Hardware–Software Co-Design of Embedded Systems

WAYNE H. WOLF, SENIOR MEMBER, IEEE

Invited Paper

This paper surveys the design of embedded computer systems, which use software running on programmable computers to implement system functions. Creating an embedded computer system which meets its performance, cost, and design time goals is a hardware–software co-design problem—the design of the hardware and software components influence each other. This paper emphasizes a historical approach to show the relationships between well-understood design problems and the as-yet unsolved problems in co-design. We describe the relationship between hardware and software architecture in the early stages of embedded system design. We describe analysis techniques for hardware and software relevant to the architectural choices required for hardware–software co-design. We also describe design and synthesis techniques for co-design and related problems.

I. Introduction

This paper surveys the state of the art in the design of **embedded computer systems** products which are implemented using programmable instruction-set processors. While embedded systems range from microwave ovens to aircraft-control systems, there are design techniques common to these disparate applications. Furthermore, embedded system design often requires techniques somewhat different than those used for either the design of general-purpose computers or application software running on those machines. Embedded computing is unique because it is a **hardware–software co-design** problem—the hardware and software must be designed together to make sure that the implementation not only functions properly but also meets performance, cost, and reliability goals.

While a great deal of research has addressed design methods for software and for hardware, not as much is known about the joint design of hardware and software. Microprocessors, and in particular high-performance 32-bit microprocessors cheap enough to use in consumer products, have stimulated research in co-design for embedded systems. So long as embedded processors were small and executed only a few hundred bytes of code, hand-crafted techniques were sufficient to satisfy functional and performance goals in a reasonable amount of time. However, modern embedded systems may include megabytes of code and run at high speeds to meet tight performance deadlines. In such large projects, building a machine and seeing whether it works is no longer satisfactory. To be able to continue to make use of the ever-higher performance CPU's made possible by Moore's Law (which predicts that the number of transistors per chip doubles every year), we must develop new design methodologies and algorithms which allow designers to **predict** implementation costs, **incrementally refine** a design over **multiple levels of abstraction**, and create a **working first implementation.**

We will use the embedded system design process as a framework for the study of co-design. The goal of this paper is to identify technologies which are important to co-design and to provide examples which illustrate their role in co-design. We must, due to space limitations, ignore certain topics, such as the design of fault-tolerant systems and verification. Our description of the literature on covered topics is also meant to be illustrative, not encyclopedic. In spite of these limitations, we hope that the juxtaposition of topics presented here will help to illustrate both what is known about co-design and what remains to be done.

The next section surveys the uses of embedded computers and the embedded system design process. Section III describes performance analysis of hardware and software elements. Section IV surveys techniques for the design of hardware–software systems.

II. Embedded Systems and System Design

A. Characteristics of Embedded Systems

The earliest embedded systems were banking and transaction processing systems running on mainframes and arrays of disks. The design of such a system entails hardware–software co-design: given the expected number and type of transactions to be made in a day, a hardware configuration must be chosen that will support the expected traffic and a software design must be created to efficiently

Manuscript received December 29, 1993; revised April 4, 1994.
The author is with the Department of Electrical Engineering, Princeton University, Princeton, NJ 08544 USA.
IEEE Log Number 9402008.

make use of that hardware. Because early transaction processing systems were built from very expensive equipment, they were used for relatively few but important applications. However, the advent of microprocessors has made the average embedded system very inexpensive, pushing microprocessor-based embedded systems into many new application areas. When computers were used for only a few types of applications, design techniques could be developed specific to those applications. Cusumano [16] documented the labor-intensive techniques used by Japanese computer system manufacturers to build mainframe- and minicomputer-based systems for industrial automation, banking, and other capital-intensive applications. When the hardware is expensive, it is easier to justify large personnel budgets to design, maintain, and upgrade embedded software. When microprocessors are used to create specialized, low-cost products, engineering costs must be reduced to a level commensurate with the cost of the underlying hardware. Now that microprocessors are used in so many different areas, we need a science of embedded system design which can be applied to previously unforeseen application areas.

Because microprocessors can be used in such a wide range of products, embedded systems may need to meet widely divergent criteria. Examples of embedded systems include:

- simple appliances, such as microwave ovens, where the microprocessor provides a friendly interface and advanced features;
- an appliance for a computationally intensive task, such as laser printing;
- a hand-held device, such as a cellular phone, for which power consumption and size are critical but digital signal processing and other sophisticated tasks must be performed;
- an industrial controller in a factory, for which reliability, maintainability, and ease of programmability are often concerns;
- a safety-critical controller, such as an anti-lock brake controller in a car or an autopilot.

Most readers would agree that each of these examples is an embedded computing system, but a comprehensive definition of embedded computing has not yet achieved wide acceptance. There are clearly examples which may or may not fit varying definitions of a system. For example, many industrial and scientific control applications are implemented on PC's. Since these applications are dedicated, many (though not all) would consider such systems embedded. But is a PC used solely to run a spreadsheet in an embedded computing system? Is a personal digital assistant (PDA) which uses the same microprocessor and runs the same spreadsheet software an embedded computer? It is difficult to come up with a simple definition which meets everyone's intuitive notion of embedded computing.

Different applications place primary importance on different factors: design time, manufacturing cost, modifiability, reliability, etc. What embedded systems share is a belief by the designers that implementing some of the system's functions on microprocessors will make one or more of those goals easier to achieve. One lurking problem with any kind of software design that also holds for embedded systems is the desire, well documented by Brooks [7], to add features at the expense of schedule and design elegance. In addition, embedded systems have added problems due to their design constraints. Designing code to meet a performance deadline or squeezing code into the given amount of ROM can be very difficult without a well-understood design methodology to help guide decisions. The design of embedded systems is not as well understood as the design of integrated circuits, which have several methodologies for different cost–performance tradeoffs—sea-of-gates, standard cell, full-custom—and design tools for the phases of design in each methodology. While embedded system designers can make use of existing tools for the hardware and software components once the design has been partitioned, much remains to be learned about how a system is partitioned into hardware and software components. Methodologies and tools for hardware–software co-design are critical research topics for embedded system design.

Hardware–software co-design of embedded systems must be performed at several different levels of abstraction, but the highest levels of abstraction in co-design are more abstract than the typical software coder or ASIC designer may be used to. Critical architectural decisions are made using abstract hardware and software elements: CPU's and memories in hardware, processes in software. As a result, the initial hardware and software design problems are high-level: the first hardware design decision is to build a network of CPU's, memories, and peripheral devices; the first software design problem is to divide the necessary functions into communicating processes. At first blush, a hardware designer in particular may not consider CPU selection to be true hardware design. For example, the major hardware architectural decision may be to choose between a 386-based or 486-based PC. However, that task is not so different from the design choices faced by VLSI designers. A chip designer does not design the threshold voltage, transconductance, and other transistor parameters to suit a particular application—rather, digital logic design requires choosing a circuit topology and computing transistor W/L's. The typical ASIC designer will not deal with transistors at all, but will choose logic gates from a library and wire them together to implement the desired function. Whether the components to be selected and interconnected are logic gates or CPU's, the designer faces the same problem: **characterizing** the components; understanding the **operation of networks** of components; and choosing a **network topology** based on the requirements.

Embedded system design can be divided into four major tasks:

- **partitioning** the function to be implemented into smaller, interacting pieces;
- **allocating** those partitions to microprocessors or other hardware units, where the function may be imple-

mented directly in hardware or in software running on a microprocessor;

- **scheduling** the times at which functions are executed, which is important when several functional partitions share one hardware unit;
- **mapping** a generic functional description into an implementation on a particular set of components, either as software suitable for a given microprocessor or logic which can be implemented from the given hardware libraries.

(This taxonomy is similar to that given by McFarland *et al.* for high-level synthesis [73] with the exception of adding partitioning as a first-class design problem.) The design goals in each task depend on the application: performance, manufacturing cost, testability, etc. The solutions to these problems clearly interact: the available choices for scheduling are controlled by how the design was partitioned, and so on. To make matters worse, not only can each of these steps be applied to the software and hardware components separately, but also to the division into hardware and software components itself, and the design decisions made for the hardware and software components separately interact with the co-design problem. We will frame our discussion of co-design techniques by reference to the partitioning, allocation, scheduling, and mapping steps. Mapping is the least understood part of co-design: while it is often possible to estimate the overall amount of computation required to complete a task, it is much more difficult to determine whether a particular hardware structure and software organization will perform the task on time.

Several disciplines help form the basis of embedded system design. Software engineering and VLSI computer-aided design (CAD) provide implementation techniques for the software and hardware components of the system, and those techniques may be useful during co-design as well. Because many embedded systems are implemented as networks of communicating microprocessors, distributed system design is an important foundation for co-design. Real-time system design is another critical foundation since many embedded systems include performance constraints as part of their requirements. Real-time systems are usually divided into **hard** real-time, for which failure to complete a computation by a given **deadline** causes catastrophic system failure, and **soft** real-time, where performance is important but missing a deadline does not cause the system to fail. A clear example of a hard real-time system is an autopilot, where failure to compute a control command from a given control input in a certain interval causes the airplane to go out of control. A laser printer is an example of a machine with soft performance constraints while the user bought the system based in part on its pages-per-minute rating, the rate at which the printer actually typesets pages can vary without causing the machine or the customer physical harm. The control of the print engine within the laser printer is, however, a hard real-time task—data must be delivered to the print drum at specified times and rates or the printed image will be destroyed. Many embedded systems have at least a few hard real-time constraints, derived from deadlines imposed by the operation of peripherals.

B. Embedded Processors and Software Architectures

Any **central processing unit (CPU)** may be used in an embedded computer system. A CPU whose design is optimized for embedded applications is called an **embedded processor.** Embedded processors may be compatible with workstation CPU's or may have been designed primarily for embedded applications. Many embedded processors do not include memory management units; the structure of the application software makes a memory management unit less useful and the chip area occupied by that logic can be put to better uses. An embedded processor optimized for digital signal processing is called a **digital signal processor (DSP).**

An **embedded controller** or **microcontroller** devotes on-chip area to peripherals commonly used in embedded systems. Embedded controllers add peripheral devices such as timers, analog-to-digital converters, or universal synchronous/asynchronous receiver transmitters (USART's) to the core CPU. Timers are used to count events, to measure external time, and to measure the length of time slices for process scheduling. A serial port like a USART may be used to control some simple devices, may be used for debugging, or may be used to communicate with other microcontrollers. Most 4- and 8-bit microcontrollers include on-board **random-access memory (RAM)** and some sort of **read-only memory (ROM),** but 16- and 32-bit embedded controllers may not include on-chip ROM. The amount of memory available on a microcontroller is usually small: 256 bytes of RAM and 1024 bytes of ROM is a common configuration for an 8-bit machine. Some applications have been forced to move to 16-bit embedded controllers not because of data size, but rather because the application code could not fit into an 8-bit controller's address space.

Some microprocessor manufacturers provide embedded controller design and manufacturing: a customer may design a chip using a microprocessor core, standard peripherals, and standard cell logic; the controller is then manufactured in quantity for the customer. Customers in this market niche must have a large enough demand for the custom controller to justify the design and added manufacturing costs. An alternative approach to the design of customer-specific microcontrollers has been proposed by a number of people, though we do not know that such a chip has yet been manufactured: a microcontroller with an on-board RAM-based **field-programmable gate array (FPGA)** would allow the customer to add custom peripherals by downloading a personality to the FPGA.

An **application-specific processor (ASIP)** is a CPU optimized for a particular application. A DSP is an example of an ASIP, though ASIP is generally used to refer to processors targeted to application niches much narrower than the audio-rate signal processing market. Paulin pointed out that some telecommunications applications require parts in hundreds of thousands to millions, making the gain in performance and reduction in cost provided by an ASIP

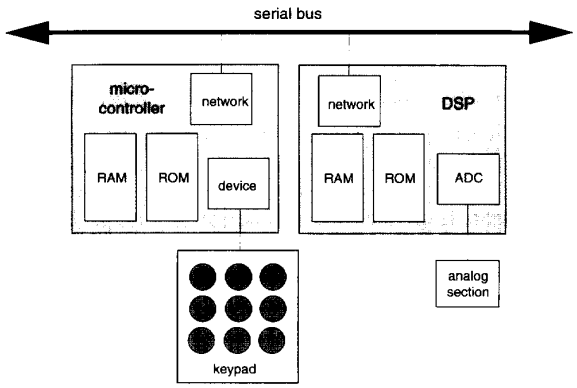

Fig. 1. An embedded system with distributed computing for signal processing and user interface.

worth the added design effort [74]. Most ASIP's available today were designed by CPU manufacturers for niche markets: the laser printer market has attracted several processors, such as the AMD 29000; the Motorola MC68302 is optimized for execution of telecommunication protocol code, such as ISDN. A CPU manufacturer may optimize a design in several ways: the memory bus is often tuned for laser printer applications; peripherals may be added to support special tasks; or, as in the case of FFT-related instructions for DSP's, application-specific instructions may be added. There is growing interest in design tools for ASIP's which can select an instruction set appropriate to the application.

Many embedded systems are implemented as **distributed systems**, with code running in multiple processes on several processors, with **interprocessor communication (IPC)** links between the CPU's. For example: most marine and general aviation navigation devices communicate via RS-232 serial lines; modern automobiles contain many microprocessors physically distributed throughout the car which communicate with each other to coordinate their work; cellular telephones today generally include at least one general-purpose microcontroller and an embedded DSP, and some telephones are built from a half-dozen embedded processors. Figure 1 illustrates a hypothetical distributed system which is used to implement a machine which must perform both signal processing and user interaction. A DSP is used to implement the signal processing functions, while an 8-bit microcontroller handles the user interface. Each microprocessor has on-board peripheral interfaces, an analog-to-digital converter on the DSP, and a parallel port on the microcontroller, which are used in this application. Since each microprocessor has on-board memory and only low-speed communication is required between the signal processing and interface tasks, a serial interface is used to connect the two CPU's.

