Abstract

We present a co-simulation environment for multiprocessor architectures, that is based on SystemC and allows a transparent integration of instruction set simulators (ISSs) within the SystemC simulation framework. The integration is based on the well-known concept of bus wrapper, that realizes the interface between the ISS and the simulator.

The proposed solution uses an ISS-wrapper interface based on the standard gdb remote debugging interface, and implements two alternative schemes that differ in the amount of communication they require.

The two approaches provide different degrees of tradeoff between simulation granularity and speed, and show significant speedup with respect to a micro-architectural, full SystemC simulation of the system description.

1 Introduction

Today’s complex systems-on-chip (SoCs) are usually built from processor-based templates, and contain one or more processor cores, with a significant amount of on-chip memory and complex communication busses. Core processors for on-chip integration are often legacy or third-party components, and are viewed as resources. Therefore, designers do not need a detailed description of the processor’s micro-architecture, but they do require correct functional (behavioral) models and I/O interface descriptors that accurately track the interaction of the core with the rest of the chip. These models should also provide information about the run-time of the software application they execute; such estimates should be reliable enough to cross-validate a design against performance specifications.

Embedded software designers working on processor cores routinely employ cross-development toolkits to validate functionality and assess performance of applications. A minimal cross-development toolkit contains a cross-compiler, a timing-accurate instruction-set simulator (ISS) and a debugger. On the other hand, hardware designers validate their work using hardware-description language (HDL) simulators. The latter are quite inefficient in simulating complex processor cores, because they model their micro-architecture in too much detail.

Designing a complex systems-on-chip requires thus a single, integrated hardware-software simulation platform, for both exploration and validation. For this reason, a large number of co-simulation platforms has been developed both by academic groups and EDA vendors [1, 2, 3, 4, 5, 6, 7, 8]. Initially, co-simulation focused establishing a solid link between event-driven hardware simulators and cycle-based ISSs.

In the last few years, hardware descriptions and design flows based on C/C++ have gained momentum because of their potential for bridging the gap between hardware and software description languages [9, 10, 11], thanks to the possibility of using the same language for describing software and hardware. In addition, co-simulation becomes easier and more efficient, because the entire system can be simulated within a single simulation engine, eliminating the overhead of communication between different simulators.

SystemC is one of the leading C/C++ design environments: it provides an open-source, free simulation environment and several class packages for specifying hardware blocks and communication channels [10]. Software in SystemC can be specified algorithmically, as a set of functions embedded in SystemC abstract modules. Software modules can communicate among themselves and with hardware components via abstract SystemC communication channels.

When software is specified at this level of abstraction, it is very hard to estimate its execution time and analyze its detailed synchronization with hardware. Excluding the possibility of resorting to a cycle-accurate, microarchitectural description of the core, because of its high inefficiency, two approaches are possible.

One possibility is that of resorting to a description of the
core in SystemC, so that the execution of the software can be modeled consistently with the rest of the system. We will refer to this solution as “RTL” simulation, to emphasize its cycle-based accuracy.

The other option is to simulate the core at a higher abstraction level, by embedding instruction-set simulators within the co-simulation environment. Most previously published approaches [11, 8, 13, 14, 15], are based on inter-process communication (IPC) and the concept of bus wrapper. The ISS and the C/C++ co-simulator run as distinct processes on the host system, and they communicate via IPC primitives. The bus wrapper has two key functions: (i) it ensures synchronization between the system simulation and the ISS, and (ii) it translates the information coming from the ISS into cycle-accurate bus transactions that are exposed to the rest of the system.

Two are the main limitation of these approaches. First, the IPC paradigm is effective when the communication between the ISS and the rest of the system is sparse in time. This is the case when the ISS model includes not just the core but also a significant amount of local memory (e.g., the D-cache), so that communication with the rest of the system is required only for few instructions (e.g., on explicit reads and writes on memory-mapped I/O). Second, most approaches define a proprietary interface between the bus wrappers and the ISS. This choice greatly complicates the integration of new processor cores within the co-simulation framework: The ISS needs to be modified to support the IPC communication primitives defined by the co-simulation system.

This work addresses the two above-mentioned limitations. Our first contribution is an implementation of the IPC interface between bus wrapper and ISS based on the remote debugging primitives of gdb [16]. This can be considered a de-facto standard for IPC, since almost every core processor is provided with a GNU-based software cross-development environment (cross-compiler, ISS and debugger). In this way, any ISS that can communicate with gdb can also become part of a system-level co-simulation environment.

