SystemC Cosimulation and Emulation of Multiprocessor SoC Designs

The SystemC simulation framework provides a generic design environment for multiprocessor architectures that enables transparent integration of instruction-set simulators and prototyping boards. Experimental results demonstrate significant speedup in both cosimulation and emulation.

Luca Benini
Davide Bertozzi
Davide Bruni
Università di Bologna

Nicola Drago
Franco Fummi
Massimo Poncino
Università di Verona

Developers usually use processor-based templates to build today's system-on-chip (SoC) designs, which contain one or more cores with considerable on-chip memory and complex communication buses. Because on-chip processor cores are often legacy or third-party components, designers need correct functional models to accurately track the core's interaction with the rest of the system.

Embedded software designers routinely use cross-development toolkits containing a cross-compiler and instruction-set simulator (ISS) to validate functionality and assess application performance. Hardware designers use Hardware Description Language (HDL) simulators to validate their work, but these simulators model the microarchitecture in too much detail to efficiently simulate complex processor cores.

Thus, exploring and validating a complex SoC design requires a single, integrated hardware-software cosimulation platform. Academic groups as well as electronic design automation vendors have developed numerous such platforms.[1-4] Traditional cosimulation design environments use multilanguage system descriptions—HDL for hardware and C or similar languages for software—to construct an efficient link between event-driven hardware simulators to cycle-based ISSs.

More recently, using C/C++ for hardware descriptions and design flows has gained popularity because using the same language for describing software and hardware can potentially bridge the gap between hardware and software description languages.[5-7] Using the same language also makes it possible to simulate the entire system within a single engine.

SYSTEM DESIGN ENVIRONMENT

SystemC is an open source C/C++ simulation environment that provides several class packages for specifying hardware blocks and communication channels (www.systemc.org/).[6] The design environment specifies software algorithmically as a set of functions embedded in abstract modules that communicate with one another and with hardware components via abstract communication channels. Using a SystemC description of the core models software consistently with the rest of the system. However, a more detailed simulation is required because specifying software at this level of abstraction does not permit synchronization with the hardware.

One option for modeling software execution consistently with the rest of the system is microarchitectural-level (MAL) simulation, a cycle-accurate method that details the core's microarchitecture and is faster than register-transfer level (RTL) simulation. Another option is to embed the ISSs within the cosimulation environment to simulate the core at a higher abstraction level.

0018-9162/03/$17.00 © 2003 IEEE

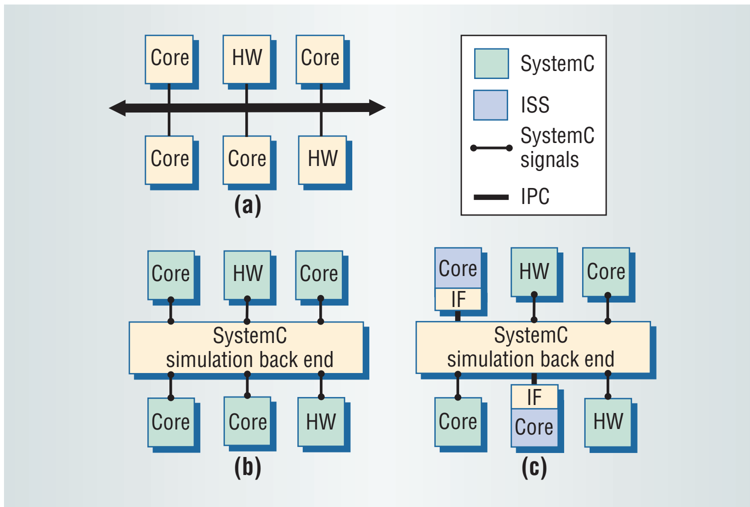

Figure 1. Architectural template and simulation alternatives. (a) System functionalities are partitioned over a set of cores and other custom hardware (HW) blocks. (b) Full SystemC simulation implements all blocks as modules. (c) Instruction-set simulator (ISS)/SystemC cosimulation utilizes interprocess communication (IPC) as well as interface (IF) modules to make granularity consistent.