A distributed system may be the best implementation of an embedded computer for any of several reasons:

- Time-critical tasks may be placed on different CPU's to ensure that all their hard deadlines are met. In the above example, sampling the keyboard might interfere with the DSP process if both were on the same CPU.
- Using several small CPU's may be cheaper than using one large CPU. In many cases, several 8-bit controllers can be purchased for the price of one 32-bit processor. Even after board real-estate costs are added in, distributing a task over several small CPU's may be cheaper than combining all the tasks onto one processor.
- Many embedded computing systems require a large number of devices. Using several microcontrollers with on-board devices may be the cheapest way to implement all the device interfaces.
- If the system includes a subsystem purchased from a supplier, that subsystem may include its own CPU. The subsystem's CPU will impose a communications interface, but it is usually not possible to move other system tasks onto the subsystem's processor.

Rosebrugh and Kwang described the design of a pen-based computer which was implemented as a distributed system [81]. Their device had to mix time-sensitive input/output operations, such as tracing the pen on the screen, with computer-bound tasks such as updating its internal database. Their design used four microprocessors: a Motorola MC68331, a 68000-family processor, as the core processor; a Motorola MC68HC05C4, an 8-bit microcontroller, for power management; a Hitachi 63484 for graphics; and an Intel 80C51, another 8-bit microcontroller, for the pen digitizer. The processors were connected heterogeneously: both the MC68331 and 63484 were connected to main memory, but the 63484 graphics processor was the only device connected to the display memory; the 80C51 was connected to the digitizer and to the 68HC05, while the 68HC05 talked to the MC68331.

A variety of interconnect schemes are used in distributed embedded systems. The processor bus may be used for simple systems—all processors not only share common memory, they also contend for the bus to access memory and devices. Co-processors like floating-point units often have their own dedicated link to the CPU. Many embedded controllers use serial links, either RS-232 or special-purpose links, between their CPU's. The I^2C bus [89] is a serial communications system popular in 8-bit distributed controllers: it requires two wires, provides multiple bus masters, and can run at up to 400 kb/s. SAE-J1850 is an emerging standard for automotive communication networks. The Echelon Neuron architecture also uses a medium-performance serial link between the processors. Bandwidth limitations on the commonly used interprocessor communication systems make process allocation very important to ensure that deadlines are met in the face of IPC delays.

Embedded software can usually be thought of as a system of communicating **processes**, though the underlying code may not have clearly defined processes. A process is an instance of a sequential machine, which we will use to refer to either a hardware or a software implementation. **Task** is a synonym for process; we will use the two

terms interchangeably, usually choosing the word used by the author whose work we describe. The term **thread** or **lightweight process** is used by some authors; a thread is usually used to describe a process which shares its memory space with other threads, rather than assuming that each process has its own address space.

As will be described in more detail in Section III-D, software processes can be implemented in several different ways: using preemptive scheduling, as in time-sharing systems; through nonpreemptive scheduling, in which a process voluntarily passes control to the next process; as a cyclostatic machine, which periodically executes a fixed sequence of operations; and as an interrupt-driven system. The software architecture appropriate for a task depends in part on the match between the CPU chosen, particularly the speed at which the CPU can switch between processes, and the performance requirements of the application.

C. The Engine Metaphor

If an embedded system is thought of as a jumble of microprocessors and code, it can be difficult to discern the structure which leads to a successful architecture. The *engine metaphor* helps us understand the roles hardware and software play in the implementation of an embedded computer system. While the designs of the hardware and software components clearly interact with each other, establishing the roles that hardware and software play in the system is critical to developing a methodology which manages the design process and categorizes the design interactions to guide the designer toward a satisfactory solution.

Our model of an embedded computer systems is a **hardware engine** which runs **application software**, as shown in Fig. 2. The engine includes the one or more CPU's and memory as well as peripherals. The engine provides the raw computing power for the system both instruction execution and peripheral operations. Most of the features of the system, however, are not directly implemented in the hardware but are instead designed into the application software. Viewing the hardware as the engine which provides the power for the application software's features helps us decide whether to solve a particular design problem by attacking the software or hardware.

Hardware engine design for embedded systems is reminiscent of engine selection for vehicles such as automobiles or airplanes. The engine is selected very early in the design of a motor vehicle [98]. While the mission requirements (gross weight, maximum speed) determine a horsepower range for the engine, the particular engine to be used is selected from a small set of available engines which meet the requirements. Once the engine has been selected, its particular characteristics—exact horsepower, torque, weight, shape, cooling requirements, etc.—constrain the design of the vehicle. Similarly, the CPU for a hardware engine must be selected from among the available processors. The characteristics of that processor—execution speed of various instructions, bus throughput, etc.—help determine the design of the software which runs on the engine.

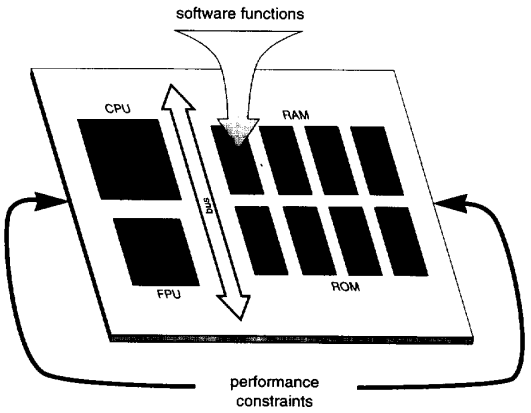

Fig. 2. A hardware engine.

The design of the **software architecture**—the division of the function into communicating processes—is closely related to engine design. Two systems with identical functions but different process structures may run at very different speeds, require vastly different amounts of memory, etc. For example, in one case, dividing one process into two may increase the amount of concurrency exposed in the system and reduce the execution time required; in another case, dividing a task into too many processes may introduce too many context switches and reduce performance. You cannot choose a hardware engine without also choosing a software architecture, since the software's process structure help determine size and speed. Similarly, a vehicle's body and aerodynamics are closely related to the choice of engine: a narrow airplane cannot accommodate a wide engine; the classic Bugatti limousines have long, elegant shapes in large part to accommodate the straight 16 cylinder engines underneath their hoods. Embedded computer system design is hardware–software co-design precisely because system architecture design must simultaneously consider the hardware engine and the software process structure.

Performance constraints, both general throughput requirements (such as average page printing rate for a laser printer) and hard real-time deadlines, determine the minimum-size hardware engine needed for the application. The designer's job is to choose an engine which is large enough to meet the application's performance demands, which is no more costly than necessary (why pay for more horsepower than you need?), and also satisfies the other nonfunctional requirements like physical size, power consumption, etc. Performance constraints for an embedded system play the role of mission requirements in vehicle design. In the absence of performance constraints, any hardware engine will do. It is performance constraints, particularly hard real-time deadlines, which determine the basic requirements of the hardware engine.

However, unlike in vehicle design, we do not at present have simple rules-of-thumb to relate embedded computer mission requirements, analogous to maximum gross weight in vehicle design, to a simple measure of processor performance like an internal combustion engine's horsepower.

Benchmarks such as the SPEC benchmark set provide some means to measure processing power, but it is often difficult to extrapolate from benchmark performance to the execution time for the application at hand. If we had such performance prediction rules, they would almost certainly be domain-specific, just as the back-of-the-envelope calculations for automobile and aircraft design are very different.

It is likely that today's large number of choices in CPU's for embedded applications is a historical anomaly. In the future, it is likely that one or two CPU's will be available for each performance/feature regime, much as vehicle designers today have a limited selection of engines available to them. Today, VLSI technology is advancing and embedded computing markets are growing, so semiconductor manufacturers are still incented to invest the large sums required to design new CPU's—it is still possible to gain production volume by growing with the market. When VLSI technology and its semiconductor markets mature, manufacturers will probably find CPU design to be too expensive to be justified simply to take market share away from another manufacturer. The difficulty of developing efficient compilers and their associated development environments for new processors adds another barrier to entry for new CPU's. There will always be opportunities for customized CPU's, either offered by manufacturers for particular market segments or designed for a particular application by a customer. As with internal combustion engines, however, simple design changes can be made cheaply but some kinds of engine redesigns require large investments in engineering. In that steady-state condition, distributed system design will probably become even more important, as system designers try to compose an engine which meets their requirements from a collection of interconnected CPU's.

The most general way to estimate the required size of an embedded hardware engine by performing an initial synthesis of both the hardware and software subsystems. By choosing one or more processor types and dividing up the software tasks among n such processors, we can determine whether the given architecture can meet its deadlines and indicate where an application-specific co-processor will be required. As we will see later in this paper, many techniques have been developed for mapping a functional specification onto a given hardware engine with constraints, but less is known about the design of the hardware engine itself. Even less is known about the joint optimization of the application software and the hardware engine.

D. Design Flow

The design process of an embedded system must vary considerably with the application: the design of a pager is very different from the design of an autopilot. However, we can identify common steps. Furthermore, a study of a typical design flow shows that the hardware and software components of an embedded system have common abstractions, a fact which we can use to our advantage in hardware–software co-design.

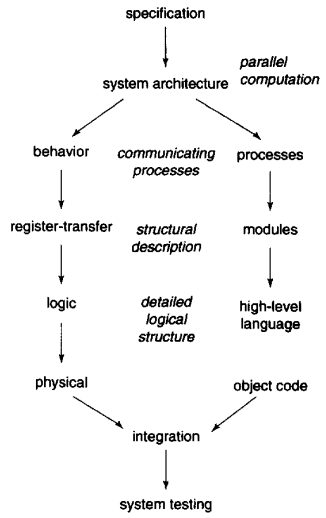

Fig. 3. A top-down design process for an embedded system.

Figure 3 shows a typical sequence of steps in a top-down design of an embedded system; of course, most actual design processes will mix top-down and bottom-up design. For comparison, Smailagic and Siewiorek describe a concurrent design methodology used to design VuMan 2, a portable smart display device [90]. Design of the hardware and software components must proceed fairly separately at some point, just as two hardware components must be designed separately once the system has been broken down into a system of components. However, as the figure shows, the hardware and software tracks refine the design through similar levels of abstraction. Processes are used to represent both the hardware and software elements of the initial partition: a system of hardware components in a block diagram is equivalent to a system of communicating processes; and the interprocess communication links in a software specification correspond to signals in a block diagram. The fact that both hardware and software are specified as processes gives us hope that we can use a common modeling technique to simultaneously design both the hardware engine and the application code to meet performance constraints. It is because hardware and software have related abstractions that co-design can make architectural choices which balance the design problems of each.

Design starts with the creation of a **specification** (also known as **requirements** or **requirements specification**). A system specification includes not only **functional requirements**—the operations to be performed by the system but also **nonfunctional requirements**, including speed, power, and manufacturing cost. The importance of the specification should not be underestimated—most systems have at least some predefined requirements or goals, such as a target page-per-minute production and maximum sale price for a laser printer. Books by Davis [21] and Dorfman and Thayer [23] give more information on system specification techniques and standards.

During the writing of the specification and the design of the initial system architecture, phases which often overlap, the system architects must determine that the design goals are in fact feasible. Feasibility checks are simpler in the design of **application-specific IC's (ASIC's)** because the chip to be designed fits into a predefined system and performs a small enough function that it can be specified with confidence. In contrast, embedded computer systems usually employ microprocessors precisely to add a variety of sophisticated features which, in turn, complicate the specification of the system and validity tests. And because embedded software is often used to decrease design turnaround time, some decisions on the hardware engine must be made from guesses as to the ultimate function to be implemented. An extreme example [3] is offered by several avionics manufacturers: even before the Federal Aviation administration had defined the technical standards for new GPS-based navigation services, these manufacturers guaranteed to purchasers of current-model radios a cap on their cost to upgrade the radios to the new standards. In many other cases, manufacturers design a single hardware engine which is used both for several products at a time and several generations of products; since the exact features of successive generations cannot be precisely predicted due to competitive pressure, it may be necessary to design the hardware engine architecture using only guesses as to the resources required by future features. Embedded system design methodologies must be able to support incomplete specifications, design of a single engine to satisfy multiple product specifications, or changes to the specs during design.

Requirements specification inherently deals with descriptions which are too informal to be defined mathematically—most customers describe their requirements in English that is often incomplete and inconsistent. A great deal of work has been done on system and software specification in general, and some of that work has concentrated on the design of real-time systems. Hatley–Pirbahi analysis [36] is a well-known technique for the design of real-time systems. A system is described in terms of two models: a system requirements model and a system architecture model. These two models are jointly refined in a **spiral development cycle**: once initial requirements have been given, an initial architecture is proposed; analysis of the architecture suggests refinements to the architecture, which in turn suggest changes to the architecture; and so on until the requirements are well-understood and an implementable architecture has been identified. A **requirements model** consists of a data flow diagram, a control flow diagram, response time specifications, and a **requirements dictionary**. (A dictionary in software specification is similar to a common dictionary—it lists definitional information, such as type, references, and so on, for elements of the specification.) The architecture model includes an architecture flow diagram which allocates functional elements of the requirements model to physical units in the architecture, an architecture interconnect diagram (a block diagram), and a dictionary. Hatley–Pirbahi analysis is intended to be used first to define the complete system, then to refine the software and hardware sections of the system.

