Hardware-Software Co-design: Tools for Architecting Systems-On-A-Chip

Rajesh K. Gupta
Information and Computer Science
University of California
Irvine, California 92697.
rgupta@ics.uci.edu

Abstract— This paper examines the issues and progress in the design of highly integrated microelectronic systems. These microsystems rely on an array of diverse components such as processors, memory, network interfaces, graphics and DSP 'cores.' In particular, we discuss problems in the combined design of hardware and software for these systems. We present a decomposition of the co-design problem, and identify the needed technologies in specification/modeling, synthesis and validation for efficient and error-free system designs. Co-design tools along with domain-specific design methodologies provide a key advantage to the system integrator in building complex single-chip systems. We illustrate this point in the specific area of architectural evaluation using co-simulation tools.

I. Introduction

The continuing increases in the capacity and the integration density of single-chip systems have now made it possible for the system designer to assemble complex systems using pre-designed core or mega-cells and glue logic on a single chip or substrate. Currently available cores include microcontrollers, DSP processors, PCI/VME network interfaces, encryption and decryption engines, signal and image processing elements, modem elements, DMA interfaces and analog interface circuits. The diversity of core-cells continues to increase with new offerings from specialized design houses. This integration capability is now leading to utilization of intellectual content from diverse sources, even from widely different domains such as digital signal processing, networking, graphics and imaging to be co-located on the same chip to create new product categories and applications. These components are often integrated together with a general-purpose microprocessor that allows maximum flexibility in the product design and evolution through programming and software updates.

With the increase in system complexity the system architects are facing new challenges in system design, integration and validation. Testability of these microelectronic systems is also a challenging task. As the use of commodity hardware components increases, the design of software, especially commodity software such as operating system components and device drivers, becomes equally crucial to a good system design. A systematic approach to design begins with the definition of a target architecture. A system architecture addresses a whole host of design choices in system functionality, components used, the macro-level organization of system components and their interconnection. Depending upon the target application domain the system architecture can be quite diverse. We visit this issue of architectural variation later in Section III.

Given a system application and a preliminary target architecture, co-design tasks can be divided into three major areas: specification analysis, design/synthesis, and validation. Typically various subtasks in these areas are carried out in an iterative fashion since many of these tasks are inter-related. As described in the following section, while some of these subtasks can be automated using CAD theoretic tools, a good co-design methodology is built in the context of the application domain and the designer experience.

II. Co-design Challenges

Figure 1 schematically illustrates the major problem areas in the design of single-chip hardware/software systems. The hardware and software components are built using synthesis and compilation tools. In addition, a design environment that allows easy integration and testing of processor with different peripherals is crucial to a successful co-design methodology. We briefly consider the co-design challenges in each of these areas below.

A. System Specification and Modeling

System specification here refers to a description of the application that an embedded system is designed to deliver. Since embedded systems are dedicated to specific applications a system specification also describes the desired system functionality. A model is an abstraction

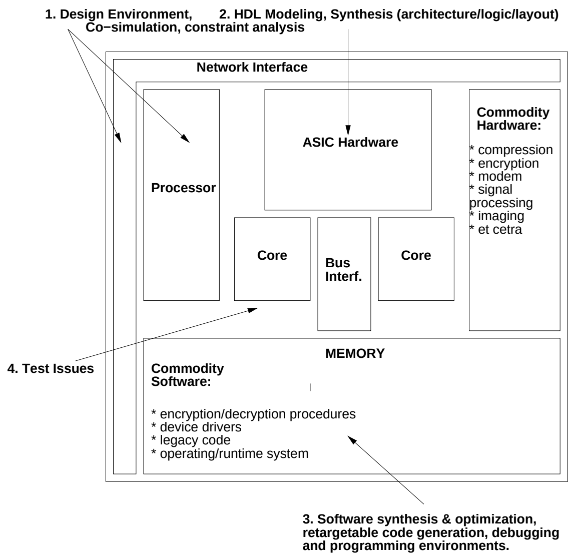
1. Design Environment, Co–simulation, constraint analysis
2. HDL Modeling, Synthesis (architecture/logic/layout)
Network Interface
Processor
ASIC Hardware
Commodity Hardware:
* compression
* encryption
* modem
* signal processing
* imaging
* et cetra
Core
Bus Interf.
Core
4. Test Issues
MEMORY
Commodity Software:
* encryption/decryption procedures
* device drivers
* legacy code
* operating/runtime system
3. Software synthesis & optimization, retargetable code generation, debugging and programming environments.

