COMPILED HW/SW CO-SIMULATION

Vojin Živojnović and Heinrich Meyr

Integrated Systems for Signal Processing
Aachen University of Technology
Templergraben 55, 52056-Aachen, Germany
zivojnov[meyr]@ert.rwth-aachen.de

ABSTRACT

This paper presents a technique for simulating processors and attached hardware using the principle of compiled simulation. Unlike existing, in-house and off-the-shelf hardware/software co-simulators, which use interpretive processor simulation, the proposed technique performs instruction decoding and simulation scheduling at compile time. The technique offers up to three orders of magnitude faster simulation. The high speed allows the user to explore algorithms and hardware/software trade-offs before any hardware implementation. In this paper, the sources of the speedup and the limitations of the technique are analyzed and the realization of the simulation compiler is presented.

I. Introduction

Simultaneous design of hardware and software can take place at different abstraction levels. At the *HLL-level* compiler and processor are designed jointly in order to obtain optimum performance on selected high-level language constructs. At the *application-level* the on- and off-chip hardware have a role of a processing accelerator, or external interface, and are optimized to deliver optimum results for a specific application or a class of them. The goal of *instruction-level* HW/SW co-design is to make frequently used instructions fast by appropriate design of the instruction set architecture of the processor. All three levels correspond to *software-based* HW/SW co-design, where the realization in software is the starting point, and hardware alternatives are introduced in order to speedup execution. Independently of the abstraction level, the co-design cycle has to be closed by intensive verification of hardware and software.

Debugging and verification can be done using hardware or software models, i.e. emulators or simulators, respectively. The main advantage of hardware models, like emulators is their speed, which is mostly only an order of magnitude slower than the speed of the final system. However, emulators are costly, offer low visibility of the internal state of the device, possess low flexibility, deliver inaccurate timing, and the design has to be specially adapted in order to be run on an emulation platform. Also, with emulators the boundary between hardware and software is mostly a priori fixed. This contradicts directly the main philosophy of HW/SW co-design — to take advantage of a flexible boundary between hardware and software, and to position it in an optimum way.

All these drawbacks are easily circumvented using a software model. The price paid is the significantly reduced speed. Although selecting the appropriate simulation accuracy can deliver faster simulation, there are still up to four orders of magnitude difference in speed between emulators and simulators.

In this paper we describe a new technique for HW/SW co-simulation. It relies on the principle of compiled simulation for simulation of both hardware and software. Whereas compiled simulation is a well known approach to hardware simulation, its use for simulation of software is new. All reported HW/SW co-simulation environments rely on the classical interpretive processor simulation technique. We show that compiled simulation is able to deliver bit-true, clock-true simulation of the instruction set architecture of the processor with a speedup of up to three orders of magnitude compared to the classical interpretive technique. The new simulation technique can be applied equally well to verification of *HLL-*, *application-*, or *instruction-level* HW/SW co-designs.

According to Amdahl's law, even a significant speedup in software simulation can be of minor value for HW/SW co-simulation if hardware simulation is the bottleneck. However, if the cycle-based behavioral or RTL model of the hardware is appropriate, the amount of

33rd Design Automation Conference ®

DAC 96 - 06/96 Las Vegas, NV, USA

co-simulated hardware is limited, or if the interaction between software and hardware is localized to specific code fragments or initiated only by events which happen less frequently than the clock edge of the processor, the increased software simulation speed can influence the overall HW/SW co-simulation speed significantly. Our experience shows that in a great deal of embedded systems with DSP functionality both of these conditions are met.

Additional advantage of the compiled approach is comfortable HW/SW debugging with a single source level debugger for hardware and software. If the C code is selected as the intermediate format for software simulation, and the behavioral model of the hardware is written in C, the standard source level debugger of the host can be used to debug hardware and software. Thereby, the HW/SW debugger has all the program-control and state-observation features of standard instruction level debuggers, and at the same time permits cycle-based hardware debugging.

