Hardware/Software Co-Design

GIOVANNI DE MICHELI, FELLOW, IEEE, AND RAJESH K. GUPTA, MEMBER, IEEE

Invited Paper

Most electronic systems, whether self-contained or embedded, have a predominant digital component consisting of a hardware platform which executes software application programs. Hardware/software co-design means meeting system-level objectives by exploiting the synergism of hardware and software through their concurrent design. Co-design problems have different flavors according to the application domain, implementation technology and design methodology.

Digital hardware design has increasingly more similarities to software design. Hardware circuits are often described using modeling or programming languages, and they are validated and implemented by executing software programs, which are sometimes conceived for the specific hardware design. Current integrated circuits can incorporate one (or more) processor core(s) and memory array(s) on a single substrate. These "systems on silicon" exhibit a sizable amount of embedded software, which provides flexibility for product evolution and differentiation purposes. Thus the design of these systems requires designers to be knowledgeable in both hardware and software domains to make good design tradeoffs.

This paper introduces the reader to various aspects of co-design. We highlight the commonalities and point out the differences in various co-design problems in some application areas. Co-design issues and their relationship to classical system implementation tasks are discussed to help the reader develop a perspective on modern digital system design that relies on computer-aided design (CAD) tools and methods.

I. INTRODUCTION

Most engineering designs can be viewed as systems, i.e., as collections of several components whose combined operation provides useful services. Components can be heterogeneous in nature and their interaction may be regulated by some simple or complex means. Most examples of systems today are either electronic in nature (e.g., information processing systems) or contain an electronic subsystem for monitoring and control (e.g., plant control). Moreover, the implementation of electronic systems and subsystems shows often a predominant digital component.

We focus in this paper on the digital component of electronic systems, and refer to them as (digital) systems for brevity. The majority of such systems are programmable, and thus consist of hardware and software components. The value of a system can be measured by some objectives that are specific to its application domain (e.g., performance, design, and manufacturing cost, and ease of programmability) and it depends on both the hardware and the software components. *Hardware/software co-design* means meeting system-level objectives by exploiting the synergism of hardware and software through their concurrent design. Since digital systems have different organizations and applications, there are several co-design problems of interest. Such problems have been tackled by skillful designers for many years, but detailed-level design performed by humans is often a time-consuming and error-prone task. Moreover, the large amount of information involved in co-design problems makes it unlikely that human designers can optimize all objectives, thus leading to products whose value is lower than the potential one.

The recent rise in interest in hardware/software co-design is due to the introduction of *computer-aided design* (CAD) tools for co-design (e.g., commercial simulators) and to the expectation that solutions to other co-design problems will be supported by tools, thus raising the potential quality and shortening the development time of electronic products. Due to the extreme competitiveness in the marketplace, co-design tools are likely to play a key strategic role. The forecast of the worldwide revenues of integrated circuit sales (Fig. 1), and in particular for those used in dedicated applications (Fig. 2), explains the high demand of electronic system-level design tools, whose volume of sales is expected to grow at a compound annual rate of 34% in the 1993–1998 time frame, according to Dataquest.

The evolution of integrated circuit technology is also motivating new approaches to digital circuit design. The trend toward smaller mask-level geometries leads to higher integration and higher cost of fabrication, hence to the need

Manuscript received February 1, 1996; revised December 2, 1996. This work was supported in part by DARPA, under Contract DABT 63-95-C-0049, and in part by NSF CAREER Award MIP 95-01615.

G. De Micheli is with the Computer Systems Laboratory, Stanford University, Stanford, CA 94305 USA (e-mail: nanni@galileo.stanford.edu).

R. K. Gupta is with the Department of Computer Science, University of California, Irvine, CA 92797 USA (e-mail: rgupta@ics.uci.edu).

Publisher Item Identifier S 0018-9219(97)02017-3.

0018–9219/97$10.00 © 1997 IEEE

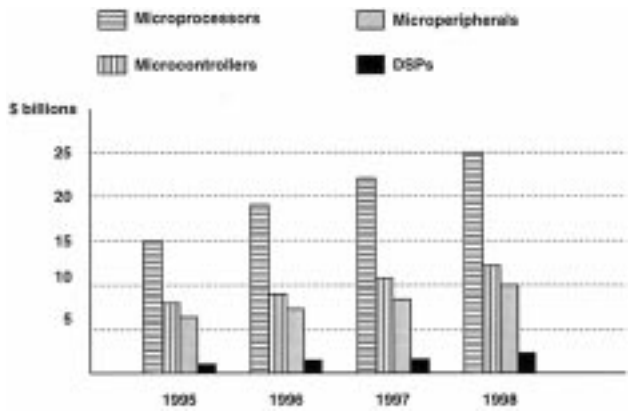

Fig. 1. Forecast of the growth of electronic components. (Source: Dataquest.)

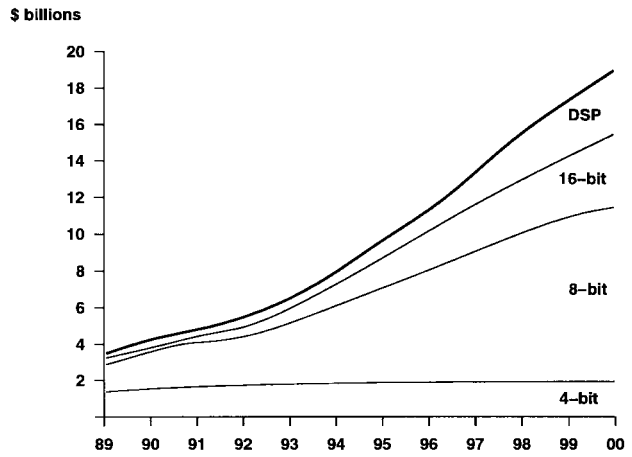

Fig. 2. Worldwide revenues for the sales of microcontrollers and DSP integrated circuits. The bold line shows the total growth. (Source: ICE.)

of amortizing hardware design over large production volumes. This suggests the idea of using software as a means of differentiating products based on the same hardware platform. Due to the complexity of hardware and software, their *reuse* is often key to commercial profitability. Thus complex macrocells implementing *instruction-set processors* (ISP's) are now made available as processor *cores* (Fig. 3 [38]). Standardizing on the use of cores or of specific processors means leveraging available software layers, ranging from operating systems to embedded software for user-oriented applications. As a result, an increasingly larger amount of software is found on semiconductor chips, which are often referred to as *systems on silicon*. Thus hardware (e.g., cores) and software (e.g., microkernels) can be viewed as commodities with large *intellectual property* values. Today both the electronic market expansion and the design of increasingly complex systems is boosted by the availability of these commodities and their reuse as system building blocks.

The recent introduction of *field-programmable gate array* (FPGA) technologies has blurred the distinction between hardware and software. Traditionally a hardware circuit used to be configured at manufacturing time. The functions of a hardware circuit could be chosen by the execution of a program. Whereas the program could be modified even at run-time, the structure of the hardware was invariant. With field-programmable technology it is possible to configure the gate-level interconnection of hardware circuits after manufacturing. This flexibility opens new applications of digital circuits, and new hardware/software co-design problems arise. For example, one (or more) FPGA circuits may be configured on-the-fly to implement a specific software function with better performances than executing the corresponding code on a microprocessor. Subsequently, the FPGA can be *re*programmed to perform another specific function without changing the underlying hardware. Thus from a user perspective, a reprogrammable hardware board can perform a function indistinguishable from that of a processor. Nevertheless the programming mechanisms and the programmer's view of the hardware is very different.

Hardware/software co-design is a complex discipline, that builds upon advances in several areas such as software compilation, computer architecture and *very large scale integration* (VLSI) circuit design. Co-design is perceived as an important problem, but the field is fragmented because most efforts are applied to specific design problems. Thus co-design has a different flavor according to the context in which it is applied. For example, co-design can be seen as a management discipline to achieve complex system products [15].

It is the purpose of this special issue to shed some light on the recent developments of co-design in different application domains. In this paper, we want to put several co-design problem in perspective, to show differences and similarities, as well as to show the cross fertilization in different scientific fields. For this reason, we describe first distinguishing features of electronic systems that are useful to classify co-design problems. We consider next system-level co-design issues for different kinds of electronic systems and components. Eventually, we review the fundamental algorithmic approaches for system-level design and organization of hardware/software systems, that form the foundations for system-level design tools.

II. Distinguishing Features of Electronic Systems

We associate co-design problems with the classes of digital systems they arise from. We attempt to characterize these systems using some general criteria, such as domain of *application*, degree of *programmability*, and *implementation* features.

A. Application Domains

A digital system may be providing a service as a self-contained unit, or as a part of a larger system. A traditional computer (with its peripherals) is an example of the first kind of systems. A digital control system for a manufacturing plant is an example of the latter case. Systems that fall in this second category are commonly referred to as *embedded systems*.

