, the task graph is an important modeling tool. The task graph does not capture all aspects of functionality, but it does describe the various rates at which computations must be performed and the expected degrees of parallelism available. This level of detail is often enough to make some important architectural decisions. A useful adjunct to the task graph are the technology description tables, which describe how processes can be implemented on the available components. One of the technology description tables describes basic properties of the processing elements, such as cost and basic power dissipation. A separate table describes how the processes may be implemented on the components, giving execution time (and perhaps other function-specific parameters like precise power consumption) on a processing element of that type. The technology description is more complex when ASICs can be used as processing elements, since many different ASICs at differing price/performance points can be designed for a given functionality, but the basic data still applies.

A more detailed description is given by either high-level language code (C, etc.) for software or hardware description language code (VHDL, Verilog, etc.) for software components. These should not be viewed as specifications—they are, in fact, quite detailed implementations. However, they do provide a level of abstraction above assembly language and gates and so can be valuable for analyzing performance, size, etc. The control-data flow graph (CDFG) is a typical representation of a high-level language: a flowchart-like structure describes the program's control, while data flow graphs describe the behavior within expressions and basic blocks.

78.4.2 Co-Simulation

Simulation is an important tool for design verification. The simulation of a complete embedded system entails modeling both the underlying hardware platform and the software executing on the CPUs. Some of the hardware must be simulated at a very fine level of detail—for example, buses and I/O devices may require gate-level simulation. On the other hand, the software can and should be executed at a higher level of abstraction. While it would be possible to simulate software execution by running a gate-level simulation of the CPU and modeling the program as residing in the memory of the simulated CPU, this would be unacceptably slow.

We can gain significant performance advantages by running different parts of the simulation at different levels of detail: elements of the hardware can be simulated in great detail, while software execution can be modeled much more directly. Basic functionality aspects of a high-level language program can be simulated by compiling the software on the computer on which the simulation executes, allowing those parts of the program to run at the native computer speed. Aspects of the program which deal with the hardware platform must interface to the section of the simulator which deals with the hardware. Those sections of the program are replaced by stubs which interface to the simulator. This style of simulation is a multi-rate simulation system, since the hardware and software simulation sections run at different rates: a single instruction in the software simulation will correspond to several clock cycles in the hardware simulation. The main jobs of the simulator are to keep the various sections of the simulation synchronized and to manage communication between the hardware and software components of the simulation.

78.4.3 Performance Analysis

Since performance is an important design goal in most embedded systems, both for overall throughput and for meeting deadlines, the analysis of the system to determine its speed of operation is an important element of any co-design methodology. System performance—the time it takes to execute a particular

aspect of the system's functionality—clearly depends both on the software being executed and the underlying hardware platform. While simulation is an important tool for performance analysis, it is not sufficient, since simulation does not determine the worst-case delays. Since the execution times of most programs are data-dependent, it is necessary to give the simulation of the program the proper set of inputs to observe worst-case delay. The number of possible input combinations makes it unlikely that one will find those worst-case inputs without the sort of analysis that is at the heart of performance analysis.

In general, performance analysis must be done at several different levels of abstraction. Given a single program, one can place an upper bound on the worst-case execution time of the program. However, since many embedded systems consist of multiple processes and device drivers, it is necessary to analyze how these programs interact with each other, a phase which makes use of the results of single-program performance analysis.

Determining the worst-case execution time of a single program can be broken into two subproblems: determining the longest execution path through the program and determining the execution time of that program. Since there is at least a rough correlation between the number of operations and the actual execution time, we can determine the longest execution path without detailed knowledge of the instructions being executed—the longest path depends primarily on the structure of conditionals and loops. One way to find the longest path through the program is to model the program as a control-flow graph and use network flow algorithms to solve the resulting system.