Fig. 1. Co-Design Problem Areas

of this functionality as the system implementation goes through various stages of design and synthesis tasks. Embedded microsystems with hardware and software are a class of reactive systems [15] that are in continuous interaction with their environment. In contrast to general-purpose computing systems, the execution of software in these systems can be considered as 'open' to system interactions. Because of the multiplicity of system interactions, not only with a user but also physical processes and other systems which constitute its environment, it is generally difficult to establish a complete chain of causal relations from input events at the system interface to system responses. It is, therefore, common to specify a desired system behavior using *timing constraints* between events and actions. These timing constraints are an important part of system functionality in building embedded micro-systems. Specification methods for embedded systems must find ways to express not only system functionality but also the timing constraints. The same requirement also holds for system models which must capture functionality along with the constraints.

High-level programming languages are increasingly being used to describe application functionality, for instance as a C program or using Verilog/VHDL hardware description languages which adhere closely to semantic conventions of popular programming languages [9]. Designer experience with problem solving using programming is providing the momentum to this proliferation even though often these languages are severely inadequate for constraint modeling. Constraints are often expressed as language annotations and/or derived using constraint models. A notable language based on the notion of reactivity, as described earlier, is the synchronous language Esterel [2]. The concept of reactivity appears in Verilog and VHDL as signals and events (a change in a signal's value) as well as the ability to recognize and respond to events.

There are two important challenges in system specification methods: first how to capture the designer intent in the description language in a manner that makes a productive use of the designer time, and its use in a co-simulation environment that enables the designer to build a system prototype capable of running actual applications. To address the first issue, several researchers have taken the approach of building visual aids that allow the designer to use graphical formalisms and entry systems to aid in system conceptualization and design (see for instance, [11, 7]). In [1, 17] the authors have proposed, with mixed success, embedded system prototyping using *several* languages such as a type language, an execution language, an architecture language et cetra. So far the visual formalisms appear to be the most promising candidate for adoption in micro-system design.

B. System Validation

System validation refers to task of verifying that the requisite functionality is delivered by an implementation while maintaining certain desired properties. Among these, a high-speed system-level simulation is a necessary prerequisite for correct and efficient design of these highly integrated systems. Since these systems use general-purpose computing elements, correct system functionality can often be tested by running existing target software applications. In order to test these applications, however, the system designer must be able retarget the application to the new machine's architecture, and execute it along with the application-specific hardware assists and cores. This requires an integrated simulation environment where detailed architectural models consisting of hardware and software components can be built and simulated for performance on new and existing applications. Currently, large scale system design at the highest level of abstraction is done using the programming language C or C++ on simulators such as Tango [8] and MINT [22]. To obtain a meaningful evaluation of system designs, simulation speed is of utmost importance. To improve simulation speed, tradeoffs against simulation accuracy, hardware and language assists have been tried [19, 4].

In general, almost any simulation framework that supports the notion of *events* and provides mechanisms to process these events, can be used to build simulatable system prototypes. This is even more true of modeling reactive hardware systems. Indeed, most existing HDLs can be classified as source languages for a corresponding event-driven simulator [12]. An event-driven model is powerful enough to describe most hardware systems at *any* level of abstraction: from algorithms to gate-level circuits [21]. However, this generality also imposes a significant burden on the simulation efficiency due to the extra work (or overhead) needed in event maintenance and processing. Event processing often requires *interpretation* of event generation, propagation and disposition by the simulation model. "Interpretation" of an object in a simulation model refers to the evaluation of the object by a separate procedure that provides semantic interpretation of the object in the context of the simulation model. Mixed hardware/software simulations are slowed down due to their interface with an event-driven hardware simulator. Recent efforts in this direction (e.g., [6]) have demonstrated the utility of simulation backplanes in integrating various simulators. Though the achieved simulation speeds are not mentioned it is unlikely that hybrid machine simulations can be used to run moderately large application benchmarks which require simulation efficiencies upwards of 100,000 simulated cycles per second.

A critical bottleneck in achieving higher simulation speeds stems from the integration of hardware description language (HDL) models. The HDL simulations often require operation-level interpretation, that is, each operation (statement) requires a call to the interpreter. This is because, each signal assignment statement, in a language such as VHDL, can potentially generate an event, therefore, the system simulations are significantly slowed by frequent calls to the interpreter (or the event manager). An alternative is to build models that work with a cycle-based simulation. A cycle-based simulation, though not necessarily efficient in terms of work required, is often able to use native execution (against software interpretation) to achieve significant reductions in simulation time. Cycle-based simulation using conditional control flow in high-level programming languages has been used to demonstrate speedup in gate-level hardware simulations [23]. Simulation frameworks such at Ptolemy [3] present a promising approach to co-simulation in specific application domains by allowing integrated simulation of heterogenous components through hierarchical encapsulation of model components and extensive library support.

