Hardware-Software Co-Design and ESDA

Kurt Keutzer
Synopsys, Inc.
Mountain View, CA
keutzer@synopsys.com

1 Introduction

Hardware-software co-design is not a new problem; systems with significant portions of hardware and software have been designed for decades. Similarly, *system design* is not new but the prospect of Electronic System Design Automation (ESDA) has received a great deal of recent attention. Several factors have contributed to the growing importance of these areas but two are particularly relevant: Computer-aided design can claim some modest success in responding to the productivity challenge posed by exponential improvements in processing technology. At the same time, time-to-market challenges for system designers require that complex systems of hardware and software be designed and deployed in 18 months. As a result the formerly distant relatives of hardware and software development have now been brought together and the importance of taking a system perspective has been stressed.

This abstract will very briefly identify the problems associated with hardware-software co-design and ESDA and will attempt to identify the most promising of the current responses to these problems. To aid in identifying these problems it may be useful to understand two different trends in the evolution of design automation. The first trend is the familiar *raising the level of abstraction*, which we associate with ESDA, and this will be treated in Section 2 as well as in [4]. The second trend is *broadening design coverage*, which we associate with hardware-software co-design, and this will be the focus of Section 3 as well as [6].

2 Raising the Level of Abstraction

To supply the increase in productivity required by the exponential improvements in processing the most consistently useful approach has been to raise the level of abstraction. Within a relatively short span of about 15 years the primary level of design entry has evolved from the transistor level captured in mylar, to the gate level captured in schematics, to the register-transfer level captured in hardware description languages such as VHDL or Verilog. To effectively raise the level of abstraction entails providing a formal model at the new level of abstraction. Gate-level netlists or register-transfer level HDL models may seem like simple data representations but each of these is also associated with a clear semantics that not only captures data but allows for analysis and synthesis. Simply providing a higher level of abstraction is not sufficient. What must also be supplied is a set of design tools that allow for entry, analysis and verification at that level, as well as synthesis tools to target to lower levels in the hierarchy. Thus, to truly be a new design paradigm, ESDA must address each of these aspects and this will be detailed in the subsequent paragraphs.

Level of Abstraction By definition the appropriate level of abstraction for ESDA is the system level. At present ESDA does not provide a *single* model of system behavior that spans both hardware and software but instead a number of different models of behavior have been proposed each of which can be used to describe a portion of system behavior.

Work on formal models of dataflow behavior grew up in the DSP application domain and synchronous dataflow [5] is a formalization of a model of computation that DSP designers have successfully used for some time. Variants of the synchronous dataflow model are reflected in the commercial offerings of COSSAP from Cadis and SPW from Comdisco. Work on formal models of control-oriented behavior has evolved up in domains associated with control-dominated systems such as mechanical control systems or telecommunication protocols. Among the various models of control, *statecharts* [3] has been particularly successful and is reflected in the i-Logix toolset.

To properly model many complex systems requires the integration of more than one computational or application domain. This raises the problem for any ESDA environment as to how to provide for integrating multiple domains within a single environment. Addressing the need for such a *heterogeneous* environment for simulation has been the focus of the Ptolemy environment [2]. In Ptolemy multiple computational domains can be integrated and their communication scheduled.

Tools for Verification and Analysis System-level verification entails verification of both the functionality and the performance of the system. Simulation remains the primary workhorse for verification of system-level functionality and simulation issues will be discussed in Section 3. A radically different approach to verifying functionality is to use formal verification, but current formal techniques appear best suited for deeply analyzing particular problems, such as cache coherency, and not for providing broad functional verification. Comprehensive system-level performance analysis has almost no tool support. A system designer wishing to make significant system-level trade-offs typically enjoys the use of only a few specialized tools, such as tools for queing analysis, or more generalized versions of the same in a tool such as the WORKBENCH from SES. Moreoever, given a system-level description in any one of the domains mentioned in the previous section there are currently no reliable tools for estimating the performance of the final implementation of that description in either hardware or software. In the absence of reliable system-level estimation tools for either hardware or software the prospects of automated hardware-software partitioning are dim.