Most previously described approaches[7-9] use interprocess communication (IPC) and a bus wrapper: The ISS and the C/C++ cosimulator run as distinct processes on the host system and communicate via IPC primitives; the bus wrapper ensures synchronization between the system simulation and the ISS and translates the information coming from the ISS into cycle-accurate bus transactions.

These approaches have two main limitations. First, the interface between the bus wrappers and the ISS typically is proprietary. This complicates the integration of new cores within the cosimulation framework because the ISS requires modification to support the IPC communication primitives that the cosimulation system defines. Second, the IPC paradigm is only effective when communication between the ISS and the rest of the system is infrequent. This occurs when the ISS model also includes a significant amount of local memory so that communication with the rest of the system is required only occasionally—for example, as read/writes on memory-wrapped I/O.

To address these problems, we developed an implementation of the IPC interface between wrappers and the ISS based on the GNU project's open-source-level debugger, GDB. Any ISS that can communicate with GDB can also become part of the cosimulation environment. To address the IPC performance bottleneck, we also developed library functions to be called from within the top module of a legacy GNU ISS, which is embedded as a process within the SystemC simulator. This fully embeds the ISS in the system simulator executable and completely eliminates the IPCs.

SIMULATION INFRASTRUCTURE

Our SystemC simulation environment targets heterogeneous multiprocessor systems.[6] As Figure 1a shows, we partition system functionalities over a set of cores and other custom hardware blocks—in this case, four cores and two hardware blocks. "Core" here represents a generic programmable resource for which some type of executable model is available.

Given our simulation infrastructure, one intuitive strategy is to implement all blocks as SystemC modules, as Figure 1b shows. The abstraction level of the SystemC descriptions may range from functional, in which the core is basically a software model, to RTL, in which all modules are cycle-accurate descriptions. Both have inadequate accuracy levels. Cores are usually too complex to be modeled at cycle-accurate granularity. Further, such modeling is not always feasible, as in the case of proprietary cores. On the other hand, a purely functional model is of little help.

Instruction-set simulators offer an acceptable level of granularity for a core simulation. ISSs can provide detailed functional information, such as register values, as well as program execution times and other timing information. Figure 1c depicts a cosimulation scheme that employs an ISS in place of some of the hardware descriptions.

The use of an ISS cannot be transparent from the software architecture point of view. First, the ISS must communicate somehow with the SystemC simulation back end; this usually occurs through the host operating system's IPC primitives. Second, a proper interface module is required to make granularity consistent. If the underlying SystemC simulation is cycle-accurate, the interface modules translate ISS events into cycle-accurate events.

Our cosimulation methodology is based on this HDL/ISS mixed-level simulation paradigm, which most existing cosimulation schemes[7-9] have adopted. These approaches implement the conceptual scheme of Figure 1c by relying on IPC and instantiation of bus wrappers; the ISS and the cosimulator run as distinct processes on the host system and communicate via IPC primitives.

Our approach is similar to that proposed by Luc Séméria and Abhijit Ghosh.[7] Their use of SystemC eliminates the simulator's need for an explicit distinction between the bus wrapper and the ISS. As Figure 2a shows, integrating wrappers as SystemC objects makes it possible to use an IPC interface between the wrappers and the ISS rather than only between the wrappers.

Our implementation relies on GDB's remote debugging features.[10] Compliance with this nonproprietary interface is the only constraint for inclusion of an ISS in the cosimulation environment. This permits cosimulation when an actual processor replaces the ISS, as in the emulation scheme. GDB's remote debugging interface provides a serial or TCP/IP interface for use with particular debugging targets.

