Hardware/Software Co-Simulation Strategies for the Future

by Brian Bailey, Russ Klein, & Serge Leef
Mentor Graphics Corporation

Abstract

The majority of the systems being designed today are embedded systems that consist of standard and custom hardware as well as standard and custom software. Traditionally, the EDA vendors have focused on tools and methodologies for automation of hardware design. As a result, the effects of the software as a component of the system have been largely ignored by the commercial EDA tools. While it is possible to execute microscopic amounts of software on the simulated, the performance and usability of current generation of tools is grossly inadequate for true hardware/software co-simulation. Similarly, the software vendors have not looked at much outside of the processor or the interaction between the software and the hardware processes. This paper examines common approaches to hardware/software co-simulation problem and proposes an idealized strategy that merges the capabilities that have been developed in both of these domains into a single environment.

Introduction

Embedded systems are task-specific computing devices that consist of standard and hardware and software. Standard hardware is typically made up of a commercial microprocessor or microcontroller, memory and a small number of standard parts. Custom hardware is implemented as Application Specific Integrated (ASICs) or in some cases as Field Programmable Gate Arrays (FPGA's). Hardware architecture binds and constrains these resources and provides the framework on which the software processes execute. Standard software frequently consists of a Real Time Operating System (RTOS), and configurable device drivers. Custom software is the embedded application. Software architecture defines how these processes communicate.

The complexity of these systems varies widely from low to high end depending on the market segment and product goals. They can be found in almost everything that we in our daily lives, such as communication systems ranging from the phone your desk, to the large switching centers, automobiles, consumer electronics, avionics and many others. In fact some of today's luxury cars contain more than 40 independent embedded systems, controlling such things as the suspension, engine, anti-lock brakes and user consoles. Some typical systems and their attributes are shown in figure 1.

System type	**CPU**	**Memory**	**ASIC**	**RTOS**	**Embedded Application**
Basic Consumer Electronics Device	Intel 8051	256KB	none	Simple Scheduler	10K lines of assembly code
Automotive Control Sub-System	Motorola 68300	512KB - 1MB	20K gate FPGA	Custom Microkernel	10K to 50K lines of assembly and C code
Telecom Switching Module	Multiple Pentiums	2MB+	Multiple ASICs from 30K - 100K gates each	Full RTOS	Up to 2M lines of C and C++ code

Figure 1. The Range of application domains

Embedded systems can be software dominant. That is, standard or previously designed hardware is utilized while the software makes up all or most of the design's value added. These, cost sensitive systems are quite common in the consumer electronics products. Software dominant systems do not present a hardware/software co-simulation challenge: the system can be validated by executing software on the existing hardware.

Hardware dominant systems are usually found in the applications where performance is a critical success factor. In these systems most of the design team's efforts are focused on implementing the functionality in hardware (e.g. boards, ASICs). The original software content of these systems tends to be small, making co-simulation a very small part of the system integration.

The largest and most complex systems being designed today are neither software nor hardware dominant, where both parts play an equally important role in the complete system. It is in this type of system that the challenge for hardware/software co-simulation exists.

Hardware/Software Integration

The typical electronic system development project is divided into three basic phases: System level design, Hardware/Software design and implementation, and Integration and Test. In interviews with a large number of customers, a striking consistency was observed in the relative duration of each phase, with integration and test taking a surprisingly large portion of the total time. This is below in shown in Figure 2.

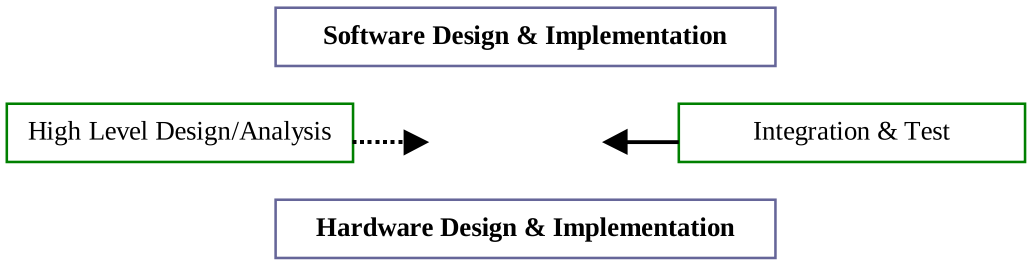
Software Design & Implementation
High Level Design/Analysis
Integration & Test
Hardware Design & Implementation