Once the longest path has been found, we need to look at the instructions executed along that path to determine the actual execution time. A simple model of the processor would assume that each instruction has a fixed execution time, independent of other factors such as the data values being operated on, surrounding instructions, or the path of execution. In fact, such simple models do not give adequate results for modern high-speed microprocessors. One problem is that in pipelined processors, the execution time of an instruction may depend on the sequence of instructions executed before it. An even greater cause of performance variations is caching, since the same instruction sequence can have variable execution times, depending on whether the code is in the cache. Since cache miss penalties are often 5X or 10X, the cost of mischaracterizing cache performance is significant. Assuming that the cache is never present gives a conservative estimate of worst-case execution time, but one that is so over-conservative that it distorts the entire design. Since the performance penalty for ignoring the cache is so large, it results in using a much faster, more expensive processor than is really necessary. The effects of caching can be taken into account during the path analysis of the program—path analysis can determine bound how often an instruction present in the cache.

There are two major effects which must be taken into account when analyzing multiple-process systems. The first is the effect of scheduling multiple processes and device drivers on a single CPU. This analysis is performed by a scheduling algorithm, which determines bounds on when programs can execute. Ratemonotonic analysis is the simplest form of scheduling analysis—the utilization factor given by ratemonotonic analysis tells one an upper limit on the amount of active CPU time. However, if data dependencies between processes are known, or some knowledge of the arrival times of data is known, then a more accurate performance estimate can be computed. If the system includes multiple processing elements, more sophisticated scheduling algorithms must be used, since the data arrival time for a process on one processing element may be determined by the time at which that datum is computed on another processing element.

The second effect which must be taken into account is interactions between processes in the cache. When several programs on a CPU share a cache, or when several processing elements share a second-level cache, the cache state depends on the behavior of all the programs. For example, when one process is suspended by the operating system and another process starts running, that process may knock the first program out of the cache. When the first process resumes execution, it will initially run more slowly, an effect which cannot be taken into account by analyzing the programs independently. This analysis clearly depends in part on the system schedule, since the interactions between processes depends on the order in which the processes execute. But the system scheduling analysis must also keep track of the

...ate—which parts of which programs are in the cache at the start of execution of each process. ...curacy can be obtained with a simple model which assumes that a program is either in the cache ...of it, without considering individual instructions; higher accuracy comes from breaking a process ...eral sub-processes for analysis, each of which can have its own cache state.

Hardware/Software Co-Synthesis

...are/software co-synthesis tries to simultaneously design the hardware and software for an embed- ...mputing system, given design requirements such as performance as well as a description of the ...nality. Co-synthesis generally concentrates on architectural design rather than detailed component ...—it concentrates on determining such major factors as the number and types of processing ...nts required and the ways in which software processes interact.

...most basic style of co-synthesis is known as hardware/software partitioning. As shown in ...78.4, this algorithm maps the given functionality onto a template architecture consisting of a CPU ...one or more ASICs communicating via the microprocessor bus. The functionality is usually specified ...single program. The partitioning algorithm breaks that program into pieces and allocates pieces ...to the CPU or ASICs for execution. Hardware/software partitioning assumes that total system ...rmance is dominated by a relatively small part of the application, so that implementing a small ...tion of the application in the ASIC leads to large performance gains. Less performance-critical sections ...the application are relegated to the CPU.

The first problem to be solved is how to break the application program into pieces; common techniques ...lude determining where I/O operations occur and concentrating on the basic blocks of inner loops. ...Once the application code is partitioned, various allocations of those components must be evaluated. ...Given an allocation of program components to the CPU or ASICs, performance analysis techniques can ...be used to determine the total system performance; performance analysis should take into account the ...time required to transfer necessary data into the ASIC and to extract the results of the computation from ...the ASIC. Since the total number of allocations is large, heuristics must be used to search the design ...space. In addition, the cost of the implementation must be determined. Since the CPU's cost is known ...in advance, that cost is determined by the ASIC cost, which varies as to the amount of hardware required ...to implement the desired function. High-level synthesis can be used to estimate both the performance ...and hardware cost of an ASIC which will be synthesized from a portion of the application program.