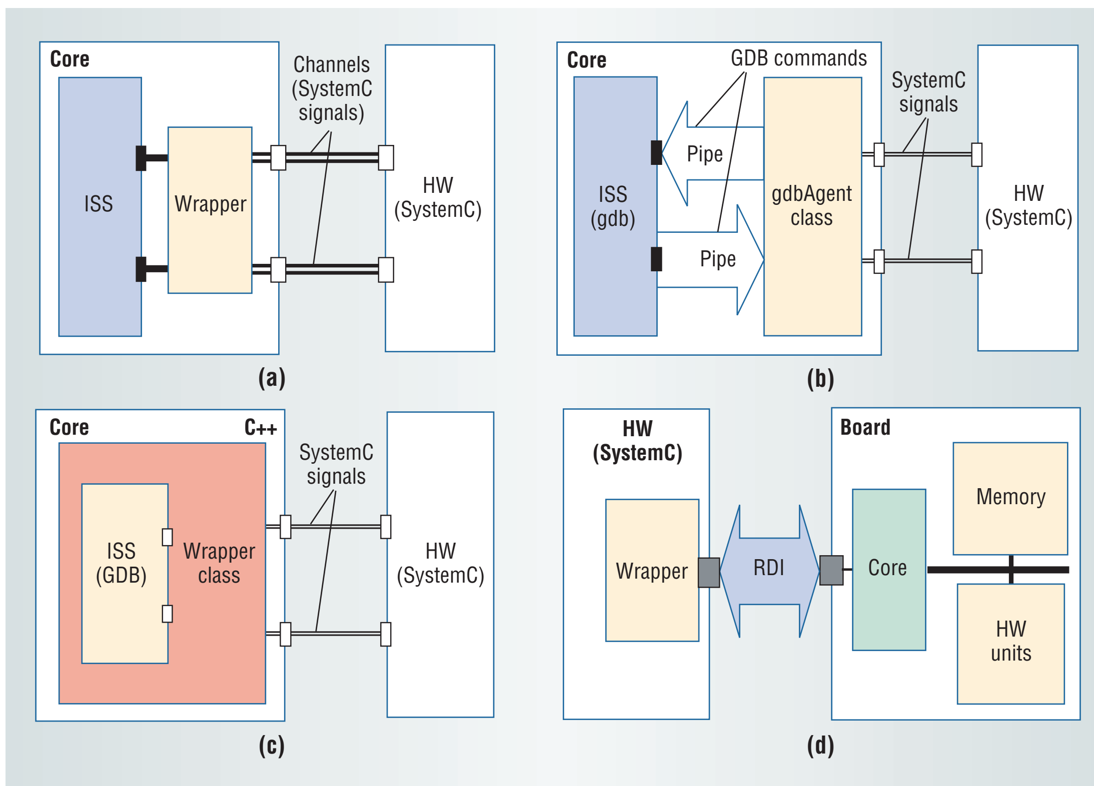

Figure 2. Cosimulation architectures. (a) Instantiation of bus wrappers as SystemC modules; (b) remote ISS cosimulation; (c) linked ISS cosimulation; (d) virtual in-circuit emulation.

COSIMULATION INTERFACES

Figure 2 depicts ISS/SystemC interface implementations that span different degrees of execution granularity. *Remote ISS cosimulation* is based on the instantiation of a wrapper that exchanges GDB commands via IPC, while *linked ISS cosimulation* embeds the ISS within the SystemC simulator.

Remote ISS cosimulation

This strategy implements the wrapper as an ad hoc class gdbAgent, whose main function is to execute the GDB and control its execution. This class is an extension of a similar class contained in the GNU project's Data Display Debugger.[11] Figure 2b shows the remote ISS cosimulation architecture.

The gdbAgent class constructor loads and executes the GDB, and creates two Unix pipes to establish a bidirectional communication channel for exchanging GDB commands. The class then implements various methods to drive GDB execution:

- Quit, Run, Next: sends the corresponding GDB commands, terminated by a newline
- setFile: sends the command "file <*filename*>"
- setBreakpoint, setBreakOnCondition: sends the break or break-on commands
- sendCommand: sends a GDB command to the ISS
- contBreak: continues execution until next breakpoint
- getVariable, setVariable: reads or modifies the value of a variable—these two methods allow data transfer to or from the ISS

The SystemC environment compiles the gdbAgent together with descriptions of the other modules and possibly with other wrappers to create a single executable of the whole system description. The simulation's granularity depends on which GDB commands are used to synchronize program execution as well as on the system architecture. The finest granularity corresponds to tracing instructions step by step via the next command. In this case, however, IPC overhead becomes sizable.

