An interactive codesign environment for domain-specific coprocessors

PATRICK SCHAUMONT AND DORIS CHING
University of California at Los Angeles
and
INGRID VERBAUWHEDE
University of California at Los Angeles, and Katholieke Universiteit Leuven

---

Energy-efficient embedded systems rely on domain-specific coprocessors for dedicated tasks such as baseband processing, video coding, or encryption. We present a language and design environment called GEZEL that can be used for the design, verification and implementation such coprocessor-based systems.
The GEZEL environment creates a platform simulator by combining a hardware simulation kernel with one or more instruction-set simulators. The hardware part of the platform is programmed in GEZEL, a deterministic, cycle-true and implementation-oriented hardware description language. GEZEL designs are scripted, allowing the hardware configuration of the platform simulator to be changed quickly without going through lengthy recompiles. For this reason we call the environment interactive. We present the execution ladder as an optimization framework to balance interactivity against simulation speed.
We demonstrate our approach using several designs including an AES encryption coprocessor and a Viterbi decoding coprocessor. We discuss the advantages of our approach as opposed to more conventional approaches using SystemC and Verilog/VHDL.

Categories and Subject Descriptors: B.5.2 [Register-transfer Level Implementation Design Aids] Hardware Description Languages, C.3 [Special-purpose and Application-based Systems] Embedded Systems
General Terms: Design
Additional Key Words and Phrases: Cosimulation, Hardware Description Language, Hardware-software codesign.

This research was supported by NSF (Grant CCR-0310527) and SRC (Grant 2003-HJ-1116)..
Authors' addresses: Electrical Engineering Department, University of California at Los Angeles, CA 90095-1594 USA (e-mail: {schaum,dorisc,ingrid}@ee.ucla.edu), and Electrical Engineering Department, Katholieke Universiteit Leuven, B-3001, Belgium.

1. INTRODUCTION

For reasons of energy-efficiency, modern embedded systems use specialized and distributed processing components. For example, a contemporary mobile phone contains multiple processing units for signal processing and control, specialized baseband signal-processing hardware, along with a number of hardware acceleration units for selected application domains including graphics and cryptography. Newer generations of such embedded systems tend to increase the number of functions they support. As a result, they require an increasing number of specialized processing units to maintain the same level of energy-efficiency. Those units must be designed, validated and integrated under a

shrinking design time schedule and design cost budget. This makes a split hardware/software design path and the use of non-programmable hardware less suited.

We present an interactive codesign environment called GEZEL that targets such hardware-accelerated multiprocessor System-on-Chip (SoC) platforms. The platform is modeled in terms of custom hardware components as well as instruction-set simulators (ISS). We call our approach interactive because it allows quick modification of the simulation models of the platform hardware. Thus, the SoC platform can be modified easily during development of the software that runs on the SoC.

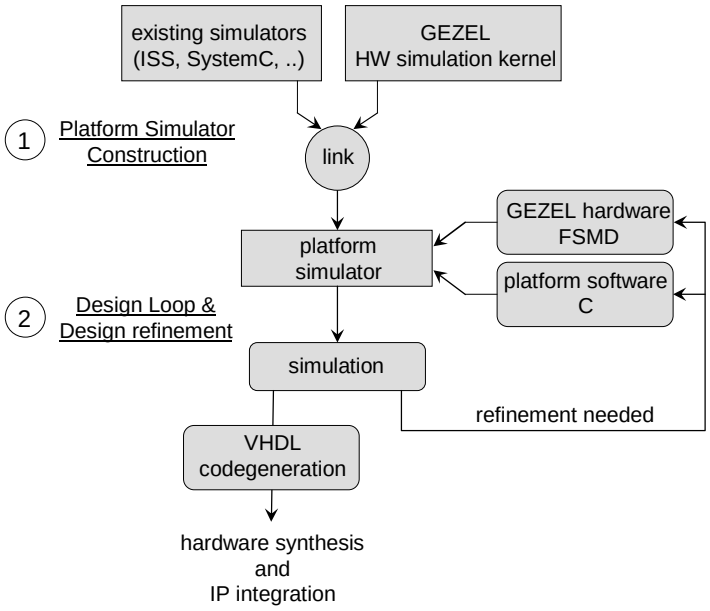


Figure 1: The GEZEL design flow.

1.1 The GEZEL design flow

Figure 1 shows the typical design flow followed using GEZEL. Two design phases can be identified during system-level design. In the first phase, a platform simulator is created by combining the GEZEL hardware simulation kernel with one or more instruction-set simulators (ISS). In the second phase, the platform simulator is configured with a description of the custom platform hardware as well as the embedded software running on the platform. In subsequent design iterations, changes to the platform hardware or software do not require a complete rebuild of the simulation platform, but

just reconfigure it. In this paper, we will provide a description of this reconfiguration process.

Once an adequate hardware design for the platform is created, GEZEL also provides a path to implementation by code-generation of synthesizable VHDL. This VHDL can be targeted to reconfigurable hardware (FPGA) or standard-cells using register-transfer-level (RT-level) synthesis tools.

The GEZEL hardware simulation kernel can model and simulate various components of an SoC, including (co)processor micro-architectures as well as networks-on-chip. The components are expressed in the GEZEL language, which captures cycle-true models in the finite-state-machine-with-datapath (FSMD) model-of-computation.