Figure 2. Typical Phases of Development

1. *System Phase* -- during this phase the entity being designed is viewed as an overall system, rather than as distinct hardware and software components. This phase, typically completed by system, results in specifications for how the system will behave. Other deliverables include the architecture and functional specification of the system. Additionally, functional requirements and budgets for both hardware and software components of the system are created. These include constraining costs, size, performance, and physical attributes. At the conclusion of this phase, the system is partitioned into software and hardware sub-systems. At the present time, there are very few automation available in this area, but is the subject of considerable research work being carried out in both industry and academia. In fact there is currently an un-bridged gap between this phase and the subsequent design phase caused by ambiguous, informal documents being used as module specifications.

2. *Hardware/Software Design Phase* -- during this phase separate organizations (except in the case of small projects) address their respective problems. The Hardware and Software design and implementation efforts typically start at the same time and ideally end at the same time. However, most customers that software development continues well beyond the start of integration and test, and in some cases may not be complete at the time of first customer ship. In other cases, software teams are ineffective at producing quality code prior to availability of hardware prototypes. This is because they do not have adequate means to perform meaningful tests. Spanning the gap between the two design teams are the firmware engineers. Firmware engineers normally develop low level software that may interfaces to the hardware, or provide software task scheduling. These low-level routines provide a software foundation for the higher level software. The higher level, or application, software is where the unique functionality of the product is usually implemented.

3. *Integration and Test Phase* -- In theory, Integration and Test is the final series of checks prior to the shipment of the system. In practice, it is the first time that the "completed" hardware and independently developed software come together as a system. At this time numerous issues surface, namely: the effects of misinterpretations of interface definitions, out-of-date specifications, poorly communicated changes between the teams, and ineffective performance modeling, etc. Consequently, one third or more of the total

development time is spent in this phase. As the tools progress and become more friendly to both the hardware and software developers, the overlap between the test phase and the design and implementation phases will increase.

Faced with cost and schedule deadlines, developers are often forced to redesign and/or lower product objectives when integration problems are found. Given the long fabrication lead times and costs associated with redesigning and re-spinning ASICs, the rework is frequently performed in software. This is not always the best solution for the end product. Integration and Test becomes Redesign and re-implementation, and can take about the same amount of time to complete as the original design and implementation.

Software change costs also tend to be less visible and tangible than the cost of an ASIC turn and so the eventual product may be compromised. Also in some cases, the first product release will not contain all of the intended software functionality because it has not been possible to start the integration effort earlier in the design. Two things are necessary before virtual integration and test can be accomplished. The first is the ability to simulate the hardware at speeds sufficient to make software execution a reality. In most cases, this means that the overall simulation performance must be increased by a factor of at least 1000 over the current execution speeds for hardware oriented simulation products. The second is the need to bring the debug and development environments for the hardware and software closer together. Simulation waveforms do not provide a natural way for a software engineer to debug high level languages. The original source form for both the software and hardware must be maintained within a single unified debugging environment. Various levels of detail about the operation of the system should be available to the user, depending upon the type of problem that is being tracked down.

Driving Forces

There are three primary driving forces affecting the market for hardware-software tools: the need to reduce time-to-market, increased software content, and increased design complexity. These are discussed below.

Need to Reduce Time-to-Market

There are a number of studies that clearly show that early detection of design errors will dramatically reduce the amount of redesign/rework needed during Integration and Test. It has also been observed that the cost of testing during the development is relatively small compared to the cost of rework in the latter stages of the process. It seems to be generally true that investing energy in the front-end activities reduces downstream costs and results in a better overall product. This approach has worked in other instances, and is one of the foundations of the quality movement (do it right the first time, correct by construction, etc.)

Developers and management both view the reduction in the length of the Integration and Test phase to be the most immediate way to achieve their improved time-to-market objectives and to improve the risk levels associated with on-time delivery. Methodology changes during the Design and Implementation phase are widely seen as the most likely source for overall time-to-market improvements. While there is a reluctance to accept new methodologies, due to the risks associated with less proven technologies, some of the pioneers in this area are reaping large rewards.