Compiled simulation achieves the high speed by additional compile-time preprocessing which influences the overall turnaround time. The increased preprocessing time is the price which has to be paid for improved runtime performance and presents the main drawback of the technique. However, using incremental compilation only the redesigned parts of the code can be preprocessed and thereby the overall preprocessing time reduced.

The paper is organized as follows. After the introduction in Section II the motivation guiding this work is explained. Section III discusses previous work which is related to those presented in the paper. The principle of compiled simulation for programmable architectures is presented in Section IV. The realization of the simulation compiler for three off-the-shelf DSP processors with different architectures is reported in Section V. Section VI provides a detailed discussion about HW/SW co-simulation and debugging using the compiled technique. Finally, in Section VII the conclusions are given.

II. Motivation

The main motivation for the work presented in this paper was the low speed of the instruction-level simulators found in HW/SW co-simulation environments. The following example arises from the development of the ADPCM G.721 and G.726 speech transcoders for the Digital European Cordless Telecommunications (DECT) and Digital Circuit Multiplication Equipment (DCME).

First, we used an off-the-shelf DSP processor. Offline verification of the hand-written software implementation ($\sim$93 millions instructions) on the standard set of CCITT-ITU test sequences (13 seconds of speech signals) on the target hardware took 7 seconds. The same verification using the instruction set simulator (4K insns/s) provided by the DSP chip vendor took approximately 6.4 hours on an 86 MIPS machine (Sparc-10).

Next, we wanted to explore ways to speedup execution of the transcoder introducing changes in the architecture. It is well known that the FMULT procedure of the G.726 algorithm is the time-critical part of the algorithm. We extended the processor model with a simple hardware accelerator executing the normalization operation of the FMULT procedure. The additional clock-accurate behavioral model of the accelerator had almost no impact on the verification speed. Multiple instructions have been replaced by a single I/O write/read function, so that the simulation speed was decreased only modestly, However, we needed additional 7 hours of simulation to validate correctness and performance of the new design. Experimentation with finite-word length issues could not be done with this simulator.

If the same algorithm is expressed in C, and compiled using the C compiler provided by the chip-vendor, off-line verification of the resulting code ($\sim$750 millions instructions) on the simulator would last for 2 days and 3 hours. Obviously, the turnaround time has to be measured in days and any experimentation with application-oriented compiler and processor adaptations is impossible.

We observed that for the kind of HW/SW co-designs we are interested in, the software simulator is the bottleneck. It is well known that in most cases the clock-accurate model of the attached hardware consumes more simulation time than the simulation of a single clock-cycle of the processor. However, in *software-based* HW/-SW co-designs the interaction with the hardware is mostly localized to specific code fragments of the software. In this case the hardware can be modeled using a less accurate model during periods of no interaction, and a more accurate when the interaction with the software takes place. As a consequence, the overall simulation speed of the hardware is significantly higher than the speed of the software simulator, and the software simulator becomes a limiting factor.

III. Previous Work

Processor simulators such as instruction set simulators are almost always supplied with off-the-shelf or in-house DSP processor. They enable comfortable debugging and verification through controlled program execution and provide visibility of processor resources necessary for code development. All currently available instruction set simulators use the interpretive simulation technique. Their main disadvantages are the low simulation speed (2K-20K insns/s [1]) and their inability to be extended by the user.

Instruction set simulators are standard components of HW/SW co-design environments [2,3]. The speed of these simulators ranges from 300 insns/s to 20K insns/s depending on the character of the processor model, the simulation technique applied or the accuracy level provided.

The compiled simulation technique we use for our simulator is well known in simulation of hardware circuits, e.g. [4]. We follow the same general idea, but apply it to the simulation of the instruction set architecture. Our approach resembles binary translation used for migrating executables from one machine to another [5], or collecting run-time statistics [6]. However, clock/bit-true translation and debugging are not objectives of binary translation.