In addition, we address the performance bottleneck created by IPC when the processor interacts very tightly with the rest of the system. We leverage the standardized structure of GNU’s instruction-set simulators to develop a small library of functions to be called from within the top module of a legacy GNU’s ISS. This top-level module is embedded as a process in the SystemC simulator, and it calls the standard GNU’s ISS interface functions, whose implementation is ISS specific. In this way, the ISS is fully embedded in the system simulator executable, and slow interprocess communication is completely eliminated.

Results on a system consisting of two processor cores with local and shared memories shows the effectiveness of the two proposed co-simulation schemes.

2 Co-Simulation Methodology

The proposed co-simulation methodology targets heterogeneous, multi-processor architectures, and is based on the SystemC simulation environment [10].

With respect to the design flow, we assume that the assignment of tasks to either hardware or software (HW/SW mapping) has already been decided. In practice, the multi-processor architectures under analysis consist of a set of hardware blocks that implement part of the tasks, and a set of processor cores that execute the other part. Processor mapping is also given, in the sense that specific core platforms have been decided. In this context, the term “core” identifies a generic programmable resource for which either an ISS or a HDL model is available. The generic architectural template is shown in Figure 1-(a), where the tasks of the system have been mapped to four cores and two generic hardware block (labeled HW).

![Figure 1. Architectural Template (a) and Simulation Alternatives: Full SystemC simulation (b), and ISS-SystemC co-simulation (c).](image)

Figure 1-(b) shows the case of a full SystemC, cycle-accurate simulation. The co-simulation scheme is depicted in Figure 1-(c)), where, as an example, the SystemC models of some cores are replaced by the corresponding ISSs (the light grey blocks), communicating via IPC with the SystemC simulation back-end.

The latter co-simulation scheme is the one followed by most existing approaches [8, 13, 14], that are based on IPC and on the instantiation of bus wrappers; the ISS and the co-simulator run as distinct processes on the host system, and communicate via IPC primitives.

The methodology proposed in this work is based on the idea proposed by Semeria and Ghosh [11]. The use of SystemC allows to eliminate the need of an explicit distinction (from the simulator point of view) between the wrapper and
the ISS. The integration of wrappers as SystemC objects allows to restrict the use of IPC just between the bus wrappers and ISS, rather than between bus wrappers (Figure 2). In particular, our methodology overcomes some of the limitations of previous approaches, and has two distinctive features:

1. The implementation of the IPC interface between the bus wrapper and the ISS through non-proprietary interface, namely, the remote debugging primitives of the GNU gdb. Compliance of an ISS to this interface becomes then the only constraint for its inclusion in the co-simulation environment. The issue of non-proprietary interfaces for an ISS in co-simulation environments was mentioned in [12], but it was not implemented inside the wrapper abstraction of Figure 2.

2. The complete elimination from the co-simulation of the bottleneck of IPCs. This is achieved by adapting the ISS code so that it can be directly embedded as a process in the SystemC simulator. This solution requires the availability of the ISS source code, hence, although most core processors are supported by the GNU cross-development toolkits, it is not viable in the case of proprietary ISSs.

The proposed co-simulation can be realized under two different strategies, that span different degrees of granularities of execution, and are both based on the integration of wrappers into SystemC. The first strategy, called triggered co-simulation is based on the instantiation of an ad-hoc wrapper that exchanges gdb commands via IPC. The second strategy, called legacy co-simulation implements the scheme that embeds the ISS within the SystemC simulator.

3 ISS-SystemC Co-Simulation
3.1 Triggered Co-Simulation

The conceptual architecture of the triggered co-simulation approach is depicted in Figure 3.

The proposed co-simulation can be realized under two different strategies, that span different degrees of granularities of execution, and are both based on the integration of wrappers into SystemC. The first strategy, called triggered co-simulation is based on the instantiation of an ad-hoc wrapper that exchanges gdb commands via IPC. The second strategy, called legacy co-simulation implements the scheme that embeds the ISS within the SystemC simulator.

The wrapper consists of a class gdbAgent whose main function is that of executing the gdb and controlling its execution. This class is an extension of a similar class contained in the DDD package [17], a GNU GUI for the the gdb.