C. System Partitioning and Synthesis Subtasks

Partitioning defines a subtask of system implementation, the other being synthesis subtasks such as operation/task scheduling, hardware resource allocation and binding; and instruction selection, code generation in case of software. The goal of partitioning may be one of following: application-speedup on a given target architecture, greater resource utilization, size and performance constraint satisfaction. In contrast to traditional approaches to system engineering where the hardware and software partitions are relatively rigid, system co-design is characterized by flexible partitions that may be shifted to meet the changing performance criteria, sometimes as a part of the product evolution. Indeed, one of the goals of CAD for embedded systems is to delay this determination of hardware and software to as late in the design process as possible, thus allowing the system designer flexibility in evaluating design tradeoffs while reducing total cost. However, this attempt at delayed determination of hardware and software must be considered against the ability of the designer to use the partitioning results for improved system design and/or synthesis tasks. A late determination of hardware and software reduces its effectiveness in the design of the individual hardware and software components, whereas an early determination is limited by the modeling capabilities. Further, accurate performance characterization and constraint analysis are essential for effective system partitioning. In absence of requisite modeling capability, the system partitioning simply can not be carried out using computer-aided design 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.

A partitioning problem typically has two inputs: a description and/or model of the application and a target architecture. While the description methods often rely on popular programming languages, there exists a great diversity in the models used as input to hardware/software partitioning formulations. For a specific architecture, partitioning of an algorithmic description results in a partition of operations (or tasks) into hardware and software portions. An exact solution to such a partitioning problem even in the simplest cases requires solution to known intractable problems. This provides motivation for heuristic solutions (see for instance [13]). Hardware synthesis for application-specific blocks may use traditional architectural and logic synthesis techniques. Software for embedded systems builds upon techniques for compilation and retargetable code generation. However, there are important differences. The notion of machine description is no longer limited to a specific instruction set architecture. Instead, often detailed machine organization is needed to make the right design tradeoffs. This may include information about the major building blocks, their interconnection or even RT-level behavior of the machine [18]. This makes the problem of generation of good compiler tools especially hard. On the other hand, the relatively fixed nature and the tight memory footprints of the embedded software also afford the possibility of using expensive algorithms to search of optimal solutions by incurring higher one-time compilation costs.

III. Rapid Evaluation of Architectural Alternatives

Design of high performance single-chip systems requires rapid evaluation of possible architectural alternatives both in conventional machine organizations as well as novel architectures for embedded systems. Architectural choices may dictate the hardware and software resources used and the hardware interconnection. For the architectural choices made, parametric determination often requires extensive simulations in the target application environment.

Since embedded systems use combination of hardware and software elements to physically divide the implemen-

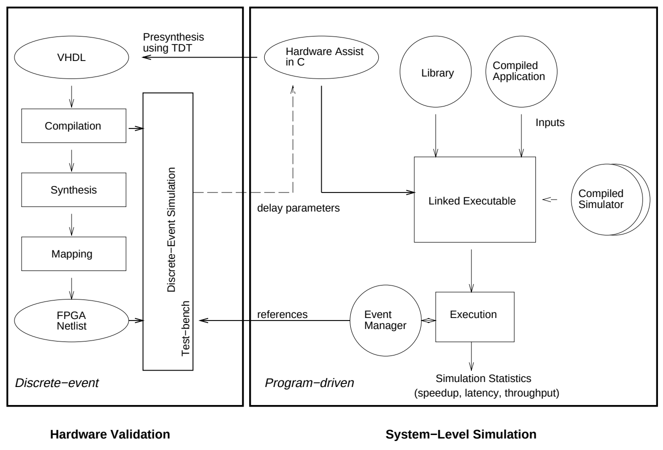

Fig. 2. Architectural Simulation Environment

tation task, the most common architecture in these systems can be characterized as one of *co-processing*, i.e., a processor working in conjunction with dedicated hardware to deliver a specific application. The specific implementations of the co-processing architecture vary in the degree of parallelism supported between hardware and software components. For instance, the co-processing hardware may be operated under direct control of the processor which stalls while the dedicated hardware is operational [5], or the co-processing may be done concurrently with software [10]. The instruction and data paths to the processing and co-processing hardware may or may not be shared depending upon application characteristics. The hardware/software interface defines another architectural variable that strongly affects the partitioning problem formulation.