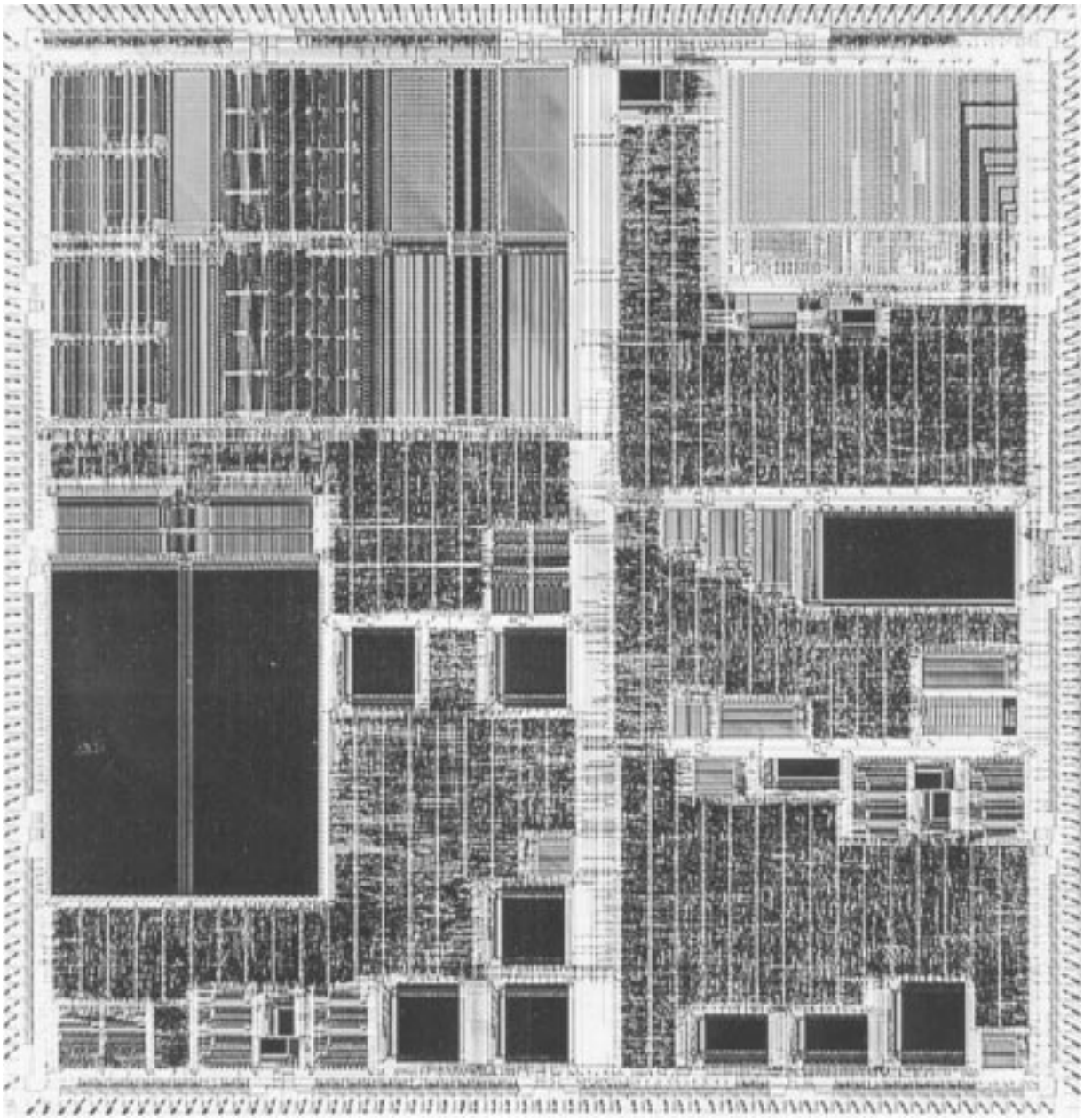

Fig. 3. Example of an integrated circuit with programmable cores. The VCP chip has two processors: the VC core, which is based on the MIPS-X processor, is placed in the top right corner, while the VP+ DSP processor occupies the top left part of chip above a memory array. (Courtesy of Integrated Information Technology.)

The term embedded means being part of a larger unit and providing a dedicated service to that unit. Thus a personal computer can be made the embedded control system for manufacturing in an assembly line, by providing dedicated software programs and appropriate interfaces to the assembly line environment. Similarly, a microprocessor can be dedicated to a control function in a computer (e.g., keyboard/mouse input control) and be viewed as an embedded controller.

Digital systems can be classified according to their principal domain of application. Examples of self-contained (i.e., nonembedded) digital systems are information processing systems, ranging from laptop computers to supercomputers, as well as emulation and prototyping systems. Applications of embedded systems are ubiquitous in the manufacturing industry (e.g., plant and robot control), in consumer products (e.g., intelligent home devices), in vehicles (e.g., control and maintenance of cars, planes, ships), in telecommunication applications, and in territorial and environmental defense systems.

Digital systems can be geographically distributed (e.g., telephone network), locally distributed (e.g., aircraft control with different processing units on a local area network), or lumped (e.g., workstations). In this paper, we consider a system to be lumped when it is concentrated in a physical unit, although it may involve more than one processing unit.

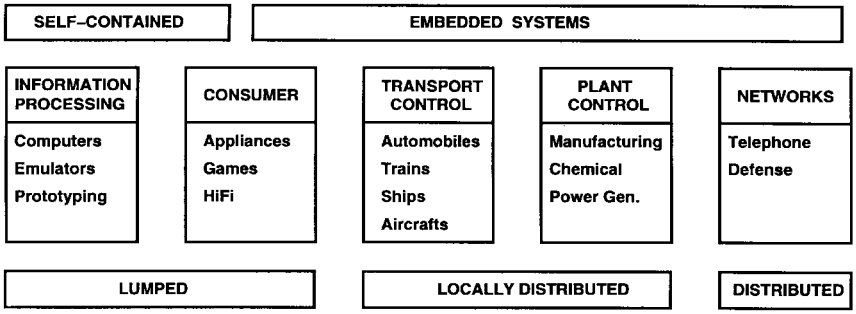

Fig. 4. Some application domains of electronic systems.

B. Degree of Programmability

In most digital systems, the hardware is programmed by some software programs to perform the desired functions. (Nonprogrammable hardwired systems are few and not relevant to this survey.) Hence the abstraction level used for programming models is the means of interaction between hardware and software. There are two important issues related to programming: 1) who has access to programming the system and 2) the technological levels at which programming is performed.

1) Access to Programming: To understand the extent to which system programmability has an effect on system design, we distinguish end-users from application developers, system integrators, and component manufacturers. Historically, each of these groups represents a separate industry or a separate organization. The application developer requires systems to be retargetable so as to port a given application across multiple hardware platforms. The chief use of system programmability for a system integrator is in ensuring compatibility of system components with market standards. Finally, the component manufacturer is concerned with maximizing component or module reuse across its product lines.

Let us take the personal computer as an example. The end-user programming is often limited to application-level programming (such as a spreadsheet) or scripting. An application developer relies on the programming language tools, operating system, and the high-level programming environment for application development. Most of these ingredients have, to a large extent, become off-the-shelf commodity products for the personal computer industry, bringing programming closer to the end-user as well. Component manufacturing for personal computers is driven by interconnection bus and protocol standards. Increasing semiconductor densities have resulted in coalescing system components, even with very diverse functionalities, onto single chips, leading to fewer but more versatile system hardware components.

When considering embedded systems, the end-user has limited access to programming because most software is already provided by the system integrator, who is often also the application developer. For example, the motion-control system of a robot arm for use in manufacturing contains embedded software for coordinating the movement of the different mechanical parts. The user can program only the moves and the actions.

2) Levels of Programming: Digital systems can be programmed at different levels, namely the *application*, *instruction*, or *hardware* levels. The highest abstraction level is the application level, where the system is running dedicated software programs that allow the user to specify desired functionality options using a specialized language. Examples range from programming a videocassette recorder (VCR) to setting navigation instructions in an automated steering controller of a ship.

Most digital systems use components with an *instruction set architecture* (ISA). Examples of such components are microprocessors, microcontrollers, and programmable *digital signal processors* (DSP's). The instruction set defines the boundary between hardware and software, by providing a programming model of the hardware. Instruction-level programming is achieved by executing on the hardware the instructions supported by the architecture. It is important to note that in some systems the end-user can compile programs to execute on the ISA, as in the case of computers. In other systems, such as in some embedded systems, the ISA is not visible to the user because it runs embedded software. In the former case, the compiler must have user-friendly features (e.g., descriptive diagnostic messages) and adequate speed of compilation, while in the latter case the compiler algorithms can afford to be more computationally expensive in the interest of a better final result [74] (i.e., more compact machine code).

Hardware-level programming means configuring the hardware (after manufacturing) in a desired way. A simple and well-known example is microprogramming, i.e., determining the behavior of the control unit by a microprogram, which can be stored in binary form in a memory. Emulation of other architectures can be achieved by altering the microprogram. Today microprogramming is common for DSP's, but not for general purpose microprocessors using RISC architectures [52] mainly due to performance reasons.

Reconfigurable circuits are the limiting case of hardware-level programming. Field-programmable technology allows us to configure the interconnections among logic blocks and to determine their personality. Reconfiguration can be global or local (i.e., the entire circuit or a portion thereof can

be altered), and may be applied more than once. Whereas microprogramming allows us to (re)configure the control unit, reconfigurable systems can be modified in both the data path and controller. Moreover, such circuits need not to be partitioned into data path and control unit, but they can be organized with wide freedom.

Overall, reconfigurabilty increases the usability of a digital system, but it does not increase its performances except on tailored applications. For general-purpose computing, top performance is achieved today by superscalar RISC architectures [52], which are programmed at the instruction level. For dedicated applications, hard-wired (nonprogrammable) *application-specific integrated circuits* (ASIC's) achieve often the best performance and the lowest power consumption. In both cases, hardware design may be expensive, because of *nonrecurrent engineering* (NRE) costs, and not flexible enough to accommodate engineering changes and upgrades. Thus the challenge for reconfigurable design technologies is to arrive at a competitive level of performance, while exploiting the hardware flexibility in addressing other important system-level issues, such as support for engineering changes, self-adaptation to the environment, and fault-tolerance.

C. Implementation Features

System implementation deals with circuit design style, manufacturing technology and integration level. We touch briefly on these issues, because we want to maintain a high-level view of the problem which is fairly independent of the physical implementation.

Digital systems rely on VLSI circuit technology. The circuit design style relates to the selection of circuit primitives (e.g., choice of library in a semicustom technology), clocking strategy (e.g., single/multiple clocks and asynchronous), and circuit operation mode (e.g., static and dynamic).

A system may have components with different scale of integration (e.g., discrete and integrated component) and different fabrication technologies (e.g., bipolar and CMOS). The choice of hardware technology for the system components affects the overall performance and cost, and therefore is of primary importance.

System-level field programmability can be achieved by storing programs in read/write memories and/or exploiting programmable interconnections. In the former case, the software component is programmed, while in the latter the hardware is configured. With field-programmable technologies, circuit configuration is achieved by programming connections using transistors driven by memory arrays [110] or by antifuses [44]. Circuits of the first type are reprogrammable and will be considered in this survey, while the others can be programmed only once.

When considering co-design problems for lumped systems, we can distinguish between systems consisting of components (like ASIC's, processors, and memories) mounted on a board or chip carrier and single-chip systems consisting of an ASIC with one or more processor cores and/or memories. The programmable core is usually a processor provided as a macro-cell in the ASIC implementation technology [38, Fig. 3]. Whereas a core may provide the same functionality as the corresponding standard part, cost and performance considerations may bias the choice of integration level. The advantages of higher integration are usually higher reliability, lower power-consumption budget, and increased performance. The last two factors come from the lack of I/O circuitry in the core and its direct connection to the application-specific logic. The disadvantages are larger chip sizes and higher complexity in debugging the system.

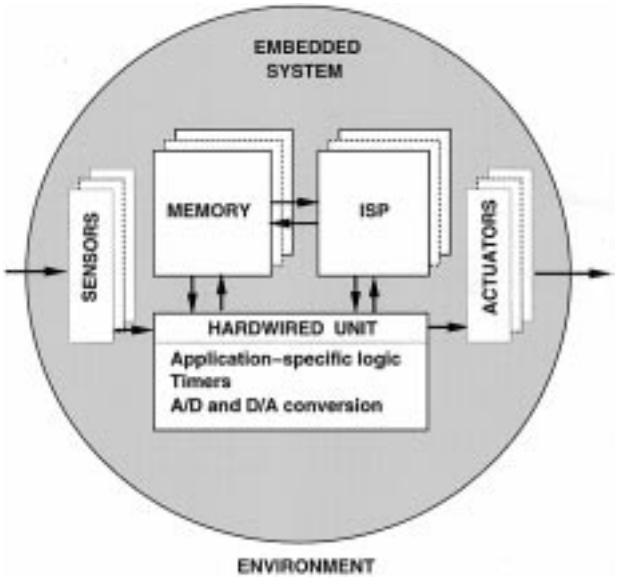

Fig. 5. Scheme of the essential parts of an embedded control system with one (or more) ISP(s).

III. General Co-Design Problems and Design Approaches

We consider now different facets of co-design. Namely, we present first the major objectives in embedded system design. We describe next the design of ISA's and their use in both self-contained information processing systems and embedded systems. Last but not least, we address the major co-design issues for reconfigurable systems.