The GEZEL hardware simulation kernel is implemented as a scripting engine for models in the GEZEL language. It will parse GEZEL models, convert these models into executable C++ objects, and initiate simulation without going through a compilation phase. The GEZEL kernel is written in C++ and can be linked easily to various cosimulation environments in order to obtain an SoC platform simulator. The GEZEL environment is available as an open-source package from the World-Wide-Web [GEZEL Homepage 2004].

1.2 Paper Outline

In this paper, we will put emphasis on the description — and simulation aspects of the GEZEL design flow. In Section 2, we review the GEZEL hardware description language and explain the major differences with conventional hardware description languages. We will also explain our hardware-software codesign model. In Section 3, we consider the GEZEL cosimulation strategy in more detail, and clarify the advantages of a scripting approach to simulator construction. In Section 4, we discuss several experiments that compare our approach to a more conventional approach that uses SystemC. In Section 5, we discuss related work and we conclude the paper in Section 6.

2. THE GEZEL LANGUAGE

In this section, we review the features of the GEZEL modeling language and compare it with existing hardware description languages. The language is presented once-over-lightly, by means of an example. For a more formal treatment of the modeling

characteristics, we would like to refer the reader to the GEZEL Language Reference Manual [GEZEL Homepage 2004].

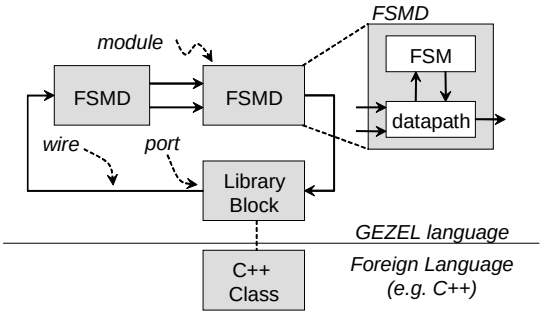


Figure 2: Elements of the GEZEL language.

2.1 An up-and-down counter in GEZEL

Figure 2 shows the composing elements of the GEZEL language. A design model in GEZEL is a network of custom hardware blocks modeled as FSMD, and library blocks. An FSMD is a combination of a finite-state machine controller with a datapath, expressed using the GEZEL language. A library block is a black box with an interface defined in GEZEL and a behavior modeled in C++.

Listing 1 shows an example of the cycle-true FSMD model of an up-and-down counter in GEZEL. It counts from 0 to 3 and then back to 0.

Listing 1. An up-and-down counter in GEZEL.

```
 1. dp updown(out a : ns(3)) {
 2.    reg c : ns(7);
 3.    sfg up { c = c + 1; a = c; }
 4.    sfg dn { c = c - 1; a = c; }
 5. }
 6. fsm fsm_updown(updown) {
 7.   initial s0;
 8.   state   s1;
 9.   @s0 if (c < 3) then (up) -> s0;
10.       else (dn)            -> s1;
11.   @s1 if (c > 0) then (dn) -> s1;
12.       else (up)            -> s0;
13. }
```

The example will be helpful as we enumerate the elements of the GEZEL language.

- **Variables and Data Types**: There are two kinds of variables in GEZEL programs: registers and signals. Each of those variables can represent an unsigned or a two's-complement signed number of selectable precision. The example creates a register `c` with a 7-bit unsigned type (line 2). The ports on a datapath, such as the 3-bit unsigned output `a` (line 1), are signals.
- **Expressions**: Expressions, such as on line 3 and 4, are formed using operators on registers and signals. Almost all operators from the C programming language are supported, and a few new ones such as for bit-selection and bit-concatenation are added.
- **Datapath Instructions**: Expressions are grouped together into datapath instructions to represent a single clock cycle of register-transfer behavior. The data-path in Listing 1 has two instructions called `up` and `dn` (lines 3 and 4). These instructions (also called signal flowgraph or `sfg`) represent a single clock-cycle of behavior using expressions. All expressions within a signal flowgraph execute concurrently.
- **Datapaths**: Several datapath instructions can be grouped together to form a datapath. A datapath also defines an interface with inputs and outputs (lines 1—5). A datapath can include as many signal flowgraphs as needed. At any particular clock cycle an arbitrary combination of signal flowgraphs can execute under direction of a controller attached to the datapath.
- **Finite State Machine Controllers**: An FSM controller defines a schedule for datapath instructions in a datapath (lines 6—13). It defines an initial state (line 7), other states (line 8), and state transitions (line 9—12). State transitions can be conditionally dependent on the value of registers in a datapath. Instructions selected by the controller each correspond to the execution of one or more signal flowgraphs in the datapath.
- **Library Blocks**: Library blocks are prebuilt datapaths, with a behavior that is defined within the GEZEL kernel. Library blocks are used to model hardware-software interfaces, RAM blocks, intellectual-property user models (IP), and so on.
- **Hierarchy and Instantiation**: GEZEL handles complexity in a similar manner as most other hardware description languages, using hierarchy and datapath instantiation.

2.2 Comparing GEZEL to existing hardware description languages

The GEZEL language is a cycle-true, deterministic, and implementation-oriented hardware description language. Most existing hardware description languages on the other hand are event-driven, non-deterministic and simulation-oriented. GEZEL also makes explicit distinction between modeling of data and control. We will clarify these properties and point out the differences with other hardware description languages.

