Software Timing Analysis Using HW/SW Cosimulation and Instruction Set Simulator

Jie Liu
Department of EECS
University of California
Berkeley, CA 94720
liuj@eecs.berkeley.edu

Marcello Lajolo
Dipartimento di Elettronica
Politecnico di Torino
Torino, ITALY 10129
lajolo@polito.it

Alberto Sangiovanni-Vincentelli
Department of EECS
University of California
Berkeley, CA 94720
alberto@eecs.berkeley.edu

Abstract

Timing analysis for checking satisfaction of constraints is a crucial problem in real-time system design. In some current approaches, the delay of software modules is precalculated by a software performance estimation method, which is not accurate enough for hard real-time systems and complicated designs. In this paper, we present an approach to integrate a clock-cycle-accurate instruction set simulator (ISS) with a fast event-based system simulator. By using the ISS, the delay of events can be measured instead of estimated. An interprocess communication architecture and a simple protocol are designed to meet the requirement of robustness and flexibility. A cached refinement scheme is presented to improve the performance at the expense of accuracy. The scheme is especially effective for applications in which the delay of basic blocks is approximately data-independent. We also discuss the implementation issues by using the Ptolemy simulation environment and the ST20 simulator as an example.

1. Introduction

Timing is one of the most important issues in real-time embedded system designs. The correctness of designs largely depends on the correctness of the interaction of hardware and software modules. Hardware/software cosimulation provides an integrated way to simulate this interaction, because it not only simulates the functionality but also simulates the delay of each module and the timing relations among the events. So it is essential for the simulator to use timing information for each module, especially software modules, which is as close to reality as possible. An incorrect delay may cause the simulation result to differ from the behavior of the implemented system.

Polis[1] is a hardware/software codesign environment for control dominated embedded systems. Polis is based on a formal model of computation called codesign finite state machine(CFSM). In Polis, systems are modeled as a group of communicating CFSM's, each of which is originally described in a formal language. e.g. Esterel[3]. In the simulations phase, both hardware and software modules are simulated in the Ptolemy environment.

Ptolemy[2] is a complete design environment for simulation and synthesis of mixed hardware-software embedded systems. In Ptolemy jargon, each functional block (a software or a hardware module) is called a star. Each star has one or more input and output ports. Stars talk to each other through links between ports that carry discrete events as FIFO queues.

A typical design flow in Polis is as follows (of course there will be feedback among different stages):

1. Create the system specification using synchronous-reactive system specification tools, such as Esterel.
2. Compile the source code and generate CFSM models.
3. Build simulation modules (stars) in the Discrete Event domain of Ptolemy.
4. Select target resources (microcontroller and real-time scheduler), assign each star as either hardware or software.
5. Run the simulation in Ptolemy, paying special attention to deadline violation and timing consistency.
6. If the simulation result is satisfying, synthesize the system into software and/or hardware.
7. Repeat more detailed simulation.

The software performance issue (clock cycles needed for a software module to execute) is involved in step 5 above.

Software performance has to be estimated in hardware/software codesign tools[4][5][10]. The advantages are small simulation overhead, ease of integration, and flexibility of porting to multiple microprocessors[8]. The disadvantage is the poor accuracy. For hard real-time embedded system, it is crucial to provide accurate timing information during the simulation phase.

Instruction Set Simulators (ISS) are software environments which can read microprocessor instructions and simulate their execution. Most of these tools can provide simulation results like values in memory and registers, as well as timing information (e.g. clock cycle statistics). Thus, ISS's provide a way to refine the timing calculation during the simulation phase.

We begin in section 2 with the analysis of existing soft-

ware timing estimation methods and their restrictions. In section 3, an interprocess communication architecture and a simple protocol are designed for integrating Ptolemy with ISS's. In order to improve performance, a cached timing refinement scheme is presented in section 4. An implementation example is given in section 5.

2. Related Approaches

Although the idea of performing a hardware/software cosimulation by combining RTL hardware simulation with cycle accurate instruction set simulators (ISS) is not new, a seamless integration of the two environments with high accuracy and good performance is still an unsolved problem.