Increased Software Content in Electronic Systems

As the microprocessors and microcontrollers have become an integral part of embedded designs, the percentage of electronic manufacturers' R&D budgets allocated to software engineering has increased radically. It is estimated that the ratio of hardware to software engineers at most has reversed over the past 11 years, such that there are now about 2-3 software engineers for every hardware engineer. That means hardware is decreasingly the dominant factor in the time-to-market equation. At the same time, the complexity of the software content is also increasing, with greater percentages of the code now being written in high level languages rather than at the assembly level.

In addition, product families are being created by migrating functionality into software. While this gives higher levels of re-programmability, and customized solutions based on a single hardware platform, it also puts more strain of the overall development process. This occurs because the hardware become fixed, and possibly in production well before some of the software is written. This is also likely to stimulate changes in methodology.

Design Complexity Renders Current Techniques Impractical

Hardware simulation has been reasonably addressed by the tools available today. Since typical hardware validation calls for the examination of micro to millisecond simulation time frames, the necessary performance level matches up well with today's simulation technology, which is approximately 7 orders of magnitude slower than real time. This allows only small portions of real-time activities to be simulated.

Embedded software requires a different level of performance. Validation of the functionality of embedded software frequently calls for execution of seconds to minutes of real time. This means that software developers would have to wait 14 days to execute 1 second of real time at current simulation speeds, which is clearly not acceptable.

Between the hardware and software components of the system, the is inserted. The firmware is typically made up of device drivers, kernels, diagnostics and boot code. The performance requirements for firmware simulation fall somewhere between the two extremes described above. It is clear, however, that current simulation tools are about three orders of magnitude too slow for adequate firmware validation.

It is because of the gap between the performance needed by the software community and the performance and usability currently available from the hardware community, that the co-simulation market has not yet evolved.

Software Execution

The software and the processor that it runs on can often be removed from the traditional hardware simulation environment. Since the host processor on which the simulation is running has much of the same capabilities as the target processor, large performance gains can be made by using these capabilities rather than simulating the same operation happening on the target processor. This use of the host capabilities can take place at different levels and results in the following kinds of execution models.

Bus Functional Model

These models do not contain any of the internal processor functionality associated with the instruction set. Only the interface circuitry is modeled and it is usually driven by a command language or small programming language such as a subset of C. The directives expressed in the programming language are converted into a timed sequence of signal transitions, which are fed as events into a traditional hardware simulation environment. This is the most commonly used technique today. While it is a powerful hardware debug technique, it does little to help embedded software developer since it does not use a standard language for its interface and is at too low a level of abstraction. If this deficiency is corrected, then it naturally evolves to the next option of compiled code.

Compiled Code

A mild improvement on the above method is known as Compiled Code Execution. Instead of a C subset, the entire language is supported. Here the embedded software is compiled for a host other than the target CPU. As the software executes on the host the I/O transactions are trapped and selectively translated into bus cycles (as above) that excite the surrounding hardware. The presence of the Bus Functional Model is still required as shown below in figure 3.

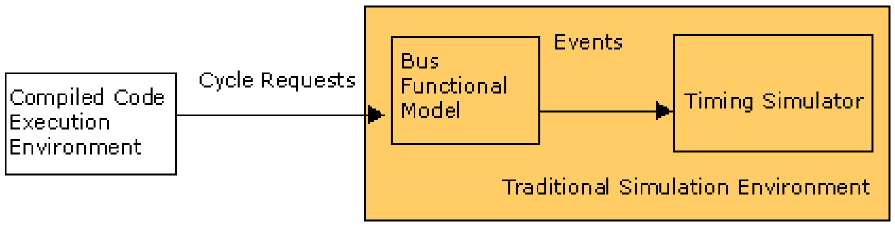
Compiled Code Execution Environment
Cycle Requests
Bus Functional Model
Events
Timing Simulator
Traditional Simulation Environment

Figure 3. Simple Co-simulation solution

While this methodology gives high performance, in some cases faster than real time, it does have a number of severe disadvantages --

- Only suitable for high level code. Since the host and target computer systems are different, machine code or legacy code cannot be executed with this kind of model. It also makes commercial RTOS or software libraries unusable.
- No view of the internal registers. The compiler only knows about the internal structure and registers of the host machine and not the target. Thus low level debug is not possible.
- Results may be inaccurate. While it is expected that integer operations would be common amongst computers, floating point operations are not. Difficulties can also exist where the word length of the two machines is not the same.
- No use for performance evaluation. So much of the real system has been abstracted away that cycle counts or real time constraints cannot be evaluated.
- Unrealistic interaction with hardware. Since code fetches and most data accesses are going to occur completely within the host system, a rather distorted view of the executing software is presented to the hardware system. This makes it impossible to debug bus arbitration or contention logic, or evaluate the basic performance characteristics of the hardware.