GEZEL is a cycle-true hardware description language

GEZEL does not have clock or reset signals. The clock- and reset-behavior is implicit to the design description. At the start of the simulation, a GEZEL design is initialized by bringing all FSM descriptions in a known initial state. After that the simulation advances at the upgoing edge of each clock cycle. In existing HDL on the other hand, the clock and reset signals are explicit.

The hardware implementation of registers and wires is directly visible from the GEZEL source code. A GEZEL register will always translate to a synchronously-clocked flip-flop and a GEZEL signal will always translate to a wire. In contrast, a shared `variable` in VHDL, or a `reg` in Verilog, may or may not translate to a register depending on the way it is used. This can lead to subtle but annoying mistakes, such as the introduction of latches instead of flip-flops.

GEZEL is a deterministic hardware description language

A GEZEL program has deterministic behavior. This means that, for a given set of stimuli, the simulation outcome of that program will always be equal. The only way to introduce non-determinism would be to use a library block that is known to be non-deterministic. The FSMD modeling in GEZEL itself is always deterministic.

This does not mean that a user may never want to write a non-deterministic program. Indeed, some applications such as random-number generation may want to use non-deterministic simulation. But the problem with most hardware-oriented languages (including Verilog, VHDL and SystemC) is that they do not tell the user if the program is deterministic or not. The non-determinism in traditional HDL originates from mixing the concepts of shared variables and concurrency. A reg in Verilog can have global visibility, and that variable may be updated by multiple concurrent modules. Such concurrent

updates result in race conditions, for which the actual outcome can be simulator-dependent.

GEZEL avoids the non-determinism described above by verifying that registers and variables are assigned only once per clock cycle. In addition, GEZEL ensures that all signals that are used as expression operands, are also produced within the same clock cycle. While a detailed description of the deterministic aspects of GEZEL lies outside the scope of this paper, the property has several useful consequences. A GEZEL program will not generate unknown ('U') or undetermined ('X') values. As one of the main goals of GEZEL is cosimulation with software, it makes sense to use a uniform abstraction level for data values between hardware and software. Another property is that a GEZEL program is free from race conditions. Note that GEZEL does not prevent non-determinism if a user would require it. In that case, the non-deterministic part must be included in a library block.

GEZEL is an implementation-oriented language

In GEZEL, the logic is structured around instructions of a datapath. Each of these instructions represents a clock cycle of behavior. In HDL on the other hand, logic is structured around processes. When a synthesizable result is required, designers often rely on a systematic two-process modeling style, with one process for combinational logic, and a second process for sequential logic. Such a modeling style is obviously redundant, yet it is recommended by synthesis tool vendors [Xilinx, 2004] as well as designers [Gaisler, 2004]. GEZEL programs correspond to this two-process HDL style by definition, and are therefore easier to keep consistent.

GEZEL separates control modeling from data modeling

GEZEL models separate between control and data processing by means of the FSMD model. In traditional HDL languages, this separation is not explicit. Often a state machine is encoded in HDL by means of a case statement, tightly mixing data processing with control processing. The problem with the case-statement approach for modeling of control is that it is deceptive. It gives the impression of being simple and straightforward, but in fact it is not. In one experiment, we translated a VHDL model of an independently published state-machine [Edwards, 2004] by hand into GEZEL. The resulting GEZEL program is less than half the size of the VHDL program (31 lines of GEZEL against 75

lines of VHDL). Separate modeling of control and data-processing in GEZEL results in more compact and easier-to-understand code.

A summary of the differences between GEZEL and other hardware description languages is listed in Table I. The GEZEL language is focused to modeling of synchronous digital systems, but we believe that it covers an adequate range of design cases to justify a dedicated language. Some examples of published GEZEL designs are enumerated next.

- A coprocessor IC for AES cryptography and biometric processing, implemented using side-channel leakage free CMOS technology [Tiri, 2005].
- A coprocessor for the Advanced Encryption Standard (AES [NIST, 2001]), attached to the SH-3 processor from Renesas and executed from within embedded Java [Matsuoka 2004].
- A network-on chip architecture consisting of one-dimensional and two-dimensional routers that cosimulate with multiple ARM processors [Ching, 2004].
- A microcontroller called MIC-1, and used as a design lab in an undergraduate course on hardware-software codesign [Madsen, 2002].
- A coprocessor for the Discrete Fourier Transform (DFT), attached to the LEON-2 processor and used in a fingerprint authentication application [Yang, 2003].