A. Co-Design of Embedded Systems

Embedded systems are elements of larger systems. Some embedded systems provide monitoring and control functions for the overall system (e.g., vehicular control, manufacturing control), while others perform information-processing functions within a network (e.g., telecommunication systems).

Embedded control systems usually regulate mechanical components via *actuators* and receive input data provided by *sensors*. Single-chip implementation of embedded controllers often integrate on the same chip analog to digital conversion (and vice versa) of some I/O signals and sometimes the sensors themselves (e.g., accelerometer for airbag control). Often embedded control systems have also a data-processing component (Fig. 5).

Control systems are *reactive* systems, because they are meant to react to stimuli provided by the environment. *Real-time* control systems [99] implement functions that must execute within predefined time windows, i.e., satisfy some *hard* or *soft* timing constraint [117], [118]. Timers are thus important components of real-time controllers.

The required functions and size of embedded controllers may vary widely. Standard programmable microcontrollers and microcontroller cores provide usually low-cost and flexible solutions to a wide range of problems. Nevertheless, control systems that are either complex (e.g., avionic controls) or that require high-throughput data processing (e.g., radio navigation) need specific designs that leverage components or cores such as microprocessors or DSP's.

Whereas performance is the most important design criterion for information processing systems, *reliability*, *availability*, and *safety* are extremely important for control systems. System reliability measures the probability of correct control operation even in presence of failures of some component, whereas availability contemplates the on-line repair of faulty components. Safety measures the probability of avoiding catastrophic failures. For example, the availability of a nuclear steam supply system is the probability (as a function of time) that a nuclear reactor can produce energy under a scheduled maintenance, while its safety is the probability that the system has no failure leading to a hazardous situation for the operators and the environment.

Since control functions can be implemented both in hardware and in software, specific design disciplines must be used to insure reliability, availability, and safety. Some formal verification techniques for embedded controllers are nowadays available to insure design correctness by comparing different representation levels and assessing system properties. System-level testing techniques must be used to check the correctness of operation of the physical system implementation. Functional redundancy may be used to enhance reliability.

Specific co-design problems for embedded systems include modeling, validation, and implementation. These tasks may be complex because the system function may be performed by different components of heterogeneous nature, and because the implementation that optimizes the design objectives may require a specific hardware/software partition. The design of embedded control systems is surveyed in [36].

Embedded systems for telecommunication applications involve data processing, where data can be a digital form of audio and video information. Data compression, decompression, and routing is often performed with the aid of programmable processors of various kinds. Co-design issues in this domain are described in [17], [43], and [88].

B. Co-Design of ISA's

The concept of ISA plays a fundamental role in digital system design [52]. An ISA provides a programmer's view of hardware by supporting a specific programming model. The definition of an instruction set permits the parallel development of the hardware and of the corresponding compiler. Components based on ISA's, i.e., (e.g., microprocessors, DSP's, and programmable microcontrollers) are commonly used in (self-contained) information processing systems and in embedded systems. Therefore a good ISA design is critical to achieving system usability across applications. In addition, increasing applications now demand processor performance that pushes the limits of semiconductor technology. Performance critical design of processors requires combined design of hardware and software elements.

The primary goal of co-design in ISP development is to optimize utilization of the underlying hardware. This is done by customizing the software development ranging from application programs to operating systems. The operating system is the software layer closest to the underlying hardware, and its role is different in computing and embedded systems (see Section IV-B3) The extent of the operating system layer in embedded systems varies from specialized real-time kernels [99] to lightweight run-time schedulers [45], according to the design goals and requirements.

Compiler development should start as early as ISA definition. Indeed, compilers are needed for the evaluation of the instruction-set choices and overall processor organization, in order to verify whether the overall performance goals are met [53]. Whereas retargetable compilers are useful in the architectural development phase, optimizing compilers are key to achieving fast-running code on the final products. In addition, speed of compilation is important for information processing applications (e.g., computers), where the end-user programs the system at the instruction-level (i.e., via programs in programming languages). Such a requirement is less important for applications of ISA's within embedded systems, where the user has limited access to programmability, as mentioned in Section II-B.

The organization of modern general-purpose processors exploits deep pipelines, concurrency, and memory hierarchies. Hardware/software trade-off is possible in pipeline-control [57] and cache-management mechanisms [53]. The selection of an instruction set is usually guided by performance and compatibility goals. This task is generally based on experience and on the evaluation of software simulation runs, although recent efforts have aimed at developing tools for computer-assisted instruction set selection [55].

Let us now consider the use of components based on ISA's for embedded data-processing systems. Such systems often use dedicated software programs with specific instruction profiles. Therefore significant performance improvements may be achieved by selecting particular instruction sets which match the application requirements. In some application domains, such as data processing for telecommunications (see Fig. 4), it has been shown practical to replace standard processors by *application-specific instruction set processors* (ASIP's), which are instruction-level programmable processors with an architecture tuned to a specific application [43], [88]. In an ASIP design, the instruction set and hardware structure are chosen to support

efficiently the instruction mix of the embedded software of the specific application. For example, ASIP's feature particular register configurations and specific interconnections among the registers, busses, and other hardware resources.

It is possible to view ASIP's as intermediate solutions between ISP's and ASIC's. ASIP's are more flexible than ASIC's, but less than ISP's. Nevertheless, they can be used for a family of applications in a specific domain. The performance of an ASIP on specific tasks can be higher than an instruction-set processor (because of the tuning of the architecture to the instruction mix) but it is usually lower than an ASIC. Opposite considerations apply to power consumption. The ASIP design time and nonrecurring engineering costs can be amortized over a larger volume than an ASIC, when the ASIP has multiple applications. Moreover, engineering changes and product updates can be handled graciously in ASIP's by reprogramming the software and avoiding hardware redesign. Unfortunately, because an ASIP is a specific architecture, it requires a compiler development which adds to the nonrecurring engineering costs. Such a compiler must also produce high-quality machine code to make the ASIP solution competitive. On the other hand, compilation speed is not a major requirement, since most ASIP-based system programmed (once only) by the manufacturer and not by the end-user.

Differently from general-purpose and digital-signal processors, ASIP's may be designed to support fairly different instruction sets, because compatibility requirements are less important and supporting specific instruction mixes is a desired goal. Unfortunately the price of the flexibility in choosing instruction sets is the need of developing application-specific compilers. Despite the use of *retargetable-compiler* technology [79], the computer-aided development of compilers that produce high-quality code for specific architectures is a difficult problem and solved only in part to date, namely for fixed-point arithmetic operations. Problems and solutions in retargetable compilation are addressed by [43] and [88] in this issue.

C. Co-Design of Reconfigurable Systems

Reconfigurable systems exploit FPGA technology, so that they can be personalized after manufacturing to fit a specific application. The operation of reconfigurable systems can either involve a configuration phase followed by an execution phase or have concurrent (partial) configuration and execution. In the latter case, the systems are called *evolvable*.

We consider first nonevolvable systems and their applications to the acceleration of computation and to prototyping. In both cases, the overall digital systems include a reconfigurable subsystem that emulates the software or the hardware execution, and sometimes a combination of both.

Let us turn our attention to co-design techniques that can accelerate software execution. There are often bottlenecks in software programs that limit their performance (e.g., executing transcendental floating-point operations or inner loops where sequences of operations are iterated). ASIC coprocessors can reduce the software execution time, when they are dedicated to support specific operations (e.g., floating-point or graphic coprocessors) or when they implement the critical loops in hardware while exploiting the local parallelism. Whereas ASIC coprocessors accelerate specific functions, coprocessors based on reconfigurable hardware can be applied to the speedup of arbitrary software programs with some distinctive characteristics (e.g., programs with parallelizable bit-level operations).

One of the first examples of programmable coprocessors is provided by the *programmable active memories* (PAM's) [13], which consist of a board of FPGA's and local memory interfaced to a host computer. Two models of PAM's, named *PeRLe-0* and *PeRLe-1*, were built. They differ in the number and type of FPGA used, as well as operating frequency. The hardware board for *PeRLe-1* is shown in Fig. 6 [112].

To accelerate the execution of a program with a PAM, the performance-critical portion of the program is first extracted and compiled into the patterns that configure the programmable board. Then, the noncritical portion of the program is executed on the host, while the critical portions are emulated by the reconfigurable subsystem. Experimental results show a speedup of one to two orders of magnitude, on selective benchmark programs, as compared to the execution time on the host [13].

The major hardware/software co-design problems consist of identifying the critical segments of the software programs and compiling them efficiently to run on the programmable hardware. The former task is not yet automated for PAM's and is achieved by successive refinement, under constraints of communication bandwidth and load balancing between the host and the programmable hardware. The latter task is based on hardware synthesis algorithms, and it benefits from performance optimization techniques for hardware circuits [13], [30]. Several other systems for software acceleration have been implemented [7], [86].[1]

A different application of reconfigurable systems is in *computer-aided prototyping*. In this case, we are interested in validating a target system yet to be manufactured by configuring and executing a prototype implemented with a reconfigurable medium. Prototypes provide design engineers with more realistic data on correctness and performance than system-level simulation [81], thus reducing the likelihood of an expensive redesign of the target system.

Prototyping of complex digital systems including multiple hardware components and software programs is appealing to designers, because it allows testing software programs on hardware, while retaining the ability to change the hardware (and software) implementation concurrently. Once the hardware configuration has been finalized, it can be mapped onto a "hard" silicon implementation using synthesis systems [30] that accept as inputs hardware models compatible with those used by the emulation systems (e.g., *VHDL* [91] and *Verilog HDL* [107] models).

[1] See URL http://www.io.com/~guccione/HW_list.html for a comprehensive list.

