SimSoC: A SystemC TLM integrated ISS for full system simulation

Claude Helmstetter
INRIA
LIAMA, 95 Zhongguancun East Road
Haidian District, 100080, Beijing, CHINA
Email: claude@liama.ia.ac.cn

Vania Joloboff
INRIA
LIAMA, 95 Zhongguancun East Road
Haidian District, 100080, Beijing, CHINA
Email: vania@liama.ia.ac.cn

***Abstract*—The development of embedded systems requires the development of increasingly complex software and hardware platforms. Full system simulation makes it possible to run the exact binary embedded software including the operating system on a totally simulated hardware platform. Whereas most simulation environments do not support full system simulation, or do not use any hardware modeling techniques, or have combined different types of technology, SimSoC is developing a full system simulation architecture with an integrated approach relying only upon SystemC hardware modeling and transaction-level modeling abstractions (TLM) for communications. To simulate processors at reasonably high speed, SimSoC integrates instruction set simulators (ISS) as SystemC modules with TLM interfaces to the other platform components. The ISS's use a variant approach of dynamic translation to run binary code. The dynamic translator uses pre-compiled code that consists of specialized functions for instruction execution, using partial evaluation techniques. It is generated by a configurable code generator, which makes it possible to tune the generated code to optimize simulation speed for the target software application.**

I. Introduction

The development of embedded systems platforms requires increasingly large pieces of software running on complex System On Chips. A simulation environment is necessary to simulate the system under design so that software developers can test the software and hardware developers can investigate design alternatives.

For the embedded software developers, the simulation environment must achieve full system simulation: it must run the exact binary software that will be shipped with the product, including the operating system and the embedded application; and the simulation must be fast enough for interactive testing and fast software verification cycles. A full system simulation at low level of hardware detail (cycle accurate) is much too slow for software testing.

These requirements call for an integrated, modular, full simulation environment where already proven components can be simulated quickly whereas new IP under design can be tested more thoroughly. Modularity and fast prototyping also have become important aspects of simulation frameworks, for investigating alternative designs with easier re-use and integration of third party IPs.

The SimSoC project[1] is developing a framework geared towards full system simulation, mixing hardware simulation and one or more ISSs, able to simulate complete System-on-Chips. The SimSoC simulation environment combines two technologies in a single framework: SystemC/TLM to model the new IPs and interconnects, and one or more instruction set simulators (ISS). Each ISS is designed as a TLM model.

In this paper, we present the overall system architecture and the ISS technology. To achieve fast processor simulation, the SimSoC ISS technology uses a form of dynamic translation, using an intermediate representation and pre-generated code using specialization, a technique already used within virtual machines.

The hardware models are standard SystemC TLM abstractions and the simulator uses the standard SystemC kernel. Therefore, the simulation host can be any commodity commercial off-the-shelf computer and yet provide reasonable simulation performance.

The rest of the paper is organized as follows. Section II describes related work in the area of full system simulation, instruction set simulation and SystemC TLM. Section III explains the overall structure of the simulator, the integration between SystemC, TLM and the ISS, and it describes the dynamic translation technology. Section IV details some benchmarking. Finally the conclusion offers perspectives for improving simulation speed.

II. Related Work

Simulation platforms can be characterized by the technologies they use for simulating hardware components, either some Hardware Description Language (HDL) or only software emulation; and the extent of the simulation with regard to the overall platform, whether or not a complete software binary such as an operating system can be run over the simulator.

To support simulation at reasonable speed for the software developers FPGA solutions can be used [1]. These solutions tend to present slow iteration design cycles, they are costly, and anyway they can only be used when the hardware design has reached enough maturity to be modeled in FPGA.

Other approaches using software based simulation usually implies two separate technologies, typically one using a Hardware Description Language, and another one using an instruction set simulator (ISS). Then some type of synchronization and communication between the two must be designed and maintained using some interprocess communication. Gerin et. al. [2] have presented such a SystemC co-simulation environment. It offers modularity and flexible usage but it uses an external ISS. Fummi et. al have implemented [3] an integrated simulation environment that reaches fair integration, however there are still two main simulation software interconnected through the use of external GDB debugger program, and the SystemC kernel has to be modified. In SimSoC, we use standard, unmodified, SystemC, and no additional synchronization mechanism is required.

A. Overview of SystemC-TLM

SystemC has become the standard to represent hardware models, as it is suitable for several levels of abstraction, from functional models to synthetizable descriptions. It is defined by an IEEE standard [4], and comes with an open-source implementation.