Table I. Comparative feature list in GEZEL

	GEZEL	**Verilog**	**SystemC**
Model of Computation	cycle-true	event-driven	event-driven
Modeling Unit	FSMD	HDL process	HDL process
Deterministic Model	yes	no	no
Language	dedicated	dedicated	general-purpose
New Lang. Primitives	yes (lib. blocks)	no	yes (classes)
Simulation	scripted	scripted/compiled	compiled
Implementation-oriented	yes	yes	no
Application	platform implementation	hardware design	system modeling
Cosimulation interfaces	user-defined; library blocks	prog. lang. interface (PLI)	C++

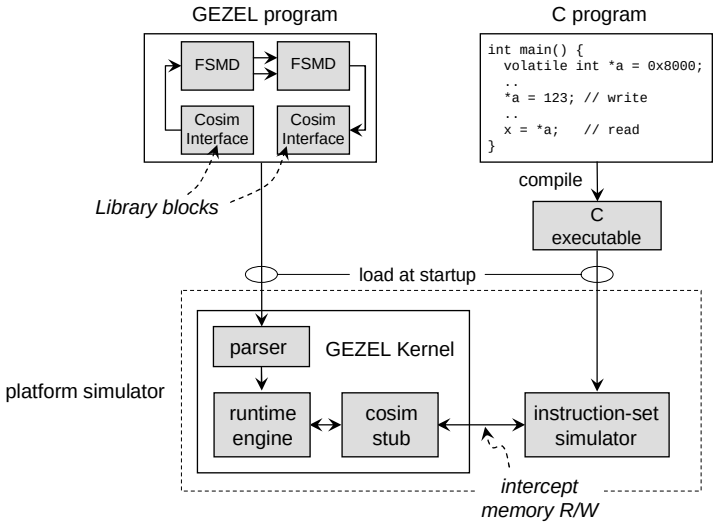


Figure 3: The GEZEL Codesign Model

2.3 Codesign Model

As shown in Figure 3, our codesign model is based on combining cycle-accurate FSMD models for hardware with instruction-set simulation for software. We have developed memory-mapped interfaces for several different instruction-set simulators. At the language level, a memory-mapped interface is supported by a library block in GEZEL, and by initialized pointers in C.

At the start of the simulation, the platform simulator loads the C executable into the instruction-set simulator, and parses the GEZEL program using the GEZEL kernel. The runtime engine of the GEZEL kernel however is not an interpreter of the GEZEL program. Instead, the GEZEL program is converted into a series of C++ objects that directly implement the behavior of the hardware.

During the simulation, the instruction-set simulator and the GEZEL kernel run in lockstep: for each simulation cycle of the ISS, there is one simulation cycle of the GEZEL hardware. However, the simulation works equally well with derived clock rates - for example with an ARM that runs at five times the frequency of the GEZEL hardware. Data communication between GEZEL and C is implemented using memory-mapped interfaces. Memory write- and read-operations on the ISS are intercepted and their address is matched against the address decoded by the GEZEL library blocks. If a match is found, a value is transferred from the GEZEL program to the C program or vice versa.

A designer uses these memory interfaces to attach and interface a coprocessor to the program running on the core. The design of such interfaces is adequately discussed in literature [De Micheli, 2001][Rowen, 2004].

We have created several cosimulators for various purposes, as listed in Table II. All of them use a scheme similar to that in Figure 3.

Table II. Cosimulators using GEZEL

Simulator	**Configuration GEZEL + ...**	**Kernel added to GEZEL**	**Codesign Interfaces** [1]	**Applications**
armcosim	Single ARM	SimIt-ARM [Qin, 2003]	MemMapped, CPMapped	Teaching
armzilla	Multiple ARM	SimIt-ARM	MemMapped, CPMapped	NoC research [Ching 2004]
gezelsh	SH3-Mobile	SH-ISS (Renesas)	MemMapped	Secure Java [Matsuoka 2004]
fdl_tsim	LEON2	tsim (www.gaisler.com)	MemMapped	ThumbPod [Tiri 2005]
gezel51	8051	Dalton ISS [Vahid 2001]	PortMapped	Sensor-Network research
libgzlsysc.a	SystemC	SystemC (www.systemc.org)	PortMapped	Legacy code integration

[1] MemMapped = Shared memory locations; CPMapped = Using coprocessor interface; PortMapped = Using dedicated ports.

3. COSIMULATOR IMPLEMENTATION

With the codesign model described above, we are now interested in obtaining an optimized cosimulation. We present the execution ladder, a framework to formulate this optimization. Two optimization strategies are described: partial evaluation and runtime optimization. The discussion will rely on the following definitions:

- **Model build-time**: The time it takes to create an executable simulation model out of source code for that model.
- **Design iteration-time**: The time it takes, given a fixed testbench, to create an executable simulation model out of source code for that model, and then execute the testbench. Design iteration-time is thus is the sum of the model build-time and the simulation execution time.

