Interactive Cosimulation with Partial Evaluation

Patrick Schaumont, Ingrid Verbauwhede
Electrical Engineering Department, University of California at Los Angeles
{schaum,ingrid}@ee.ucla.edu

Abstract

We present a technique to improve the efficiency of hardware-software cosimulation, using design information known at simulator compile-time. The generic term for such optimization is partial evaluation. Our contribution is that we apply the optimization transparantly to the user, and at multiple abstraction levels in the simulation.

We use the technique to create an interactive codesign environment, and evaluate it on several designs including an AES encryption coprocessor and a Viterbi decoder, and for several instruction-set simulators. Compared to SystemC-based cosimulation, we achieve comparable cosimulation performance at only a fraction of the model-build time.

1 Introduction

Because of reasons of energy-efficiency and performance, system-on-chip (SoC) for embedded applications rely on hardware coprocessors. These coprocessors are used for a variety of specialized functions, such as baseband signal processing, sensor data reduction, encryption, and a myriad of peripheral I/O functions. System verification is a major issue because all these coprocessors cooperate to achieve a single goal, for example doing an encrypted videolink on a mobile phone.

The development of such hardware-enabled platforms is a trial-and-error process that requires design exploration and the shifting of functionality from software to hardware and back. We developed an interactive design environment that targets such hardware-enabled SoC platforms. It provides interpreted yet highly efficient simulation of hardware models. We call it *interactive* because it combines simultaneous development of the SoC platform and the application for that platform. In contrast, compiled SoC hardware models require lengthy recompiles for each modification to the SoC platform.

Our design environment, illustrated in Figure 1, combines intruction-set simulation (ISS) with application-specific hardware models writtten in a dedicated language. The hardware models are RT-level accurate, and thus can be readily translated into synthesizable code. We made this design choice because RT-level simulation is still regarded by many as the ultimate golden model of an SoC.

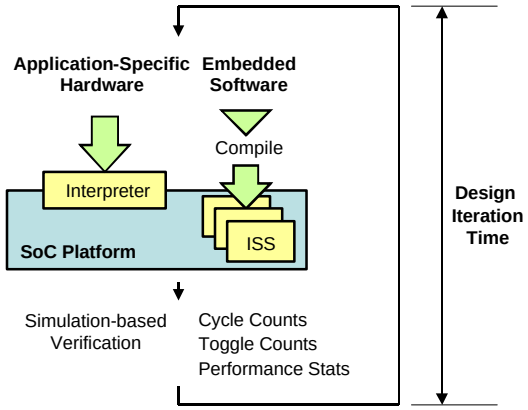

Figure 1: Interactive Codesign Optimization

This SoC cosimulation therefore has conflicting goals: we need high simulation speed, but also cycle-accurate simulation accuracy and minimal model-build-time. In the paper we discuss how our design environment handles this problem. The key is to use knowledge about the properties and structure of the design to optimize the simulator. The general technique that achieves this is called *partial evaluation* [1]. For example, when the wordlength of a particular signal is known to be smaller than 32 bits, then the simulation can use a 32-bit integer instead of an arbitrary-precision datatype. We will show that the ideas of partial evaluation apply to many different aspects of simulation.

In the paper we first discuss cosimulation approaches related to our work. Next, we present our codesign data model and cosimulator architecture. We then consider the optimization strategy for the cosimulator, and illustrate this with design examples. The examples are compared against Verilog modeling as well as SystemC modeling. Finally we draw conclusions and point to future work.