The constructor of the gdbAgent class first loads and executes the gdb, and creates two UNIX pipes to establish a bidirectional communication channel with gdb, over which conventional gdb commands are exchanged. The class implements then the various methods for driving the execution of the gdb:

- Quit, Run, Next: send the corresponding gdb commands, terminated by a newline;
- setFile: send the command “file <filename>”;
- setBreakpoint, setBreakOnCondition: sets breakpoints on a source line or on some specified condition;
- sendCommand: sends a gdb command to the ISS;
- contBreak: continues after a breakpoint, and return the breakpoint identifier;
- getVariable, setVariable: allows to read or modify the value of a variable (correspond to print var and set var=val commands).

The gdbAgent is compiled within the SystemC environment simulation together with the descriptions of the other modules (possibly with other wrappers), to get a single executable of the whole system description.

The granularity of the simulation depends on which gdb command are used to synchronize the execution of the program, and on the system architecture.

Figure 4 shows an example of the typical sequence of operations of the triggered scheme, where an object cpu of the gdbAgent class is communicating with the ISS. Notice the breakpoints in correspondence of two auxiliary functions that expose memory reads and writes.

In the case of multiprocessor systems, for instance, synchronization between processors is realized through the access to specific variables in the shared memory. In this
case, the coarsest possible granularity is obtained by setting breakpoints in correspondence to reads and writes to those memory cells.

The finest granularity is clearly equivalent to tracing instructions step-by-step (via the next command). In this case, however, the overhead due to the IPC becomes non-negligible.

3.2 Legacy ISS Co-Simulation

When the interaction between a processor instance and the rest of the system is very tight, interprocess communication becomes burdensome. In these cases, a tighter link between ISS and SystemC simulations is sought, in an effort to alleviate the speed penalty caused by frequent IPC calls. An alternative approach to the triggered approach is to completely embed the ISS within the SystemC simulation. In other words, we want to transform the ISS in a C++ class. Locally-scoped variables can remain untouched. Two methods are defined: the CPU constructor and run. The constructor performs all initialization procedures of the stand-alone ISS, and prepares all data structures required for simulation. The run method performs the simulation.

The SystemC SC_MODULE WRAPPER instantiates a CPU object, and initializes it in its constructor. The ISS simulation is started by a dedicated process (a SystemC SC_THREAD), called Start_simulation. Clearly, if no provisions are made, the run method would run until simulation completion, with no interaction between ISS and its environment. To enable interaction, the code within the run method must be marginally modified. In detail, we change the code around ISS memory and IO access functions, in such a way that accesses to specific memory or IO regions can be detected. An example of this is shown in Figure 5: when an access is detected, some information is made available to the wrapper and execution is suspended with a call to the SystemC wait function.

In particular, the SystemC wrapper has to receive information about external memory or I/O addresses, data to write on the bus and the type of bus request (read or write access). Moreover, data read from the bus must be passed back to the ISS. This communication between the ISS and the SystemC wrapper is implemented by allocating the parameters of interest as public variables of

The CPU class is created starting from a stand-alone ISS (in C or C++). The class declaration is shown in Figure 5. All global variables in the stand-alone ISS must be made internal variables of the CPU class. Locally-scoped variables can remain untouched. Two methods are defined: the CPU constructor and run. The constructor performs all initialization procedures of the stand-alone ISS, and prepares all data structures required for simulation. The run method performs the simulation.
the CPU class. In these way, they can be accessed by both sides. Two other public variables have been used. 

\text{Mem\_access triggers the SystemC Bus\_Iface\_Out process, which in turn generates the cycle accurate bus configuration. Return\_from\_mem\_access is the variable watched by the ISS at each recovery from the sleep state, indicating whether the bus access has been completed or not. In this way, the timing penalty for accessing an external memory is taken into account.}

It is important to note that synchronization between the SystemC time and the ISS simulated time has been implemented. The ISS simulation is suspended by means of wait calls until the SystemC time tracks the simulated time. Only at that time the bus transaction is carried out, thus generating a realistic bus traffic.

4 Experimental Results

We have implemented the proposed methodology in the SystemC 2.0 simulation framework, and we have applied it to a system consisting of two core processors accessing to a shared memory through a bus. A block diagram of the system is shown in Figure 6.