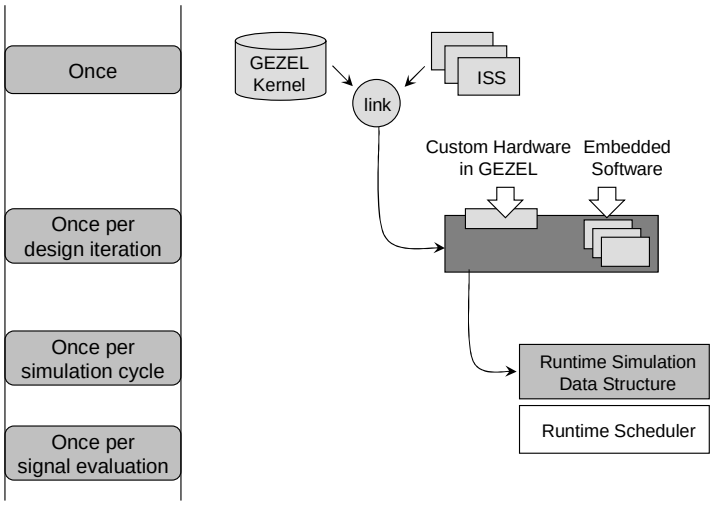


Figure 4: The Execution Ladder

3.1 The Execution Ladder

The execution ladder, first published as [Schaumont, 2004a], is a framework to organize the optimizations that we will consider. At the heart of the execution ladder sits the idea that some tasks in a design are done more frequently than others. For example, a simulator is created a single time (once), but it is then used to simulate millions of clock cycles. When we optimize the design iteration-time, we should try to optimize the most frequently executed portions first, but we should not ignore the overhead introduced at parts that are executed less often. In terms of the example, this means that we should optimize the time it takes to simulate a single clock cycle, but we should not ignore the time it takes to create the simulator in the first place. Indeed we will show that C++-based simulators can take a long time to compile, and that this compilation time can overshadow the execution time.

As illustrated in Figure 4, the execution ladder organizes tasks per design iteration according to their execution frequency. The top-level of the execution ladder concerns the activities that are done only once for a design. It includes the setup of the ISS/GEZEL cosimulation environment as well as creation of testbenches and the initial version of the code. The next level concerns activities that are done per design iteration. A GEZEL design description will be parsed before the simulation starts. A simulation itself consists of many clock cycles, therefore clock cycles are the 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.

3.2 Overall 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 an efficient cycle simulator. At the lower two levels of the execution ladder we also apply runtime optimization of the cycle simulation.

Table III illustrates for each level of the execution ladder: the input, output and evaluation program. For the upper two levels, the output is a program by itself on a lower level - this is what makes partial evaluation possible. In the next sections we discuss the optimizations at the individual levels.

Table III. Partial Evaluation and Runtime Optimization of the Execution Ladder.

Level	**Input**	**Program**	**Output**
Once	GEZEL C++ Library	GNU g++	Compiled GEZEL + ISS
Once per Design Iteration	GEZEL Program	Compiled GEZEL + ISS	RT-Simulator (C++ Objects)
Once per Clock Cycle	Simulator State FSMD Inputs FSM State	RT-Simulator Simulation Loop	Simulator State FSMD Outputs FSM Next-State
Once per Signal Evaluation	Expression Inputs	RT-Simulator Eval Loop	Signal Values

3.3 Partial Evaluation

First, consider a generic definition of partial evaluation. Given a program P that uses a static (constant) input Is and a dynamic input Id to evaluate an output O, then a partial evaluation of program P with input Is will create a specialized program Q. Program Q can create the output O using only dynamic input Id. 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, as illustrated by the following examples.

- Strength reduction with software compilation. Expressions using loop counters may be simplified based on the knowledge of the static loop increment value [Muchnick 1997].

- Add-shift expansion of hardware multiplication with constant values [Pasko, 1999].
- Fixed-point refinement in Digital Signal Processing, which relies on knowledge of the limited dynamic range of input signals [Kim, 1998].
- Redundancy removal in hardware compilation, which relies in part on the propagation of constants into gates [De Micheli, 1994].

Partial evaluation translates as follows to the case of GEZEL. At the upper level of the execution ladder, a platform simulator is created. This is done by compiling the GEZEL C++ library, and by linking it to an instruction-set simulator. At the next level of the execution ladder, this simulator will read a GEZEL description and one or more embedded software binaries and will create a runtime simulation architecture. We therefore identify two opportunities for partial evaluation: one while creating the platform executable, and one while creating the runtime simulation architecture. The optimization during creation of the platform simulator is provided by the C++ compiler, and consists of well-known optimizing compiler techniques.

The second optimization step concerns translation of a program written in the GEZEL language into C++ objects. First, the parsing process itself can be optimized, such as by using hash tables. This minimizes the overhead of symbol table management. In addition, when GEZEL language is translated into C++ objects we can create a C++ object structure that is application-specific.

Procedural, Optimized Operators

The C++ runtime architecture works with custom data types to represent arbitrary-wordlength bit vectors. It is common practice to implement operations on these data types using custom C++ operators, because it results in clear and easy-to-maintain source code. However, the use of such operators introduces extra temporaries. For a statement such as

```
my_custom_type a,b,c;
b = a + (c >> 5);
```

the C++ compiler will create two intermediate results - one to hold the result of the shift operation, and one to hold the result of the addition before it is assigned to b. These temporary objects are created and destroyed for each evaluation of the expression. Note that a C++ compiler will not optimize these temporary objects away, because they are not native machine types. By using procedural versions of the operators, we obtain control

over allocation of temporary objects and can select an optimal version of each operation. For example, the expression above can be written as

```
my_custom_type a, b, c, tmp;
constant_shift_right(tmp, c, 5);
add(tmp,a,tmp);
assign(b,tmp);
```

This code uses only a single temporary as well as a specialized version of the shift operator. While it can be tedious to write for a C++ designer, it is easy to create these objects out of GEZEL code. Thus, a data type that uses operators (looks ‘nice’) in GEZEL, can have an efficient procedural implementation in C++. In addition, GEZEL data types are converted into C++ objects during parsing. Operator optimizations such as the selection of the constant-shift operator are done before the simulation starts. Without the partial evaluation process, we would need to do these tests at runtime.