[1]This project has been partly funded with a grant from Schneider Electric Corporation in China

SystemC is a C++ library that provides classes to describe the architecture (`sc_module`...) of heterogeneous systems and their behavior thanks to processes (`SC_THREAD`...) and synchronization mechanisms (`sc_event`...). The architecture is built by executing the *elaboration phase*, which instantiates modules and binds their ports. Next, the SystemC simulator *schedules* the SystemC processes. A SystemC process is either *eligible* or *running* or *waiting* for a SystemC event. There is at most one running process at a time. A process moves from eligible to running when it is elected by the scheduler. The elected process explicitly suspends itself when executing a `wait` instruction (i.e. the scheduling policy is not *preemptive*). If the running process notifies an event, then all processes waiting for this event move from waiting to eligible.

Transactional level modeling (TLM) refers both to a level of abstraction [5] and to the SystemC-based library used to implement transactional models [6]. The *transaction* mechanism allows a process of an *initiator* module to call methods exported by a *target* module, thus allowing communication between TLM modules with very few synchronization code.

B. Dynamic translation

An extensive body of work has addressed instruction set simulation (ISS). Early instruction set simulation experiments used interpretive simulation of each instruction. It suffers from a performance penalty due to the tedious decoding sequence repeated uselessly. The decoding phase can be removed thanks to a preliminary compilation phase, also called *translation*, that preserves this work for later re-execution of the code.

One can directly translate the target binary into software for the simulation host machine. This technique is known as static translation. Reports [7], [8] show dramatic performance improvements, but static translation does not work well when simulating applications with self-modifying code, or applications like Java Virtual Machines that include a compiler itself generating new code [9].

In the past decade, dynamic translation technology has been favored, such as [10]–[12] . The target code to be executed is dynamically translated into an executable representation. Although dynamic translation introduces a compile time phase as part of the overall simulation time it is expected that this translation time is amortized over time.

Full system simulation is also achieved in so called Virtual Machines such as QEMU [13] and GXemul [14] that emulate the behavior of a particular hardware platform. These emulators are each using ad-hoc techniques to simulate hardware components. Although they contain many hardware components emulation, these models are non standard and non interoperable. For example any of each device model from one emulator cannot be reused into the other emulator. In particular, simulating parallel system on one computer requires some form of scheduling. How these tools schedule parallel entities is not well specified enough to guarantee the compatibility between third-party models. SimSoC relies on the SystemC norm to avoid this problem.

III. SimSoC

SimSoC is implemented as a set of SystemC TLM modules. The global architecture is depicted in figure 1. The hardware components are modeled as TLM models, therefore the SimSoC simulation is driven by the SystemC kernel. The interconnection between components is an abstract bus. Each processor simulated in the platform is abstracted as a particular class.

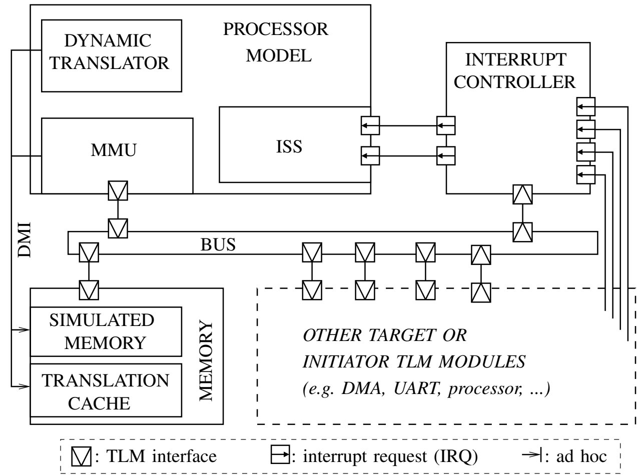

Fig. 1. SimSoC architecture

The goal of the SimSoC ISS is to simulate the behavior of the target processor with instruction accuracy. It emulates execution of instructions, exceptions, interrupts and virtual to physical memory mapping. The processor drives the translation of binary code. When the program counter points to an instruction that has not been translated yet, the translation is called, otherwise the cached translated code is executed. The translation is actually achieved on a memory page basis.

A. Dynamic translation

SimSoC dynamic translation uses an intermediate representation that is partly dependent on the target architecture, but does not involve the maintenance cost of a compiler, similar to [14]. SimSoC intermediate representation is totally independent of the host (both machine architecture and operating system), as long as the host platforms supports the C++ language.

To optimize performance, we have pursued two paths. First, offload most of the compiling work by pre-compiling most of the simulation code with maximum optimization. Second, exploit partial evaluation specialization techniques to optimize generated code.