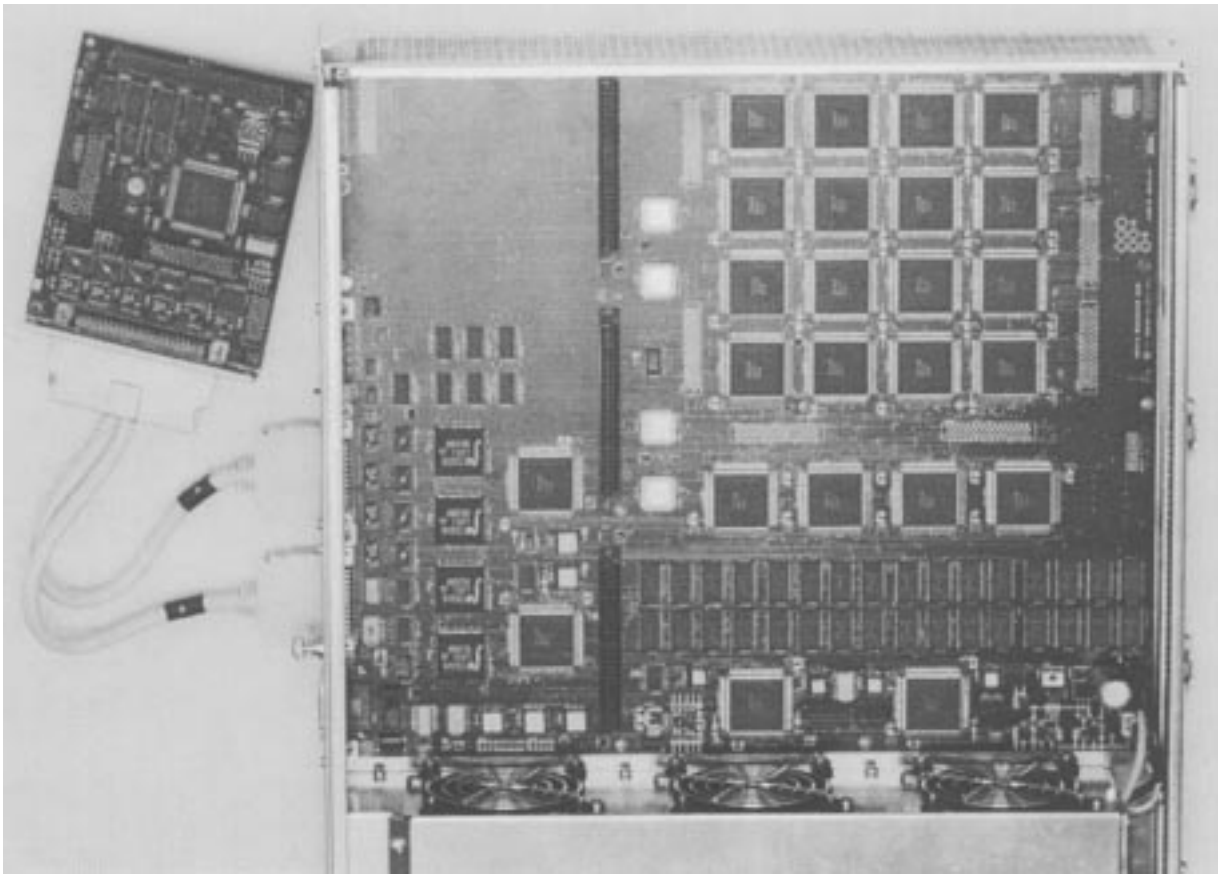

Fig. 6. The *PeRLe-1* implementation. (Courtesy of P. Bertin.)

Evolvable systems [97] are digital systems where reconfiguration of one of its parts is concurrent with execution. One of the goals of evolvable systems is to adapt automatically to the environment. As an example, consider a network interface unit, that receives and retransmits data with different formats. Upon sensing the protocol and format of the incoming data, such a unit configures itself to optimize data translation and transmission. Whereas such a unit could be implemented with nonevolvable technology, the ability to reconfigure the hardware may be the key to sustain higher data rates.

Fault tolerance in evolvable systems can be obtained by detecting the malfunctioning unit, and by reconfiguring a part of the system to regenerate a fault-free replacement of the faulty unit. This can be achieved under several assumptions, some of which are typical of fault-tolerant system design, including that of having enough spare reconfigurable circuits to implement the faulty unit on-the-fly.

Evolvable systems are the subject of several research efforts [97]. An interesting application of reconfigurable hardware for fault-tolerance applications is *embryonics* (embryological electronics) [28], [78], a discipline where biological models of organization are used for electronic system design. There are a few implementations of embryonic systems, relying on this general implementation strategy [78]. The underlying hardware is a memory-based field-programmable circuit that uses a decision diagram structure. The hardware is organized as a rectangular matrix of cells, each one addressable by its coordinates and communicating with its neighbors. The overall system function is mapped onto the programmable cells. Circuit configuration is performed by feeding each cell with a compiled software program (bit stream) containing the functionality of the entire system. This parallels the organization of multicellular living beings, where the genome of each cell is a repository of information of the entire being. The program is transmitted from an initial cell to the others. Then each cell extracts the portion of the overall program pertinent to its operation (using the coordinate information) and configures itself. This parallels the synthesis of a living cell from the gene structure.

After a boot phase in which the information is transmitted to all cells and the cells self-configure, the system can start operating. Upon failure of a cell in providing the required function, the neighboring cells readapt their operations so that the faulty cell is replaced by a working clone, and the overall system function is preserved. This reconfiguration, called *cicatrization*, allows the system to recover from failures after a finite delay.

Interesting applications of embryological circuits include embedded system applications with high reliability requirements, such as control of unmanned spacecrafts or of robots operating in hostile environments. Hardware/software codesign problems relate to how software programs are used to configure and program the underlying hardware circuit, as well as to how the reconfigurable circuit is organized.

IV. Design of Hardware/Software Systems

We consider here the high-level (i.e., technology independent) steps in the design of hardware/software systems. We consider the problem in its breadth, rather than in its depth, to highlight similarities and differences in co-design of digital systems of different nature. We also draw parallels among techniques applicable to hardware and software (e.g., scheduling). We refer the interested reader to the other articles in this issue for an in-depth analysis applicable to different domains.

The design of hardware/software systems involves *modeling*, *validation*, and *implementation*. We call modeling the process of conceptualizing and refining the specifications, and producing a hardware and software model. We call validation the process of achieving a reasonable level of confidence that the system will work as designed, and we call implementation the physical realization of the hardware (through synthesis) and of executable software (through compilation).

When considering embedded systems, different modeling paradigms and implementation strategies may be followed [36]. We exclude here pure hardware (e.g., ASIC) and pure software (e.g., embedded software running on an ISA) implementations, because we concentrate on co-design. Therefore, the overall model of an embedded system involves both hardware and software. The modeling style can be *homogeneous* or *heterogeneous*. In the former case, a modeling language (e.g., the C programming language) or a graphical formalism (e.g., Statecharts [51]) is used to represent both the hardware and software portions. A hardware/software *partitioning* problem can then be stated as finding those parts of the model best implemented in hardware and those best implemented in software. Partitioning can be decided by the designer, with a successive refinement and annotation of the initial model, or determined by a CAD tool. We will consider computer-aided partitioning in more detail in Section IV-A.

When using a heterogeneous modeling style, the hardware/software partition is often outlined by the model itself, because hardware and software components may be expressed in the corresponding languages. Nevertheless, system designers may want to explore alternative implementations of some components. For example, the first release of a product may contain a sizable software component (for time to market and flexibility reasons) while later releases may implement part of this software in hardware for performance and/or cost reasons. Tools which support implementation *retargeting* help the designer avoiding manual translation of the models or parts thereof. A few CAD environments for heterogeneous design and retargeting have been realized [8], [24], [26], [29], [64], [114].

ISA's are modeled at different levels. Instruction sets provide the essential information about the architecture, supporting both hardware and software development. The processor organization is usually described in a *hardware description language* (HDL) for hardware synthesis purposes, while processor models (e.g., bus functional models) are often used for cosimulation.

In the case of reconfigurable systems, we need to distinguish between modeling the target application and modeling the host. The first task is pertinent to the system user, while the second to its developer. Thus, these two modeling tasks are very different aspects of co-design.

As systems become more complex, validation is necessary to insure that correct functionality and required performance levels are achieved in the implementation of a system model. Moreover, validation takes different flavors according to the system's application domain. For example, satisfaction of performance objectives (in addition to correctness) is extremely important in processor design. Performance validation is often based on cosimulation of hardware and software [53]. On the other hand, embedded controllers may have less stringent performance requirements to be validated, but their correctness of operation must be verified under all possible environmental conditions, to insure overall system safety levels [36].

The implementation of a hardware/software system may involve several subtasks, the major being hardware synthesis and software compilation. Both topics are complex and several references describe them in depth [2], [30]. We review in Sections IV-A–B techniques affecting the macroscopic system implementation characteristics, and in particular the boundary between hardware and software. Namely, we focus our attention on: 1) partitioning and allocation of system functions to hardware and software and 2) scheduling hardware operations, program instructions and software processes. These two topics address *where* and *when* the system functions are implemented respectively.

A. Hardware/Software Partitioning

The partition of a system into hardware and software is of critical importance because it has a first order impact on the cost/performance characteristics of the final design. Therefore any partitioning decision, performed either by a designer or by a CAD tool, must take into account the properties of the resulting hardware and software blocks.

The formulation of the hardware/software partitioning problem differs according to the co-design problem being confronted with. In the case of embedded systems, a hardware/software partition represents a physical partition of system functionality into application-specific hardware and software executing on one (or more) processor(s). Various formulations to this partitioning problem can be compared on the basis of the architectural assumptions, partitioning goals and solution strategy. We consider each of these issues in detail in the next subsections.

When considering general purpose computing systems, a partition represents a logical division of system functionality, where the underlying hardware is designed to support the software implementation of the complete system functionality. This division is elegantly captured by the instruction set. Thus instruction selection strongly affects the system hardware/software organization.

In the case of reconfigurable systems, the flavor of the partitioning problem depends on the available primitives. For systems consisting of arrays of FPGA chips only, partitioning the system function into the components corresponds to performing technology mapping [16]. On the other hand, for systems consisting of processors and FPGA components, partitioning involves both a physical partition of system functionality (as in the case of embedded systems) and mapping.

1) Architectural Assumptions: Since usually embedded systems are implemented by processors and application-specific hardware, the most common architecture in these systems can be characterized as one of *coprocessing*, i.e., a processor working in conjunction with dedicated hardware to deliver a specific application. The particular implementation of the coprocessing architecture varies in the degree of parallelism supported between hardware and software components. For instance, the coprocessing hardware may be operated under direct control of the processor, which stalls while the dedicated hardware is operational [37], or the coprocessing may be done concurrently with software execution [47]. Similarly, the choice of one (or more than one) processor(s) for the target architecture strongly affects the partitioning formulation [11]. As an example [77], an evaluation of possible coprocessing architectures for a three-dimensional (3-D) computer graphics application leads to an architecture where a processor controls the application-specific coprocessor which maintains its independent data storage. The reported speedup varies from 1.32 to 2.0 across different benchmarks.

The hardware/software interface defines another architectural variable that strongly affects the partitioning problem formulation. It is common to assume that communication operations are conducted using memory-mapped I/O by the processor [26], [49]. However, memory-mapped I/O is often an inefficient mechanism for data transfer [63]. More efficient methods, including dedicated device drivers, have been considered for co-processing architectures [21], but their relation to partitioning has not been articulated yet. Yen and Wolf [119] developed analysis and synthesis methods of bus-oriented communication schemes among processing elements. Other researchers [84] use explicit scheduling of the communication operations in the partitioning loop to improve the quality of the resulting partition in terms its ability to satisfy external timing constraints.

2) Partitioning Objectives: Coprocessing architectures are often chosen to improve the system performance in executing specific algorithms [5], [34]. Accordingly, in some approaches partitioning seeks to maximize the overall speedup for a given application [13], [37], [63], [108]. The speedup estimation is almost always done by a profiling analysis that takes into account typical data sets over which the application behavior is estimated. Due to this data dependence, in some application areas the overall speedup may not be a well-defined metric. Furthermore, in some applications and particularly in those with real-time response requirements, it may not be a useful metric either. In such cases, metrics such as size of implementation and timing constraint satisfaction are used to drive the partitioning subtask. For instance, partitioning is used to improve the hardware utilization by pipelining multiple functions [14].

Constrained partitioning formulations often use capacity constraints, such as size of individual hardware or software portions, to generate a physical division of functionality such that each block can be implemented in a single component. This capacity-constrained partitioning formulation is commonly used in system prototyping applications, where an application is mapped onto multiple FPGA components by a partitioning method [16].