In [9] a method which loosely links a hardware simulator with a software process is proposed. Synchronization is achieved by using the standard interprocess communication (IPC) mechanisms offered by the host operating system. One of the problems with this approach is that the relative clocks of software and hardware simulation are not synchronized. This requires the use of handshaking protocols, which may impose an undue burden on the implementation. This may happen, for example, because hardware and software would not need such handshaking since the hardware part runs in reality much faster than in simulation.

In [5] a method, which keeps track of time in software and hardware independently and synchronizes them periodically, is described. In [7] an optimistic and non-IPC approach for improving the performance of single-processor timed cosimulation are presented.

The biggest problem with all the current strategies is how to optimize the communication overhead required to synchronize the execution of the ISS and hardware simulator. Simply synchronizing the hardware and the software simulator in a lock-step fashion at every clock cycle would result in very limited performance due to the very low simulation speed obtainable. The other typical drawback of this approach is that the system has to be recompiled and resynthesized when the partition is changed.

The commercial tool Mentor-Seamless CVE [12] can offer a certain number of optimization techniques such as an instruction-fetch optimization that can directly fetch operations to the memory-image server, thus eliminating these cycles from the logic simulator workload. With a memory image server for memory read/write optimization, it can process operations 10,000 time faster than a logic simulator. This optimization is controlled by the user: early in the co-verification process, all memory operations should be directed to the logic simulator to debug all the operations of the memory sub-system. As the co-verification progresses, larger amounts of memory access can be optimized gradually, further speeding up the software execution.

The link with an ISS is not only applicable to hardware software cosimulation, but also useful for the general problem of embedded program timing analysis. For example, in [4] an approach which combines simulation with formal techniques is presented. They address the timing analysis problem by trying to approach each step in the analysis with the best methods currently known. They perform an architecture classification and a successive analysis and combine the classical approach of Instruction Timing Addition (ITA), in which the addition of the execution times in a basic block or in a path segment is computed, with a Path Segment Simulation (PSS), in which a cycle true processor model is used on a specific program path. Basically, they use the ITA approach for addressing the problem of data dependent instruction execution timing typical of some microcoded CISC architectures (e.g. multiplication) and PSS for all the other problems related to the impact of the architecture (pipelining, caching, superscalarity). The ISS is run only once on all basic blocks in the program and then the information collected is used with a formal analysis to determine the worst case execution time of the program.

Our approach is close to [4], but we determine which basic blocks are used at run-time by using a delay caching technique. Our approach is to leverage on an existing approach to time-approximate cosimulation[1], based on source-code estimation of execution time, and refine its precision by using an ISS. In particular, we do not require the designer to change the system specification. Hence we preserve the partitioning approach based on rapid interaction with the simulator, without the need to recompile or modify the specification (other than change implementation attributes for modules). The performance remains acceptable, with only a slight decrease in accuracy, thanks to a totally automated caching approach.

We also standardize the interface between the ISS and the system simulator, thus making the porting to a new ISS very easy.

3. Architecture and Protocol Design

The integration of Ptolemy and ISS should have the following properties:

- Supporting different ISS's
- Uniform interface
- Minimize code generation

To achieve these requirements, an interprocess communication architecture is provided. More precisely, a wrapper program is designed to glue Ptolemy and the ISS together.

The IPC architecture is shown in Figure 1. The wrapper

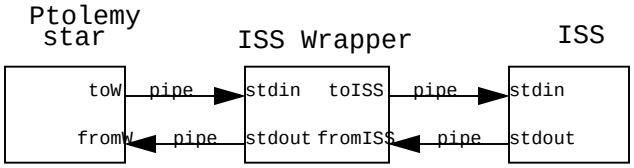

Figure 1. Interprocess Communication Architecture

program acts both as an interface unifier and a command translator. For each kind of ISS, a wrapper is built specifically so that it accepts the predefined ISS commands from Ptolemy, translates them into ISS syntax, and vice versa.

The syntax is defined to be concise enough so that the communication on each pipe is minimized. In other words, the commands adapt the CFSM simulation model to the ISS. The set of commands are listed in Table 1.