As mentioned earlier, fast system simulation is key to architectural evaluation. In the following we describe our approach to system simulation that uses a combination of interpretation and native code execution such that the native codes are encapsulated into non-event producing blocks and interpretation is done at a coarse-level. This is done by separating the two very different time-scales of concern related to system simulation and hardware verification. Cycle-based system-level simulation is uses a modified program-driven simulator MINT that performs the system simulation as the application executes. The simulator front end represents interface to the application program, whereas its back end models the underlying micro-architecture. The front end handles the application program execution. When the program execution leads to generation of event (either through reference to modeled blocks in MINT or through directives embedded in the source program), the front ends sends the event to the back end. The underlying simulation library provides event processing and disposition. As an example, consider interconnection to a processor core with a peripheral on an on-chip system bus. A read event generated by the processor will be processed by executing a corresponding procedure call. The back end defines this function, and can do almost anything inside this function representing its handling by the corresponding dedicated hardware block. For instance it can call new functions, create new events, et cetra. This allows us to conveniently customize system simulations for specific hardware blocks. Due to the high-level synchronous nature of the system level simulations, the hardware blocks are modeled at the behavioral level while detailed implementations are only considered in hardware validation.

The compiled simulator (based on MINT) can interpret almost any program that runs natively, and can generate a events specific to any hardware block. The original design of MINT was used to generate events representing references to (cache) memory blocks. However, with appropriate definition of an event, the system simulator can be used to address references to any hardware block. This way the simulation environment allows us to associate events and reduce runtime interpretation to those only on the blocks that are under design. The back end is customized to reflect the underlying system micro-architecture. For architectural blocks that can be identified in the compiled simulator, the applications do not need to be preprocessed or recompiled. (The original MINT simulator interprets the applications that are compiled and linked on a MIPS-based machine. Variants of MINT for other Intel and HP architectures are also commonly available.)

Hardware validation is done using traditional event-driven simulation on hardware blocks. The input to these simulations is derived from translation of C models of hardware into HardwareC [14] blocks. These blocks are subject to presynthesis optimizations based on Timed Decision Tables [16]. (Currently this translation is done manually.) The hardware validation does not require online application executions. Instead, the results from application executions are used to create a test-bench for the event-driven hardware validation. This reduces the redundancy in detailed hardware simulations (for the purposes of hardware validation), while modeling its effect on application-level simulation statistics.

To test the usefulness the simulation environment, we have implemented a micro-architecture in the simulator back end that uses application-specific hardware assists to improve system performance. In particular, hardware is used to improve the performance of memory hierarchy. The memory hierarchy consists of a L1 data cache, a L2 data cache, and the main memory. The L2 cache can be configured as either shared or private among on-chip functional blocks. Four application kernels (Jacobi, SAXPY, Gather-scatter array references and large-scale irregular vector stride) and two complete application benchmarks (MP3D and sparse matrix manipulation package) [20] were compiled using standard C compiler for simulation on this new architecture. Results indicated system simulation speeds from 300,000 to over a million instruction per second for machine models in high-level C programs. Since these simulations use only an abstract view of the hardware blocks and thus are limited in comparing detailed designs of these blocks. However, these simulations are useful in making application-driven design decisions.

IV. Conclusions

Efficient and correct design of a 'system-on-a-chip' requires selection of a suitable system architecture that matches to application needs followed by detailed implementation subtasks. Hardware/software co-design attempts assist this process by providing tools and methodology to describe desired application, carry out performance and constraint analysis followed by compilation and synthesis tasks where both hardware and software are optimized under specific cost criteria. Research efforts currently in progress are focussed on various aspects of modeling and (constraint) analysis, software analysis and synthesis problems. At this time, tool support is limited to only a few co-design tasks such as system specification and timing analysis. However, continued progress in co-design is expected to lead to tools for interactive design partitioning and exploration, a library of (multithreaded) application and operating system code that can be used to build specialized embedded kernels, library of hardware modules that can be easily combined with a given network/bus interface module, static analysis and compilation tools that can help user tightly couple special-purpose hardware units to the code generation process. The special-purpose hardware can be a predesigned library component or synthesized from algorithmic descriptions using high-level/architectural synthesis tools.