All I/O locations or memory that exist within the simulated hardware need to be identified to the software execution environment. When the host system attempts to access these locations it must either make the requisite calls to the rest of the simulation system automatically, or the user must insert special function calls to do it manually. The Bus Functional Models are required to transform these requests into a timed sequence of signal changes that are fed into the hardware simulator. As the number of locations in the hardware system increase, the performance advantages of this method diminish.

Instruction Set Simulation

Instruction set models provide full functional accuracy of the processor as viewed from the pins of the device and correct most of the restrictions of the compiled code execution model. All of the important state storage elements of the processor are preserved, but the data path that connects them is abstracted out of the model. The processor is described in terms of its instruction set. That is, a collection of instructions is modeled where each instruction defines a relationship between constructs that are internal (registers, on-chip memory) or external (on-board memories) to the processor. Full bit level accuracy exists within the models. Instruction set models allow both high level and assembly code to be executed and debugged. Instruction Set Models can exist at two different levels of accuracy:

Instruction accurate

These models are accurate at instruction boundaries only. While they do guarantee that all of the correct bus cycles will be performed and that the total number of cycles for the instruction will be correct, they do not guarantee that the bus operations will occur during the correct clock cycles. They also do not attempt to assure the correct internal state at individual clock boundaries. These models are adequate for most situations. Their performance is good and they closely approximate the eventual code performance executing on the real hardware. Some inaccuracy may creep in when bus sharing or contention exists, since the exact time of the bus cycles may not be correct.

Cycle accurate

Cycle accurate models guarantee the internal and external state of the model at every bus cycle. This means that the exact bus behavior of the target device will be seen. These models, while running considerably slower than instruction accurate models, are still significantly faster than full behavioral models of the processor since all of the individual events within the processor data path are suppressed. In many cases, these models may actually mimic the micro architecture of the processor. An example of this is the Motorola PowerPC timing simulation model (PPCSim603), where each of the functional units of the processor is modeled and communication between them occurs through synchronized interfaces [1].

Hardware Simulation

While event-driven simulation tools have achieved relatively high levels of performance, they all face the same performance barrier: management of the event queue to service all transitions introduces a theoretical limitation on overall performance. Behavioral models serve to reduce the total number of events in the system and compiled code techniques transfer large amounts of work from the run-time to the compile stage, however, simulation performance is still orders of magnitude away from the minimum levels required for the realistic execution of software. We must thus look to new techniques to achieve the required performance.

Cycle

Several emerging techniques exist that can achieve higher performance rates at the cost of reduced resolution. Cycle simulation allows the simulator to abstract away the timing details of internal transactions by statically scheduling the evaluations to occur in the correct sequence when clock edges occur. This eliminates a vast amount of computation and can lead to 10X to 100X performance gains over traditional event based simulation. Other techniques have evolved from Binary Decision Diagrams (BDDs) which effectively pre-compute functions given a current state and a set of inputs.

These tools have not been adopted widely. Cycle simulation puts significant strain on the design methodology without delivering performance level sufficient for execution of meaningful amounts of software.

Emulation

Another technique that has been explored for this application is Hardware Emulation. In this approach, the design that would ultimately be implemented as one or more ASICs is mapped onto programmable hardware. The programmable hardware assumes the personality of the loaded design and realizes its execution at 1MHz to 10 MHz. While the execution speed of Hardware Emulation is more than sufficient for embedded software debug, introducing companion elements of pre-existing hardware (such as CPU) can be cumbersome. Historically, these devices have suffered from high cost, very long design iteration times, and a lack of comprehensive debug capabilities. New emulation technologies are now coming onto the market which will address and correct many of these deficiencies.

Key Issues

The techniques described above have all been explored in the field. While the results have been uneven, several key issues surface repeatedly.