IV. Compiled Simulation of Programmable Architectures

Interpretive simulators process instructions using a software model of the target processor. A virtual processor is built using a data structure representing the state of the processor, and a program which changes the processor state according to the stimuli — either a new instruction pointed to by the program sequencer or some external events, such as interrupts. In general, interpretive simulators can be summarized as a loop in which instructions are fetched, decoded, and executed using a "big switch" statement, such as the one below:

```
        while(run) {
           next = fetch(PC);
           insn = decode(next);
           switch (insn) {
              ...
add:          exe_add(); break;
              ...
           }
        }
```

Our approach translates each target instruction directly to one or more host instructions. For example, if the following three target instructions

```
add r1,r2;
mov r2,mem(0x175);
mul r2,r3;
```

are interpreted, the above simulation loop iterates once for each instruction. The compiled simulation approach translates the target instructions into the following host instructions, represented here as macros:

```
ADD(_R1,_R2); SAT(_R2); ADJ_FL(_R2); PC();
MOV(_R2,MEM(0x175)); ADJ_FL(); PC();
MUL(_R2,_R3); SAT(_R3); ADJ_FL(_R3); PC();
```

where `SAT()`, `ADJ_FL()`, and `PC()` model the saturation logic, adjustment of the flags, and the change of the program counter, respectively. The translation completely eliminates the fetch and decode steps, and loop overheads of interpretation, resulting in a faster simulation. For target processors with complex instruction encoding, the decode step can account for a significant amount of time. Additional speedup is created because the compiled-simulation generates code tailored to the required accuracy level, while an interpreter provides a fixed level of accuracy. For example, if interrupts are not required, compiled-simulation suppresses the simulation of the interrupt logic already at compile-time, and no run-time penalty is payed.

For large programs, the speed of compiled simulation could be degraded by low locality of reference if the generated simulation code is much larger than the available cache. In this situation, an interpreter would perform better. DSP programs, however, typically exhibit high locality; as a result, the generated simulation program does also. Moreover, the program memory of DSP processors, especially fixed-point ones, is small compared to typical host-machine cache sizes. Our measurements show no difference in simulation speed between small and large DSP programs. However, a detailed analysis still has to be done.

However, compiled-simulation assumes that the code does not change during run-time. Therefore self-modifying programs will force us to use a hybrid interpretive/compiled scheme. Fortunately, self-modifying programs are rare. The isolated cases we encountered so far are limited to programs that change the target address in branch instructions. This type of self-modifying code, however, can be easily handled without interpreting.

The binary-to-binary translation process can be organized in two ways. The direct approach translates target binary to host binary directly (Fig. 1a). It guarantees fast translation and simulation times, but the translator is more complex and less portable between hosts. To simplify the translator and improve its portability, we split the translation process into two parts — compile the target code to a program written in a high-level language such as C (front-end processing), and then compile the program into host code (back-end processing) (Fig. 1b). In this way we take advantage of existing compilers on the host and we reduce the realization of the simulation compiler to building the front-end. Portability is greatly improved but with a possible loss in simulation speed.

Some features of machine code are difficult to represent in a high-level language like C. For example, in the absence of very sophisticated analysis, compiled simulation must assume that every instruction can be a target of an indirect branch statement. Therefore, every

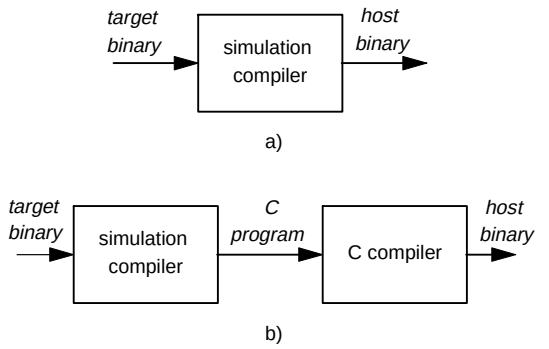

Figure 1: Two Approaches to Binary-To-Binary Translation.

compiled instruction must have a label, and computed `goto` or `switch` statements are used to simulate indirect branching. These labels reduce the effectiveness of many compiler optimizations. If indirect branching is not used in the code, and this is reported to the simulation compiler by an appropriate flag, the generated intermediate code is more amenable to compiler optimizations.