Shlaer–Mellor analysis [88] is an object-oriented approach to real-time system specification, where an object is a data structure plus a set of member functions which operate on that data structure. In object-oriented designs, functions are performed not by executing monolithic functions on large data structures, but rather by member functions updating their objects and calling member functions of other objects. Shlaer–Mellor analysis views objects as state machines: the values in the object's data structure define its state while its member functions can update the object's state and produce outputs based on that state. The behavior of an object can be described as an ASM-like state transition graph. Analysis helps the designer identify which objects make up the system, the behavior of each object, and how objects communicate to implement system behavior.

It is critical that specification not bias implementation. If the specification method is too operational, it will contain an implicit or explicit architectural model. The architectural model which is best suited to specification may not be the most efficient implementation. D'Anniballe and Koopman [20] studied techniques bias-free specification of distributed systems. They developed a specification methodology which mixed object-oriented analysis with a set-theoretic formal specification of the behavior of the objects. This technique allowed them to better separate the specification of function from the allocation of those operations to architectural components.

After the architectural decisions have been made, hardware and software design can proceed somewhat separately; if proper co-design has selected a good architecture, then components designed to the architectural specifications can be put together to build a satisfactory system. Hardware design proceeds through several steps: a description of behavior, which may include communicating machines and in which operations are only partially scheduled in time; a register-transfer design, which gives combinational logic functions between registers but not the details of logic design; the logic design itself; the physical design of an integrated circuit, placement and routing in a field-programmable logic device, etc. Software design starts with a set of communicating processes since most embedded systems have temporal behavior which is best expressed as a concurrent system; the decomposition of function into modules is an intermediate step in software design, which proceeds to coding in some combination of assembly and high-level languages. The software and hardware components must be integrated and then tested to be sure that the system meets its specifications.

The separation of design tasks into concurrent design of hardware and software components highlights the importance of early selection of a hardware engine and an accompanying software architecture. The architectural choices made early in the design process guide the detailed implementation choices for the hardware and software components. As in any complex design, the architect must make key choices early, looking ahead to possible implementation

problems, but without completing a full implementation. When an aircraft designer selects an engine for a new airplane, he or she does not need to design the arrangement of rivets on the airplane's tail to determine the horsepower requirements of the new engine; such detailed implementation decisions would only be an abstraction. The detailed design of the airframe must, however, be completed in a way that does not violate the assumptions about gross weight, drag, and other factors that were used to determine the engine requirements. In an embedded computing system, the implementation of the software processes and the hardware components must be consistent with the assumptions made about computing loads.

Once the hardware and software components have been implemented, they must be separately tested, integrated, and tested again. Unfortunately, the hardware and software design communities use the word **testing** very differently: hardware designers use it to mean **manufacturing testing**, or tests which ensure that each manufactured copy of a component was correctly manufactured; software designers use it to mean system **validation**, or ensuring that the design meets the specification. (We prefer to reserve the word verification for mathematical techniques which give proofs of the correctness of certain system properties, leaving validation for informal techniques which give reasonable assurance but fall short of proofs.) Both forms of testing are necessary: the hardware and software elements must be executed together to ensure that the system satisfies its specification; and each copy must be tested for defects as it comes off the manufacturing line. Manufacturing test of embedded systems does not introduce major new problems, since the hardware can be tested independently of the software and the integrity of ROM code can be easily checked. Design validation of the integrated hardware–software system does, however introduce some problems.

The first problem to be considered is that software development must rely as little as possible on the completion schedule of the hardware design. Design methodologies in which the software designers must wait for the hardware so that developers can execute their code result in unacceptably long development times. More important, such a serial design methodology ensures that the hardware engine will have design flaws which introduce software performance problems. Hardware designers (or synthesis tools) must be able to make use of the results of a more detailed software design to refine the design of the engine and conversely, the software implementation can be affected by the details of the hardware engine. It may be possible to develop the code on a completely different platform, such as a personal computer or workstation. In other cases, it may be necessary to use the target processor but to use a standard development system in place of the final hardware engine.

Once the first versions of the engine and application code are available, integration tests can begin. Integration testing must check for both functional bugs and performance bottlenecks. As mentioned in Section III-B, the events in an embedded computer system are not always easy to observe. It may not be possible to gather large enough traces to determine detailed system behavior; on cached machines, behavior may not be externally visible. Many modern microprocessors provide hooks for tracing during execution, which are useful in functional debugging but may not be feasible for systems which must be run at full speed. Sampling is often used to generate approximate performance measures: the address bus can be sampled periodically to generate histograms of address execution rates; the kernel's scheduler can be used to generate a trace of active processes to determine how frequently each process was executed; counters in the hardware engine can be programmed as event counters to measure certain performance statistics by adding a small amount of measurement code to the system.

III. Performance Analysis

Soft and hard performance goals are essential parts of the specifications of most embedded systems. Performance analysis is also critical to cost minimization—the usual approach to designing a time-critical system in the absence of accurate performance analysis is to overdesign the hardware engine, producing a system that is more expensive than may be necessary. As a result, performance analysis is critical at all stages of design. This section describes two categories of performance analysis methods: those used during requirements analysis and architecture design; and those used to measure the performance of software. We will not consider here how to determine the cycle time of hardware components, which is a relatively well-understood problem whose outlines can be found elsewhere [100].

Malik and Wolfe [64] argue that embedded system performance analysis requires solving two problems: modeling the underlying hardware engine and analyzing the behavior of the code running on that engine. Furthermore, performance estimation tools are required at each level of abstraction through which the design proceeds. This section considers performance analysis of large systems in Section III-A, modeling of CPU performance in Section III-B, performance of a single task in Section III-C, performance of multiple tasks running on a shared processor in Section III-D, and co-simulation in Section III-E.

A. System Performance Analysis

The goal of system performance analysis is to translate performance specifications, which are typically given on user-level functions, into constraints on the design of the hardware engine and the application software. System performance analysis includes several tasks: determining the implications of performance specifications; estimating the hardware costs of meeting performance constraints; identifying key development bottlenecks; and estimating development time. System performance analysis is not necessary for most ASIC's due to the simple form of the performance constraints. It is, however, a necessary step in the design of a modern CPU—the sizes of queues and buffers, the number of hardware resources available, and the interconnections between those resources all determine the

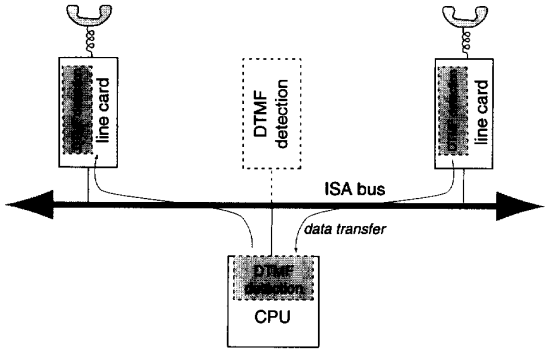

Fig. 4. Data transfers and tone detection options in TigerSwitch.

performance of the CPU. Similarly, memory, interconnect, and function units (in the form of CPU's or special-purpose processors) all influence the overall performance of an embedded system.

An example helps to show the role of system performance analysis in the design of an embedded computer system. TigerSwitch is a PC-based telephone switching system designed at Princeton University. As shown in Fig. 4, the PC system bus (known as the ISA bus) serves as the switching fabric. Line cards connect telephone lines to the switch; an analog–digital converter connects to the telephone microphone, while a digital–analog converter connects to the speaker. During a call between two phone lines, the PC first reads the current microphone sample from one line, then writes it to the other line; it then reverses the procedure to provide a full-duplex connection. Each call must be sampled at 8 kHz, the sampling rate for telephone-quality audio.

Several factors influence the number of phone lines which can be supported by the switch. The ISA bus bandwidth certainly limits the number of calls which can be switched. However, bus bandwidth divided by sampling rate is only an upper bound on phone line capacity, since some bus operations are required for execution of the program which controls the switch. Furthermore, the CPU itself is used both to implement the switching fabric and for other switching functions: an 8-kHz timer interrupts to switch data between all active lines by executing I/O instructions on the CPU; a foreground process keeps track of call state; other processes determine how to route calls and bill time as well as other functions.

The function which caused the most concern in the design of TigerSwitch was tone detection. Dialing tones (known as DTMF, for dual-tone multifrequency) must be detected at the start of the call: each digit on the phone keypad is signaled by a pair of tones, where each row and each column on the keypad has its own distinct tone frequency. DTMF detection can be performed by a filter bank or by Fourier analysis. Because the two tones must be sustained for at least 0.1 s to make a valid signal, tone detection requires a great deal of computation.

As shown in Fig. 4, we had three possible locations for DTMF detection in the architecture: on each line card, using analog DTMF detectors; on the main CPU, as a background process, using digital signal processing algorithms; and on an auxiliary processor, using the same DSP algorithms, on a card plugged into the ISA bus. In most switching systems, DTMF detection is performed digitally by one of a few tone detection units, since DTMF detection is required for only a small fraction of the call. In this scheme, when a phone is taken off the hook, the switch searches for a free tone-detection unit and does not issue a dial tone until one is available. We decided not to implement a tone-detection unit on a separate card, because we could not design the additional hardware and meet our desired completion date (the end of the semester). A test program was created to measure the amount of time required to run the DTMF detection algorithm on the main CPU. Experiments showed that DTMF detection took sufficiently long on a 386 processor that the switching fabric process did not have enough time left over to switch any calls at the 8-kHz rate. A Pentium® processor was fast enough to execute the DTMF algorithm and still switch calls. However, we decided that we could add DTMF detection to our line card much more cheaply than the cost of a faster host processor, so we added analog DTMF detection to the line card. A similar project which had different requirements and manufacturing volumes would probably opt for one of the other two possible solutions (the line card set is the single most expensive element of a large switching system). Design choices must be evaluated in light of the system requirements.

TigerSwitch illustrates the typical architecture design process in an embedded system:

- first, a candidate architecture must be proposed and potential performance bottlenecks in that architecture must be identified;
- second, modifications to the architecture must be proposed and the performance of each analyzed;
- finally, one configuration must be chosen based on the results of performance analysis and other requirements, such as manufacturing cost, design time, and reliability.

Performance analysis of an initial architecture, before a complete implementation of any part of the system is available, is critical because performance bottlenecks may not be obvious from the system specification. Performance approximations derived from simplified models of the system, providing that the approximations are sufficiently accurate, help us avoid completing an unsatisfactory implementation which must be thrown away.

Morfit's description of one embedded system design illustrates the role of performance analysis and optimization [67]. After the design and implementation of a cellular telephone system, the design team used a software monitoring system, which recorded the active process at each scheduler interrupt, to measure which processes were dominating CPU utilization. Measurement showed that unexpected processes were taking up most of the CPU and

® Pentium is a trademark of Intel.

that repartitioning operations between processes could substantially reduce CPU requirements. In one case, bundling display writes into a single processor reduced that process's utilization from 20%–50% to less than 10%. In another case, modifying a process to write multiple-byte rather than 1-byte records reduced the number of function calls to a critical routine in the range 20:1 to 300:1, depending on the data. While some performance data cannot be fully created without a complete implementation, a more detailed model with accurate analysis would allow such problems to be caught in time to change the design of both the software and the hardware engine.

Queueing system models are useful for analyzing systems where inputs arrive sporadically or the processing time for a request may vary. In a queueing model, customers arrive at the queue at some rate; the customer at the head of the queue is immediately taken by the processing node, but the amount of time spent by the customer in processing must be specified. Typically, both the customer arrival rate and processing time are modeled as Poisson random variables. Useful characteristics a queue include the average **residence time**, which is the amount of time a customer resides in the queue plus processing time, and the average **queue length**, which is given by Little's Law [54].

Queues are assembled into networks to model systems: the output of one processing center feeds into another queue to enter the next phase of processing. Computer system models have traditionally used for data-processing style applications: stochastic models are appropriate not only for CPU's which must handle requests which may take varying amounts of time to process, but also disks whose access time depends on the disk state at the time of the request. Queueing network models are usually solved by simulation; SES/Workbench® is one well-known queueing network analysis system. Kobayashi [54], Lazowska *et al.* [56], and Smith [91] survey computer system modeling using queueing networks.

B. CPU Performance and Modeling

Advanced architectures usually include components, like caches, which provide high peak performance at the cost of greater variance in execution times. Hard real-time systems must meet their deadlines under worst case conditions. Some techniques exist for narrowing the variance of execution times.

Most CPU's are offered in several different models and the choice of model can substantially affect the cost of the CPU. Integrated circuits may be packaged in ceramic or plastic; ceramic packages are much more expensive to make than injection-molded plastic packages, but ceramic packages provide more pins and can run at faster rates. For example, a 1991 Intel catalog lists the price of a 20-MHz i960KB in a plastic quad flat pack (PQFP) as 17% less than the same chip in a ceramic pin-grid array (PGA); the 25-MHz version of this processor is available only in the PGA package.

In some cases, the processor can be packaged in plastic only by reducing the number of pins. In such cases, a modified bus is designed for the processor with a smaller number of data pins. For example, the Intel i386DX has a 32-bit data bus, while the i386SX has a 16-bit data bus. The smaller bus width is invisible to the software running on the CPU because the bus subsystem breaks a write into operations which can fit on the bus, and assembles the results of a read to produce a datum of the proper size. The smaller bus width also makes the memory system less expensive, since it requires fewer separate chips. The ability to use a plastic package at all, or to use a cheaper plastic package with fewer pins, can make a CPU substantially cheaper: the 1991 Intel catalog prices a 16-MHz i960SB, a variation of the KB with a 16-bit bus, as 40% less expensive than the 20-MHz chip in a ceramic package. While a 16-bit bus on a 32-bit processor requires two bus transactions to fetch a 32-bit datum, the effect of reduced bus width on program performance cannot be simply calculated. Code executing from the internal cache or data in registers runs at the same rate on the narrow- and wide-bus systems; if the program fetches smaller data values, such as bytes, the narrow bus will be as efficient. A study of the program's dynamics is required to determine the true performance penalty of a narrow bus.