2 Related work

Cosimulation is traditionally done by connecting multiple simulation engines, for example an ISS and a HDL simulator [2]. Contemporary ISS achieve over 1 MHz cycle-accurate simulation performance on a workstation [3], moving the simulation bottleneck to the integration of HW and SW simulation. By using a programming language such as

1530-1591/04 $20.00 (c) 2004 IEEE

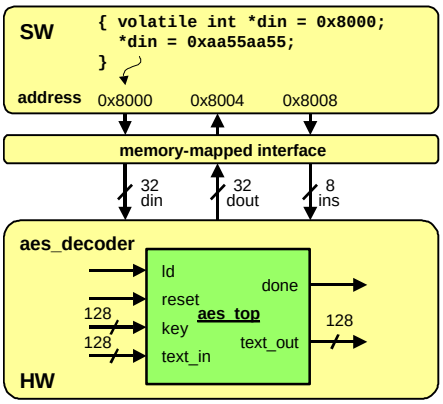

Figure 2: Codesign Data Model Example

SystemC, a tight and efficient coupling between hardware model and ISS can be achieved [4]. The hardware simulation efficiency can be further increased at the expense of simulation accuracy by using abstracted models [5]. These approaches use a compiled programming language for hardware modeling. Our work targets to combine the benefits of a compiled programming language with a design interactivity. We use an interpreted, dedicated language to avoid the compilation overhead, and as will be shown this does not have to imply slow execution speed. In addition, the use of a dedicated language allows to issue feedback and error messages that are directly related to the hardware model. In contrast, with a general-purpose language such as C or C++, one has first to create a correct C(++) program before the semantics of the hardware model can be checked.

Modern SoC platforms increasingly consist of 'soft' hardware [6] in the form of FPGA and other configurable technologies. This makes model-build time an important parameter, and motivates why we want to minimize design iteration time (the loop of Figure 1) instead of simply going for the fastest simulation speed possible. For the latter, very efficient techniques are available [7].

A key insight in our work is that an extra interpretion step allows to do partial evaluation [8] - the use of design properties to specialize the simulator. It can be done transparantly to the designer and can take away some of the design burden.

3 Codesign model

3.1 The GEZEL language

Our codesign model is based on combining cycle-accurate FSMD (finite-state machine + datapath) models for hardware with instruction-set simulation for software. We will illustrate the codesign model in the most simple form, as a single ISS combined with a memory-mapped interface to the hardware.

Figure 2 illustrates the main features of the codesign datamodel by the example of an AES encryption coprocessor. The AES encryption coprocessor is controlled out of C code running on an embedded core. The encryption coprocessor includes an AES IP core (`aes_top`) with 128-bit input/output busses. This core is instantiated inside of a

```
typedef volatile char* vcp;
typedef volatile int*  vip;
vcp ins  = (vcp) 0x80000000;
vip din  = (vip) 0x80000008;
vip dout = (vip) 0x80000004;

enum {ins_idle, ins_load, ins_key};

void load_key(int w0, w1, w2, w3) {
  *din = w0;   *ins = ins_load;   *ins = ins_idle;
  *din = w1;   *ins = ins_load;   *ins = ins_idle;
  *din = w2;   *ins = ins_load;   *ins = ins_idle;
  *din = w3;   *ins = ins_key;    *ins = ins_idle;
}
```

(a)

```
dp aes_decoder(in   ins : ns(8);                    <- Module
               in   din : ns(32);
               out dout : ns(32)) {
  reg key                       : ns(128);
  reg wrkreg0, wrkreg1, wrkreg2 : ns(32);
  reg ir                        : ns(8);
  reg dinreg                    : ns(32);

  use aes_top(rst, ld, sigdone, key, txtin, txtout);

  sfg decode   { insreg = ins;
                 dinreg = din; }
  sfg putword  { wrkreg0 = dinreg;
                 wrkreg1 = wrkreg0;
                 wrkreg2 = wrkreg1; }
  // The '#' operator bit-concatenates
  sfg setkey   { key     = wrkreg2 # wrkreg1 #
                           wrkreg0 # dinreg; }

}

fsm faes_decoder(aes_decoder) {                     <- Module control
  initial s0;
  state s1, s2;
  @s0 (decode)                                          -> s1;
  @s1 if (ir == 1)      then (decode, putword) -> s2;
      else if (ir == 2) then (decode, setkey)  -> s2;
      else (decode)                                     -> s1;
  @s2 if (ir == 0) then (decode)                        -> s1;
                   else (decode)                        -> s2;
}

ipblock b_ins(out data : ns(8)) {                   <- HW/SW Interface
  iptype "armsource"; ipparm "address=0x80000000";
}

ipblock b_datain(out data : ns(32)) {
  iptype "armsource"; ipparm "address=0x80000008";
}

ipblock b_dataout(in data : ns(32)) {
  iptype "armsink"; ipparm "address=0x80000004";
}

system S {
  aes_decoder(ins, din, dout);
  b_ins(ins);
  b_datain(din);
  b_dataout(dout);
}
```