In multiprocessor systems, access to specific variables in the shared memory achieves synchronization. In this case, setting breakpoints in correspondence to reads and writes to those memory cells achieves the coarsest possible granularity.

The typical sequence of operations for a remote ISS scheme involves first creating a gdbAgent object that sets the proper executable file, connects to the target, and sets breakpoints to the functions that trap shared-memory accesses. After running the program, the agent manages accesses to the shared memory; from a given breakpoint code, the agent determines the corresponding type of access (read/write) and triggers the relative bus transaction.

Linked ISS cosimulation

When interaction between a processor instance and the rest of the system is tight, interprocess communication becomes burdensome. In these cases, a closer link between the ISS and SystemC simulation is necessary to alleviate the speed penalty caused by frequent IPC calls. An alternative to the remote ISS approach is to completely embed the

Figure 3. SystemC wrapper architecture. (a) The SC_ MODULE WRAPPER instantiates a CPU object, launches the ISS simulation, and synchronizes it with the signals from the environment. (b) The class CPU is created from a stand-alone ISS. (c) Achieving ISS/ SystemC interaction requires modifying the code to detect accesses to specific memory or I/O regions.

```
#include <systemc.h>
#include "CPU.H"

SC_MODULE(WRAPPER){
sc_in .....
sc_out ...
sc_inout ...
CPU cpu; // ISS Class allocation
void Start_Simulation();
void Bus_Iface_Out();
void Bus_Iface_In();

SC_CTOR(WRAPPER) {
   SC_THREAD(Start_Simulation);
   SC_CTHREAD(Bus_Iface_In, clock.pos());
   sensitive_pos << clock;
   SC_METHOD(Bus_Iface_Out);
   sensitive <<cpu.mem_access;
}
```

(a)

```
Class CPU {
..........
public:
CPU(..); // Constructor

sc_signal<bool> return_from_mem_access;
sc_signal<bool> mem_access;
uint address;
uint data;
.........
}
```

(b)

```
#include <systemc.h>
#include "CPU.H"

CPU::CPU(..) {
// Initialize simulation
}

CPU::Run() {
...
...
...
}

uint Memory_Read(uint add)
{
...
  address = add;
  mem_access.write(true);
  while (return_from_mem_access = = 0)
     wait();
  ...
}
```

(c)

ISS within the SystemC simulator as a C++ class. Upon instantiation of an ISS class object, it is possible to start, manage, and synchronize an instruction-set simulation with the rest of the system. This solution transforms the scheme of Figure 2a into that of Figure 2c, where embedding of the ISS within the wrapper is explicit.

As Figure 3a shows, achieving such embedding requires defining an SC_MODULE WRAPPER and the ISS simulator class CPU, shown in Figure 3b. The wrapper instantiates a CPU object, launches the simulation, and synchronizes it with environmental signals such as memories or peripherals. It also implements a virtual socket interface that translates ISS interface events into legal system bus transactions. The CPU class is created from a standalone ISS. Global variables in the ISS become internal variables of the CPU class, while locally scoped variables remain untouched.

The CPU constructor performs all ISS initialization procedures and prepares all data structures required for simulation. The Run method performs the simulation and can be taken directly from the ISS after some changes.

The typical sequence of operations is as follows: The wrapper instantiates a CPU object and starts ISS simulation through an SC_THREAD command. By default, the Run method runs until the simulation completes, with no interaction between the ISS and its environment. Therefore, we modified the code as Figure 3c shows to detect accesses to specific memory or I/O regions. Now, execution of a memory write instruction in the simulation loop, contained in Run, invokes Memory_Write to communicate to the wrapper the generation of an event requiring synchronization; this suspends execution by calling wait() for a restart condition to become true.

Allocating quantities of interest as public variables of the CPU class moves information between the ISS and the wrapper. For example, mem_ access triggers the Bus_Iface_Out process, which generates cycle-accurate bus signals outside the CPU wrapper.