Instruction execution time is normally given in tabular form. For a nonpipelined processor, one entry per instruction is sufficient for simple instructions. More complex instructions may have execution times which depend on data values, just as program execution times depend on the trace taken through the program. Integer multiplication, floating-point, and especially transcendental functions are likely to have data-dependent execution times. A transcendental operation can take tens of thousands of clock cycles to execute.

Execution behavior in a pipelined machine depends not just on one instruction, but on a set of instructions. For example, a processor may use a prefetch unit to fetch instructions in order after the present program counter location and store those instructions in a queue. When a branch is taken, the pending instruction queue is no longer valid, so the CPU must wait for the branch target to be fetched, which causes a longer interval between successive instruction completion times. Other pipelining mechanisms also cause the execution time of one instruction to depend on which other instructions are pending; for example, the Intel i960KB manual gives the execution time for register operations depending on whether the processor can bypass a register access [45]—a bypass hit occurs if the source of one of the instruction's operands was the result of the previous instruction, saving one clock cycle. Schmit [83] describes techniques for optimizing Pentium code to take advantage of multiple instruction issue.

It is often not possible to obtain from the microprocessor supplier a CPU simulator which accurately models performance. Simulation models which mimic only bus

® SES/Workbench is a trademark of Scientific and Engineering Software, Inc.

behavior, known as **bus-level models**, are more common. Such a model accurately reflects the number of cycles required to perform a bus transaction, such as a read or write, but does not model the action of instructions. On the other hand, some manufacturers do not publish instruction performance data, even in tabular form. In such cases, the only recourse for accurate performance measurement is to measure execution times on a hardware system. Measurement may be difficult on cache-based systems, since not all CPU operations will be reflected on the bus. An **in-circuit emulator** is a version of a CPU with the same pinout as the standard CPU but which keeps traces of instructions, allows breakpoints to be set, etc. While an emulator is useful, emulation suffers the same fate as simulation in that worst case performance is hard to elicit, and the emulator may not be able to execute the instruction stream at the full rate of the standard CPU. An emulator is often more useful for functional debugging than for performance analysis.

Caches affect CPU performance even more than pipelining within the execution unit. Many texts, such as Patterson and Hennessy [38], describe cache organization and operation. Since the static RAM (SRAM) used in cache is ten times or more faster than the dynamic RAM (DRAM) typically used in main memory, the penalty for a cache miss is very large. Interrupt-driven systems are very poorly matched to the assumptions which typically justify caches. Rather than have a loop or another relatively small section of code which is executed repeatedly, an interrupt-driven system switches at irregular intervals between routines residing in very different parts of memory. In a typical processor, an interrupt routine invalidates most or all of the cache; not only does this slow down the initial execution of that routine, but it introduces contention for the cache between interrupt routines which slows down the entire system.

The stochastic nature of cache-based systems presents a problem to the design of systems with hard real-time deadlines. While an operation may run fast if it happens to reside in the cache, it will run much slower if the routine is not in the cache. It is often difficult or impossible to predict from macroscopic program structure whether a particular piece of code can be guaranteed to be in the cache. As a result, real-time system designers often assume that a memory fetch will always miss the cache. While this assumption does ensure that the process will always meet its deadline, it is very pessimistic caches are one of the most effective means to the improvement of CPU performance.

Larger caches provided by increasing SRAM density make it easier to reserve sections of the cache for critical code. Two schemes—one hardware and one software—have been proposed to ensure that selected routines reside in the cache. Kirk proposed the SMART (Strategic Memory Allocation for Real-Time) cache organization [51], [52], which partitions the cache into a shared partition plus several partitions allocated for critical tasks, as shown in Fig. 5. Each time-critical routine may receive one or more segments of the cache. A processor flag determines whether the currently running process is mapped into the shared partition or into one of the private partitions. An ID register identifies which private partitions are owned by this process. Hardware ensures that a segment can be accessed only by the task to which it was allocated. Kirk and Strosnider [52] developed an algorithm to analyze instruction traces to choose sections of code which should be allocated their own cache segments.

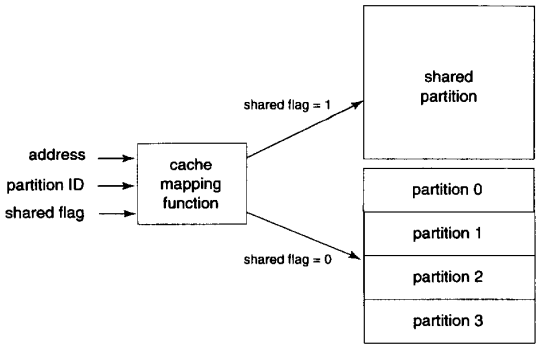

Fig. 5. Address mapping in Kirk's hardware cache partitioning scheme.

Wolfe [101] proposed a partitioning scheme which requires no additional hardware. His scheme chooses addresses for code and data during linking such that critical routines are the only addresses in the program which map into certain sections of the cache. Since no other routine can knock that code out of the cache, the routine is guaranteed to be cache resident. Figure 6 shows an example for which the cache has an 8-byte line: the first two lines are allocated contiguously to one process, the third line to another process, and the fourth line to yet another process. This scheme maps addresses into the cache into very small chunks, equal to the size of the cache line—frequently 16 bytes or smaller. As a result, a process's address space is broken into many small chunks, and it may not be possible to allocate a process's code to contiguous addresses; it also requires widely separated addresses. Wolfe presented another scheme which allows contiguous addressing: by extracting the tag from the middle of the address, not the top, larger blocks of contiguous memory are mapped into the cache. While this scheme requires that the cache hardware be redesigned to use different address bits, it does not add either area or delay to the cache implementation.

Interrupt latency—the time required for the CPU to execute the first instruction of an interrupt handler after an interrupt is raised—can significantly affect performance in two ways. First, interrupt-driven or scheduling-based systems must add interrupt latency into their calculations of total processing time. Second, interrupt latency puts a lower bound on the time in which the system can respond to an interrupted request. Tasks which require very fast response to an event may not be implementable as interrupt routines—busy-wait I/O or addition of a special hardware unit to handle the task are alternative implementations which decrease response time at the expense of added hardware cost (either in the form of required additional

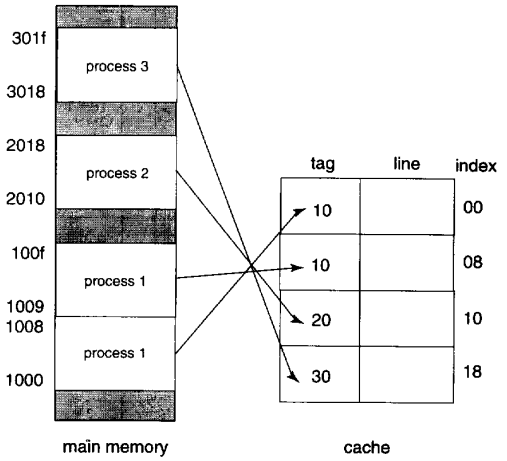

Fig. 6. Address mapping in Wolfe's software cache partitioning scheme.

CPU capacity to make up for the lost time incurred by busy wait polling or for the special-purpose hardware).

When a CPU services an interrupt, it must typically reference interrupt vector tables, change the CPU state, and other assorted tasks. Responding to an interrupt usually takes much longer than simple instructions and the penalty generally grows with CPU size. For example, the Motorola MC68HC16 16-bit microcontroller requires 16 clock cycles to respond to an interrupt [68]; the Motorola MC68020 requires a minimum of 26 cycles [69]; the Intel i960KB requires a minimum of 85 cycles [45]. While these times are not completely comparable, since these processors perform somewhat different actions on interrupts (the i960KB, for instance, saves the current register set), but they do show that the penalty is significant and increases with architectural complexity. The time required to respond to an interrupt may depend on the state of the processor: for example, multiple-cycle instructions may not be interruptible. Interrupt latency is not the only cost of handling an interrupt: interrupt latency is overhead incurred in addition to the execution time of the interrupt handler; furthermore, the change in instruction flow induced by the interrupt changes the state of the cache.

C. Software Performance Estimation

To estimate the performance of a concurrent system, we must be able to estimate the performance of a single process. **Software performance estimation** estimates bounds on the running time of a single-threaded code fragment when run on a specified processor. While CPU modeling concentrates on a stream of instructions small enough to fit in the processor pipeline and cache, software performance estimation analyzes larger sections of code. Software performance estimation can be broken down into two steps: identifying legal paths through the code; and determining the execution time of each path. Identifying all legal paths through a program with unbounded memory is equivalent to solving a halting problem, making exact path identification undecidable, but good identification of false paths through the program tightens the bounds on execution time.

We would like to develop a hierarchy of performance estimation models, including at least high-level language and assembly language descriptions of the program. Lower level models, such as assembly language, will obviously give more accurate information on the effects of register allocation, instruction interactions, and caching than will a high-level language model. However, we often synthesize a program from some specification by creating a high-level language description which is then passed to a compiler. We may not want to include compilation time in the interval required to generate an initial performance estimate; furthermore, we may want to compare the merits of potential target CPU's without purchasing the compilers for all those architectures. Therefore, in addition to path analysis, software performance estimation must also consider how to calculate the execution times of the primitive operations in the software description at whatever level of abstraction is available.

The earliest techniques for software performance analysis were manual analysis methods developed for data processing systems. Smith [91] gives a good overview of the analysis of program representations she calls **execution graphs**, which are flow charts which use *fork* and *join* operators to specify concurrent activity. Execution graph analysis is intended primarily for modifications to existing systems because it depends on performance measurement of code. The execution time of the graph is computed by reducing subgraphs. However, the accuracy of the execution time clearly depends on the accuracy of the performance estimates for the primitive operations, the estimates of the numbers of times loops are executed, etc. Smith describes techniques for obtaining accurate execution time and workload information from existing systems. However, performance estimates based on measurements must always be used with caution because the system tests may not have exercised worst case behavior. Automatic software performance estimation offers the promise of performance figures which are both conservative, i.e., guaranteed to be bounds on actual worst case performance, and more precise than can be generated by hand.

Ein-Dor and Feldmesser [24] performed an early experiment in performance prediction. Their goal was to predict the relative performance of a computer system from basic characteristics of that system. They executed a set of benchmark programs on 209 computer systems and created a regression model of computer performance as a function of six variables: cache memory size, minimum number of I/O channels, maximum number of I/O channels, machine cycle time, minimum main memory, and maximum main memory. Their model could predict performance relatively accurate over a range of medium-performance machines.

Shaw developed techniques for reasoning about the execution time of both single and communicating processes [86], [87]. Shaw assumed that bounds $[t_{\min}, t_{\max}]$ could be found for the execution times of program statements. He

defined the execution times of code in terms of schema which describe the execution times of combinations of statements. He used Hoare-style assertions to describe timing properties of the program: given a statement S in the program, if a predicate P is true before S is executed, then Q is true after S is executed, which is written in the form $\{P\}\ S\ \{Q\}$. To reason about the real-time behavior of the program, P and Q can be functions of the real time: if a statement S's execution must be completed in the interval $[t_{dl,\min}, t_{dl,\max}]$ and rt represents the value of real time, then the deadline can be expressed in the form

$$\{rt \leq T_{dl,\max} - t_{\max}(S)\}S\{rt < t_{dl,\max}\}, \quad \text{when} \quad t_{dl,\min} = -\infty.$$

Shaw used these techniques to, as one example, reason about programs which recognized single and double mouse-clicks, behavior which relies intimately on real time.

Puschner and Koza [79] used simple bounds declarations to capture user execution information which could not be directly derived from the program and more accurately estimate maximum execution time. Their declarations took the form of annotations in the program source code: scope identifiers delimited a sequence of statements in the program; a marker statement specified the maximum number of times the program would pass through that marker between entering and leaving the scope which enclosed it; a loop sequence declaration gave an upper bound on the total number of times a sequence of loops would be executed (useful when the loop bounds of sequential loops are correlated but not independently fixed). They found that these declarations were sufficient to greatly improve the accuracy of execution time bounds in the examples they studied.

Park and Shaw created a timing tool which combined performance models for C language statements with path analysis algorithms [71], [72]. Park developed his performance model for one CPU–compiler pair; namely, a 68010-based Sun-3 and Gnu C-1.34. Park's model relies on instructions being executed deterministically: instruction execution times do not depend on nearby instructions and neither instructions nor data are cached. He found that the most accurate means for estimating program times was to consider all the code in a basic block at once: he extended the C compiler to mark boundaries of basic blocks; for each basic block in the C program, he identified the assembly language generated by that block and looked up the execution times of each instruction in a table. He applied corrections to take into account two types of system-level interference in program execution: clock interrupts and memory refresh. Park compared measured execution times to computed estimates to show that, in most cases, these techniques produced tight bounds and that most uncertainty could be removed by more accurate prediction of execution paths.

While a program's control flow graph gives the set of all possible execution paths through the program, there may be paths which are never executed: the data values supplied to the program may be restricted so as to, for example, limit the number of times through a loop; relationships between variable values in the code may also make some paths infeasible. Park's model for execution paths was regular expressions extended with intersection and negation operators. To gather user execution information, he developed an information description language: important statements in the program were given names, restrictions such as nopath (A, B) and loop A K times could be placed on feasible paths. This information can be used to ignore illegal paths during timing analysis. Experiments showed that adding path analysis information tightened execution bounds.