Acknowledgments

The author would like to acknowledge support from National Science Foundation CAREER award number MIP 95-01615, NSF/DARPA ASC-96-34947 and from NSF EEC 89-43166.

References

[1] Belz, F. C., and Luckham, D. C. A new language-based approach to the rapid construction of hardware/software system prototypes. In *Proc. Third International Software for Strategic Systems Conference* (Feb. 1990), pp. 8–9.

[2] Berry, G., and Gonthier, G. The Esterel synchronous programming language: Design, semantics, implementation. *Science Of Computer Programming 19*, 2 (1992), 87–152.

[3] Buck, J., Ha, S., Lee, E. A., and Messerschmitt, D. G. Ptolemy: A Platform Heterogeneous Simulation and Prototyping. In *European Simulation Conference* (June 1991).

[4] Covington, R. C., Madala, S., Mehta, V., Jump, J. R., and Sinclair, J. B. The Rice Parallel Processing Testbed. In *Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (1988), pp. 4–11.

[5] Ernst, R., Henkel, J., and Benner, T. Hardware-Software Cosynthesis for Microcontrollers. *IEEE Design & Test of Computers* (Dec. 1993), 64–75.

[6] S. Brown *et. al.* Experience in Designing a Large-Scale Multiprocessor using Field-Programmable Devices and Advanced CAD Tools. In *Design Automation Conference* (June 1996), pp. 427–432.

[7] Gajski, D. D., Vahid, F., Narayan, S., and Gong, J. *Specification and Design of Embedded Systems.* Prentice Hall, 1994.

[8] Goldschmidt, S. Simulation of Multiprocessors: Accuracy and Performance. Ph.D. Thesis, Stanford University, June 1993.

[9] Gupta, R. K. Panel on opportunities and pitfalls in hdl-based ic design. In *Proceedings of the International Conference on Computer Design* (Oct. 1996), pp. 56–57.

[10] Gupta, R. K., and Micheli, G. D. Hardware-Software Cosynthesis for Digital Systems. *IEEE Design & Test of Computers* (Sept. 1993), 29–41.

[11] Harel, D. Statecharts: a visual formalism for complex systems. *Science of Computer Programming 8* (1987), 231–274.

[12] IEEE. *IEEE Standard VHDL Language Reference Manual, Std 1076.* IEEE Press, New York, 1987.

[13] Kalavade, A., and Lee, E. A. A global criticality/local phase driven algorithm for constrained hardware/software partitioning problem. In *International Workshop on Hardware-Software Co-design* (Sept. 1994).

[14] Ku, D., and Micheli, G. D. HardwareC - A Language for Hardware Design (version 2.0). CSL Technical Report CSL-TR-90-419, Stanford University, Apr. 1990.

[15] Kurshan, R. P. Reducibility in analysis of coordination. *LNCS 103* (1987), 19–39.

[16] Li, J., and Gupta, R. K. HDL Optimizaiton using Timed Descision Tables. In *Proceedings of the* 33^{rd} *Design Automation Conference* (June 1996).

[17] Luckham, D. C., Vera, J., Bryan, D., and Augustin, L. Partial Ordering of Event Sets and Their Application to Prototyping Concurrent Timed Systems. *Journal of Systems and Software* (July 1993).

[18] Marwedel, P., and Goosens, G. *Code generation for embedded processors.* Kluwer Academic, 1995.

[19] Reinhardt, S. K., Hill, M. D., Larus, J. R., Lebeck, A. R., Lewis, J. C., and Wood, D. A. The wisconsin wind tunnel: Virtual prototyping of parallel computers. In *ACM SIGMETRICS* (1993).

[20] Singh, J. P., Weber, W.-D., and Gupta, A. SPLASH: Stanford parallel applications for shared memory. Technical report CSL-TR-91-469, Computer Systems Laboratory, Stanford University, Stanford University, CA 94305, April 1991. Available from `ftp://mojave.stanford.edu/pub/splash/report/splash.ps`.

[21] Thomas, D., and Moorby, P. *The Verilog Hardware Description Language.* Kluwer Academic Publishers, 1996.

[22] Veenstra, J. E., and Fowler, R. J. MINT: A front end for efficient simulation of shared-memoy multiprocessors. In *Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)* (Jan. 1994), pp. 201–207.

[23] Wang, L. T., Hoover, N. E., Porter, E. H., and Zasio, J. J. Ssim: A software levelized compiled-code simulator. In *Proceedings of the Design Automation Conference* (1987), pp. 2–8.