EMULATION INTERFACES

Using GDB's remote debugging interface (RDI) makes it possible to apply remote ISS cosimulation with an actual core in place of the ISS. *Virtual in-circuit emulation* is similar to conventional in-circuit emulation[12] in that hardware replaces the processors, while HDL models the rest of the system under design. A virtual ICE connector links the simulation model's virtual socket, which is left open, to an available hardware implementation. This solution can perform both functional and timing-accurate validation with the same accuracy as hardware and with greater speed than simulation software, but at a fraction of the cost of conventional emulation schemes.

By explicitly instantiating the wrapper into the SystemC description, as Figure 2d shows, virtual

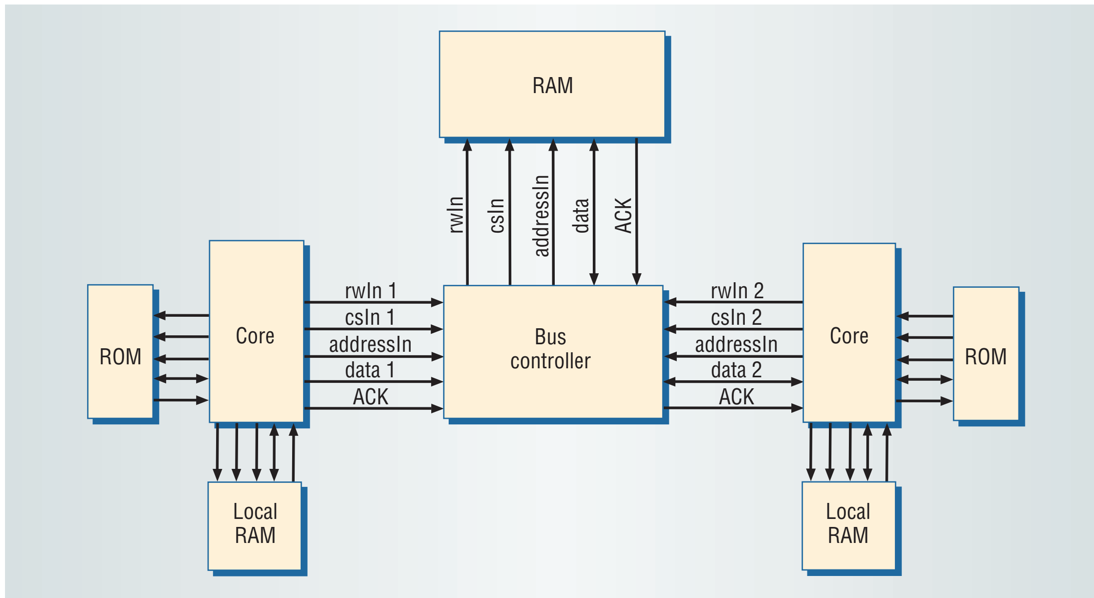

Figure 4. ISS/SystemC cosimulation test architecture. The interface between the bus and the cores, and between the bus and memory, consists of a read/write signal, a chip select, an address, data, and an acknowledge signal.

ICE lets the simulator exchange GDB commands through RDI commands during communication with the target. It also synchronizes SystemC with the memory board by trapping C/C++ read/write operations and replacing them with gdbAgent class getVariable and setVariable commands. These commands move data from software variables (memory board) to SystemC variables (hardware device buffers) and vice versa.

Proper pragmas in the application code ensure the automatic substitution of read/write commands. For example, the system can replace the generic function that reads data from the hardware device actually modeled in SystemC with a dummy function that serves as a pure placeholder for the target of a GDB breakpoint.

Adding timing information—namely, the distance between two consecutive breakpoints—can improve synchronization with software execution. Accomplishing this requires inserting a timing control point around the operation of interest; the time values that the system reads depend on which time reference is available on the board and, in general, must be properly converted to obtain the correct clock value. The difference between the two times will then yield the exact number of clock cycles. Instructing the SystemC code to wait for the same number of clock cycles implements a timing-accurate cosimulation.