V. Realization of the Simulation Compiler

The simulation environment *SuperSim* SS-21xx has been implemented for the Analog Devices ADSP-21xx family of DSP processors. It consists of the simulation compiler (`ssc`), host C compiler (`gcc`), and C source level debugger (`dbx`). This enables cycle- and bit-true behavioral simulation of the processor in a comfortable debugging environment.

The `ssc` simulation compiler has a form of a two-pass translator with a translation speed of about 1500 target insns/s (Sun-10/64MB). Translating the whole program memory (16 Kinsns) of the ADSP-2105 into intermediate C representation takes less than 11 seconds. To enable additional trade-off between recompilation and execution speed, the simulation compiler can translate target instructions into intermediate C code using macros or function calls.

Compiling the intermediate C code to the host executable takes most of the overall translation time. For the version with function-calls the compilation speed of the `gcc-2.5.8` compiler with optimization `-O1` was about 240 target insns/s (120 target insns/s for -O3). For all 16 Kinsns the compilation with `-O1` takes less than 2 minutes. Using macros the compilation speed slows down almost 5 times compared to the function-call version. In the same time the speedup in execution time is only about 30%. Our current work is concentrating on speeding up the compilation by recompiling only those parts of the target binary which have been changed.

Table 1 presents some real-life examples of SS-21xx performance. Simulation speed measured in insns/s depends on the complexity of instructions found in the target code. The FIR filter example is generated by the C compiler of the target that generates compound instructions rarely. However, the ADPCM example is hand-coded optimally and uses complex compound instructions frequently. The results from Table 1 show that our simulator outperforms the standard simulator by almost three orders of magnitude on the FIR example and by about 200 times on the ADPCM example. The same verification which took 6.4 hours with the standard ADSP-21xx simulator is reduced to less than 2 minutes using *SuperSim*.

The speed improvement we obtained has two main sources. One source is the compile-time decoding and scheduling of the instructions. The other source is that the final simulation program does not include any debugging-related code, but still offers complete debugging support. All the necessary debugging information is inserted by the compiler of the host, and the host-specific debugger. The existing interpretive simulators are designed to support host-independent debugging, and are forced to insert debugging-related operations (e.g. breakpoint checking) at the source level. This introduces an additional, significant slowdown of the simulation.

The ADSP-21xx does not have a visible pipeline. In order to prove our concepts on architectures with pipeline effects, we have written compiled simulation examples for the TI's TMS320C50 and NEC's μPD77016 processors. Despite overhead introduced for pipeline modeling, results from Table 1 show that our approach still achieves significant speedup. Our analysis has shown that the compiled simulation technique fails if indirect delayed branches have to be simulated. In this case the simulator has to switch to the interpretive simulation. More details about compiled simulation of pipelines can be found in [7].

VI. HW/SW Co-Simulation

Designers frequently, during an early stage of the design process, create a software prototype of the design. At this stage, designers can explore implementation options in which some of the functions are shifted into hardware. *SuperSim* supports this exploration because it attaches easily to behavioral models of the hardware. Later, the behavioral models can serve as a starting point in hardware design. Co-simulation becomes useful again, once the behavioral models have been refined into hardware, perhaps rendered using a hardware description language (HDL such as VHDL or Verilog) or as a net list. One can verify such hardware components by attaching either a HDL simulator or a logic simulator to *SuperSim*.

example	simulator	optimization	insns/s	speedup
FIR filter	ADSP-21xx	-	3.9K	1
	SS-21xx	-O3	2.5M	640
	"	-O2	2.0M	510
	"	-O1	1.6M	420
	TI-C50	-	2.4K	1
	SS-C50†	-O3	0.4M	160
	SS-77016†	-O3	0.4M	-
ADPCM	ADSP-21xx	-	4.0K	1
	SS-21xx	-O3	0.8M	200
	"	-O2	0.6M	150
	"	-O1	0.4M	100
host: Sun-10/64MB; SS-21xx flags: -f; compiler: gcc 2.5.8; †-preliminary;				

Table 1: Simulation Examples - Performance Results.