Ye *et al.* [102] developed a fast timing analysis technique for use in hardware–software partitioning. They needed to accurately estimate the system performance even when the software partition executed on a high-performance CPU whose execution times depend on data dependencies, instruction order, etc. They extracted a basic block of the software and executed it once; the measured execution time automatically takes into account processor-specific timing. They then used that time as one timing label in a control flowgraph; along with annotations for execution times of the hardware units, the control flowgraph can be analyzed to predict the total execution time of the hardware–software system.

D. Performance of Tasks on Shared CPU's

It is not sufficient to analyze the performance of each process in isolation. When several processors are allocated to a system CPU, system performance depends on how the processes are scheduled on the CPU. The scheduling of processes on a CPU determines the CPU's **utilization**, a key measure of architectural efficiency. An underutilized CPU adds unnecessary cost to the system since it could be replaced with a smaller CPU. However, it can be shown that in many cases a CPU cannot both be fully utilized and meet all the deadlines on its processes.

The processes executing on a CPU may be scheduled either **statically** (in an order determined when the program was designed) or **dynamically** (during system execution). A **cyclostatic** scheduler is an example of a statically scheduled system: a cyclostatic scheduler is called periodically by a timer and executes a set of tasks in a fixed order. A dynamically scheduled set of processes may be scheduled either **preemptively** or **nonpreemptively**. In a nonpreemptively scheduled system, each process explicitly gives up control to the next process. System calls are spread periodically through the code which allow the next process to run, usually determined by a list of active processes maintained by the kernel. Microsoft Windows is an example of a nonpreemptively scheduled system. Nonpreemptive code must be carefully implemented to ensure that each task gives up control of the CPU in a bounded amount of time on any execution path—failure to relinquish control causes the system to fail to respond to other inputs. A preemptively scheduled system uses a timer to periodically return control of the CPU to a scheduling process in the kernel. The process with the highest **priority** is chosen to run in the next time slot.

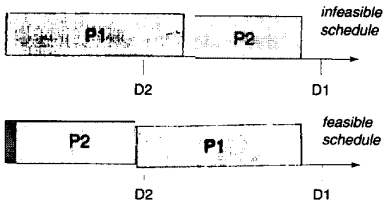

Fig. 7. Process priorities and deadlines.

Each scheduling scheme has advantages and disadvantages. A nonpreemptive scheduler allows the designer to verify properties such as deadlock; because we do not know the exact order of execution of processes in a preemptively scheduled system, they cannot guarantee liveness. Chiodo *et al.* [12] use an FSM-based description to verify the properties of hardware–software systems before choosing a hardware–software partitioning for implementation. However, a nonpreemptively scheduled system requires a more powerful CPU to ensure that it meets all its deadlines than does a preemptive scheduler preemptive scheduling is more CPU-efficient because a task with a closer deadline can be assigned a higher priority and preempt a lower priority task.

Figure 7 shows a simple example of how process scheduling determines deadline satisfaction. The processes $P1$ and $P2$ have deadlines $D1$ and $D2$, respectively. In the upper schedule, $P1$ has been scheduled first, for example because it was activated first by an outside event. $P1$'s deadline is well after $P2$'s, but $P1$ prevents $P2$ from executing and makes $P2$ miss its deadline. If process priorities are assigned to ensure that the process with the shortest deadline receives the highest priority, allowing it to start executing as soon as it is activated, then both processes are able to complete before their deadlines.

The fundamental result in scheduling of hard real-time tasks was discovered by Liu and Layland, who introduced the rate-monotonic scheduling algorithm [62] for scheduling periodic task sets. In their model, a system consists of a set of tasks, each of which has a deterministic computation time and a period. While those times are fixed, the phasing of the tasks relative to each other is not fixed. They showed that such a task set can be scheduled to meet all deadlines, irrespective of the phasing of task initiations in the period. Interestingly, only fixed priorities are required: priorities are assigned inversely to task period, with the shortest period task receiving the highest priority. They also showed a worst case bound for CPU utilization of 69.3%. Thus excess CPU capacity must be available to ensure that the CPU can respond to the worst possible combination of task phasings. However, for preemptive scheduling, the CPU capacity is determined by the task specifications, while in nonpreemptively scheduled systems, the CPU capacity required is determined by the structure of the code.

This work has been extended in a number of ways to handle more general hard real-time systems. The priority ceiling protocol was developed by Sha, Rajkumar, and Lehoczky [85] for systems in which low-priority tasks can obtain critical resources. When a process requires a resource which must be shared by P/V (semaphore) synchronization, a lower priority process can use the resource's lock to block execution of a higher priority process which needs the resource, a situation known as **priority inversion**. Strosnider [96] introduced deferrable servers for aperiodic tasks, in which a collection of aperiodic tasks is modeled as a regularly scheduled process which periodically checks the status of the aperiodic requests. Leinbaugh [60] developed an algorithm to bound worst case performance for systems of processes which talked to devices and executed critical sections.

E. Co-Simulation

Simulation will be an important co-design tool for the foreseeable future because of the complex nature of the embedded computing systems. Because embedded system components are so complex, it may be difficult to develop comprehensive analytic models for their performance. Simulation also helps the designer verify that the system satisfies its requirements. **Co-simulation** mixes components which have different simulation models. Co-simulation usually refers to some sort of mixed hardware–software simulation—for example, one part of the system may be modeled as instructions executing on a CPU while another part may be modeled as logic gates. Co-simulation is difficult because the system's components operate at different levels of abstraction—analog components operate over voltages, PLD's operate over binary values, and microprocessors operate over instructions—and run at different rates one instruction in a microprocessor may take several cycles to execute, during which time analog components may have moved to a drastically state.

Multimode simulators allow a system to be described as a mixture of components at different levels of abstraction. As shown in Fig. 8, two different techniques have been developed for multimode co-simulation. A **simulation backplane** provides a top-level simulation model through which different types of simulators can interact, so long as their external behavior meets the modeling requirements of the backplane. A **heterogeneous** simulator does not require all simulation events to be reduced to the same level of abstraction. Ptolemy [9] is a well-known framework for the construction of co-simulators. A simulation universe in Ptolemy consists of several domains, where each domain has a single simulation model. Ptolemy does not enforce a single simulation model: domains may be combined arbitrarily and developers may create new domains. Several different domains have been implemented for Ptolemy: synchronous data flow, dynamic data flow, discrete event, and a digital hardware modeling environment. Each domain has its own scheduler. Wormholes allow data to pass between domains. A wormhole between two domains includes an event horizon which translates events as they move from one domain to another. As an event moves between domains, it is first translated into a universal event type and then to the domain of the destination. Because wormholes hide the details of operations in other domains, a domain's scheduler does not have to know the semantics of the other

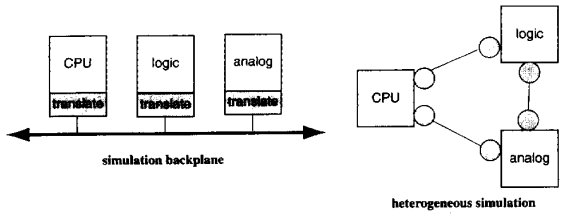

Fig. 8. Two techniques for multimode simulation.

domains to execute its simulation. Ptolemy also supports interfaces to code synthesis for DSP's. Kalavade and Lee describe the use of Ptolemy-based co-simulators to develop a hardware–software system [49].

Hierarchical simulators allow a design to be modeled and simulated at several different levels of abstraction, but with the same level of abstraction for all components in each model. ADAS (Architecture Design and Assessment System) [92] was an early co-design tool which was targeted to signal processing applications. ADAS could simulate a system at three levels of abstraction: at an algorithm level, modeled by data flow; at an architectural level, modeled by scheduled processes; and at an implementation level, described as a register-transfer system. The designer could simulate the system at each level of abstraction and compare the system performance at two different levels of abstraction to guide the refinement of the design from one level of abstraction to the next.

Roth *et al.* [82] created a custom simulator for a pair of graphics accelerator ASIC's they designed. The design project needed a simulator which was both fast enough for software development and accurate enough to be used in debugging the hardware models. They developed a custom simulator in C which could simulate either of the two chips and provided a programming interface equivalent to the chips themselves. They extracted the simulated chips' internal states during simulation and compared that state trace to a state trace generated by a Verilog-based register-transfer or gate-level simulation of the chips. Differences in the state traces indicated potential bugs. Because the ASIC's had multiple function units and worked on an asynchronous memory, accurately comparing the two state traces required considerable effort.

IV. Hardware Engine and Software Process Design

In this section, we will survey methods for the design of an embedded system's hardware and software. We do not yet fully understand how to jointly design the hardware and software for embedded applications. Most work has concentrated on the design of either the hardware or software, using a very simplified model for the other. Recent work on hardware–software partitioning takes a more balanced view of the two elements, but existing algorithms still rely on restricted architectures for both the hardware and software components. Methodologies and algorithms for truly general hardware–software co-design of embedded systems are a primary goal of research in this

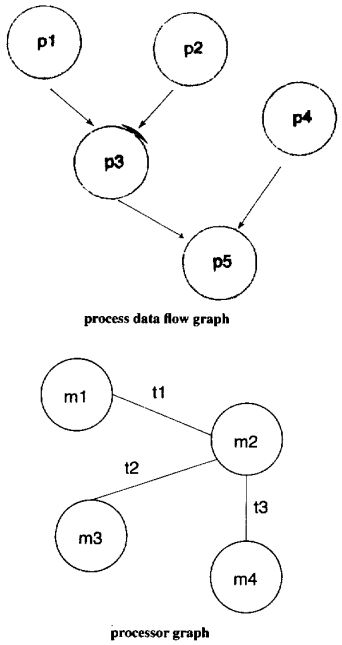

Fig. 9. Graph models for hardware and software design.

area. As a result, we will describe both previous work in co-design proper and also related work in distributed system design, which takes the hardware engine as a given and designs a software architecture.

Joint design of the hardware and software requires representations for both the processes and the distributed engine, as shown in Fig. 9. An example of a software model is a data flowgraph to represent the software processes, which identifies sources and sinks of data between the processes. It may also be important to represent control flow between the processes, in which case the general form is called a **process graph**. The **processor graph**, which represents the hardware engine, has CPU's as nodes and communication links as edges. Some research considers processor graphs where all CPU's are identical, while other work allows each node to have different characteristics.

Figure 10 illustrates how a process graph is mapped onto a processor graph. The figure can only easily depict the allocation of processes to CPU's, but the complete design process also entails scheduling those processes on their assigned CPU's, partitioning the process set to provide an efficient scheduling and allocation, and mapping the processes onto particular types of CPU's. In a traditional design flow, the processes and the processor network would be designed relatively separately. However, to obtain the highest performance, lowest cost solution, we must simultaneously design both the processes and processor system.

We first consider alternative models for processes and their embodiment in programming languages. Section IV-B describes techniques developed for the design of hardware engines; Section IV-C describes recent work in hardware–software partitioning, which is the joint design of hardware engine and software architectures from a hard-

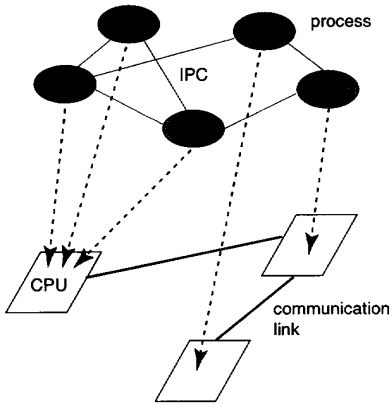

Fig. 10. Mapping application processes onto a hardware engine.

ware architecture template. The remaining subsections describe techniques developed by the distributed systems community for the scheduling, partitioning, and allocation of processes given the hardware architecture of a distributed system.

A. Process Models and Program Specification

Since the system's function will be described as a system of communicating processes, the choice of a process model is fundamental. Given a process model poorly suited to the problem, the system description may become unwieldy or the system may not be describable at all. A number of different process formalisms have been developed over the years, each with its own advantages and disadvantages.

The coarsest distinction between process models is their level of parallelism. A Petri net model [76] is a highly parallel model of computation. The synchronous dataflow model [59], a data flow model in which the number of samples consumed by a node at activation is specified *a priori*, is another highly parallel model of computation. A communicating sequential processes model describes each process as executing sequentially, but allows the processes to execute at different rates.

There are many communicating sequential process models, which can be further classified according to the types of interprocess communication they support. Buchenrieder *et al.* [8] use the PRAM model, which is commonly used to model shared-memory parallel algorithms, to describe the communicating processes in a machine. The Solar modeling language [70] is an example of a communicating processes model for co-design. Hoare's CSP [39] is one style of communicating sequential process model which uses unbuffered communication. In **nonblocking** or **buffered** communication, a queue holds data which have not been consumed by the receiver, allowing the sender to continue executing. A finite-state machine-style model uses implicit communication—because an FSM can change its outputs at any time, communication is not separated from the rest of the machine's computation. Event-driven state machine models, including the BFSM [97] and the CFSM [12], have been proposed for use in co-design because they allow the systems to be described as a partial ordering.

Software is much more than code, particularly when that software implements a system of concurrent processes. Representations for programs are important intermediate forms for system design. Software design models ensure that the software is correctly implemented. Because most design languages for concurrent processes also define interprocess communication primitives, they provide documentation on how the processes talk to each other and hooks for the analysis of interprocess communication.