(b)

Figure 3: (a) C driver program for (b) GEZEL description of aes_decoder

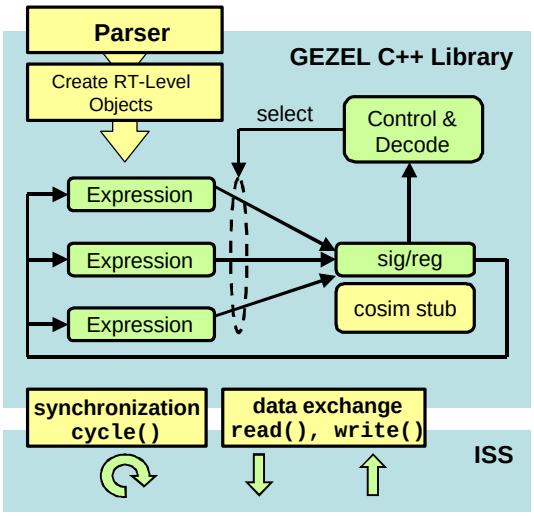

Figure 4: GEZEL (Co)Simulator Architecture

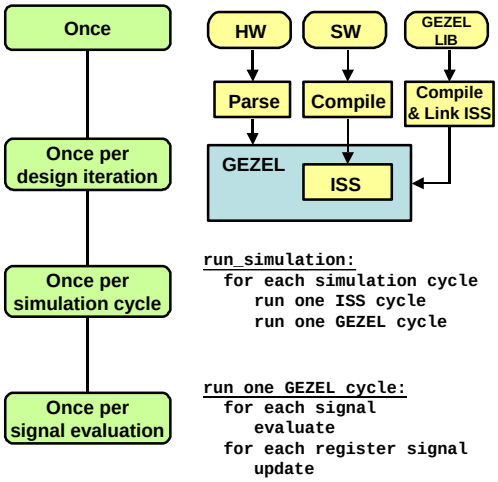

Figure 5: Execution Ladder

decoder module `aes_decoder` that multiplexes the 128-bit I/O data busses of `aes_top` on the 32-bit data connections to `aes_decoder`.

A number of memory addresses on the core have been reserved for communication with the AES coprocessor. In this case, we have reserved two 32-bit data channels (`din`, `dout`), and an 8-bit instruction bus `ins`. By accessing an absolute memory address in C, we will be able to exchange data with the AES hardware. Figure 3a shows a sample C program that provides a new key value as 12 subsequent memory writes.

The AES hardware is expressed in a dedicated language called GEZEL that uses FSMD semantics. Figure 3b illustrates (part of) the GEZEL description of the top-level `aes_decoder`. The code shows how a 128-bit key is assembled out of four subsequent 32-bit data input values. The module `aes_decoder` contains registers, in addition to signal flowgraphs (`sfg`) that contain operations on these registers. Each `sfg` represents one clock cycle of processing. A module can hierarchically include another one by means of the `use` statement. A finite state machine (`fsm`) expresses control as a state transition graph that indicates which sfg will execute each clock cycle. One transition takes one clock cycle. Conditional state transitions can be expressed using boolean conditions on datapath registers. In the example, the `fsm` decodes one of three instructions received through `ins` from the C software. Two of them (`ir==1` and `ir==2`) assemble the key using 32-bit chunks of `din`. A third one is an idle instruction that is used to synchronize the operation of `aes_decoder` to the C software on the ISS. As shown in Figure 3a, a single-sided handshake is created by providing an idle instruction after each active instruction.