We coupled our compiled simulator to a block-diagram editor, a C library of clock-accurate behavioral models of hardware components, and a C code generator. The resulting HW/SW co-simulation environment is able to deliver fast, clock-accurate simulation.

Figure 2 presents an example of an A/D converter with glue logic attached to a DSP processor. Communication between software and hardware is mediated by cycle hooks. The hooks pass control to the hardware model which is written in C. The hooks also accept data from the hardware models. We can insert different cycle hooks executing different hardware models depending on the type of instruction which is executed in the current cycle, or in the cycles before or after. In this way we are able to control the accuracy of the hardware simulator and thereby the speed. Obviously, the same procedure could be applied to interpretive simulators. However, in the case of compiled simulators the selection can be done already at compile-time, and no run-time overhead for selecting the appropriate hardware model is introduced.

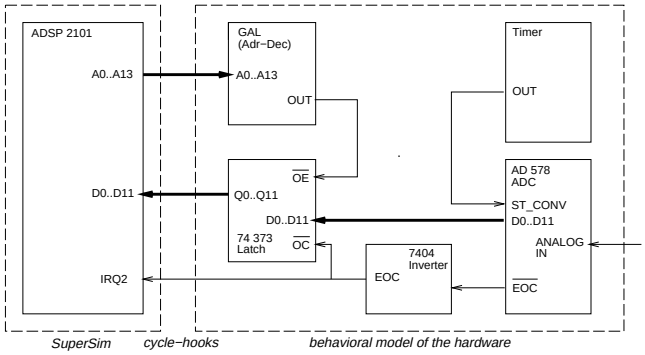

Figure 2: HW/SW Co-Simulation Using SuperSim.

Table 2 presents some simulation results. The example is taken from the front-end of a speech processing device. It consists of an FIR filter executing on a DSP processor, and external acquisition hardware. If the state of the hardware is updated at each clock tick using the same hardware model hook, the resulting speed of the compiled HW/SW co-simulator is 89.0K insns/s. Using the ADSP-21xx interpretive simulator delivering 4.0K insns/s the resulting HW/SW co-simulation speed would be only 3.8K insns/s. Attaching different hardware model hooks to different instruction instantiations, the simulation speed was raised to 1.1M insns/s with the SS-21xx compiled simulator, and to only 4.2K insns/s with the interpretive one.

When the hardware models are written in C, the hooks are simple calls. However, when the models are written in HDL, the hooks are more complicated. They must synchronize *SuperSim* to the HDL simulator and also convert data values before and after communicating with the HDL simulator.

Our simulator offers full debugging support using the standard C level debugger (e.g. `dbx` or `gdb`). It offers breakpoint setting and watching of registers, memory, flags, stack and pins. This is a large advantage compared to standard interpretive debuggers which are highly target dependent. Figure 3 shows an example of the graphical user interface of the `dbxtool` debugger which was adapted to execute C code of the simulator, and in the same time display assembly instructions of the target or the C code of the simulator. As soon as the simulation program reaches the clock-cycle hook, the same debugger which was used for software debugging switches to the code describing the behavioral model of the attached hardware.

Debugging of software and hardware with a standard source-level debugger is one of the main advantages of the compiled technique over the standard interpretive approach. If behavioral models of the hardware are expressed in C, and if the C language is used for the intermediate representation of the software model, compiled simulation seems to be the optimum solution for comfortable debugging of HW/SW co-designs.

simulator	model	insns(cycles)/s
SW only (ADSP-21xx)	interpretive ISA	4.0 K
SW only (SS-21xx)	compiled ISA	2.5 M
HW only	behavioral C	93.0 K
HW/SW (ADSP-21xx)	code-independent HW model	3.8 K
HW/SW (SS-21xx)	"	89.0 K
HW/SW (ADSP-21xx)	code-dependent HW model	4.2 K
HW/SW (SS-21xx)	"	1.1M
host: Sun-10/64MB; SS-21xx flags: -f; compiler: gcc 2.5.8; optimization -O3		

Table 2: HW/SW Co-Simulation - FIR Filter with Acquisition Hardware.