Performance

The techniques described above all offer different levels of stand-alone performance. It is, however, the combined hardware/software performance that is important. For instance, it is irrelevant how fast the software side of the system executes if the hardware simulation is left unaccelerated. Even if the processor, memory and embedded code execute infinitely fast, the overall performance of systems with high original hardware content (greater than 40K gates) can remain close to unaccelerated levels. When dealing with performance, it is crucial to address the acceleration of the entire system rather than that of the separate hardware or software components. This requires a merging of the techniques and capabilities developed over many years by the traditional EDA vendors and the embedded software tool companies

Accuracy

While it may be practical to abstract significant amounts of detail out of the analysis session, accuracy must be maintained. No level of performance can be justified, if it cannot faithfully represent the expected behavior of the system under test. Optimizations that cause a reduction in the level of accuracy must be under user control and discretion. For example, many interactions between hardware and software are sensitive to timing constraints. The absence of accurate instruction timing may make it impossible to truly validate the hardware/software interface. Once the user has established that the interface transactions are occurring correctly, it may be practical to reduce or eliminate all further occurrences of these, in order to get higher performance levels. It should however, be possible at any point in time to re-establish full hardware/software interactions.

Usability
It is pivotal to have an accurate machine-specific view of the software execution in addition to the views traditionally provided by the hardware simulation tools. A common debug environment must offer a layered view of the software at both the process and code levels and the hardware depending on the user needs. Visibility of the processor states and ability to create composite breakpoints are important elements of effective hybrid system debug environment.

Model Availability
All of the above approaches require some type of processor model. As a minimum, a Bus Functional Model is required for the Compiled Code Execution method. We believe that an Instruction Set Model is needed for meaningful Hardware/Software Co-simulation. Neither of these model types is trivial to create, thus it is critical to have access to a sizable portfolio of CPUs commonly used in embedded system design. Furthermore, performance of the interface connecting such models and the rest of the system must be highly optimized.

Cross-Domain Optimization
In addition to the combination of hardware and software acceleration techniques outlined above, there are also many cross-domain optimizations that can be made. This is the area in which the various vendors will bring unique and differenciatable value to the co-validation marketplace. In essence, it relies on the ability to selectively suppress activity from crossing the hardware/software boundary in a manner that does not result in a loss of accuracy, or accessibility to data. Many ideas in this area are currently the subjects for patent applications.

Summary

	ASIC Stimulus	Protocol Simulation	H/S Interface Simulation and Firmware Debug	System Debug (RTOS +Apps)
Bus Functional Model	**V**			
Compiled Code Execution + Event Sim	**V**	**V**		
Instruction Set	**V**	**V**	**V**	
ISM + Hardware Emulator	**V**	**V**	**V**	**V**

A number of application domains exist, for which different co-simulation solutions can be created. Some of these are shown in Figure 4. The current method of choice for most hardware engineers today is the use of bus functional models to drive stimulus into their ASIC.

The major EDA vendors have been experiencing pressure from their customers for some time now, to push the solution envelope into hardware/software co-design and co-simulation arenas. Mentor Graphics Corporation intends to fill this need, and has started to develop and acquire the technological expertise necessary to put in place a complete product offering to cover this domain.

Microtec Research has one of the largest portfolios of instruction set models on the market today, along with complete development environments for these processors. They also have many years of experience in the creation of debuggers and real-time development tools for the embedded software developer. Mentor Graphics and Microtec Research have been working together for some time now to create a breakthrough in the co-simulation market place. This will result in significant product introduction during the coming year.

In addition, the pending acquisition of Meta Systems in France, gives Mentor Graphics access to the next generation of emulation system. This technology offers turn around times and debugging capabilities that enable it to be the first emulation system capable of being used at very early points in the design process. Work is underway to fully integrate this into the co-simulation solution. Products can also be expected from this work later in the year, with stand alone emulators available now.

In the longer term, Mentor Graphics and its partners are working towards the definition and implementation of hardware/software co-design solutions and the tools necessary to support the next major wave of productivity improvements in the electronic design market place.

References

1. Anderson W. An Overview of Motorola's PowerPC Simulator Family. Communications of the ACM June 1994 Vol. 37, No. 6

2. Schulz S.E. Bridging the Gap to Software. ASIC and EDA, September 1994

3. Mancini G. HW-SW coverification in ATM. Proc 7th International Symposium on High-Level Synthesis. May 1994.

4. Wolf W. H. Hardware-Software Co-Design of Embedded Systems. Proc IEEE Vol. 82, No. 7 July 1994.

5. Thomas D.E., Adams J.K., Schmit H. A Model and Methodology for Hardware-Software Codesign. IEEE Design and Test. Sept. 1993