The `aes_decoder` module is interfaced to a driver C program by describing the characteristics of each memory-mapped interface. In GEZEL, an `ipblock` is used for such a memory-mapped interface. An `ipblock` can have a custom implementation, both for the purpose of simulation as well as for mapping. Finally, the memory-mapped interfaces are connected to the `aes_decoder` in a `system` block, which is the top-level GEZEL module.

3.2 Cosimulation architecture

The architecture of the cosimulator for this single-processor system is illustrated in Figure 4. GEZEL is organized as a C++ library with a built-in parser. The library is linked against the ISS to create a cosimulation environment. A GEZEL description is parsed and converted into simulation objects. These objects are sets of expressions, extracted out of `sfg` descriptions. The expressions define the values of signals or registers. The control description of each module determines for each clock cycle which expression is valid.

The cosimulation interface consists of two elements: a *synchronization* interface and a *data exchange* interface. The synchronization interface keeps the GEZEL description running in coordination with the ISS. The data exchange interface allows to exchange data from the C software to the GEZEL program. To implement a memory-mapped interface, we intercept memory read/write in the ISS and forward those with a matching address to the GEZEL simulation, where they are available as port values on an `ipblock`.

4 Cosimulator optimization

4.1 The execution ladder

With the codesign model described above, we are now interested in what factors influence the design iteration time. Given a particular design including the testbench, we define the design iteration time as the time it takes to make

Table 1: Overview of Partial Evaluation in optimization of GEZEL simulation

	Input	Program	Output	Partial Evaluation/ Runtime Technique
Once	GEZEL C++ Library	GNU g++	**Compiled GEZEL Lib** linked to ISS	O3 Flag
Once per Design Iteration	GEZEL Program	**Compiled GEZEL Lib**	**RT-Simulator** (C++ Objects)	Static Allocation and Strength Reduction
Once per Cycle	GEZEL Simulator State Module Inputs	**RT-Simulator** Main simulation Loop	GEZEL Simulator State Module Outputs	Cycle-skip Detection
Once per Signal Eval	Expression Inputs Module Control State	RT-Simulator Expression Evaluator	Signal Values Module Control Next-State	Demand Driven Evaluation

a modification to the input description, rerun the cosimulation testbench and evaluate the results that will guide the next modification to the source code.

We rely on the concept of an *execution ladder* to represent design iteration time systematically. As illustrated in Figure 5, the execution ladder organizes tasks per design iteration according to their execution frequency, similar to the concept of nested loops in the execution of a software program. The outer loop of the execution ladder concerns things that are done only once for a design. It includes setting up the ISS/GEZEL cosimulation environment as well as creation of testbenches and the initial version of the code. Next, for each design iteration, a GEZEL design description will be parsed before simulation will start. A simulation itself consists of many clock cycles, therefore clock cycles are the natural next level in the execution ladder. Finally, the evaluation of each clock cycle will include many different signal evaluations. So the signal evaluations form the bottom of the execution ladder.

4.2 Optimization strategy

We will consider each step of the execution ladder separately for minimal design iteration time. At the top two levels of the execution ladder, we use a technique called *partial evaluation* to create the most efficient cycle simulator. At the lower two levels of the execution ladder we also apply *runtime optimization* of the cycle simulation.

A generic definition of partial evaluation is as follows. Given a program P that uses static (constant) input I_s and dynamic input I_d to evaluate an output O. Then a partial evaluation of program P with input I_s will create a specialized program Q. Program Q can create the output O using only dynamic input I_d. With careful design, Q will also be faster than P because it needs to consider less input data. The idea of partial evaluation is found in many optimizations in design automation, for example in strength reduction with software compilation or redundancy removal in hardware compilation.