Performance-constrained partitioning formulations focus either on global constraints (such as overall latency) [65], or on the satisfaction of local timing constraints between operations [45]. In this case the partitioning goal is to reduce system cost by migrating part of the system functionality to software, thus reducing the application-specific hardware to implement, while satisfying the performance requirements.

3) Partitioning Strategies: A common misconception in partitioning formulations is that automated methods are the only viable approach to solving partitioning problems when using CAD tools. Often a determination of hardware versus software implementation of a specific functionality is done at levels of abstraction that are not even modeled in system specifications. In the absence of requisite modeling capability, the system partitioning simply can not be carried out using automated tools. Thus there exists a strong relationship between the models used for capturing system functionality and the abstraction level at which the partitioning is carried out. Even when it is possible to create a detailed mathematical model of the partitioning problem, the complexity of the resulting formulation renders it useless for conventional algorithmic methods. Thus we will consider in this section both heuristic approaches and a problem decomposition strategy to handle the complexity of the partitioning problem under some architectural assumptions.

Given a specific architecture, partitioning of a system-level functional description results in a labeling of its tasks as hardware or software operations. The exact solution of such a partitioning problem, even in the simplest cases, requires a solution to computationally intractable problems [41]. In an attempt to mathematically model the variables affecting the partitioning problem, *integer programming* (IP) and *integer linear programming* (ILP) formulations have been proposed recently [11], [42], [84]. Comparison of mathematical programming approaches to hardware/software partitioning is difficult, because the quality of results is often strongly affected by the parametric accuracy of the variables used and by the complexity of the cost/performance model.

Heuristic approaches to partitioning consist primarily of two strategies: *constructive* methods such as clustering techniques [9], [33] and *iterative* methods such as network flow [101], binary-constraint search [111], and dynamic programming [63]. By far, the most used methods are based on variable-depth search methods such as variants of the

Kernighan–Lin (KL) migration heuristics, or probabilistic hill-climbing methods such as simulated-annealing or genetic algorithms. Ernst *et al*. [37] used profiling results on a C-like description to estimate the potential speedup in extracting blocks of code for hardware implementation. The actual selection of code blocks for hardware is done by a simulated annealing algorithm that is used to maximize overall speedup. Reported results indicate speed-up of up to a factor of three on a *chromakey* algorithm for HDTV [37]. Design space search methods, such as using KL's algorithm, are often used following a constructive initial solution to arrive at a feasible solution that meets imposed cost/performance constraints. Reference [45] presents a KL-based algorithm that is used to drive the partition toward meeting timing constraints. Vahid *et al*. [111] use a combination of clustering followed by binary-constraint search that dynamically adjusts the optimization function balance between performance and hardware size as the algorithm progresses to minimize size of hardware while meeting constraints. Another similar approach [65] uses a composite objective function (taking time criticality and local affinity into account) to drive the partition. The results are shown to be qualitatively close to optimal while taking much less computing time.

Let us now examine what makes the problem of partitioning hard. The partitioning and synthesis subtasks are closely interrelated. The cost function of a partitioning problem needs to be evaluated using estimates of the resulting hardware and software. However, the abstraction level at which partitioning is carried out is so high that only rough estimates are available. As an example, consider a hardware/software partitioning problem whose objective is to maximize the application speedup under a constraint on the hardware size. The results of operation scheduling can be used to better estimate the effect of partitioning on the overall latency, and more importantly on the communication cost. This information can be used to select operations for partitioning that results in maximum overall speedup. On the other hand, partitioning a set of scheduled tasks would sacrifice the flexibility of optimizing the communication cost. This dilemma is addressed directly [84] using an integer programming formulation for the partitioning problem that uses an approximate schedule of operations. Solution to this IP programming is followed by a solution to the IP formulation of the scheduling problem that updates the schedule.

The above solution points to a problem decomposition strategy that is a straightforward extension of the decoupling of scheduling and binding in high-level synthesis [30]. In principle, task partitioning, scheduling, and binding to resources should be performed concurrently. Nevertheless, most practical approaches serialize these tasks, while some suggested interactive and iterative approached to partitioning and synthesis [62], [68].

Let us consider first the case where scheduling is followed by concurrent binding/partitioning subtasks. An example of this approach targets pipelined ASIP's [14]. Partitioning is done simultaneously with binding, and the clock-cycle constraint of partitioning is derived from pipeline scheduling done prior to partitioning. This approach works well in cases where the objective of partitioning is to minimize hardware resource requirements. A scheduled input identifies the temporally mutually exclusive operations that can be implemented on a common resource. The partitioner can use the schedule information to divide operations into compatible groups such that binding subtask is able to maximize resource utilization. Such designs are typically resource dominated, therefore, an optimal resource utilization results in reduction of overall size.

The approach of applying partitioning prior to scheduling/binding is fairly common. A difficulty with this approach is the loss of parametric accuracy and controllability of the final result since the partitioning decisions are made early on. As a result most methods in this category either rely on extensive profiling or preprocessing of the input in order to make intelligent decisions about the hardware versus software implementations. Eles [35] presents a two-stage partitioning approach where a prepartitioning of VHDL input is followed by a detailed partitioning using simulated annealing. Constructive methods are quite popular to derive initial hardware/software grouping. For instance, different forms of clustering methods based on similarity measures [9] or closeness criteria [4], [8] are used to group together operations into hardware and software parts.

In addition to handling the strong relationship between different implementation subtasks, a partitioning problem formulation faces the classical dilemma of having to choose between accurate performance/cost measures on partition results versus the efficiency of the partitioning algorithm that determines the extent of design space search. While good estimation methods for hardware performance and size exist, the software component is generally characterized by a significant variability in performance parameters primarily due to architectural features such as caches and a very strong dependence of delay on the input data values. Recent research effort in this direction has been directed at accurate modeling of software delay using analysis of control paths [72], [87], [93], and program annotations [82]. Architectural modeling for software uses pipelining [120], instruction caches [73] and bus-activity using DMA [60]. These continuing efforts have successfully improved the estimation accuracy to be within 50–100% of the actual worst-case delay. The need for timing predictability continues to adversely affect the design of tightly constrained systems to such an extent that many systems use distressingly simple architectures (such as turning off cache memories), thus rarely exploiting the peak performance of the underlying hardware in real applications.

B. Scheduling

The *scheduling* problem has many facets. Scheduling algorithms have been developed in both the operation research and computer science community, with different models and objectives. The techniques that are applicable

today to the design of hardware and software systems draw ideas from both communities.

Generally speaking, hardware and software scheduling problems differ not just in the formulation but in their overall goals. Nevertheless, some hardware scheduling algorithms are based on techniques used in the software domain, and some recent system-level process scheduling methods have leveraged ideas in hardware sequencing.

Scheduling can be loosely defined as assigning an execution *start time* to each *task* in a set, where tasks are linked by some relations (e.g., dependencies, priorities,...). The tasks can be elementary (like hardware operations or computer instructions) or can be an ensemble of elementary operations (like software programs). When confusion may arise, we will refer to tasks as *operations* in the former case, and to *processes* in the latter. The tasks can be *periodic* or *aperiodic*, and task execution may be subject to *real time* constraints or not. Scheduling under timing constraints is common for hardware circuits, and for software applications in embedded control systems. Tasks execution requires the use of *resources*, which can be limited in number, thus causing the serialization of some task execution. Most scheduling problems are computationally intractable [41], and thus their solutions are often based on heuristic techniques.

We consider next scheduling algorithms as applied to the design of hardware, compilers, and operating systems.

1) Operation Scheduling in Hardware: We consider now the major approaches to hardware scheduling. These techniques have been implemented (to different extent) in CAD tools for the design of ASIC's and DSP's [32], [66], [85], [106], which are modeled with a behavioral-level HDL (e.g., VHDL, Verilog HDL, and DFL [115]). The behavioral model can be abstracted as a set of operations and dependencies. The hardware implementation is assumed to be synchronous, with a given cycle-time. Operations are assumed to take a known, integer number of cycles to execute. (We will consider removing this assumption later). The result of scheduling, i.e., the set of start times of the operations, is just a set of integers. The usual goal is to minimize the overall execution *latency*, i.e., the time required to execute all operations.

Constraints on scheduling usually relate to the number of resources available to implement each operation and to upper/lower bounds on the time distance between start times of operation pairs. Usually, the presence of resource constraints makes the problem intractable [30], [41].

The scheduling problem can be cast as an integer linear program [30], [42], [50], where binary-valued variables determine the assignment of a start time to each operation. Linear constraints require each operation to start once, to satisfy the precedence and the resource constraints. Latency can also be expressed as a linear combination of the decision variables. The scheduling problem has a dual formulation, where latency is bounded from above and the objective function relates to minimizing the resource usage, which can also be expressed as a linear function. Timing and other constraints can be easily incorporated in the ILP model [42].

The appeal of using the ILP model is due to the uniform formulation even in presence of different constraints and to the possibility of using standard solution packages. Its limitation is due to the prohibitive computational cost for medium-large cases. This relegates the ILP formulation to specific cases, where an exact solution is required and where the problem size makes the ILP solution viable.

Most practical implementations of hardware schedulers rely on *list scheduling*, which is a heuristic approach that yields good (but not necessarily optimal) schedules in linear (or overlinear) time. A list scheduler considers the time slots one at a time, and schedules to each slot those operations whose predecessors have been scheduled, if enough resources are available. Otherwise the operation execution is deferred. Ties are broken using a priority list, hence the name.

Another heuristic for scheduling is *force-directed* scheduling [89], which addresses the latency-constrained scheduling problem. Here, operations are scheduled into the time slots one at a time, subject to time-window constraints induced by precedence and latency constraints. Ties among different time slots for each operation are broken using a heuristic based on the concept of *force*, which measures the tendency of the operation to be in a given slot, to minimize overall concurrency. The computational cost of force-directed scheduling is quadratic in the number of operations.

When resource constraints are relaxed, the scheduling problem can sometimes be solved in polynomial time. For example, scheduling with timing constraints on operation time separation can be cast as a longest-path problem [30]. On the other hand, scheduling under release times and deadlines is intractable, unless the operations take a single cycle to execute [41].

There are several generalizations of the scheduling problem. In some cases, operations are not restricted to take an integral number of cycles to execute, and more than one operation can be *chained* into a single time slot. Pipelined circuits require specific constraints on data rates, and additional resource conflicts have to be taken into account due to the concurrent execution of operations in different pipestages. Periodic operation subsets, e.g., iteration construct bodies, may be advantageously scheduled using *loop pipelining* techniques [30], which is an example of a method borrowed from software compilers [116]. Chaining and pipelining can be incorporated in ILP, list, and force-directed schedulers.

The *synchronization* of two (or more) operations or processes is an important issue related to scheduling. Synchronization is needed when some delay is unknown in the model. *Relative scheduling* is an extended scheduling method to cope with operations with unbounded delays [67] called *anchors*. In this case, a static schedule cannot be determined. Nevertheless, in relative scheduling the operations are scheduled with respect to their anchor ancestors. A finite-state machine can be derived that executes the

operations in an appropriate sequence, on the basis of the relative schedules and the anchor completion signals. The relative scheduling formulation supports the analysis of timing constraints, and when these are consistent with the model, the resulting schedule satisfies the constraint for any anchor delay value. Scheduling with *templates* [70] is a similar approach, where operations are partitioned into templates that can be seen as single scheduling units. Thus templates are useful for hierarchical scheduling and scheduling multicycle resources (e.g., pipelined multipliers).