Table 1: Ptolemy-ISS Command Stack

Name	Syntax	Description
start	s <module> <?output?>	Initialize ISS, load binary code, set breakpoints at the beginning of the module and the corresponding output event emission point. If <output> is not set, the default end point is the end of the module.
write	w <variable> <value>	set a variable; variables could be CFSM states, event flags, star parameters, input values.
run	r	simulate up to a breakpoint
statistic	z	get the clock cycle statistic
quit	q	terminate the simulator.

The system works as follows:

❒In the setup phase of each star that uses an external ISS, it first checks if the wrapper has been started. If not, it forks a subprocess and executes the wrapper.

❒Each time the star gets fired in Ptolemy, it detects whether the delay needs to be refined (see section 5). If so, it sends the module information together with the variables to the wrapper.

❒After the wrapper has received all the information needed to run the simulator, it translates it into the ISS syntax, and sends it to the ISS.

❒The ISS loads the executable code of the corresponding star, sets the variable values and breakpoints, and executes the code up to the required breakpoint.

❒The wrapper waits until the ISS finishes the requested execution, and gets the clock cycle counting information.

❒The wrapper sends the clock cycle counting result to the Ptolemy star in a pre-defined format.

When this value is returned from the wrapper, the star adds it to the timestamp of the input event and gets the time-stamp of the output event. Then, one communication session is considered finished.

4. Cached Refinement Scheme

It is easy to imagine that, for a long run, if an ISS is used for every firing of a software star, the simulation could be quite slow. In order to speed up the timing refinement process and to keep the advantage of accurate clock cycle counting, we designed a cached refinement scheme, based on the properties of the s-graph and the discrete event semantics.

4.1. S-graph

An s-graph is a model of software execution that is used by the Polis cosimulation. It is a directed acyclic graph (DAG) with one source node called BEGIN and one sink node END. In s-graphs, there are two more types of nodes, ASSIGN and TEST. ASSIGN and BEGIN nodes have only one successor, and TEST nodes have two or more. An expression is associated with each TEST node, and according to the value of the expression (boolean or integer) the TEST node selects one of its children. An execution of an s-graph starts from the BEGIN node, traverses several nodes, until it reaches the END node. The sequence of nodes and edges during one execution forms a path of the execution. From the structure of the s-graph, it is easy to conclude:

Property 4.1: An s-graph can only have a finite number of paths, and for each execution the path is completely determined by the expression values at TEST nodes.

A node is called an EMIT node if its function is to emit an event to an output port of the CFSM. The time delay of emitting an output event is the time cost of the execution from the BEGIN node to the corresponding EMIT node. If we treat the END node as a special kind of event emission, the expressions at TEST nodes together with the final point of the path (EMIT or END node) uniquely specify a path to generate an event.

Generally speaking, the time delay of an event is not equal to the sum of the delays of each node along the path. For example, optimizations of the compiler, caching and pipelining can largely effect the delays. In particular, deep pipelining can overlap the execution of several basic blocks in the program. In addition, program and instruction fetches introduce dependences of the execution time of a path on the sequence of instructions/data fetches. We can thus have a great variance in the delay information for different executions of the path under different input stimuli and internal conditions. However, when we look at an entire path, the delay often varies in a relatively small range. In other words, when considering a medium-large level of granularity, most software modules have execution

delays that are approximately independent of past executions and depend mostly on input data. For this sort of applications, the delay of an execution path can be stored and be used the next time when the same path is traversed. Although this approximation reduces the accuracy, the simulation execution time can be dramatically improved.

4.2. The Ptolemy Discrete Event Domain

The semantics of Ptolemy DE domain[6] is that all signals (events) have two fields, a tag and a value. The tag is the timestamp, which is totally ordered among all the events. A star simply receives events from its the input ports and generates events to its output ports. The process of generating new output events can be divided into two parts: calculating the values and finding the timestamps. These two tasks are independent of each other, and can be done separately. So it is possible to use Ptolemy for behavior simulation and the ISS for timing. Furthermore, in the process of finding out the value of the event, the s-graph is executed from the BEGIN node to the proper EMIT node or END node. If all the predicates on this path are recorded, the internal path information is extracted. This unique internal path can then be used for caching timing information.