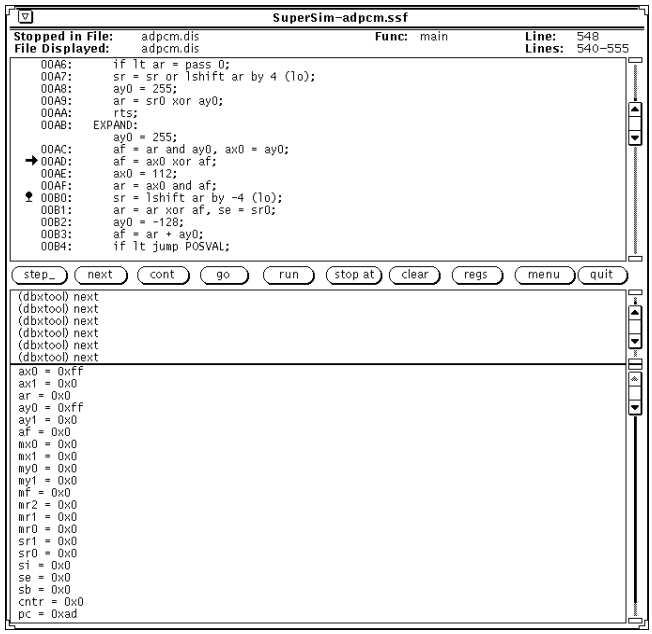

Figure 3: Debugging with *SuperSim*.

VII. Conclusions and Further Research

Compiled simulation provides very fast and accurate instruction set simulation. The presented simulation environment generates bit-, cycle-, and pin-accurate HW/SW co-simulation engines that are two to three orders of magnitude faster than interpretive simulators. Moreover, standard source level debuggers offer a comfortable debugging environment and the intermediate representation in C is open for extensions by the designer. The presented compiled simulator is easily interfaced to behavioral hardware models. In addition to fast simulation, it offers a comfortable debugging environment in which hardware and software are debugged using the same debugger.

Currently, recompilations (with *SuperSim*) after design changes are relatively slow. Though recompilation will always take additional time relative to interpretation, we believe that we can reduce the time by limiting recompilation only to code that has changed. Moreover, a *SuperSim*-interpreter hybrid, in addition to alleviating the problems of indirect delayed branches, can provide fast simulation speed, as well as fast turn-around time on design changes.

We are also investigating two key problem areas in interfacing *SuperSim* to hardware simulators: how accurate do we need to model the processor pin interface. With behavioral models, we have idealized the processor interface to a small set of pins: the data, the address, and interrupt request lines, but not detailed handshaking signals. With more detailed hardware models, however, it may be advantageous to use a detailed processor interface that simulates all pins accurately. We are investigating the attachment of commercially-available processor-interface models to *SuperSim*.

VIII. References

[1] J. Rowson, "Hardware/Software co-simulation," in *31st ACM/IEEE Design Automation Conference*, 1994.

[2] A. Kalavade and E. Lee, "A hardware-software codesign methodology for DSP applications," *IEEE Design & Test of Computers*, pp. 16–28, Sept. 1993.

[3] S. Sutarwala, P. Paulin, and Y. Kumar, "Insulin: An instruction set simulation environment," in *Proc. of CHDL-93, Ottawa, Canada*, pp. 355–362, 1993.

[4] Z. Barzilai, *et al.*, "HSS - A high speed simulator," *IEEE Trans. on CAD*, vol. CAD-6, pp. 601–616, July 1987. 1987.

[5] R. Sites, *et al.*, "Binary translation," *Comm. of the ACM*, vol. 36, pp. 69–81, Feb. 1993.

[6] J. Davidson and D. Whalley, "A design environment for addressing architecture and compiler interactions," *Microprocessors and Microsystems*, vol. 15, pp. 459–472, Nov. 1991.

[7] V. Živojnović, S. Tjiang, and H. Meyr, "Compiled simulation of programmable DSP architectures," in *Proc. of 1995 IEEE Workshop on VLSI in Signal Processing, Osaka, Japan*, Oct. 1995.