Basic, co-synthesis heuristics start from extreme initial solutions: We can either put all program compo- ...nents into the CPU, creating an implementation which is minimal cost but probably does not meet

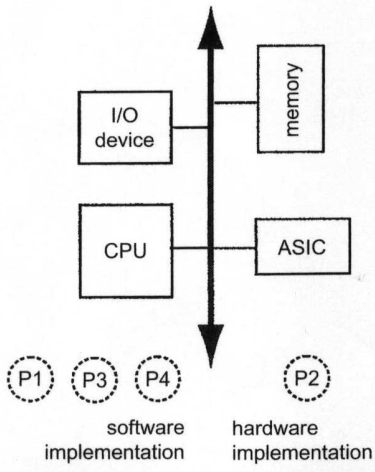

FIGURE 78.4 Hardware/software partitioning.

performance requirements, or put all program elements in the ASIC, which gives a maximal-performance, maximal-expense implementation. Given this initial solution, heuristics select which program components to move to the other side of the partition to either reduce hardware cost or increase performance, as desired. More sophisticated heuristics try to construct a solution by estimating how critical a component will be to overall system performance and choosing a CPU or ASIC implementation accordingly. Iterative improvement strategies may move components across the partition boundary to improve the design.

However, many embedded systems do not strictly follow the one CPU, one bus, n ASIC architectural template. These more general architectures are known as distributed embedded systems. Techniques for designing distributed embedded systems rest on the foundations of hardware/software partitioning, but they are generally more complicated, since there are more free variables. For example, since the number and types of CPUs is not known in advance, the co-synthesis algorithm must select them. If the number of busses or other communication links is not known in advance, those must be selected as well. Unfortunately, these decisions are all closely related. For example, the number of CPUs and ASICs required depends on the system schedule. The system schedule, in turn, depends on the execution time of each of the components on the available hardware elements. But those execution times depend on the processing elements available, which is what we are trying to determine in the first place. Co-synthesis algorithms generally try to fix several designs and vary only one or a few, then check the results of a design decision on the other parameters. For example, the algorithm may fix the hardware architecture and try to move processes to other processing elements to make more efficient use of the available hardware. Given that new configuration of processes, it may then try to reduce the cost of the hardware by eliminating unused processing elements or replacing a faster, more expensive processing element with a slower, cheaper one.

Since the memory hierarchy is a significant contributor to overall system performance, the design of the caching system is an important aspect of distributed system co-synthesis. In a board-level system with existing microprocessors, the sizes of second-level caches is under designer control, even if the first-level cache is incorporated on the microprocessor and therefore fixed in size. In a single-chip embedded system, the designer has control over the sizes of all the caches. Co-synthesis can determine hardware elements such as the placement of caches in the hardware architecture and the size of each cache. It can also determine software attributes such as the placement of each program in the cache. The placement of a program in the cache is determined by the addresses used by the program—by relocating the program, the cache behavior of the program can be changed. Memory system design requires calculating the cache state when constructing the system schedule and using the cache state as one of the factors to determine how to modify the design.

78.4.5 Design Methodologies

A co-design methodology tries to take into account aspects of hardware and software during all phases of design. At some point in the design process, the hardware and software components are well-specified and can be designed relatively independently. But it is important to consider the characteristics of both the hardware and software components early in design. It is also important to properly test the system once the hardware and software components are assembled into a complete system.

Co-synthesis can be used as a design planning tool, even if it is not used to generate a complete system architectural design. Because co-synthesis can evaluate a large number of designs very quickly, it can determine the feasibility of a proposed system much faster than a human designer. This allows the designer to experiment with what-if scenarios, such as adding new features or speculating on the effects of lower component costs in the future. Many co-synthesis algorithms can be applied without having a complete program to use as a specification. If the system can be specified to the level of processes with some estimate of the computation time required for each process, then useful information about architectural feasibility can be generated by co-synthesis.

Co-simulation plays a major role once subsystem designs are available. It does not have to wait until all components are complete, since stubs may be created to provide minimal functionality for incomplete components. The ability to simulate the software before completing the hardware is a major boon to software development and can substantially reduce development time.