CASE STUDIES

We compared ISS and SystemC cosimulations with a full SystemC simulation at the microarchitectural level. We also assessed our scheme's emulation capabilities using an actual prototyping board hosting an ARM core and various hardware peripherals running an embedded application. Two case studies demonstrated significant speedup in both cosimulation and emulation.

Cosimulation

We implemented the cosimulation methodology in the SystemC 2.0 framework and applied it to a system consisting of two cores accessing shared memory through a *bus controller,* as Figure 4 shows. The interface between the bus and the cores consists of a read/write signal rwIn, a chip select csIn, an address addressIn, the data, and an acknowledge signal ACK that the bus asserts upon completion of a reads/writes from/to memory. A similar interface exists between the bus controller and memory.

To limit the processors' access to memory, the bus controller implements an aging mechanism that decreases priorities as the number of memory accesses increases. A local ROM stores the application that the two processors execute, which consists of the *manipulation*—the computation of a moving average—of an array of integers; partitioning the data in two concurrently processed subsets results in parallelization. Testing the value of a shared memory cell, used as a semaphore, synchronizes the processors.

Given the availability of a MAL SystemC description of a DLX processor, a simplified version of the MIPS core, we built the ISS with GNU Compiler Collection (GCC) v2.95.3 and GDB v5.0 and used MIPS as a target for three different experiments. The first one consists of a SystemC reference simulation of the system architecture: The same clock synchronizes all blocks, while the cores read the respective instructions as binary codes from the ROMs and access the bus according to the memory access pattern. The other two experiments demonstrate the two proposed cosimulation schemes.

The remote ISS implementation replaces the cores with two gdbAgent classes and drives them with the standard GDB interface. We synchronize the processors by setting a breakpoint every time a location in the shared memory is modified. Only accesses to the

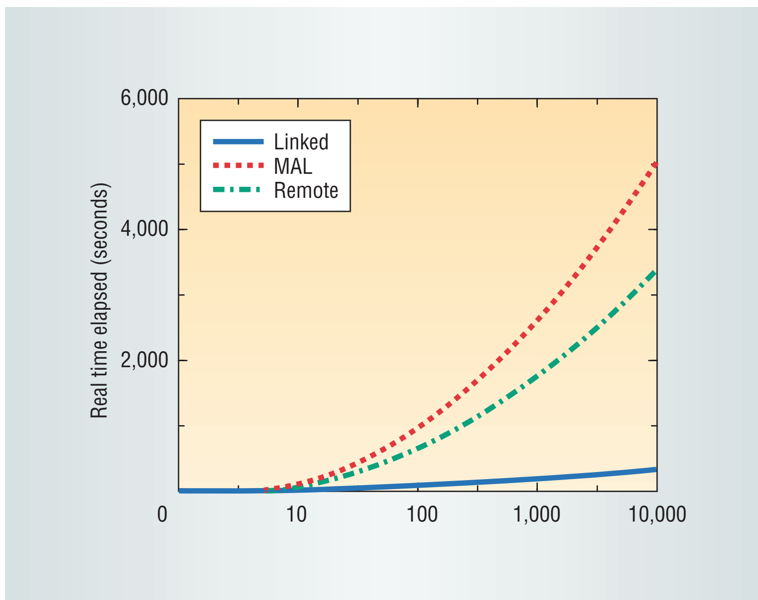

Figure 5. Speedup versus number of cycles for ISS/SystemC cosimulation and microarchitectural-level SystemC simulation. Linked ISS is an order of magnitude faster than MAL, while remote ISS is faster than MAL by a factor of about two.

shared memory require an explicit interaction through the gdbAgent; those to local memories always occur through the GDB memory. The linked ISS implementation replaces the cores with two CPU classes; again, simulation is synchronized with accesses to locations in the shared memory.

Figure 5 compares the execution time of the various simulations—measured on a Pentium II with 256 Mbytes of memory, running Linux Red Hat 7.2—versus the number of iterations of the algorithm that the application implements. The results show that the two cosimulation approaches offer tradeoffs between flexibility and simulation speed. As expected, linked ISS is much faster and should always be the choice when the target ISS sources are available. Remote ISS is slower than linked ISS yet still faster than MAL simulation by a factor of about two.