Static allocation of intermediate expression results:

The previous step can be taken further by controlling the allocation of all intermediate expression results explicitly. In GEZEL, we use a simple static allocation of all intermediate expression results.

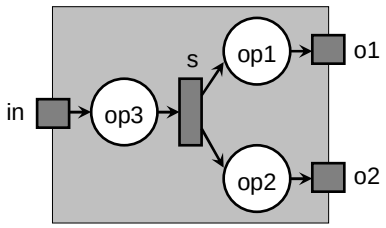
3.4 Runtime Optimization

The bottom two levels of the execution ladder are located at the level of the runtime simulation infrastructure, and therefore must be handled with runtime optimization techniques.

Cycle-skip Detection:

With this mechanism, we attempt to skip simulation of a clock cycle altogether if it can be shown that the simulator state will not change in the next clock cycle. 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 hardware/software (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. Indeed, an ISS typically is much faster then a general hardware simulator.




```
init:
c = 0; /* current cycle count */
s.n = o1.n = o2.n = 0;

simulate_sfg at cycle c:

1. eval(o1,c)
 1.1 eval(s,c)
 1.2 (s.n != c) => s=op3(in);
                   s.n = c;
 1.3 o1 = op1(s);
       o1.n = c;
2. eval(o2, c)
 2.1 eval(s,c)
 2.2 o2 = op2(s);
       o2.n = c
```

Figure 5: Demand-driven evaluation of cycle-true simulations. ‘n’ is a signal attribute that holds the clock cycle of the most recent signal update, and is called the generation of the signal.

Demand-driven Signal Evaluation:

The simulator evaluates signals for each module in a demand-driven fashion, working from the outputs to the inputs. 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 [Smith, 1987], but are effective for cycle simulation as well. Figure 5 shows an example of demand-driven evaluation in the context of cycle-true simulation. Each signal has, besides a value, also a generation. The generation indicates at which cycle the value of a particular signal is valid, and is updated when a signal is reassigned. A simple comparison of the generation of a signal with the current cycle time allows deciding if we can use the current value of the signal, or rather if we should check the expression that defines the signal. As illustrated by Figure 5, when we first evaluate output `o1`, we need to evaluate all expressions leading to the new signal value. However, when we evaluate output `o2`, we conclude that the intermediate signal value `s` is already current. Demand-driven evaluation guarantees that each operation is only evaluated once for each clock cycle.

Simulator Caches:

A third optimization technique relies on the use of simulation-specific caching tables. For example, in a GEZEL FSMD, the expression that defines a signal is dependent on the control step of the FSM. This control step selects a set of `sfg`, and each sfg selects a group of expressions. This is a double indirection that can be avoided by means of a hashing table per signal. The table is indexed by the control step and returns the expression defining this signal. Such a hashing table is filled up at runtime.

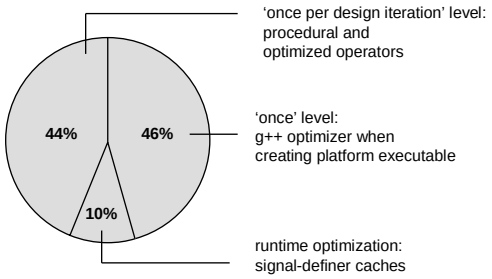


Figure 6: Relative contribution of each optimization.

Finally we illustrate the relative contribution of all these optimizations. Overall, we found that with all optimizations mentioned above turned on, the execution time for a GEZEL stand-alone simulation improves on the average by one order of magnitude. We analyzed two samples designs in detail: an encryption unit and a Viterbi decoder. Both are described in the next section. For these designs, the order-of-magnitude in improvement is divided over the different levels of the execution ladder as illustrated by Figure 6.

4. RESULTS

Using the optimized GEZEL simulator and cosimulators, we now present two sets of results. First we compare stand-alone GEZEL designs to equivalent Verilog and SystemC designs. Next, we compare the design iteration-time of GEZEL to that of SystemC for an AES coprocessor design.

4.1 Standalone Simulation

To evaluate the efficiency of our simulator, we performed two kinds 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 [GEZEL Homepage 2004].

Table IV. Non-comment, non-blank line count (NCLOC) for design exanples.

	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 [Usselman 2003], while the second is a (2,1,2) Viterbi decoder [Stojanovic 1999]. 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 efficient 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 [Synopsys 2002]. As a result, each design performs identically on a cycle-by-cycle basis in each of the three environments. The resulting design sizes are illustrated in Table IV and show that GEZEL allows for compact hardware descriptions.

Table V. Design-iteration time for stand-alone (HW-only) simulation of examples.

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

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

We next compare the design iteration-time for each design. Table V 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. The model build-time for SystemC is considerably slower, because general C++ compilation is far more complex than the use of a dedicated scripting engine. 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.

4.2 Cosimulation – Design Iteration Time

Next we considered cosimulation. We first took the AES coprocessor design and evaluated the design iteration-time in more detail. We made use of the StrongArm instruction set simulator (SimIt-ARM 1.1b) in combination with the AES coprocessor. 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 5. In the BFM, a C function is used to simulate the AES core.

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

	Build + Simulate (seconds)	**Simulation 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

Simulation Platform: PC 3 GHz, 512MB RAM with gcc 3.2

In all cases, the embedded software is 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 simulate drops to about 70K because of the hardware acceleration that is provided by the coprocessor.