Table 1 illustrates the optimizations that were done at each level of the execution ladder. For each level, the input, output and evaluation program is shown. For the upper two levels, the *output* is a *program* by itself on a lower level - this is what makes partial evaluation possible. At lower levels, we rely on runtime-optimization techniques. We briefly discuss the optimizations.

(a) At the *once* level, the GEZEL C++ library is compiled with full compiler optimization (GCC O3 level) and linked to an ISS.

(b) For each *design iteration*, a GEZEL program is parsed in by the GEZEL library, and transformed into an RT simulator. The optimizations include static allocation of intermediate expression results, and strength reduction of selected operations. Static allocation of intermediate results enables arguments of operations to be accessed by reference instead of by value.

(c) For each *cycle*, we apply a runtime optimization we define as *cycle-skip detection*. It consists of skipping simulation of a clock cycle altogether if it can be shown that the simulator state at the input and output of this step are identical. The conditions for skipping a cycle are: (1) no register has changed state in the previous clock cycle, (2) no controller has changed state in the previous clock cycle, (3) no HW/SW interface *ipblock* has changed state. Skipping cycles is very useful to increase HW/SW cosimulation efficiency, since they allow to 'wake-up' the hardware simulation out of the ISS only when it is needed.

(d) For each signal evaluation, the RT simulator determines the expression inputs using the module control state. The simulator evaluates signals for each module from the outputs to the inputs, in a demand-driven fashion. We also ensure that each signal is evaluated only once during each clock cycle. This is done by tagging signals with the clock cycle time of their last evaluation. Demand driven techniques were originally proposed for event driven simulation [9], but are effective for cycle simulation as well.

The next section will present design results obtained with the optimized GEZEL cosimulator. Before that, we consider the relative contribution of partial evaluation to the overall result. Figure 6 shows the relative design iteration time, averaged over a number of testcases. Optimization at

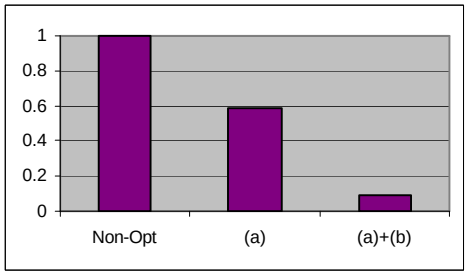

Figure 6: Relative Design IterationTtime with Partial Evaluation.

the 'once' (a) and 'once per design iteration' (b) level results in one order of magnitude improvement. This is on top of the runtime optimizations (c) and (d).

5 Results

To evaluate the efficiency of our simulator, we performed two sets of experiments. The first are stand-alone hardware simulations, the second are cosimulations. We compare with two existing simulation environments: SystemC 2.0.1 and Verilog-XL 2.8. SystemC was selected because it can be easily used for cosimulation purposes. Verilog-XL was selected because we started from Verilog reference code. All code developed for the examples is available on the World Wide Web [10].

Table 2: Non-comment, non-blank Line Count for design examples

	AES	Viterbi
Verilog	522	426
RTL SystemC	506	374
GEZEL	312	265

We started from two open-source Verilog designs. The first is an AES128 encryption processor [11], while the second is a (2,1,2) Viterbi decoder [12]. Both were translated into SystemC 2.0.1 and GEZEL. During translation into SystemC, care was taken to optimize for execution speed, using the most efficicient data types and minimizing the amount of signals. However we did not abstract the execution model into a bus functional model (a model with a cycle-accurate interface and functional-level internal behavior). Rather, the guidelines for synthesizable SystemC RTL code were followed [13]. As a result, each design performs identically on a cycle-by-cycle basis in each of the three environments.