Emulation

The virtual ICE experimental setup consists of a Linux box—a Pentium II with 64 Mbytes of RAM—running the hosted part of the simulation written in SystemC, an advanced reduced-instruction-set machine evaluation board with an ARM7 microcontroller, and Red Hat's open source embedded configurable operating system (eCos) as runtime support. The development suite's preinstalled Angel Debug Monitor is transparent to the gdbAgent class, which supports both remote debugging with an eCos GDB stub (target remote) and the Angel Debug Monitor stub (target rdi). In the case of rdi targets, however, a timing-accurate evaluation is impossible because the gdbAgent class does not support eCos time-tracking primitives.

For this case study, we chose an integer-parameterizable fast Fourier transform (FFT) based on a precalculated sine table. The first experiment involves the rdi target. The source is compiled with the GCC compiler targeted for arm-elf.

The hosted part of the virtual ICE drives the simulation and emulates an analog-to-digital converter that provides samples at a given rate. When the buffer is filled with the samples needed for the FFT evaluation, the program passes control to the gdbAgent, which uses the setVariable method to transfer the buffer to the board.

A serial link with a maximum transfer rate of 38,400 bits per second sends the data to the board. When the data transfer is complete, control passes to the board, which performs the FFT and returns control to the hosted part of the simulation. The simulator can obtain computation results with the getVariable method, thereby functionally validating the algorithm that the hardware platform executes.

Because eCos supports the ARM evaluation board and all the primitives necessary to calculate a task's required time, it is possible to evaluate algorithm performance—in this case, the eCos GDB stub (target remote) is required. Figure 6 shows the results of this analysis, reporting execution time and sustainable frame rate as a function of the number of FFT points.

Inserting SystemC wait() commands in the hosted part of the simulation that correspond to FFT computation time achieves synchronization. The exact number of commands depends on the board's time granularity—10 milliseconds in our case. Our board features a 24-MHz clock with a 41.66-ns period. Assuming that the SystemC simulation and board share this clock, the board's time granularity corresponds to 10 ms/41.66 ns = 240,000 clock cycles. For real-time performance, the board needs 1,024 samples every 140 milliseconds. Therefore, if the FFT computation takes 140 milliseconds (10 ticks) on the board, the simulation must execute 3,360,000 ($14 \times 240{,}000$) wait() commands for the synchronization.

We are currently working to extend our methodology to more complex on-chip architectures, such as network-on-chip designs. In the case of NoCs, modeling interconnections among the cores—in addition to modeling the cores themselves—becomes crucial. For such systems, integrating SystemC descriptions and ISSs with a network simulation engine will be an essential step. ■

References

1. F. Balarin et al., *Hardware-Software Co-Design of*

Embedded Systems: The Polis Approach, Kluwer Academic Publishers, 1997.
2. J.T. Buck et al., "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems," *Int'l J. Computer Simulation,* Apr. 1994, pp. 155-182.
3. Mentor Graphics Corp., "Seamless Hardware/Software Co-Verification," www.mentor.com/seamless.
4. CoWare Inc., "CoWare N2C Design System," www.coware.com/cowareN2C.html.
5. G. De Micheli, "Hardware Synthesis from C/C++ Models," *Proc. Conf. Design, Automation and Test in Europe* (DATE 99), ACM Press, 1999, pp. 382-383.
6. T. Grötker et al., *System Design with SystemC,* Kluwer Academic Publishers, 2002.
7. L. Séméria and A. Ghosh, "Methodology for Hardware/Software Coverification in C/C++," *Proc. Asia and South Pacific Design Automation Conf.* (ASP-DAC 00), ACM Press, 2000, pp. 405-408.
8. J. Liu, M. Lajolo, and A. Sangiovanni-Vincentelli, "Software Timing Analysis Using HW/SW Cosimulation and Instruction Set Simulator," *Proc. 6th Int'l Workshop Hardware-Software Codesign,* IEEE CS Press, 1998, pp. 65-69.
9. P. Gerin et al., "Scalable and Flexible Co-Simulation of SoC Designs with Heterogeneous Multi-Processor Target Architectures," *Proc. Conf. Asia and South Pacific Design Automation* (ASP-DAC 01) ACM Press, 2001, pp. 63-68.
10. GNU Project, "Debugging with GDB," www.gnu.org/manual/gdb-4.17.
11. GNU Project, "DataDisplayDebugger v3.3," www.gnu.org/software/ddd.
12. R. Bannatyne, "Debugging Aids for Systems-on-a-Chip," *Proc. Wescon/98 Conf.,* IEEE Press, 1998, pp. 159-163.