The model build-time figures in Table 5 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 AES function regardless whether the inputs have changed or not.

5. RELATED WORK

Cosimulation is traditionally done by connecting multiple simulation engines, for example an ISS and a HDL simulator [Zivojnovic, 1996]. Contemporary ISS achieve over 1 MHz cycle-accurate simulation performance on a workstation [Qin, 2003], moving the simulation bottleneck to the integration of HW and SW simulation. By using a programming language such as SystemC, a tight and efficient coupling between the hardware model and the ISS can be achieved. The hardware simulation efficiency can be further increased at the expense of simulation accuracy by using abstracted models [Semeria, 2000]. Such abstraction can apply to the hardware models, but also to the cosimulation interfaces [Fummi, 2004]. All of these approaches use a compiled programming language for hardware modeling. Our work targets to combine the benefits of a compiled programming language with those of an interactive design environment. We use an interpreted, dedicated language to avoid the compilation overhead, but also make sure to optimize the simulation 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.

Many coprocessor design systems today are constructed as an ASIP synthesis system. In such a system, the instruction-set of a standard processor is extended or specialized to fit a dedicated task [Hoffmann 2001][Cong 2004]. The appeal of this approach is that a single environment can create the target architecture, as well as a design tool suite (compiler and simulator) to map and verify applications for this architecture. Our approach does not rely on extending instruction-sets, but on explicit description and integration of the coprocessor micro-architecture. This allows for loosely coupled coprocessors that do no fit the template of an instruction-set, for example with memory-mapped coprocessors. In general, loosely-coupled architectures can offer better energy efficiencies than tightly-coupled ones [Schaumont, 2004b].

Modern SoC platforms increasingly consist of 'soft' hardware in the form of FPGA and other configurable technologies [Vahid, 2003]. This makes model build-time an important parameter, and motivates why we want to minimize design iteration-time instead of simply going for the fastest simulation speed possible. For the latter, very efficient techniques are available [DeVane, 1997].

A key insight in our work is that an extra interpreting step allows to do partial evaluation - the use of design properties to specialize the simulator [Au, 1991]. It can be done transparently to the designer and can take away some of the design burden. A related approach that allows for fast simulation in combination with minimal model build-time is just-in-time translation (JIT). This technique has been successfully applied to performance improvement of embedded software execution as well as instruction-set simulation [Nohl 2002]. The just-in-time translation step creates a native implementation of an instruction that can be reused later in the simulation, and thus avoids repeated interpreting of that instruction. Thus, some of the simulation work is moved from an inner simulation loop to an outer one. We are not aware of any cosimulation systems that use JIT-like techniques for the hardware part.

6. CONCLUSIONS

We have demonstrated an interactive design environment for domain-specific coprocessors, called GEZEL. Using a dedicated hardware modeling language and a general-purpose cosimulation interface, various types of cosimulators can be easily created. Compared to existing cosimulation methods, we have shown that comparable performance can be achieved while at the same time minimizing the design iteration-time - hence the use of the term interactive. We also obtain compact code size. Our results

show that we can efficiently support a wide range of coprocessors, starting from tightly-coupled designs up to very loosely-coupled ones.

REFERENCES

AU, W., 1991. “Automatic Generation of Compiled Simulations through Program Specialization,” In Proceedings of the 28th Design Automation Conference, ACM Press, June 1991, San Francisco, CA, 205—210.

CHING, D., SCHAUMONT, P., VERBAUWHEDE, I., 2004. "Integrated modelling and generation of a reconfigurable network-on-chip," In Proceedings of the 18th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2004), April 2004, 139.

CONG, J., FAN, Y., HAN, G., ZHANG, Z, 2004. “Application-Specific Instruction Generation for Configurable Processor Architectures.” In Twelfth International Symposium on Field Programmable Gate Arrays, 2004, 183—189.

DEVANE, C., 1997. “Efficient Circuit Partitioning to Extend Cycle Simulation beyond Synchronous Circuits,” In Proceedings of the International Conference on Computer-Aided Design, IEEE Computer Society Press, San Francisco, CA, 154—161.

DE MICHELI, G., 1994. “Synthesis and Optimization of Digital Circuits,” McGraw-Hill Science and Engineering, 1994.

DE MICHELI, G., ERNST, R., WOLF, W., 2001. “Readings in Hardware/Software Codesign.,” The Morgan Kaufmann Systems On Silicon Series, Elsevier, Norwell, MA, 2001.

EDWARDS, S., 2004. “Design and Verification languages,” Columbia University CS Technical Report CUCS-046-04.

FUMMI, F., MARTINI, S., PERBELLINI, G., PONCINO, M., 2004. “Native ISS-SystemC Integration for the Co-Simulation of Multi-Processor SoC,” In Proceedings of the 2004 Design Automation and Test in Europe Conference, February 2004, Paris, France, 464—469.

GAILSER, 2004. “A structured VHDL design method,” online copy at <http://www.estec.esa.nl/microelectronics/vhdl/vhdlpage.html>.