In Table 2, we compare the non-comment, non-blank linecount for each of the examples. There are several reasons for the lower linecount in GEZEL. It is a purely synchronous modeling approach with an implicit clock. Also, GEZEL does not require (nor support) module declarations. And third, there is support for specific hardware constructs like lookup tables.

5.1 Standalone simulation

We compare the design iteration time for each design. Table 3 lists the results for a 20K cycle testbench for AES and a 100K cycle testbench for Viterbi. Since we are interested in design iteration time, we list the parse/compile time as well as the simulation time. For SystemC, we use the O3 flag to compile for performance. The evaluation platform is a SUN Ultra-10 (500 MHz CPU, 2GB RAM) with gcc 3.2.2.

Table 3: Design iteration time for stand-alone (HW-only) simulation of examples

	AES 20K cycles		Viterbi 100K cycles	
(seconds)	Build	Simulate	Build	Simulate
Verilog	0.3	15	0.2	46
RTL SystemC	85	21	56	15
RTL GEZEL	1	13	0.1	22

Platform: SUN Ultra-10 500 MHz, 2GB RAM with gcc 3.2.2

The model-build-time for SystemC is considerably slower, because general C++ compilation is far more complex than the use of a dedicated parser.

The testbench of the AES design consists of about 1600 subsequent encryptions. This simulation is known to have a high event density because a good encryption algorithm toggles on the average half of the bits it processes. In this case, the cycle algorithm of GEZEL performs very well. For the Viterbi simulation, we observe the reverse situation. In this case, half of the cycles are idle cycles without any events. The reason why the Verilog version is slower is that it uses a two-phase clock, which is translated to a single-edge clock in SystemC and GEZEL.

5.2 Cosimulation

We next evaluated the design iteration time for a cosimulation using a StrongArm instruction set simulator (SimIt-ARM 1.1b [3]). The evaluation platform is a Pentium PC (3GHz CPU, 512 MB RAM) with gcc 3.2.2. Our testcase is a loop in software that performs 100 encryptions using an AES coprocessor. For each encryption, a new key and plainttext (2 * 128 bits of data) are transferred between software and hardware. We wrote a cycle-accurate model (RTL) and a bus-functional model (BFM) of the AES encryption processor in GEZEL and SystemC, and collected build-time and simulation-time in Table 4. In the BFM, a C function is used to simulate the AES core `aes_top` from Figure 2.

The embedded software is in all cases compiled with O3-level optimization. A cycle-accurate simulation on the ISS by itself runs at 1 million cycles per second. This implementation takes 785K cycles to complete. When using a hardware model for the AES, the total amount of cycles to

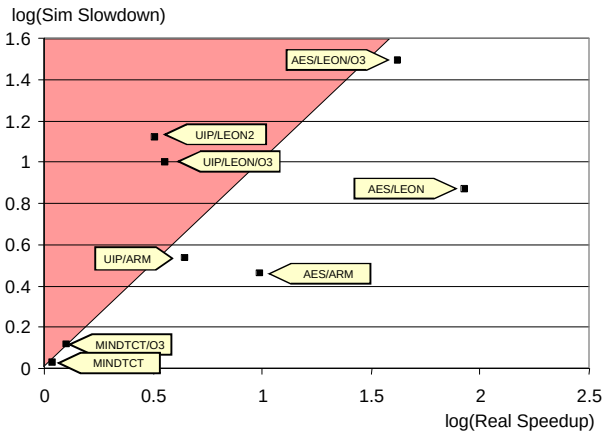

Figure 7: Codesign as a tradeoff between simulation speed and actual performance

simulate drops to about 70K because of the increased parallelism.