A number of concurrent programming languages have been developed—these languages allow computations to be expressed as systems of communicating processes but do not provide methods to specify deadlines or guarantee that they be met. A concurrent programming language provides primitives for interprocess communication. (Interprocess communication primitives may also be provided by the operating system.) The way in which the language models communication has a profound impact on the way the initial architecture is implemented in the language. Several different communication techniques are possible. A **signal** sent by one process forces another process to start executing at a specified location which should handle the signal; a signal is a software version of an interrupt and is used in the Unix® system. A **mailbox** is a variant of buffered communication usually reserved for schemes in which the names of mailboxes are known globally and any process may both send messages to and take messages from a mailbox. The language and operating system may also support blocking communication or finite-size queues, which can provide nonblocking communication given some assumptions about the maximum number of pending messages.

The term real-time programming language has been used in various ways, but generally means that the language allows deadlines to be specified and provides mechanisms to ensure that the deadlines are met. Such a language must have some sort of representation of a process in the language. A real-time programming language may also restrict itself to statements whose execution times can be bounded; other real-time languages are extensions of general-purpose languages, which usually contain statements with potentially unbounded execution times, such as while loops controlled by unrestricted input values.

The statechart [34] is an extended-state machine model for reactive systems. A statechart is a state transition graph in which a transition's source or sink may be a set of states, not just a single state; another way to view this specification style is that a state which represents several different behaviors may be decomposed into states which implement pieces of that behavior. As shown in Fig. 11, OR and AND decompositions can be used to refine a state's specification by decomposition. In the OR decomposition in the figure, the transition specifies that the system should go to state s when input $i1$ is received if the system's present state is either a or b; the source of this transition

® Unix is a trademark of Unix System Laboratories.

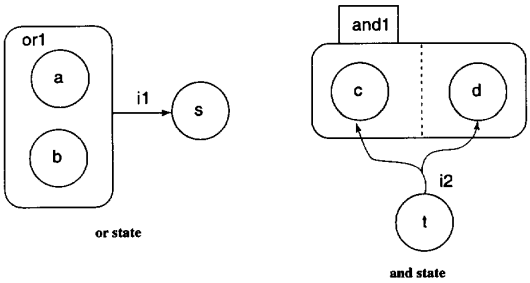

Fig. 11. Composite states in a statechart.

is the state $or1$, which represents the union of a and b. An AND decomposition represents concurrent activities—in the figure, the $i1$ transition out of state t sets one part of the system state to c and another part to d, so that the total state is the product $c \times d$. OR and AND decompositions allow exponentially fewer states and transitions than are required to describe the equivalent behavior in a state transition graph with a single level hierarchy of states. STATEMATE is a well-known tool for manipulating statecharts [35]. The tool integrates representations for block diagram structural descriptions and data flow diagrams with statecharts. It can simulate a specification and generate high-level language code to implement a specification.

A recent collection of papers in the PROCEEDINGS OF THE IEEE [5] described several **reactive programming languages**, where a reactive language interacts constantly with its environment, rather than viewing input and output as streams. For a detailed description of those languages, the reader is referred to those articles. One example of this approach, the Esterel language [6] promotes a specification style different from that used in statecharts an Esterel program is a network of communicating machines, rather than a single machine with a hierarchically decomposed set of states, as is a statechart. Processes communicate by emitting actions and waiting for actions to appear, with communication scheduled by the programmer. Esterel assumes that reactions to events are atomic and happen instantaneously. Because actions are atomic, the component machines communicate synchronously, which means that many analyses of communication behavior are much simpler than with asynchronous systems. An Esterel program can be compiled into code by generating a product machine; since in the product machine, communication between components is implemented as products of states of the component machines, the communication action does not generate any code, thus satisfying the atomicity hypothesis.

B. Hardware Engine Design

ADAS, described in Section III-E, supported early functional verification and evaluation of system performance at the first three stages of design, which the authors identified as algorithm definition, system architecture design, and detailed system design. In our terminology, the products of ADAS were an architecture for the hardware engine and an allocation of software processes onto processors. While ADAS did not provide synthesis algorithms to automatically generate the engine architecture and software partitioning, it did provide representations for the various stages of design and assessment tools to help the designer determine the best translation from one stage to the next.

ADAS used three representations of the design, one for each stage of design. The highest level of abstraction was a data flowgraph, which they called a software graph: nodes represented processes and edges represented data transfer. The algorithm to be implemented was specified as a software graph; data flow is a sufficient representation for signal processing algorithms with little control-dependent behavior. The mapping of processes to hardware components was represented by a projected processing graph, which was constructed from the software graph by adding synchronization arcs which forced sequential execution of processes. Finally, the architecture of the hardware engine was represented by a hardware resources graph, which was a form of register-transfer machine.

To design an application, the designer first specified a software graph, refined it to a projected processing graph, then added multiplexers and registers to create a hardware resources graph. The software graph could be executed to check that the desired function was properly specified. Performance analysis began with the projected processing graph—Petri net analysis was used to determine whether the sequentiality introduced by the sequentiality constraints allowed the function to be executed at the required rate. ADAS had algorithms which could determine the utilization of a node and the latency of a computation. ADAS apparently did not include tools to verify that the hardware resources graph correctly implemented the projected processing graph.

MICON [32] generates a board-level design of a microprocessor system from a loose description of the board's requirements. The system's specification includes information such as: type of CPU required; types and amounts of memory required; I/O ports required; and the types of external connectors. MICON selected a complete set of components and interconnected them to produce a complete board design. Several different components may implement a particular requirement: for example, different combinations of memory chips may be used to implement the required memory. Components may also have to be interpolated into the design to provide an interface between two required components. MICON uses AI techniques both to learn how to design boards from expert designers and to complete the design of a board from requirements.

Parkash and Parker [78] used integer programming to design a multiprocessing architecture for an embedded hardware engine. Given a partitioning of an application into tasks, their algorithm designs a heterogeneous engine to run the application, allocates tasks to processors in the engine, and schedules those tasks. Their model for a system of tasks is a data flowgraph for which a task may start computation before all its data have arrived. They model the architecture of the engine to be designed as a set of processors with direct communication links.

Each processor has local memory, communication occurs via messages over the links, and one application runs on a processor at a time. Their mathematical programming model includes both integer-valued timing variables and binary decision variables: timing variables represent data availability times, output av .ability times, task execution times, and data transfer times; decision variables represent allocation of tasks to processors and the direction of data transfer. The model includes ten types of constraints which determine: how tasks are mapped onto processors, whether data transfer is local or remote, input and output availability, task execution start and end times, data transfer start and end times, and that each processor and communication link is not used for more than one operation at a time. Their model may be solved to satisfy a performance goal or to minimize total engine cost. They rewrite several of their constraints in linear form to create a mixed integer linear program, then solve the program using standard techniques.

Srivastava *et al.* used templates to design real-time subsystems for a workstation-based embedded architecture [93], [94]. Figure 12 shows a simple architectural template with two levels of hierarchy: subsystems are connected by a bus; each subsystem consists of a CPU and several ASIC's all connected by their own bus. They defined a four-level hierarchy for their architecture template: the top level is a workstation which communicates with the layer-2 processor; custom boards form the third level of the hierarchy, with the CPU and peripherals in the subsystem forming the fourth level. They map a behavior described as a system of process onto the architecture template. After allocation, they generate the hardware and software components separately, using the bus as a framework for hardware design and a real-time kernel as the framework for the software components. They developed a large suite of reusable, parameterized subsystem generators to implement functions ranging from memory subsystems to bus interfaces, to fiber-optics communications. Software components are generated as processes which operate under the control of a real-time executive. In addition, each hardware module has a wrapper software module to provide a software interface. To illustrate the use of their system, they designed a robot control system built from a Sun workstation as the level-1 processor, a MC68020 as the level-2 processor, and two subsystems each built from a TMS320C30 processor and a DSP32C slave; software processes were distributed among the subsystem CPU's to make sure that the real-time controller's deadlines were satisfied. The subsystem CPU's also ran service processes for run-time I/O and data routing.

Gabriel [58], a precursor of Ptolemy, is a design environment for DSP which supports both simulation and direct execution of functions on processors. The function under design is specified as a block diagram. Gabriel's simulation scheduler can simulate the block diagram under synchronous data flow semantics. The same schedule can be used to guide the generation of code for the target DSP; an individual operations has assembly language templates to implement the operation on the target machine.

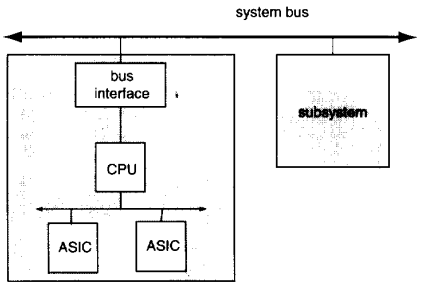

Fig. 12. A simple hierarchical architecture template.

C. System and Hardware–Software Partitioning

A system's function must be partitioned when it is implemented on either multiple physical units (chips) or onto heterogeneous units (CPU's and ASIC's). System partitioning requires some performance information to be able to compute the system's critical performance path, but partitioning should use a simple timing model which can be quickly evaluated while analyzing large partitioning problems. Lagnese and Thomas [55] developed APARTY, a system partitioning tool which partitions a control data flow graph to minimize chip area and interconnect requirements. APARTY uses a multistage clustering algorithm which operates in several stages. Each stage has its own clustering objective, which it uses to cluster nodes in the design. At the end of the stage, clusters of a given size are selected as elements for the next stage of clustering. Ismail *et al.* [44] developed an interactive system-level partitioning tool. The design is specified as a set of communicating processes; the user can specify a sequence of transformations—move, merge, split, cut, and map—to redesign the process network.

Hardware–software partitioning algorithms try to meet performance goals by implementing some operations in special-purpose hardware. The hardware unit generally takes the form of a co-processor, communicating with the CPU over its bus. In some cases, a fairly large set of routines may be implemented totally in hardware to avoid instruction interpretation overhead, but hardware–software partitioning algorithms are targeted to systems in which only a few operations need specialized hardware. However, the computation performed in the co-processor must be long enough to compensate for the time required to transfer data back and forth to the CPU. If the co-processor's computation is too short, the CPU may be able to perform the operation faster by keeping values in its registers and avoiding bus protocol overhead. Hardware–software partitioning algorithms are closely related to the process scheduling model used for the software side of the implementation.

Hardware–software partitioning algorithms generally target their hardware design to high-level synthesis algorithms. High-level synthesis generates a register-transfer implementation from a behavior description, which may be either a pure data flowgraph or a mixed control-data flowgraph. Synthesis algorithms schedule operations in time and per-

form several allocations: operations to hardware function units, values to registers, and data transfers to interconnect. Readers interested in high-level synthesis algorithms can consult books by De Micheli [22], Gajski *et al.* [29], and Michel *et al.* [66].

Most partitioning algorithms divide the behavior specification into a set of software processes running on one CPU and one co-processor. Two styles of algorithms have been proposed: ones which start with all operations in hardware and move some to software; and ones which start with everything in software and move some operations to hardware.

Gupta and De Micheli [33] developed an algorithm which migrates operations from the hardware partition to the software partition. Their algorithm accepts a behavior in the form of a control-data flowgraph and a set of rate constraints on the constituent operations. It divides the behavior into a set of threads bounded by operations with nondeterministic delays either I/O or loops with data-dependent bounds. The execution time of each thread depends on whether it will be implemented in hardware or software. Initially, threads which start with a data-dependent loop are initially assigned to the software partition and all other threads are assigned to the software partition. Threads are then moved between partitions such that rate constraints are satisfied and CPU and bus utilization constraints are met.

Ernst *et al.* [26] developed a partitioning algorithm which identifies critical operations in an instruction stream and moves those operations to hardware. They measure the performance of compiled code to identify execution paths which do not meet their performance requirements. They iteratively refine the partition due to the difficulty of estimating the results of both instruction execution times and high-level synthesis. At each step, they identify an operation to move to hardware and estimate the speedup gained by moving that operation to hardware. They use an operator table to estimate the hardware speed of the basic operation. Rescheduling an operation may not always result in instruction execution speedup if the next operation depends on other values or pipeline interlocks restrict the execution time of an instruction, the value may sit idle waiting for the instruction stream to catch up. The partitioning algorithm creates a local schedule of operations to check availabilities. Communication time overhead in the basic block is estimated using data flow analysis, counting a constant number of clock cycles per variable.

It may also be desirable to map some functions onto existing specialized hardware. ASIC's may be divided into two categories: a **catalog ASIC**, such as a cache controller, is designed for a particular function but is described in a catalog; a **custom ASIC** is one of the components being designed for the current system. If the designer is willing to partition parts of the system specification which can be implemented by catalog ASIC's, the allocation of those functions to catalog ASIC components is relatively straightforward. If the function must be assembled from several ASIC's, the synthesis task is more difficult. Haworth *et al.* [37] describe part selection algorithms.

A related but distinct problem is the design of the hardware and software portions of a device interface: given a microcontroller and a set of devices, generate the interface logic and associated driver routines. The main partitioning has been done in this case, but synthesis must determine where additional hardware is required and be able to generate efficient code to control that interface logic. Chou *et al.* [13] describe one synthesis algorithm for interface design. They use a program as the behavior specification and have models for the devices, the microcontroller, and additional interface components. Their synthesis algorithm recursively allocates operations to microcontroller ports and takes advantage of special-purpose microcontroller or interface logic functions to improve the implementation.

D. Distributed System Scheduling

The scheduling of the processes on the hardware engine clearly influences system cost. The schedule must be chosen to meet hard deadlines and soft performance constraints. If a feasible schedule cannot be found, the designer has several choices: use a faster, more expensive hardware engine; repartition the software; reallocate processes to CPU's in the engine; or some combination of the above. Process scheduling over the distributed engine is the measure of feasibility of our hardware engine and software architecture.