GEZEL HOMEPAGE, 2004. <http://www.ee.ucla.edu/~schaum/gezel>

HOFFMANN, A, KOGEL, T., NOHL, A., BRAUN, G., SCHLIEBUSCH, O., WAHLEN, O., WIEFERINK, A., MEYR, H., 2001. “A novel methodology for the design of application-specific instruction-set processors (ASIPs) using a machine description language,” In IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Nov. 2001, 20(11) : 1338—1354.

JONES, N.D., GOMARD, C.K., SESTOFT, P., 1993. “Partial Evaluation and Automatic Program Generation,” Prentice Hall International, June 1993, xii + 415 pages. ISBN 0-13-020249-5.

KIM, S., KUM, K., SUNG, W., 1998. “Fixed-point optimization utility for C and C++ based digital signal processing programs”, IEEE Trans. on Circuits and Systems II, November 1998, 45(11):1455—1464.

MADSEN, J., STEENSGAARD-MADSEN, J., CHRISTENSEN, L., 2002. “A Sophomore Course in Codesign,” Computer, Nov. 2002, 108—110.

MATSUOKA, Y., SCHAUMONT, P., TIRI, K., VERBAUWHEDE, I, 2004. "Java cryptography on KVM and its performance and security optimization using HW/SW co-design techniques," in Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2004), September 2004, 303—311.

MUCHNICK, S., 1997. “Advanced Compiler Design and Implementation,” Morgan Kaufmann Publishers, 1997.

NOHL, A, BRAUN, G., HOFFMANN, A., SCHLIEBUSCH, O., MEYR, H., LEUPERS, R., 2002. “A Universal Technique for Fast and Flexible Instruction-Set Architecture Simulation,” In Proceedings of the 39th Design Automation Conference, June 2002, New Orleans, Louisiana, 22—27.

NIST, 2001. “Specification for the Advanced Encryption Standard,” Federal Information Processing Standards publication 197, November 2001. online copy at <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>

PASKO, R., SCHAUMONT, P., DERUDDER, V., VERNALDE, S., DURACKOVA, D., 1999. “A new algorithm for elimination of common sub-expressions,” IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, January 1999, 18(1):58—68.

QIN, W., MALIK, S., 2003. “Flexible and Formal Modeling of Microprocessors with Application to Retargetable Simulation,” in Proceedings of the 2003 Design Automation and Test in Europe, March 2003, Munchen, Germany, 765—769.

ROWEN, C., 2004. “Engineering the Complex SoC,” Prentice Hall Modern Semiconductor Series, Upper Saddle River, NJ, 20004.

SCHAUMONT, P., VERBAUWHEDE, I., 2004A. “Interactive cosimulation using partial evaluation,” In Proceedings of the 2004 Design Automation and Test in Europe Conference, February 2004, Paris, France, 642—647.

SCHAUMONT, P., SAKIYAMA, K., HODJAT, A., VERBAUWHEDE, I., 2004B. “Embedded Software Integration of Coarse Grain Reconfigurable Architectures,” In Proceedings of the 11th Reconfigurable Architectures Workshop, April 2004, Santa Fe, NM, 137.

SEMERIA, L., GHOSH, A., 2000. “Methodology for Hardware/Software Co-verification in C/C++,” in Proceedings of the 2000 Asia and South Pacific Design Automation Conference, Yokohama, Japan, 405—408.
SMITH, S., 1987. “Demand Driven Simulation: BACKSIM,” In Proceedings of the 24th Design Automation Conference, ACM Press, June 1987, Miami Beach, FL.
STOJANOVIC, V., KETAKI, R., 1999. “Baby Viterbi Decoder,” <http://mos.stanford.edu/ee272/proj99/babyviterbi/>
SUTHERLAND, 2002. “The Verilog PLI Handbook: A Tutorial and Reference Manual on the Verilog Programming Language Interface,” Springer, Norwell MA, 2002.
SYNOPSYS, 2002. “Describing Synthesizable RTL in SystemC,” v 1.2, Synopsys Inc, November 2002.
TIRI, K., HWANG, D., HODJAT, A., LAI, B.C., YANG, S., SCHAUMONT, P., VERBAUWHEDE, I., 2005. “A Side-Channel Leakage Free Co-processor IC in .18um CMOS for Embedded AES-Based Cryptographic and Biometric Processing,” In Proceedings of the 42th Design Automation Conference, ACM Press, June 2005, Anaheim, CA.
USSELMAN, R., 2003. “Open Cores AES Core,” <http://www.opencores.org/projects/aes_core/>
VAHID, F., 2003. “The softening of hardware,” in IEEE Computer, IEEE Computer Society Press, April 2003, 27—34.
VAHID, F., GIVARGIS, T., 2001. “Platform Tuning for Embedded Systems Design,” IEEE Computer, March 2001, 34(3):112—114.
XILINX, 2004. “Synthesis and Simulation guide,” online copy at <http://toolbox.xilinx.com/docsan/2_1i/data/common/sim/sim4_4.htm>.
YANG, S., SAKIYAMA, K., VERBAUWHEDE, I., 2003. "A compact and efficient fingerprint verification system for secure embedded systems," Proc. 37th IEEE Asilomar Conference on Signals, Systems, and Computers, November 2003, 405—408.
ZIVOJNOVIC, V., MEYR, H., 1996. “Compiled Hardware-Software Cosimulation,” in Proceedings of the 38th Design Automation Conference, ACM Press, Las Vegas, CA, 127—136.