2) Instruction Scheduling in Compilers: Compilers are complex software tools, consisting of a front-end, a suite of optimization routines operating on an intermediate form, and a back-end (called also *code generation*) which generates the machine code for the target architecture. In the context of compilation, instruction scheduling on a uniprocessor is the task of obtaining a linear order of the instructions. Thus it differs from hardware scheduling because the resource constraints typically refer to storage elements (e.g., registers) and the hardware functional resource is usually one ALU. In the more general case, scheduling can be viewed as the process of organizing instructions into streams.

Instruction scheduling is related to the choice of instructions, each performing a fragment of the computation, and to register allocation. When considering compilation for general-purpose microprocessors, instruction selection and register allocation are often achieved by dynamic programming algorithms [2], which also generate the order of the instructions. When considering retargetable compilers for ASIP's, the compiler back-end is often more complex, because of irregular structures such as inhomogeneous register sets and connections. As a result, instruction selection, register allocation and scheduling are tightly-coupled phases of code generation [43]. In both cases, scheduling objectives are reducing the code size (which correlates with the latency of execution time) and minimizing *spills*, i.e., overflows of the register file which require memory access.

Optimizing compiler algorithms for ASIP's and general-purpose DSP's has been a subject of recent research activities [79]. Instruction selection, instruction scheduling, and register spilling problems for ASIP's are addressed by Liao *et al*. [71]. The same group formulates the instruction selection problem as a binate covering problem, that is solved using a branch-and-bound algorithm [74]. Scheduling has been modeled by resource and instruction set conflicts and solved by bipartite matching algorithms [109]. Araujo *et al*. [6] considered code generation for basic blocks in heterogeneous memory-register DSP processors and used register-transfer paths to convert basic block graphs into expression trees which are used in code generation.

The co-design of deeply pipelined microprocessors can leverage the coupling between instruction scheduling and hardware organization. Pipeline hazard avoidance can be achieved by hardware means (e.g., stall) or by software means (e.g., instruction reorder and NOP insertion). Recent research [57] has addressed the problem of the concurrent synthesis of the pipeline control hardware and the determination of an appropriate instruction reorder that the corresponding back-end compiler should use to avoid hazards. The same group [59] has also proposed a methodology for synthesizing instruction sets from application benchmarks.

3) Process Scheduling in Different Operating Systems: Process scheduling is the problem of determining when processes execute and includes handling synchronization and mutual exclusion problems. Algorithms for process scheduling are important constituents of operating systems and run-time schedulers [104].

The model of the scheduling problem is more general than the one previously considered. Processes have a coarser granularity and their overall execution time may not be known. Processes may maintain a separate context through local storage and associated control information. Scheduling objectives may also vary. In a *multitasking* operating system, scheduling primarily addresses increasing processor utilization and reducing response time. On the other hand, scheduling in *real-time operating systems* (RTOS) primarily addresses the satisfaction of timing constraints.

We consider first scheduling without real-time constraints. The scheduling objective involves usually a variety of goals, such as maximizing CPU utilization and throughput as well as minimizing response time. Scheduling algorithms may be complex, but they are often rooted on simple procedures such as *shortest-job first* (SJF) or *round robin* (RR) [92]. The SJF is a priority-based algorithm that schedules processes according to their priorities, where the shorter the process length (or, more precisely, its CPU burst length) the higher the priority. This algorithm would give the minimum average time for a given set of processes, if their (CPU-burst) lengths were known exactly. In practice, predictive formulas are used. Processes in a SJF may be allowed to preempt other processes to avoid starvation.

The round-robin scheduling algorithm uses a circular queue and it schedules the processes around the queue for a time interval up to a predefined quantum. The queue is implemented as a *first-in/first-out* (FIFO) queue and new processes are added at the tail of the queue. The scheduler pops the queue and sets a timer. If the popped process terminates before the timer, the scheduler pops the queue again. Otherwise it performs a *context switch* by interrupting the process, saving the state, and starting the next process on the FIFO.

Process scheduling in real-time operating system [100] is characterized by different goals and algorithms. Schedules may or may not exist that satisfy the given timing constraints. In general, the primary goal is to schedule the tasks such that all deadlines are met: in case of success (failure) a secondary goal is maximizing earliness (minimizing tardiness) of task completion. An important issue is predictability of the scheduler, i.e., the level of confidence that the scheduler meets the constraints.

The different paradigms for process scheduling in RTOS can be grouped as static or dynamic [100]. In the former case, a schedulability analysis is performed before run

time, even though task execution can be determined at run time based on priorities. In the latter case, feasibility is checked at run time [100]. In either case, processes may be considered periodic or aperiodic. Most algorithms assume periodic tasks and tasks are converted into periodic tasks when they are not originally so.

Rate monotonic (RM) analysis [76] is one of the most celebrated algorithms for scheduling periodic processes on a single processor. RM is a priority-driven preemptive algorithm. Processes are statically scheduled with priorities that are higher for processes with higher invocation rate, hence the name. Liu and Layland showed that this schedule is optimum in the sense that no other fixed-priority scheduler can schedule a set of processes which cannot be scheduled by RM [76]. The optimality of RM is valid under some restrictive assumptions, e.g., neglecting context-switch time. Nevertheless, RM analysis has been the basis for more elaborate scheduling algorithms [21], [27].

Let us consider now hardware/software system implementations obtained by partitioning a system-level specification, as mentioned in Section IV-A. The implementation consists of a set of software fragments executing on a processor in parallel with the execution of other tasks in dedicated hardware. A relevant problem is to determine the execution windows for both the hardware and software tasks. Since the partition depends on the specific application and design objectives, a run-time scheduler for the system is required that fits the hardware/software partition. Conversely, a given partition may be chosen because a run-time scheduler can assign schedule tasks while satisfying given deadline and rate constraints.

We summarize an approach fully described in [45]. Software tasks are represented by *threads*, each thread being a set of operations with known execution, except possibly the head of the thread. Operations within threads are statically scheduled (with respect to the head of the thread), so that timing constraints are *marginally satisfied*, i.e., within the limits of the lack of knowledge of the delay of the thread head operation. Threads execution is then dynamically determined by a nonpreemptive run-time scheduler whose task is to synchronize the execution of hardware and software. Thread-based scheduling can be seen as an application and extension of relative scheduling to the hardware/software domain, thus showing the cross-fertilization of the hardware and software fields.

We briefly describe now process scheduling in *Chinook*, a CAD environment for designing reactive real-time systems [21]. The overall system can have different *modes* of operation, each having a schedule. Timing watchdogs can disable modes and cause mode transitions. Upon changing of mode, the system starts running the corresponding schedule. Timing constraints may be intermodal or intramodal. Each mode has a periodic set of tasks, which is unrolled and scheduled under timing constraints, using an extension of the relative scheduling formulation [21]. With this scheduling technique, Chinook supports the mapping of an embedded system model to one (or more) processor and peripherals while ensuring the satisfaction of timing constraints.

In summary, process scheduling plays an important role in the design of mixed hardware/software systems, because it handles the synchronization of the tasks executing in both the hardware and software components. For this reason, it is currently a subject of intensive research.

V. Accomplishments

We underline here briefly the major accomplishments in this field, and we refer the readers to the other articles of this issue for specific results. Hardware/software co-design has attracted the attention of several research groups worldwide, as documented by books [29], [39], [95], journal articles, and publications in symposia. Some of the research addresses incremental changes to system-level design tools, to cope with software components in predominantly hardware designs. The need for addressing practical problems and for fitting into existing design methodologies where modeling styles and languages are not negotiable, limits significantly the power of these tools to search creatively the co-design solution space. Other research contributions propose paradigm shifts in system-level design, by assuming a wide freedom in the way systems are modeled and designed. While these approaches will probably provide the basis for long-term innovation, they often lead to design tools which are not readily usable by system designers because disconnected from existing design practices.

Several research computer-aided co-design environments have been released, and some of these are used for research and/or product development, e.g., *Castle* [105], *Chinook* [24], *Cosmos* [61], *Cosyma* [37], *Coware* [114], *Polis* [26], *Ptolemy* [64], *Siera* [103], *Specsym* [40], and *Tosca* [8]. (See [29] for additional information on some of these systems.)

Commercial co-design tools are available to address some of the problems mentioned in this survey. Some CAD vendors provide design entry systems and co-simulation environments. Cosimulation is widely applicable to general-purpose and digital-signal processor design, as well as to embedded system design. For the telecommunication domain, specialized environments support the vertical design of systems from design entry to physical realization. System emulators, based on field-programmable technology, have proven to be successful for validating large systems.

Commercial products in the software domain include compilers for general-purpose and dedicated processors with standard and application-specific architectures, as well as real-time operating systems and microkernels. Such products find several applications in embedded systems.

VI. Conclusion

Hardware/software co-design presents an enormous challenge, as well as an opportunity, for system designers. Use and reuse of hardware and software macro blocks can lead to products of superior quality (i.e., performance/cost, flexibility, ...) with a shorter design and development

time as compared to traditional integrated circuit design methodologies. The progress in electrical system design will depend, among other factors, on the level of support provided by CAD tools. In particular, digital system products will benefit from concurrent hardware/software design which exploits the synergism of hardware and software in the search for solutions that use at best the current manufacturing technology and the availability of hardware components and software programs.

Scientific and commercial interest in hardware/software co-design methods and tools has risen significantly in the recent years. Product-level use of co-design tools has been reported in some application domains, (e.g., co-simulation, emulation, synthesis for embedded controllers, retargetable compilers). The sector of computer-aided co-design tools is growing at a rapid pace because the potential payoffs make it an attractive area for research as well as an exciting business opportunity.

Overall, hardware/software co-design is a wide field of research, because of the diversity of applications, design styles and implementation technologies. Since this area is still not completely defined, we can expect some evolutionary and some revolutionary changes in the way digital systems are designed. Thus hardware/software co-design is the key design technology for digital systems.

REFERENCES

[1] T. Agerwala, "Microprogramming optimization: A survey," *IEEE Trans. Computers*, vol. C-25, pp. 962–973, Oct. 1976.

[2] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques and Tools*. Reading, MA: Addison-Wesley, 1988.

[3] A. Alomary, T. Nakata, Y. Honma, M. Imai, and N. Hikichi, "An ASIP instruction set optimization algorithm with functional module sharing constraints," in *Proc. ICCAD*, 1993, pp. 526–532.

[4] S. Antoniazzi, A. Balboni, W. Fornaciari, and D. Sciuto, "HW/SW codesign for embedded telecom systems," in *Proc. ICCD*, 1994, pp. 278–281.

[5] R. Amerson, R. Carter, W. B. Culberttson, P. Kuekes, and G. Snider, "Teramac-configurable custom computing," in *FPGA's for Custom Computing Machines*, Apr. 1995.

[6] G. Araujo, S. Malik, and M Lee, "Using register-transfer paths in code generation for heterogeneous memory-register architectures," in *Proc. DAC*, 1996, pp. 591–596.