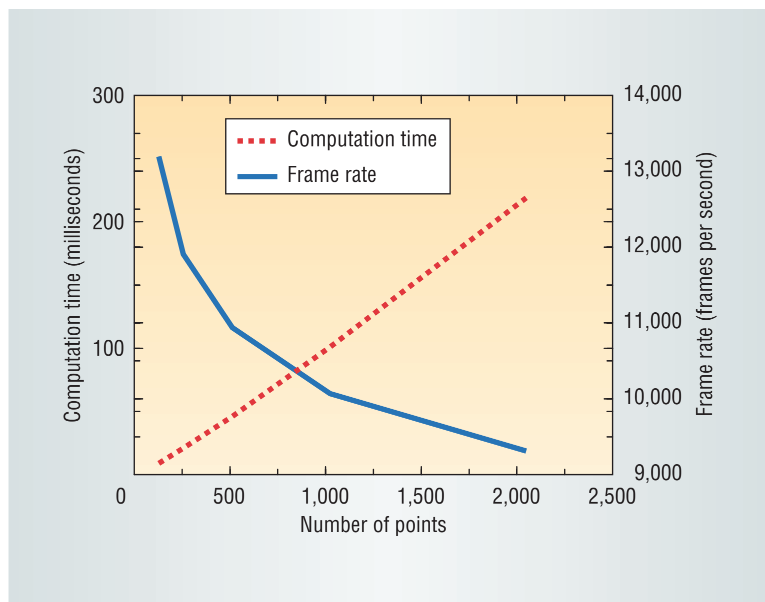

Figure 6. Speed and sustainable frame-rate estimation for fast Fourier transform. Values are from the emulation framework and can substitute for hardware measurements.

Luca Benini *is an associate professor in the Department of Electronics and Computer Science (DEIS) at the University of Bologna, Italy. His research interests include all aspects of computer-aided design of digital circuits and the design of portable systems. Benini received a PhD in electrical engineering from Stanford University. Contact him at lbenini@deis.unibo.it.*

Davide Bertozzi *is a doctoral student in the DEIS at the University of Bologna. His research interests include power modeling and optimization of multiprocessor systems-on-chip. Bertozzi received a DrEng in electrical engineering from the University of Bologna. Contact him at dbertozzi@deis.unibo.it.*

Davide Bruni *is a doctoral student in the DEIS at the University of Bologna. His research interests are in the estimation and optimization of microprocessor-based system-on-chip designs. Bruni received a DrEng in electrical engineering from the University of Bologna. Contact him at dbruni@deis.unibo.it.*

Nicola Drago *is a PhD candidate in the Department of Science and Technology at the University of Verona, Italy. His research interests include IP network protocols, automatic modeling, and synthesis of transmission protocols. Drago received a Laurea in computer science from the University of Verona. Contact him at drago@sci.univr.it.*

Franco Fummi *is a professor of computer architecture in the Department of Science and Technology at the University of Verona. His research interests include hardware description languages and electronic design automation methodologies for testing and optimization of hardware/software systems. Fummi received a PhD in electronic and communication engineering from Milan Polytechnic, Italy. Contact him at fummi@sci.univr.it.*

Massimo Poncino *is an associate professor of electrical and computer engineering in the Department of Science and Technology at the University of Verona. His research interests include the computer-aided design of integrated circuits and systems, with particular emphasis on logic synthesis, optimization, testing, and verification. Poncino received a PhD in computer engineering from Turin Polytechnic, Italy. Contact him at poncino@sci.univr.it.*