4.3. Cached Timing Refinement Scheme

A set of variables `iTrace`, one for each TEST node and termination point, are used to record the expression values on the TEST nodes. A table is created to store the timing information. The table has two fields; one of which, called the *key,* is encoded from `iTrace`; the other, called the *delay,* stores the delay value. During the firing of a star, the behavior model is first executed in Ptolemy. Then, the executed path of the s-graph is stored in `iTrace`. If this particular path has been traversed before, i.e. the same *key* is in the table, the corresponding *delay* is read and used as the delay for this execution. In this process, the ISS is not called. If there is no such *key* in the table, the ISS is called as described in section 3. The returned delay value is used for event delay calculation, and at the same time a new entry is added in the table.

The advantage of using this scheme is that for each internal path, the external ISS is called only once. So the number of Ptolemy-ISS communications is significantly reduced. It is obvious that this scheme is only appropriate for s-graphs with approximately constant delay for each path.

4.4. Stochastic Analysis

Due to the increasing complexity of the processors that are used for embedded applications, it is impossible to ignore the effects of caching and pipelining in the software part both in cosimulation and in static estimation. Moreover, a single execution of each basic block of the program is not sufficient to accurately characterize the block, because it neglects the interaction with previous, subsequent and preempting basic blocks.

Although stochastic measures are considered unacceptable for hard real-time embedded systems, we believe that applying a statistical analysis on the results obtained from linking Ptolemy with the ISS at an early design stage could improve the accuracy of our high-level trade-off evaluation method by automatically stopping the slow cosimulation when the measured variance of the delay in a path goes below a certain threshold. Alternatively, this provides the user with a measure of which module is particularly hard to characterize (due to a high variance of measured delays), in order to decide to continue simulations with the ISS, or use only cached delays.

Sometimes applications such as multi-media use processor architectures with a sophisticated non-standard instruction set. At present, compilers are not able to generate good code for that type of architectures and the user must manually optimize the code. In the Polis approach, this would result in an assembly code subroutine called inside Esterel modules. In this case, the modules cannot be directly simulated in the Ptolemy environment because the code cannot be compiled on a workstation. It is desirable to skip the execution of that function in Ptolemy and update, by using the ISS, not only the delay information but also the arithmetic/logic result of the computation.

Both activities outlined are still at a very preliminary stage, but the results seem quite promising.

5. Implementation

A prototype of this scheme has been implemented by using the Ptolemy and the ST20 Toolset[11]. Two parameters are added for each star to allow users to choose the delay calculation method--that of [10], cached refinement, and uncached refinement. A hash table is used to record and retrieve the path delay. The key of the hash table is encoded from the `iTrace` variables and the ID of an output port.The typical flow of this implementation is shown in Figure2.

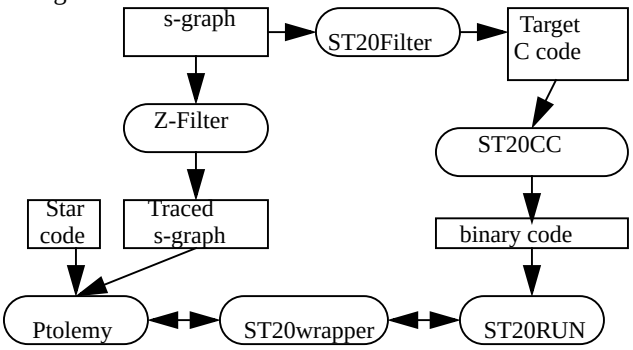

Figure 2. Implementation

Simple benchmarks are tested using these three approaches. COMPARE is a 2-input-2-output module which compares two integers A and B. If A>B, it emits output O1, otherwise it emits O2. ADDER is a floating point adder with one input A, one internal parameter B, and an output SUM. The results of using software performance estimation and the ISS are shown in Table 2 and Table 3 respectively, where `dctA` is the flag of detecting an input A; `A` stands for the value of input A; `output` is the termination point with 0 for END; `estimated` is the clock cycles obtained from the method of [10]; and `measured` is that from the ISS.