[7] J. Babb, R. Tessier, and A. Agarwal, "Virtual wires: Overcoming pin limitations in FPGA-based logic emulators," in *Proc. IEEE Workshop on FPGA's for Custom Computing Machines*, Apr. 1993.

[8] A. Balboni, W. Fornaciari, and D. Sciuto, "Cosynthesis and cosimulation of control-dominated embedded systems," *Design Automat. Embedded Syst.*, vol. 1, no. 3, pp. 257–289, July 1996.

[9] E. Barros, W. Rosenstiel, and X. Xiong, "A method for partitioning UNITY language in hardware and software," in *Proc. EURODAC*, 1994, pp. 220–225.

[10] D. Becker, R. Singh, and S. Tell, "An engineering environment for hardware-software co-simulation," in *Proc. DAC*, 1992, pp. 129–134.

[11] A. Bender, "Design of an optimal loosely coupled heterogeneous multiprocessor system," in *Proc. EDTC*, 1996, pp. 275–281.

[12] A. Benveniste and G. Berry, "The synchronous approach to reactive and real-time systems," *Proc. IEEE*, vol. 79, pp. 1270–1282, Sept. 1991.

[13] P. Bertin, D. Roncin, and J. Vuillemin, "Introduction to programmable active memories," in *Systolic Array Processors*, J. McCanny, J. McWhirter, and E. Schwartzlander, Eds. Englewood Cliffs, NJ: Prentice-Hall, 1989.

[14] N. Binh, M. Imai, A. Shiomi, and N. Hickichi, "A HW/SW partitioning algorithm for designing pipelined ASIP's with least gate counts," in *Proc. DAC*, 1996, pp. 527–532.

[15] K. Buchenrieder, A. Sedelmeier, and C. Veith, "Industrial HW/SW codesign," in *Hardware/Software Co-Design*, G. De Micheli and M. Sami, Eds. Amsterdam: Kluwer, 1996, pp. 453–466.

[16] J. Cong and Y. Ding, "Combinational logic synthesis for LUT based field programmable gate arrays," *TODAES, ACM Trans. Design Automat. Electron. Syst.*, vol. 1, no. 2, pp. 145–204, Apr. 1996.

[17] I. Bolsens, H. De Man, B. Lin, K. van Rompaey, S. Vercautern, and D. Verkest, "Hardware-software codesign of digital telecommunication systems," *Proc. IEEE*, this issue, p. 391–418.

[18] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang, "Symbolic model checking: 10^{20} states and beyond," *Informat. and Computation*, vol. 98, no. 2, pp. 142–170, June 1992.

[19] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill, "Symbolic model checking for sequential circuit verification," *IEEE Trans. CAD/ICAS*, vol. 13, no. 4, pp. 401–424, Apr. 1994.

[20] D. Bursky, "Microcontroller design exploits reusable cores," *Electron. Design*, vol. 42, no. 6, pp. 53–68, Mar. 1994.

[21] P. Chou, E. Walkup, and G. Borriello, "Scheduling strategies in the co-synthesis of reactive real-time systems," *IEEE Micro*, vol. 14, no. 4, pp. 37–47, Aug. 1994.

[22] P. Chou and G. Borriello, "Software scheduling in co-synthesis of reactive real-time systems," in *Proc. DAC*, 1994, pp. 1–4.

[23] ——, "Interval scheduling: Fine-grained code scheduling for embedded systems," in *Proc. DAC*, 1995.

[24] P. Chou, R. Ortega, and G. Borriello, "The Chinook hardware/software co-design system," in *Proc. ISSS*, Cannes, France, 1995, pp. 22–27.

[25] M. Chiodo, P. Giusto, A. Jurecska, H. Hsieh, L. Lavagno, and A. Sangiovanni, "A formal methodology for hardware/software co-design of embedded systems," *IEEE Micro*, vol. 14, no. 4, pp. 26–36, Aug. 1994.

[26] M. Chiodo, D. Engels, P. Giusto, A. Jurecska, H. Hsieh, L. Lavagno, K. Suzuki, and A. Sangiovanni, "A case study in computer-aided co-design of embedded controllers," *Design Automat. Embedded Syst.*, vol. 1, no. 2, pp. 51–67.

[27] M. Cochran, "Using the rate monotonic analysis to analyze the schedulability of ADARTS real-time software design," in *Int. Workshop on Hardware/Software Co-design*, Sept. 1992.

[28] H. de Garis, "Evolvable hardware," in *Proc. Artificial Neur. Nets and Genetic Algorithms*, Apr. 1993, pp. 441–449.

[29] G. De Micheli and M. Sami, *Hardware/Software Co-Design*. Amsterdam: Kluwer, 1996.

[30] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. New York: McGraw-Hill, 1994.

[31] ——, "Computer-aided hardware/software co-design," *IEEE Micro*, vol. 14, no. 4, pp. 10–16, Aug. 1994.

[32] G. De Micheli, D. Ku, F. Mailhot, and T. Truong, "The olympus synthesis system for digital design," *IEEE Design and Test*, pp. 37–53, Oct. 1990.

[33] E. D. Lagnese and D. Thomas, "Architectural partitioning for system level descriptions," in *Proc. DAC*, 1989, pp. 62–67.

[34] T. Ebisuzaki *et al.*, "GRAPE: Special purpose computer for classical many-body simulations," in *Proc. Advances in Computing Techniques*, 1994, pp. 218–231.

[35] P. Eles, "VHDL system-level specification and partitioning in a hardware/software co-synthesis environment," *Int. Workshop on Hardware/Software Codesign*, Sept. 1994, pp. 22–24.

[36] S. Edward, L. Lavagno, E. Lee, and A. Sangiovanni, "Design of embedded systems: Formal models, validation and synthesis," *Proc. IEEE*, this issue, pp. 366–390.

[37] R. Ernst, J. Henkel, and T. Benner, "Hardware-software cosynthesis for micro-controllers," *IEEE Design and Test*, pp. 64–75, Dec. 1993.

[38] C. Feigel, "Processors aim at desktop video," *Microprocess. Rep.*, vol. 8, no. 2, Feb. 1994.

[39] D. Gajski, S. Narayan, F. Vahid, and J. Gong, *Specification and Design of Embedded Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1994.

[40] D. Gajski, F. Vahid, and S. Narayan, "A system design methodology: Executable specification refinement," in *Proc. EDAC*,