Table 4: Simulation for SW-only, HW/SW cosimulation with a bus-functional model, and HW/SW cosimulation with RT-level Models

	Build + Simulate (seconds)	**Speed** (cycles per second)
ISS SW-only (AES in SW)	0.14 + 0.78	1M
ISS + BFM SystemC	7.0 + 0.23	318K
ISS + BFM Gezel	1.8 + 0.72	101K
ISS + RTL SystemC	20.5 + 9.0	8.1K
ISS + RTL Gezel	0.11 + 4.0	17.7K

Platform: PC 550 MHz, 256MB RAM with gcc 2.96

The model build-time figures in Table 4 are clearly faster for GEZEL-based cosimulation. As indicated before, an encryption algorithm is rich in events, therefore a SystemC BFM model will much run faster than the event-driven SystemC RTL model. For GEZEL, the skip-cycle mechanism can omit a large number of clock cycles. This, combined with the cycle-simulation algorithm makes the GEZEL RTL model faster than that of SystemC. However, the GEZEL BFM does not outperform the SystemC BFM. This is because the cycle simulation algorithm will evaluate the Rijndael function regardless whether the inputs have changed or not.

In another series of experiments, we compare a number of software-only designs with their hardware-accelerated counterparts in Figure 7. The designs are a fingerprint verification system with a DFT accelerator (MINDTCT), an embedded TCP/IP stack and webserver with IP checksum acceleration (UIP), and the AES coprocessor system described earlier.

The codesigns are based on either StrongArm with SimIt-ARM.1.1b as ISS, or on LEON-2 Sparc with TSIM 1.2 as ISS (`http://www.gaisler.com`). Each point in Figure 7 compares the accelerated design with the non-accelerated, software-only design. For designs in the non-shaded area, hardware accelerated designs cosimulate faster than their original, software-only counterparts, because the improvement in cycle count they offer exceeds the slowdown in simulation speed. Obviously a designer working in the nonshaded area will be more motivated to explore alternatives and try to improve even more. The designs in Figure 7 were not done by the authors but by actual users of GEZEL.

6 Conclusions

We have demonstrated an interactive approach to efficient cosimulation. Compared to existing methods, we have shown that comparable performance can be achieved while at the same time minimizing the design iteration time. We also obtain very compact code size. We will extend our approach to multicore systems and create an automatic path to implementation for GEZEL descriptions.

Acknowledgements

This work has been made possible by NSF (Grant CCR-0310527) and SRC (Grant 2003-HJ-1116).

References

[1] N. Jones et al, *Partial Evaluation and Automatic Program Generation*, Prentice Hall International, June 1993.

[2] V. Zivojnovic et al., *Compiled HW/SW Cosimulation*, Proc. 33th DAC, Las Vegas, 1996.

[3] W. Qin et al., *Flexible and Formal Modeling of Microprocessors with Application to Retargetable Simulation*, Proc. DATE 03, March 2003, Munchen.

[4] L. Benini et al., *SystemC Cosimulation and Emulation of Multiprocessor SoC Designs*, IEEE Computer, April 2003, pp. 53-59.

[5] L. Semeria et al., *Methodology for Hardware/Software Coverification in C/C++*, Proc. ASP-DAC 2000, Jan. 2000.

[6] F. Vahid, *The softening of hardware*, IEEE Computer, April 2003.

[7] C. DeVane, *Efficient Circuit Partitioning to Extend Cycle Simulation beyond Synchronous Circuits*, Proc. ICCAD 97, San Francisco, 1997.

[8] W. Au, *Automatic Generation of Compiled Simulations through Program Specialization*, Proc. 28th DAC, June 1991, San Francisco.

[9] S. Smith, *Demand Driven Simulation: BACKSIM*, Proc. 24th DAC, Miami Beach, 1987.

[10] http://www.ee.ucla.edu/~schaum/gezel/date04

[11] R. Usselman, Open Cores AES Core, http://www.opencores.org/projects/aes_core/

[12] V. Stojanovic et al., Baby Viterbi Decoder, http://mos.stanford.edu/ee272/proj99/babyviterbi/index.html

[13] Synopsys, *Describing Synthesizable RTL in SystemC*, v 1.2, November 2002.