COMPARE is a module without function calls, where the compiler optimization make the delay of an entire path to be less than the delay of the sum of each node.

Table 2: COMPARE MODULE

state	dctA	dctB	output	estimated	measured
0	-	-	-	57	51
1	0	0	0	77	63
1	0	1	0	77	59
1	1	0	0	77	49
1	1	1	1	106	56
1	1	1	2	106	58
2	-	0	0	67	49
2	-	1	1	98	56
2	-	1	2	98	58
3	0	-	0	41	66
3	1	-	1	100	61
3	1	-	2	100	63

Table 3: ADDER MODULE

state	dctA	A	B	output	estimated	measured
0	-	-	0.1	0	120	114
1	0	-	-	1	45	50
1	1	0	0	1	887	392
1	1	0.1	0.1	1	887	755
1	1	2.3e12	5.3e8	1	887	790
1	1	0.123	0.987	1	887	855

Table 3 shows a module with two user defined functions; the delay of these functions are obtained from the statistic of sample runs with typical inputs. Also notice that the delay of a same path varies with the input data values.

The experiments are done on a SPARC20 workstation. For a simulation of 100 firings of COMPARE or ADDER, despite the time of starting the ISS (approximately 3 Sec.), the cached timing analysis is about 10 times slower than that of [10], but 1000 times faster than the non-cached exact timing analysis.

6. Conclusion

In this paper, an ISS-based timing refinement scheme is studied in the context of the Polis codesign approach. By using the ISS, some intrinsic problems of software performance estimation are solved. An open architecture is presented to adapt the differences among ISS's. Based on the properties of the s-graph and the DE semantics, a cached timing refinement scheme is studied. The result is that for s-graphs with approximately constant delay paths, the delay of one execution can be stored and reused the next time when the same path is traversed. A prototype has been implemented by using the Ptolemy and the ST20 simulation tools. The result shows that the timing analysis scheme can be seamlessly integrated into the Polis environment.

Acknowledgments

Thanks to Dr. Luciano Lavagno for his precious comments and suggestions during the working of this paper; also to Giovanni Bestente, Fabio Bellifemine, Antonio Bonomo and Giovanni Ghigo of CSELT, Torino, Italy for their help with defining the problem and outlining solutions

References

[1] F. Balarin, M. Chiodo, etc. "Hardware-Software Co-design of Embedded Systems", Kluwer Academic Publishers, 1997

[2] Ptolemy Home Page, "http://ptolemy.eecs.berkeley.edu"

[3] Esterel Home Page, "http://www.inria.tr/meije/esterel"

[4] R. Ernst, W. Ye, "Embedded Program Timing Analysis Based on Path Clustering and architecture classification", Proceedings of ICCAD, Nov. 97, PP 598-60

[5] K. ten Hagen, H. Meyr, "Timed and Untimed Hardware Software Cosimulation: Application and Efficient Implementation", Proceedings of Int. Workshop on Hardware-Software Codesign, Oct. 1993

[6] E. A. Lee and A. Sangiovanni-Vincentelli, "A Denotational Framework for Comparing Models of Computation", ERL Memorandum UCB/ERL M97/11, University of California, Berkeley, CA 94720, January 30, 1997.

[7] S. Leef, "Hardware and Software Co-Verification - Key to Co-Design", Electronic Design, September 15, 97, PP 67-71

[8] C. Passerone, L.Lavagno, etc. "Trade-off Evaluation in Embedded System Design via Co-simulation". Proceedings of ASP-DAC, PP 291-297, 1997.

[9] J. Rowson, "Hardware/Software Co-simulation", Proceedings of DAC, 1994, PP 439-440.

[10] K. Suzuki and A. Sangiovanni-Vincentelli, "Efficient Software Performance Estimation Methods for Hardware/Software Codesign", Proceedings of DAC, 1996, PP605-610.

[11] "ST20 'Osprey' Toolset: User Manual", SGS-THOMSON Electronics, March 1997

[12] S. Yoo, K. Choi, "Synchronization Overhead Reduction in Timed Cosimulation" Proceedings of Int. High Level Design Validation.