Synthesis There is a truism in the industry that analysis proceeds synthesis but it is also useful to observe that synthesis leverages the investment in analysis. For example, simulation of HDL models of circuits was considered useful before the advent of synthesis but it was the ability of synthesis to provide a path of implementing the circuit that made the investment in a simulation model easier to justify. Similarly, the market for system level entry and analysis tools has grown slowly but through improving links to both hardware and software implementation there is a promise that the market may grow much more quickly. Substantial progress has been made in the development of translators from system level descriptions to both programming languages, such as *C*, and hardware description languages, such as *VHDL*. However, it is not sufficient to simply supply links to implementation. Those links must be able to provide efficient implementations. Nevertheless, current implementations resulting from domain specific entry and synthesis techniques have been sufficiently competitive to be used for a number of industrial designs.

3 Improving System Coverage

Whenever the level of abstraction is raised, more and more implementation details need to be addressed in some way. The migration from transistor-level to gate-level required not only gate level simulators and schematic editors but also physical design tools that could place and route the resulting netlist. As the level of abstraction is raised to the system level, all the elements of the system level need to be addressed in some way. The emerging problem is the implementation, analyis and verification of complex systems with interacting hardware and software components. To attempt to quantify the problem: In current processing technologies an entire system consisting of a microprocessor, peripherals, memory , ASIC and software can be implemented on a single chip. Described at the system level this system can consist of hundreds of interacting domain-specific modules. The final implementation of this system in hardware and software can easily result in 150,000 lines of Verilog at the register-transfer level and an additional 30,000 lines of C code that will reside in the program memory. To verify such a system entails more than simply verifying the hardware in isolation, building a prototype, and verifying the software on the prototype. Verifying such a system under strong time-to-market pressures requires *co-verifying* the hardware and software as they are developed. A more comprehensive treatment of the problems associated with hardware-software co-verification will be presented in a companion paper in this session [6] but problems associated with one approach to the problem will be briefly described. In one proposed environment ASIC hardware under development would be co-simulated with the actual software under development. Principal constituents of this environment are then: simulator, processor models and interfaces to software compilers and debuggers. In this environment the processor models and ASIC circuitry are both run in the same simulation environment. Software is executed in memory as data for the processor models. In this way software and ASIC circuitry can each be debugged in their native environments. Obstacles to this approach will be touched on below:

Simulation Speed Chief among the requirements of any viable hardware-software co-simulation environment is simulation speed. While designers of micro-controller based systems have been able to do significant software debugging using simulators running at only 1-10 cycles per second, input from other designers is that simulation speeds in excess of 100 cycles-per-second will be necessary before even the core algorithms of an audio DSP system can be debugged. To supply these speeds will require the integration of cycle-based based simulation techniques [1], as well the development of design methodologies that make cycle-accurate modeling sufficient.

Processor Models Another requirement for hardware-software co-simulation is the availability of simulation models. To simulate the execution of software in a standard simulation environment it is necessary that a simulation model of the processor be available in that environment.

Interfaces In a true hardware-software co-simulation environment both software developers and hardware developers should be able to use their own respective debugging environments. To accomplish this new interfaces need to be created between software development environments and existing hardware simulation environments.

4 Conclusion

There presently exist well-defined computation models at a higher level of abstraction than the register-transfer level HDL's and these models can be analyzed and synthesized into either hardware or software. Using these techniques offers the potential for significant productivity boosts over current register-transfer level synthesis techniques. The use of these techniques does introduce new problems, or at least exacerbate existing ones. Chief among these emerging issues is the problem of verifying a complex software system interacting with a complex hardware system. Verifying such systems in a timely manner will require signicant improvements in simulation speed, the easy availability of processor models and the existance of links from hardware simulation environments to software compilers and debuggers. As these tools come into place over the next few years it does appear that system designers of the future will be reasonably well equipped to cope with another rise in the complexity of system development.

References

[1] Z. Barzilai, L. Carter, B. Rosen, and J. Rutledge. HSS - a high-speed simulator. *IEEE Transactions on Computer-aided Design*, CAD-6(4):601–617, July 1987.

[2] J. Buck, S. Ha, E. A. Lee, and D. Messerschmidt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. In *International Journal of Computer Simulation*, 1994. to appear.

[3] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, February 1987.

[4] A. Kalavade and E. A. Lee. Manifestations of heterogeneity in hardware/software codesign. In *Proceedings of the DAC*, June 1994. to appear.

[5] E. A. Lee and D. Messerschmitt. Synchronous data flow. In *IEEE Proceedings*, 1987.

[6] James A. Rowson. Hardware/software co-simulation. In *Proceedings of the DAC*, June 1994. to appear.