Fernandez and Bussel [27] gave bounds on two problems: the number of processors required to execute the process data flowgraph in a given amount of time; and the time required to complete a computation on a fixed number of processors. Adam *et al.* [1] experimentally compared several scheduling heuristics on both benchmark programs and randomly generated examples: HFELT (highest levels first with estimated times); HLFNET (highest levels first with no estimated times); random; SCFET (smallest co-level first with estimated times); and SCFNET (smallest co-level first with no estimated times). All these schemes except random are variations of list scheduling: in their terminology, the level of a process is measured in the data flowgraph from the graph's sinks while the co-level is measured from the sources. Their experiments showed that HFELT gave the best results and performed close to optimally. Kasahara and Narita [50] proposed extensions on HFELT. Lee *et al.* [57], [43] proposed two algorithms for scheduling taking interprocessor communication delays into account. El-Rewini and Lewis proposed other improvements to the HFELT strategy, taking into account interprocessor communication and contention.

Leinbaugh and Yamani [61] developed algorithms to bound the amount of time required to execute a set of processes on a distributed system. Ramamritham *et al.* [80] developed a heuristic scheduling algorithm for real-time multiprocessors. Their algorithm greedily chooses a schedule which ensures that all processes meet their deadlines and minimizes a heuristic cost function; their experiments showed that minimizing the sum of minimum deadline-first and minimum-earliest-start-time-first gave the best results.

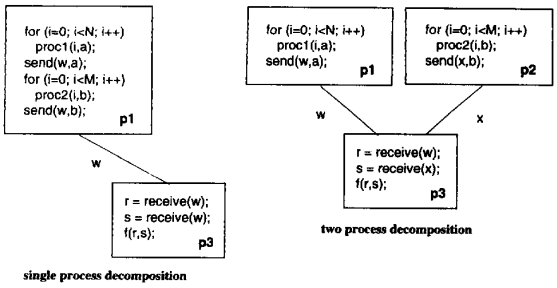

Fig. 13. How process partitioning affects distributed system performance.

E. Process Partitioning

Process partitioning affects the implementation cost of a software architecture—poor partitioning may delay a computation on one node, causing another processor to be idle while it waits for the result. Figure 13 gives one example of the problem: when process $p1$ computes two values in series, $p3$ must wait for both values to be computed, which may leave the CPU assigned to $p3$ idle. If $p1$'s computation is split into two parts which are assigned to different processors, $p3$'s processor will be idle for less time. Stalls which reduce CPU utilization may also be introduced by waits for I/O devices, particularly devices like disks which take significant amounts of time to complete a transaction.

Huang [41] studied process partitioning for distributed systems which have acyclic data flowgraphs. His algorithm merges data-flow nodes according to a simple set of rules: a pair of nodes which both precede and succeed each other are put in the same process; if a module M_i precedes all other modules in a process P_k, has all preceding, adjacent models in process P_k, or has each noninclusive preceding adjacent module precede the module included in the task with highest precedence, then M_i is included in P_k. This work only considers interprocess communicating delay, not I/O overhead.

F. Distributed Process Allocation

Process allocation affects system performance in two ways: by changing the cost of interprocess communication and by changing how tasks sharing a CPU can be scheduled. Figure 14 shows a simple example of allocation for communication. The given process graph shows that $P1$, $P2$, and $P3$ all communicate very closely, while $P4$ has little communication with the other processes. If, for example, $P1$ and $P4$ are put on one processor and $P2$ and $P3$ on another, the three tightly coupled processes will have to communicate over the link between $CPU1$ and $CPU2$. That link must have enough bandwidth to support the communication. If $P1$, $P2$, and $P3$ are put on the same CPU, they can communicate via shared memory, which is both faster and cheaper than the communication link. Process allocation can also affect the scheduling of processes on the CPU's, much as scheduling and allocation influence each other in high-level synthesis.

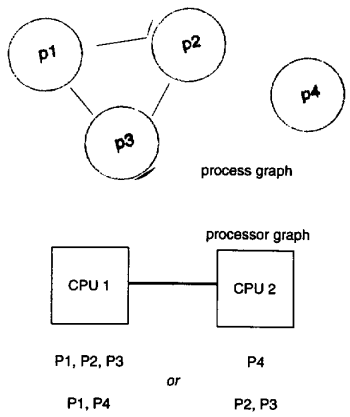

Fig. 14. Process allocation for communication efficiency.

Stone [95] developed the first algorithm for allocation of processes to processors on distributed systems. He modeled multiprocessor scheduling as a network flow problem. His formulation can be efficiently solved for two processors, but the formulation becomes more complex and its solution more difficult when there are more processors. Dasarathy and Feridun [19] developed extensions for real-time constraints.

Work on distributed system allocation quickly moved to the examination of heuristics which were effective for larger networks. Chu *et al.* [15] developed heuristics for taking interprocess communication times into account during the allocation process. Chu and Tan [14] developed heuristics to include precedence relations between processes into the optimization task. They did not use the data flowgraph directly, but instead used process size as an approximation for important precedence relationships—their heuristic assumed that smaller and larger processes are often paired: the smaller process pre-computes data and smooths the load for the larger process when both are placed on the same CPU. Shen and Tsai [84] used a graph-matching heuristic to allocate processes; their algorithm minimized interprocessor communication and balanced system load.

Researchers have also studied more computation-intensive algorithms for allocation. Ma *et al.* [63] developed a branch-and-bound algorithm for process partitioning which minimized the sum of processing and interprocess communication costs. They measured interprocess communication costs by counting the number of data objects sent from one process to another. Peng and Shin [75] developed a branch-and-bound partitioning algorithm whose objective function was minimization of the maximum normalized task response time. Their algorithm takes into account data dependencies between processes during execution.

Gopinath and Gupta [31] applied a combination of static and dynamic techniques to improve processor utilization. They statically analyze process code and assign two predictability/unpredictability and monotonicity/nonmonotonicity values to each process. They estimate

the mean and standard deviation of the execution time of each process and use that data to move less predictable code earlier in the schedule. They then use software monitors to keep track of actual execution time of processes and adjust the schedule on-line.

V. Summary

Embedded computer system design requires intimate knowledge of the interactions between the hardware and software components, even if no custom chips are designed for the system. In the mainframe world, system analysts constructed large systems for specialized tasks having relatively few choices for hardware (i.e., IBM). Microprocessors and ASIC's together provide a much larger range of choices for hardware engines than were available to early system analysts.

At present, we have a much deeper understanding of the hardware and software design disciplines separately than we do about co-design. While it is possible to design embedded systems as separate hardware and software systems, failure to consider tradeoffs between choice of hardware engine and design of application software can lead to designs that are too expensive, too slow, and never perform as intended.

A key element of our understanding of co-design is the study of system modeling in all its forms. We need a deeper understanding of the properties of CPU's, interconnect structures, and software modules. While we have many abstract models of the components of embedded systems, we do not have detailed, accurate models which reflect the idiosyncrasies of those components. It is the peculiarities of components which makes system design challenging and interesting: certain properties may cause a certain design to fail to meet a requirement, while other properties may provide an unexpectedly efficient means to meet a goal. Because embedded systems are always designed to cost and performance requirements, a comprehensive understanding of modeling which includes both high-level and detailed properties of components is essential to making the most of the components available to us.

Embedded systems also give us the opportunity for higher level software synthesis than are provided by general-purpose programming languages. Embedded applications give hard constraints, giving clear optimization goals—while difficult optimization goals require sophisticated optimization algorithms, it is impossible to optimize a system in the absence of goals. Embedded system designers are more willing than generic programmers to wait for synthesis tools to finish the compilation of embedded system software if those synthesis algorithms truly provide added value.

While embedded system designers can get the job done today, they may not always complete new designs as quickly or converge on designs as cost-effective as they could if more sophisticated design tools were available. And as powerful microprocessors become even cheaper and come into wider use, the need for tools will increase. A more comprehensive understanding of hardware–software co-design is essential to making use of the computational power provided to us by VLSI.

Acknowledgment

I would like to thank S. Malik and A. Wolfe of Princeton for any number of lively discussions on embedded system design in general and performance analysis in particular. I would also like to thank to thank A. Dunlop and N. Woo of AT&T Bell Laboratories, with whom the author collaborated on early co-design projects. The TigerSwitch design team includes S. Chinatti, R. Koshy, G. Slater and S. Sun.

References

[1] T. L. Adam, K. M. Chandy, and J. R. Dickson, "A comparison of list schedules for parallel processing systems," *Commun. ACM*, vol. 17, no. 2, pp. 685–690, Dec. 1974.

[2] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Priniciples, Techniques and Tools*. Reading, MA: Addison-Wesley, 1986.

[3] "GPS: when to buy?," *Aviation Consumer*, pp. 12–13, Apr. 1, 1993.

[4] R. A. Ballance, A. B. Maccabe, and K. J. Ottenstein, "The program dependence web: a representation supporting control-, data- and demand-driven interpretation of imperative languages," in *Proc. ACM SIGPLAN'90 Conf. on Programming Language Design and Implementation*, pp. 257–271, 1990.

[5] A. Benveniste and G. Berry, "The synchronous approach to reactive and real-time systems," *Proc. IEEE*, vol. 79, no. 9, pp. 1270–1282, Sept. 1991.

[6] F. Boussinot and R. de Simone, "The Esterel language," *Proc. IEEE*, vol. 79, no. 9, pp. 1293–1304, Sept. 1991.

[7] F. Brooks, *The Mythical Man-Month*. Reading, MA: Addison-Wesley, 1975.

[8] K. Buchenrieder and C. Veith, "CODES: A practical concurrent design environment," presented at the 1992 ACM/IEEE Int. Workshop on Hardware-Software Co-Design, Estes Park CO, Oct. 1993.

[9] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A platform for heterogeneous simulation and prototyping," in *Proc. 1991 European Simulation Conf.*, June, 1991.

[10] M. Burke, "An interval-based approach to exhaustive and incremental interprocedural data-flow analysis," *ACM Trans. Programming Languages and Systems*, vol. 12, no 3, pp. 341–395, July, 1990.

[11] M. Chiodo and A. Sangiovanni-Vincentelli, "Design methods for reactive real-time systems co-design," presented at the 1992 ACM/IEEE Int. Workshop on Hardware-Software Co-Design, Estes Park CO, Oct. 1993.

[12] M. Chiodo, P. Giusto, A. Jurecska, M. Marelli, L. Lavagno, H. Hsieh, and A. Sangiovanni-Vincentelli, "A formal specification model for hardware/software codesign," presented at the Int. Workshop on Hardware Software Co-Design, Cambridge MA, Oct. 1993.

[13] P. Chou, R. Ortega, and G. Borriello, "Synthesis of the hardware/software interface in microcontroller-based systems," in *Proc. ICCAD-92*. IEEE Computer Society Press, 1992, pp. 488–495.

[14] W. W. Chu and L. M.-T. Tan, "Task allocation and precedence relations for distributed real-time systems," *IEEE Trans. Comput.*, vol. C-36, no. 6, pp. 667–679, June 1987.

[15] W. W. Chu, L. J. Holloway, M.-T. Lan, and K. Efe, "Task allocation in distributed data processing," *IEEE Computer*, pp. 57–69, Nov. 1980.

[16] M. A. Cusumano, *Japan's Software Factories: A Challenge to U.S. Management*. Oxford, UK: Oxford Univ. Press, 1991.

[17] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "An efficient method of computing static single assignment form," *SIGPLAN Notices*, 1989; from 1989 ACM Principles of Programming Languages Conf..

[18] J. G. D'Ambrosio, S. Hu, and A. Tang, "The role of analysis in hardware/software codesign," presented at the 1993 ACM/IEEE Int. Workshop on Hardware-Software Co-Design, Cambridge MA, Oct. 1993.