![Block Diagram of the Test Architecture](image)

**Figure 6. Block Diagram of the Test Architecture.**

The bus arbitration mechanism is managed by the module labeled Bus controller. The interface between the bus and the cores consists of five signals: a read/write signal \text{rwIn}, a chip select \text{csIn}, an address \text{addressIn}, the data \text{data}, and an acknowledge signal from the bus \text{ACK}, asserted upon completion of a read/write to memory. A similar interface exists between the bus controller and the memory.

Access to the bus is based on a priority mechanism. In order to avoid the chance that one of the processors can be granted the access to the memory for the whole duration of its computation, the bus controller implements a sort of aging mechanism that decreases priorities as the number of memory accesses increases.

The application executed by the two processors are stored in a local ROM, and consists of the manipulation (a variant of the computation of a moving average) of an array of integers, executed in a parallel fashion: data are partitioned in two subsets that are processed concurrently by the two processors. The processors are synchronized by testing the value of a shared memory cell, used as a semaphore.

The availability of a SystemC description of a DLX processor, a simplified version of the MIPS [18], has determined the choice of the target architecture. The relative ISS has been built the GNU cross-compiler (gcc Version 2.95.3) and cross-debugger (\text{gdb} Version 5.0) with the MIPS as a target.

We have run three different simulation experiments: The first one represent the reference simulation, and consists of a plain SystemC simulation of the architecture of Figure 6. All blocks have thus been implemented as SystemC modules, and are synchronized on the same clock. The cores read the respective instructions as binary codes from the ROMs, and access the bus according the memory access pattern.

The other two experiments realize the two co-simulation schemes described in Section 3. One experiment uses the triggered approach: The two cores are replaced by two GDBAgent classes, and are driven by the standard GDB interface. Processors are synchronized (i.e., a breakpoint is set) every time a location in the shared memory is modified. In practice, the user issues a conventional GDB “break on <condition>” command to synchronize the execution. Notice that only accesses to the shared memory requires an explicit interaction via IPC with GDBAgent. Accesses to the local memories always occur through the \text{gdb} memory.

The other configuration uses the legacy simulation approach: The two cores are replaced by two CPU classes. The simulation is synchronized, as in the previous case, in correspondence of accesses to locations in the shared memory.

Table 1 compares the results of the various simulation approaches. The plot reports CPU time, measured on a Pentium II 400 with 256 MB of memory, running Linux Red Hat 7.2. The table shows three columns 10, 100, and 1000, corresponding to the number of iterations of the algorithm implemented by the application.

<table>
<thead>
<tr>
<th>Simulation Type</th>
<th>CPU Time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
</tr>
<tr>
<td>RTL</td>
<td>9.4</td>
</tr>
<tr>
<td>Triggered</td>
<td>2.5</td>
</tr>
<tr>
<td>Legacy</td>
<td>0.7</td>
</tr>
</tbody>
</table>

**Table 1. Co-Simulation Results.**

The results show that the two co-simulation approaches offer different tradeoffs between flexibility and simulation.
speed. As expected, the legacy approach is much faster, and should always be the choice when the type of synchronization is clearly defined and the target ISS source are available. The speedup is more than one order of magnitude with respect to a full SystemC simulation (Row RTL). The triggered approach is slower than legacy simulation, yet still faster than SystemC simulation of a factor of about 2.

Notice that, although we labeled the full SystemC simulation as “RTL”, the implementation is far from being a synthesizable description. As a reference data, the case of 1000 iterations corresponds to the execution of more than 2 million instruction, definitely much faster than a RTL simulation. Therefore, we expect more sizable speedups in the case of a “true” RTL simulation of the system.

5 Conclusions

Application of conventional co-simulation paradigms to multi-processor architectures requires efficient mechanisms for the communication between ISSs and the simulation engine. The adoption of a C++-based simulation environment such as SystemC allows to develop effective solution, because the entire system executes within a single simulation environment.

In this work, we propose two co-simulation approaches, that are based on the use of a standard interface (namely, the gdb remote debugging interface, supported by most ISS) between the ISS and the wrapper used to link it to the simulation environment.

The two proposed solutions provide various degrees of simplification of the ISS/wrapper interface, up to a minimum-overhead scheme that completely removes the need of IPC on the interface, obtained by transparently embedding the ISS within the simulation environment.

Simulation results, with respect to a full SystemC simulation of a two-processor test case, shows speed improvement by a factor of 1.5x to 15x, depending on the chosen solution.

References