1994, pp. 458–463.
[41] M. Garey and D. Johnson, *Computers and Intractability*. New York: Freeman, 1979.
[42] C. Gebotys and M. Elmasry, *Optimal VLSI Architectural Synthesis*. Amsterdam: Kluwer, 1992.
[43] G. Goossens, P. G. Paulin, J. Van Praet, D. Lanneer, W. Guerts, A. Kifli, and C. Liem, "Embedded software in real-time signal processing systems: Design technologies," *Proc. IEEE*, this issue, pp. 436–454.
[44] J. Green, E. Hamdy, and S. Beal, "Antifuse field programmable gate arrays," *Proc. IEEE*, vol. 81, pp. 1041–1056, July 1993.
[45] R. Gupta, *Co-Synthesis of Hardware and Software for Digital Embedded Systems*. Amsterdam: Kluwer, 1995.
[46] R. Gupta, C. Coelho, and G. De Micheli, "Synthesis and simulation of digital systems containing interacting hardware and software components," in *Proc. DAC*, 1992, pp. 225–230.
[47] R. Gupta and G. De Micheli, "System co-synthesis for digital systems," *IEEE Design and Test*, vol. 10, no. 3, pp. 29–41, Sept. 1993.
[48] ——, "A co-synthesis approach to embedded system design automation," *Design Automat. for Embedded Syst.*, vol. 1, nos. 1–2, pp. 69–120, Jan. 1996.
[49] R. Gupta, C. Coelho, and G. De Micheli, "Program implementation schemes for hardware-software systems," *IEEE Computer*, pp. 48–55, Jan. 1994.
[50] L. Hafer and A. Parker, "Automated synthesis of digital hardware," *IEEE Trans. Computers*, vol. C-31, no. 2, Feb. 1982.
[51] D. Harel, A. Pneuli, J. Schmidt, and R. Sherman, "Statecharts: A visual formalism for complex systems," *Sci. Computer Programming*, no. 8, pp. 231–274, 1987.
[52] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*. San Francisco: Morgan Kaufmann, 1990.
[53] M. Heinrich, D. Ofelt, M. A. Horowitz, and J. Hennessy, "Hardware/software co-design of the stanford FLASH multiprocessor," *Proc. IEEE*, this issue, p. 455–466.
[54] M. Horowitz and K. Keutzer, "Hardware-software co-design," in *Proc. SASIMI*, Nara, 1993, pp. 5–14.
[55] B. Holmer, "A tools for processor instruction set design," in *Proc. DAC*, 1994, pp. 150–155.
[56] X. Hu, J. D'Ambrosio, and D.-L. Tang, "Codesign architectures for automotive powertrain modules," *IEEE Micro*, vol. 14, no. 4, pp. 17–25, Aug. 1994.
[57] I. Huang and A. Despain, "High-level synthesis of pipelined instruction set processors and back-end compilers," in *Proc. DAC*, 1992, pp. 135–140.
[58] ——, "Hardware/software resolution of pipeline hazards in pipeline synthesis of instruction set processors," in *Proc. ICCAD*, 1993, pp. 594–599.
[59] ——, "Synthesis of instruction sets for pipelined microprocessors," in *Proc. DAC*, 1994, pp. 5–11.
[60] T.-Y. Huang, "Worst-case timing analysis of concurrently executing DMA I/O and programs," Ph.D. dissertation, Univ. Illinois, Dept. Computer Sci., Sept. 1996.
[61] T. Ismail and A. Jerraya, "Synthesis steps and design models for codesign," *IEEE Computer*, no. 2, pp. 44–52, Feb. 1995.
[62] T. Ismail, K. O'Brien, and A. Jerraya, "Interactive system-level partitioning with PARTIF," in *Proc. EDAC*, Feb. 1994, pp. 464–468.
[63] A. Jantasch, P. Ellervee, J. Öberg, A. Hemani, and H. Tenhunen, "Hardware/software partitioning and minimizing memory interface traffic," in *Proc. EURODAC*, 1994, pp. 226–231.
[64] A. Kalavade and E. Lee, "A hardware-software co-design methodology for DSP applications," *IEEE Design and Test*, vol. 10, no. 3, pp. 16–28, Sept. 1993.
[65] ——, "A global criticality/local phase driven algorithm for the constrained hardware/software partitioning problem," in *3rd Int. Workshop on Hardware/Software Codesign*, Sept. 1994, pp. 42–48.
[66] D. Knapp, *Behavioral Synthesis*. Englewood Cliffs, NJ: Prentice-Hall, 1996.
[67] D. Ku and G. De Micheli, "Relative scheduling under timing constraints: Algorithms for high-level synthesis of digital circuits," *IEEE Trans. CAD/ICAS*, June 1992, pp. 696–718.
[68] K. Kucukcakar and A. Parker, "CHOP: A constraint-driven system-level partitioner," in *Proc. DAC*, 1991, pp. 514–519.
[69] D. Lanneer, J. Van Praet, A. Kifli, K. Schoofs, W. Guerts, F. Thoen, and G. Goossens, "CHESS: Retargetable code generation for embedded DSP processors," in *Code Generators for Embedded Processors*, P. Marwedel and G. Goossens, Eds. Amsterdam: Kluwer, 1995.
[70] T. Ly, D. Knapp, R. Miller, and D. MacMillen, "Scheduling withing behavioral templates," in *Proc. DAC*, 1995, pp. 101–106.
[71] S. Liao, K. Keutzer, S. Tjiang, and A. Wang, "Code optimization techniques for embedded DSP microprocessors," in *DAC Proc. Design Automat. Conf.*, 1995, pp. 599–604.
[72] Y.-T. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," in *Proc. DAC*, 1995, pp. 456–461.
[73] Y.-T. Li, S. Malik, and A. Wolfe, "Performance estimation of embedded software with instruction cache modeling," in *Proc. ICCAD*, Nov. 1995, pp. 380–387.
[74] S. Liao, K. Keutzer, K. Keutzer, and S. Tjiang, "Instruction set selection using binate covering for code size optimization," in *Proc. ICCAD*, 1995, pp. 393–399.
[75] C. Liem, T. May, and P. Paulin, "Instruction-set matching and selection for DSP and ASIP code generation," in *Proc. Europe. Design and Test Conf.*, 1994, pp. 31–37.
[76] C. L. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, pp. 44–61, Jan. 1973.
[77] J. Madsen and J. P. Brage, "Codesign analysis of a computer graphics application," *Design Automat. for Embedded Syst.*, vol. 1, pp. 121–145, 1996.
[78] D. Mange, M. Goeke, D. Madon, A. Stauffer, G. Tempesti, and S. Durand, "Embryonics: A new family of coarse-grained FPGA's with self-repair and self-reproducing properties," in *Toward Evolvable Hardware*, E. Sanchez and M. Tomassini, Eds. Amsterdam: Springer, 1996.
[79] P. Marwedel and G. Goossens, Eds., *Code Generators for Embedded Processors*. Amsterdam: Kluwer, 1995.
[80] P. Marwedel, "Tree-based mapping of algorithms to predefined structures," in *Proc. ICCAD*, 1993, pp. 586–593.
[81] L. Maliniak, "Logic emulator meets the demands of CPU designers," *Electron. Design*, Apr. 1993.
[82] A. Mok, P. Amerasinghe, M. Chen, and K. Tantisirivat, "Evaluating tight execution bounds of programs by annotations," in *Proc. 6th IEEE Workshop on Real-Time Operating Syst. and Software*, May 1989, pp. 272–279.
[83] S. Narayan, F. Vahid, and D. Gajski, "Translating system specifications to VHDL," in *Proc. EDAC*, 1991, pp. 390–394.
[84] R. Niemann and P. Marwedel, "Hardware/software partitioning using integer programming," in *Proc. EDTC*, 1996, pp. 473–479.
[85] S. Note, F. Chattoor, G. Goossens, and H. De Man, "Combined hardware selection and pipelining in high-level performance data-path design," *IEEE Trans. CAD/ICAS*, pp. 413–423, Apr. 1992.
[86] K. Olukotun, R. Helahel, J. Levitt, and R. Ramirez, "A software/hardware co-synthesis approach to digital system simulation," *IEEE Micro.*, vol. 14, no. 4, pp. 48–58, Aug. 1994.
[87] C.-Y. Park, "Predicting program execution times by analyzing static and dynamic program paths," *J. Real-Time Syst.*, vol. 5, pp. 31–62, Mar. 1993.
[88] P. Paulin, G. Goossens, C. Liem, M. Cornero, and F. Nacabal, "Embedded software in real-time signal processing systems: Applications and architecture trends," *Proc. IEEE*, this issue, pp. 419–435.
[89] P. Paulin, C. Liem, T. May, and S. Sutarwala, "Flexware: A flexible firmware development environment for embedded systems," in *Code Generators for Embedded Processors*, P. Marwedel and G. Goossens, Eds. Amsterdam: Kluwer, 1995.
[90] ——, "DSP design tool requirements for embedded systems: A telecommunications industrial perspective," *J. VLSI Signal Process.*, no. 9, pp. 23–47, 1995.
[91] D. Perry, *VHDL*. New York: McGraw-Hill, 1991.
[92] J. Peterson and A. Silbershatz, *Operating Systems Principles*. Reading, MA: Addison-Wesley.
[93] P. Puschner and C. Koza, "Calculating the maximum execution time of real-time programs," *J. Real-Time Syst.*, vol. 1, pp. 159–176, Sept. 1989.
[94] J. Rowson, "Hardware-software co-simulation," in *Proc. Design Automat. Conf.*, 1994, pp. 439–440.
[95] J. Rozenblit and K. Buchenrieder, *Codesign: Computer-Aided Software/Hardware Engineering*. Piscataway, NJ: IEEE, 1995.
[96] M. Salinas, B. Johnson, and H. Aylor, "Implementation-independent model of an instruction set architecture in VHDL,"

IEEE Design and Test, vol. 10, no. 3, pp. 42–55, Sept. 1993.
[97] E. Sanchez and M. Tomassini, *Toward Evolvable Hardware*. Amsterdam: Springer, 1996.
[98] R. Saracco and P. Tilanus, "CCITT SDL: Overview of the language and its applications," *Computer Networks and ISDN Syst.*, vol. 13, no. 2, pp. 65–74, 1987.
[99] K. Shin and P. Ramanathan, "Real-time computing: A new discipline of computer science and engineering," *Proc. IEEE*, vol. 82, pp. 6–24, Jan. 1994.
[100] P. Ramanathan and J. Stankovic, "Scheduling algorithms and operating system support for real-time systems," *Proc. IEEE*, vol. 82, pp. 55–67, Jan. 1994.
[101] S. Agrawal and R. Gupta, "System partitioning using global data-flow," Univ. Illinois Memo. UIUC DCS, 1995.
[102] A. Smailagic and D. Siewiorek, "A case-study in embedded system design: The VuMan2 wearable computer," *IEEE Design and Test*, vol. 10, no. 3, pp. 56–67, Sept. 1993.
[103] M. B. Srivastava and R. W. Broderson, "Rapid-prototyping of hardware and software in a unified framework," in *Proc. ICCAD*, 1991, pp. 152–155.
[104] J. Stankovich, M. Spuri, M. Di Natale, and C. Buttazzo, "Implications of classical scheduling results for real-time systems," *IEEE Computer*, vol. 28, no. 6, pp. 16–25.
[105] M. Theissinger, P. Stravers, and H. Veit, "CASTLE: A design environment for co-design," in *Proc. Int. Workshop on Hardware/Software Co-Design*, Grenoble, Sept. 1994, pp. 203–209.
[106] D. Thomas, E. Lagnese, R. Walker, J. Nestor, J. Rajan, and R. Blackburn, *Algorithmic and Register Transfer Level Synthesis: The System Architect's Workbench*. New York: Kluwer, 1990.
[107] D. Thomas and P. Moorby, *The Verilog Hardware Description Language*. New York: Kluwer, 1991.
[108] D. Thomas, J. Adams, and H. Schmitt, "A model and methodology for hardware-software co-design," *IEEE Design and Test*, vol. 10, no. 3, pp. 6–15, Sept. 1993.
[109] A. Timmer, M. Strik, J. van Meerbergen, and J. Jess, "Conflict modeling and instruction scheduling in code generation for in-house DSP cores," in *DAC Proc. Design Automat. Conf.*, 1995, pp. 593–598.
[110] S. Trimberger, "A reprogrammable gate array and applications," *Proc. IEEE*, vol. 81, no. 7, pp. 1030–1041, July 1993.
[111] F. Vahid, J. Gong, and D. Gajski, "A binary-constraint search algorithm for minimizing hardware during hardware/software partitioning," in *Proc. EURODAC*, 1994, pp. 214–219.
[112] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard, "Programmable active memories: Reconfigurable systems come of age," *IEEE Trans. VLSI*, vol. 4, no. 2, pp. 56–69, Mar. 1996.
[113] A. Wenban, J. O'Leary, and G. Brown, "Codesign of communication protocols" *IEEE Computers*, no. 12, pp. 46–52, Dec. 1993.
[114] D. Verkest, K. van Romaey, I. Bolsen, and H. De man, "Co-ware—A design environment for heterogeneous hardware/software systems," *Design Automat. for Embedded Syst.*, vol. 1, no. 4, pp. 357–386, Oct. 1996.
[115] P. Willekens *et al.*, "Algorithm specifications in DSP station using data flow language," *DSP Applicat.*, vol. 3, no. 1, pp. 8–16, Jan. 1994.
[116] M. Wolf and M. Lam, "A loop transformation theory and an algorithm to maximize parallelism," *IEEE Trans. Parallel Distrib. Syst.*, vol. 2, no. 4, pp. 452–471, Oct. 1991.
[117] W. Wolf and E. Frey, "Tutorial on embedded system design," in *Proc. ICCD*, 1992, pp. 18–21.
[118] W. Wolf, "Hardware-software co-design of embedded systems," *Proc. IEEE*, vol. 82, pp. 967–989, July 1994.
[119] T.-Y. Yen and W. Wolf, "Communicating synthesis for distributed embedded systems," in *Proc. ICCAD*, 1995, pp. 288–294.
[120] N. Zhang, A. Burns, and M. Nicholson, "Pipelined processors and worst case execution times," *J. Real-Time Syst.*, vol. 5, pp. 319–343, Oct. 1993.

Giovanni De Micheli (Fellow, IEEE), for a photograph and biography, see this issue, p. 349.

Rajesh K. Gupta (Member, IEEE) received the B.Tech. degree in electrical engineering from the Indian Institute of Technology, Kanpur, India, the M.S. degree in electrical engineering and computer science from the University of California at Berkeley, and the Ph.D. degree in electrical engineering from Stanford University, Stanford, CA, in 1984, 1986, and 1993, respectively.

He is presently an Assistant Professor of Information and Computer Science at the University of California, Irvine. During 1994–1996 he was an Assistant Professor at the University of Illinois at Urbana–Champaign. From 1986 to 1989 he was with Intel Corporation, Santa Clara, CA, where he worked on VLSI design at various levels of abstraction as a member of the design teams for the 80386–SX, 80486, and Pentium microprocessor devices. He has worked on a number of successful chip designs, including CMOS, BiCMOS, ECL, and high-speed GaAs devices. He co-authored a patent on PLL-based clock circuit. He authored *Cosynthesis of Hardware and Software for Digital Embedded Systems* (Kluwer).

Dr. Gupta was nominated for the NSF Presidential Faculty Fellow Award by the University of Illinois in 1996. He received the NSF CAREER Award in 1995, the Philips Graduate Fellowship in 1991 and 1992, the Departmental Achievements by Microcomputer Division at Intel Corporation in 1987 and 1989, the Components Research Award (Technology Development Division, Intel) in 1991, and the Joseph Dias Fellowship and the David and Sylvia Gale Fellowship, both from U.C. Berkeley, in 1984 and 1985, respectively. He serves on the program committees of the Great Lakes Symposium on VLSI, CODES workshop, ICCD, ICCAD, and DAC.