[19] B. Dasarathy and M. Feridun, "Task allocation problems in the synthesis of distributed real-time systems," in *Proc. IEEE 1984 Real-Time Systems Symp.*, pp. 135–144, 1984.
[20] J. V. D'Anniballe and P. J. Koopman, Jr., "Towards execution models of distributed systems: a case study of elevator design," presented at the 1993 ACM/IEEE Int. Workshop on Hardware-Software Co-Design, Cambridge MA, Oct. 1993.
[21] A. M. Davis, *Software Requirements: Analysis and Specification.* Englewood Cliffs, NJ: Prentice Hall, 1990.
[22] G. De Micheli, *Synthesis of Digital Circuits.* New York: McGraw-Hill, 1994.
[23] M. Dorfman and R. H. Thayer, *Standards, Guidelines, and Examples on System and Software Requirements Engineering.* IEEE Computer Society Press, 1990.
[24] P. Ein-Dor and J. Feldmesser, "Attributes of the performance of central processing units: a relative performance prediction model," *Commun. ACM*, vol. 30, no. 4, pp. 308–317, Apr. 1987.
[25] H. El-Rewini and T. G. Lewis, "Scheduling parallel program tasks onto arbitrary target machines," *J. Parallel Distrib. Comput.*, vol. 9, pp. 138–153, 1990
[26] R. Ernst, J. Henkel, and Th. Benner, "Hardware-software co-synthesis for micro-controllers," *IEEE Des. & Test of Comput.*, vol. 10, no. 4, pp. 64–75, Dec. 1993.
[27] E. B. Fernandez and B. Bussell, "Bounds on the number of processors and time for multiprocessor optimal schedules," *IEEE Trans. Comput.*, vol. C-22, no. 8, pp. 745–751, Aug. 1973.
[28] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Programm. Languages Syst.*, vol. 9, no. 3, pp. 319–349, July 1987.
[29] D. Gajski, N. Dutt, A. Wu, and S. Lin, *High-Level Synthesis: Introduction to Chip and System Design.* Norwell, MA: Kluwer, 1992.
[30] N. Gehani and K. Ramamritham, "Real-time concurrent C: A language for programming dynamic real-time systems," *J. Real-Time Syst.*, vol. 3, no. 4, pp. 377–405, Dec. 1991.
[31] P. Gopinath and R. Gupta, "Applying compiler techniques to scheduling in real-time systems," in *Proc. 1990 IEEE Real-Time Systems Symp.*, pp. 247–256, 1990.
[32] A. P. Gupta, W. P. Birmingham, and D. P. Siewiorek, "Automating the design of computer systems," *IEEE Trans. CAD/ICAS*, vo. 12, no. 4, pp. 473–487, Apr. 1993.
[33] R. K. Gupta and G. De Micheli, "Hardware-software cosynthesis for digital systems," *IEEE Des. & Test Comput.*, vol. 10, no. 3, pp. 29–41, Sept. 1993.
[34] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. of Comput. Progr.*, vol. 8, pp. 231–274, 1987.
[35] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot, "STATEMATE: A working environment for the development of complex reactive systems," *IEEE Trans. Software Eng.*, vol. 16, no. 4, pp. 403–414, Apr. 1990.
[36] D. J. Hatley and I. A. Pirbhai, *Strategies for Real-Time System Specification.* Dorset House, 1988.
[37] M. S. Haworth, W. P. Birmingham, and D. E. Haworth, "Optimal part selection," *IEEE Trans. CAD/ICAS*, vol. 12, no. 10, pp. 1611–1617, Oct. 1993.
[38] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach.* Morgan Kaufman Pub. 1990.
[39] C. A. R. Hoare, *Communicating Sequential Processes.* Englewood Cliffs, NJ: Prentice-Hall, 1985.
[40] C. E. Houstis, "Module allocation of real-time applications to distributed systems," *IEEE Trans. Software Eng.*, vol. 16, no. 7, pp. 699–709, July 1990.
[41] J. P. Huang, "Modeling of software partition for distributed real-time applications," *IEEE Trans. Software Eng.*, vol. SE-11, no. 10, pp. 1113–1126, Oct. 1985.
[42] R. Hunziker and P. G. Schreier, "Field buses compete for engineers' attention. start gaining commercial support," *Personal Eng. Instrum. News*, vol. 10, no. 8, pp. 35–46, Aug. 1993.
[43] J.-J. Hwang, Y.-C. Chow, F. D. Anger, and C.-Y. Lee, "Scheduling precedence graphs in systems with interprocessor communication times," *SIAM J. Comput.*, vol. 18, no. 2, pp. 244–257, Apr 1989.
[44] T. Ben Ismail, K. O'Brien, and A. Jerraya, "Interactive system-level partitioning with PARTIF," in *Proc. EDAC'94.* IEEE Computer Society Press, 1994.
[45] Intel Corp., *80980KB Hardware Design Reference Manual*, 1989.
[46] ——, *80980KB Microprocessor Programmer's Reference Manual*, 1991.
[47] ——, *Intel Price List*, Dec. 1991.
[48] Y. Ishikawa, H. Tokuda, and C. W. Mercer, "An object-oriented real-time programming language," *IEEE Comput.*, pp. 66–73, Oct. 1992.
[49] A. Kalavade and E. A. Lee, "A hardware-software codesign methodology for DSP applications," *IEEE Des. & Test*, vol. 10, no. 3, pp. 16–28, Sept. 1993.
[50] H. Kasahara and S. Narita, "Practical multiprocessor scheduling algorithms for efficient parallel processing," *IEEE Trans. Comput.*, vol. C-33, no. 11, pp. 1023–1029, Nov. 1984.
[51] D. B. Kirk, "Process dependent static cache partitioning for real-time systems," in *Proc. 1988 Real Time Systems Symp.*. IEEE Computer Society Press, 1988, pp. 181–190.
[52] ——, "SMART (Strategic Memory Allocaiton for Real-Time) cache design," in *Proc. 1989 Real Time Systems Symp.*, IEEE Computer Society Press, 1989, pp. 229–237.
[53] D. B. Kirk and J. K. Strosnider, "SMART (Strategic Memory Allocaiton for Real-Time) cache design using the MIPS R3000," in *Proc. 11th Real Time Systems Symp.* IEEE Computer Society Press, 1990, pp. 322–330.
[54] H. Kobayashi, *Modeling and Analysis: An Introduction to System Performance Evaluation Methodology* Reading, MA: Addison-Wesley, 1978.
[55] E. Dirkes Lagnese and D. E. Thomas, "Architectural partitioning of system level synthesis of integrated circuits," *IEEE Trans. CAD/ICAS*, vol. 10, no. 7, pp. 847–860, July 1991.
[56] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantatitive System Performance: Computer System Analysis Using Queueing Network Models.* Englewood Cliffs, NJ: Prentice-Hall, 1984.
[57] C.-Y. Lee, J.-J. Hwang, Y.-C. Chow, and F. D. Anger, "Multiprocessor scheduling with interprocessor communication delays," *Operations Res. Lett.*, vol. 7, no. 3, pp. 141–147, June 1988.
[58] E. A. Lee, W.-H. Hu, E. E. Goei, J. C. Bien, and S. Bhattacharyya, "Gabriel: A design environment for DSP," *IEEE Trans. Acoust., Speech, Signal Process.*, vol. 37, no. 11, pp. 1751–1762, Nov. 1989.
[59] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proc. IEEE*, vol. 75, no. 9, pp. 1235–1245, Sept. 1987.
[60] D. W. Leinbaugh, "Guaranteed response times in a hard-real-time environment," *IEEE Trans. Software Eng.*, vol. SE-6, no. 1, pp. 85–91, Jan. 1980.
[61] D. W. Leinbaugh and M. Reza Yamani, "Guaranteed response times in a distributed hard-real-time environment," in *Proc. 1982 Real-Time Systems Symp.*. IEEE Computer Society Press, 1982, pp. 157–169.
[62] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, Jan. 1973.
[63] P.-Y. R. Ma, E. Y. S. Lee, and M. Tsuchiya, "A task allocation model for computing systems," *IEEE Trans. Comput.*, vol. C-31, no. 1, pp. 41–47, Jan. 1982.
[64] S. Malik and A. Wolfe, "Tutorial on embedded systems performance analysis," presented at ICCD'93, Cambridge MA, Oct. 1993.
[65] E. McRae, "Avoiding microcontroller processing pile-ups," *Dr. Dobb's J.*, pp. 84–92, Oct. 1993.
[66] P. Michel, U. Lauther, and P. Duzy, Eds., *The Synthesis Approach to Digital System Design.* Norwell, MA: Kluwer, 1992.
[67] J. Morfit, "A simple software profiler yields big performance gains," *Comput. Des.*, p. 68, Oct. 1992.
[68] Motorola, Inc., *CPU16 Reference Manual*, rev. 1, 1991.
[69] ——, *MC68020 23-Bit Microprocessor User's Manual*, 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, 1985.
[70] K. O'Brien, T. Ben Ismail, and A. Amine Jerraya, "A flexible communication modeling paradigm for system-level synthesis," presented at the 1993 ACM/IEEE Int. Workshop on Hardware-Software Co-Design, Cambridge MA, Oct. 1993.
[71] C. Y. Park, "Predicting deterministic execution times of real-time programs," Ph.D dissertation, Dept. Comput. Science and Eng., University of Washington, Seattle, Aug. 1992. Released as Tech. Rep. 92-08-02.

[72] C. Y. Park and A. C. Shaw, "Experiments with a program timing tool based on source-level timing scheme," *IEEE Comput.*, vol. 24, no. 5, pp. 48–57, May 1991.
[73] M. C. McFarland, A. C. Parker, and R. Camposano, "The high-level synthesis of digital systems," *Proc. IEEE*, vol. 78, no. 2, pp. 301–318, Feb. 1990.
[74] P. Paulin, C. Liem, T. May, and S. Sutarwala, "DSP design tool requirements for embedded systems, a telecommunications industrial perspective," accepted for publication in *J. VLSI Signal Process.*, Spring 1994.
[75] D.-T. Peng and K. G. Shin, "Static allocation of periodic tasks with precedence constraints in distributed real-time systems," in *Proc. 9th Int. Conf. on Distributed Computing Systems.* IEEE Computer Society Press, 1989, pp. 190–198.
[76] J. L. Peterson, *Petri Net Theory and the Modeling of Systems.* Englewood Cliffs, NJ: Prentice-Hall, 1981.
[77] L. L. Pollock and M. L. Soffa, "An incremental version of iterative data flow analysis," *IEEE Trans. Software Eng.*, vol. 15, no. 12, pp. 1537–1549, Dec. 1989.
[78] S. Prakash and A. C. Parker, "SOS: Synthesis of application-specific heterogeneous multiprocessor systems," *J. Parallel Distributed Comput.*, vol. 16, pp. 338–351, 1992.
[79] P. Puschner and Ch. Koza, "Calculating the maximum execution times of real-time programs," *J. Real-Time Syst.*, vol. 1, pp. 159–176, 1989.
[80] K. Ramamritham, J. A. Stankovic, and P.-F. Shiah, "Efficient scheduling algorithms for real-time multiprocessor systems," *IEEE Trans. Parallel Distributed Syst.*, vol. 1, no. 2, pp. 184–194, Apr. 1990.
[81] C. Rosebrugh and E.-K. Kwang, "Multiple microcontrollers in an embedded system," *Dr. Dobbs J.*, pp. 48–57, Jan. 1992.
[82] R. Roth, J. Watkins, M. Hsieh, W. Radke, D. Hejna, R. Tom, and B. Kim, "An integrated environment for concurrent development of a pixel processor ASIC and application software," in *Proc. ICCD'93.* IEEE Computer Society Press, 1993, pp. 116–125.
[83] M. Schmit, "Optimizing Pentium code," *Dr. Dobb's J.*, pp. 40–49, Jan. 1994.
[84] C.-C. Shen and W.-H. Tsai, "A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion," *IEEE Trans. Comput.*, vol. C-34, no. 3, pp. 197–203, Mar. 1985.
[85] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Real-time synchronization for multiprocessors," in *Proc. 9th IEEE Real-Time Systems Symp..* IEEE Computer Society Press, 1988, pp. 259–269.
[86] A. C. Shaw, "Reasoning about time in higher-level language software," *IEEE Trans. Software Eng.*, vol. 15, no. 7, July 1989.
[87] ——, "Deterministic timing schema for parallel programs," in *Proc. 5th Int. Parallel Processing Symp..* IEEE Computer Society Press, 1991, pp. 56–63.
[88] S. Shlaer and S. J. Mellor, *Object Lifecycles: Modeling the World in States.* Yourdon Press, 1992.
[89] Signetics Corp., "The I^2C-bus and how to use it (including specification)," Jan. 1992.
[90] A. Smailagic and D. P. Siewiorek, "A case study in embedded system design: the VuMan 2 wearable computer," *IEEE Des. & Test Comput.*, vol. 10, no. 3, pp. 56–67, Sept. 1993.
[91] C. U. Smith, *Performance Engineering of Software Systems.* Reading, MA: Addison-Wesley, 1990.
[92] C. U. Smith, G. A. Frank, and J. L. Cuadrado, "An architecture design and assessment system for software/hardware codesign," in *Proc. 22nd Design Automation Conf..* IEEE Computer Society Press, 1985, pp. 417–424.
[93] M. B. Srivastava and R. W. Brodersen, "Rapid-prototyping of hardware and software in a unified framework," in *Proc. ICCAD-91.* IEEE Computer Society Press, 1991, pp. 152–155.
[94] M. B. Srivastava, T. I. Blumenau, and R. W. Brodersen, "Design and implementation of a robot control system using a unified hardware-software rapid-prototyping framework," in *Proc. ICCD'92.* IEEE Computer Society Press, 1992.
[95] H. S. Stone, "Multiprocessor scheduling with the aid of network flow algorithms," *IEEE Trans. Software Eng.*, vol. SE-3, no. 1, pp. 85–93, Jan. 1977.
[96] J. K. Strosnider, "Highly-responsive real-time token rings," Ph.D. dissertation, Dep. Electrical Comput. Eng., Carnegie Mellon University, Pittsburgh, PA, Aug. 1988.
[97] A. Takach, M. Leeser, and W. Wolf, "An automaton model for scheduling constraints in synchronous machines," accepted for publication in *IEEE Trans. Comput..*
[98] F. K. Teichmann, *Airplane Design Manual*, 4th ed. New York: Pitman, 1958.
[99] Texas Instruments, Inc.,*TMS320C4x User's Guide*, 1991.
[100] W. Wolf, *Modern VLSI Design: A Systems Approach.* Englewood Cliffs, NJ: P T R Prentice-Hall, 1994.
[101] ——, "Software cache partitioning," accepted for publication in *Int. J. Comput. Software Eng.*
[102] W. Ye, R. Ernst, Th. Benner, and J. Henkel, "Fast timing analysis for hardware-software co-synthesis," in *Proc. ICCD'93.* IEEE Computer Society Press, 1993.

Wayne H. Wolf (Senior Member, IEEE) received the B.S., M.S., and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, in 1980, 1981, and 1984, respectively.

He is Assistant Professor of Electrical Engineering at Princeton University, Princeton, NJ. Before joining Princeton University he was with AT&T Bell Laboratories, Murray Hill, NJ. His research interests include computer-aided design for VLSI and embedded systems as well as application-driven architecture.

Dr. Wolf is a member of Phi Beta Kappa and Tau Beta Pi, and a senior member of the ACM.