The SimSoC binary decoder is actually generated by a decoder generator, the Instruction Set Compiler. It takes as input a specification file and produces the C++ architecture specific decoder. This decoder computes every possible value that can be statically determined at that time for partial evaluation and caches re-used values into the data structure of the intermediate representation. For example some ARM architecture instructions may have an immediate value argument shifted by another immediate value and the carry of the resulting shifted value is used in computing the carry bit resulting from that instruction. Such values can be pre-computed at decoding time to select the partially evaluated code that should be used as described below.

A SimSoC ISS includes pre-compiled code loaded at start-up time. Therefore, it is not dependent upon the host binary format and operating system. The decoder dynamically constructs an intermediate representation that maps the binary instructions to this precompiled code.

The precompiled code consists of specialized code. Specialization is a compiling optimization technique, also known as partial evaluation. The basic concept of specialization is to transform a generic program P, when operating on some data d into a faster specialized program Pd that executes (faster) specifically for this data.

Specialization can be advantageously used in processor simulation with a dynamic translation phase.

When data can be computed at decoding time, a specialized version of the generic instruction code can be used to execute it. For example, if an instruction using register Ri operates with register Rj and some immediate value V stored in the instruction, a specialized operation can be generated using the constant values of Ri, Rj and V. The simulation code then uses fewer tests, fewer memory accesses and more immediate values. This technique has been used to some extent in the IC-CS simulator [10] and SimSoC is moving it a step forward. Potentially there are 2^{32} specializations of a 32-bit instruction set, which would lead to a huge amount of specialized code. In practice however, many binary configurations are illegal and some instructions are more frequently executed than others. By specializing the most frequently used instructions to a higher degree than the less frequent ones, one can reduce the number of specialized functions to a manageable amount of code.

The code we are using in SimSoC can be more or less specialized for each instruction class. For almost every variant of an instruction, a specialized version of the code is maintained in a large multidimensional table storing the specialized code for this particular case. Each such element in the table is called a semantic function. The decoding phase mostly amounts to locating the appropriate semantic function for that specialized instruction. For example, regarding the ARM architecture, it is worth specializing the move and load instructions in the `always` condition code, and it is less valuable specializing arithmetic instructions in the rare case the condition code is not `always` and the `S` bit is set.

The specialized code is not manually coded. A code generator generates it. It is then pre-compiled and loaded into the table by the simulator. The code generator is parameterized to generate more or less specialized instructions [15], which can be tuned based on the analysis of the simulated application. For example, the SimSoC code generator generates for the subset of data processing and simple load store instructions 14280 semantic functions, and a total code size of 6.6 Megabyte of code for the entire simulator, which is reasonably small compared to the available memory size on simulation hosts.

B. Transaction Level Modeling

The SimSoC ISS need to access memory and other devices: 1) when it fetches an instruction which is not translated yet; 2) when it execute a load/store instruction (e.g. `ldr`, `strh`, `ldm`, etc). The SimSoC provides two modes: one basic generic mode and an optimized mode.

The basic mode uses the *Blocking transport interface* of the OSCI TLM-2draft [6], which has been designed for untimed simulation as our ISS. This interface requires that each target module exports a function `void b_transport(TRANS &trans)`. We use the default `tlm_generic_payload` for the transaction type, as recommended by the OSCI to ease interoperability. Consequently, to communicate to another component, the processor creates a transaction object, by providing at least an address, a command (read or write), a pointer to data and a data size. Next it calls the `b_transport` function on this object. The bus will next forward the transaction to the memory or a device according to the memory map. Eventually, the `b_transport` method of the corresponding target module will be executed. This way, the SimSoC ISS is compatible with all untimed models of hardware which follows the OSCI recommendation for transactional modeling.

The optimized mode uses the concept of *Direct Memory Interface* (DMI) as suggested by the OSCI TLM-2draft documentation. However, we do not use the OSCI implementation. Indeed, the dynamic translation mechanism used by the ISS requires that the translated code is stored in the memory TLM module in order to accommodate multi-core platforms with shared memory, such that the code translated by one processor may be used by another processor, or invalidated if another initiator writes into the binary code location. We wrote our own direct memory interface such that the processor can fetch a previously translated instruction, and the memory can check for code modification for each write access. The processor MMU can then access memory directly when DMI is enabled, generating a real transaction only for accesses to other devices. The DMI can be reconfigured or disabled or enabled at runtime.

The ISS communicates with other components using interrupt signals too. The OSCI TLM-2draft does not target interruption modeling, so we had to define our own interface. Each interrupt initiator (e.g. a timer) contains a port `sc_port<IT>`, and each interrupt target (e.g. a processor) contains an `sc_export<IT>`, where `IT` is the C++ interface `struct IT {virtual void interrupt(bool new_signal_state)=0};`. The `interrupt` method of our ISS sets a boolean member `irq_pending` according to the new signal state and the interruption masking bits (e.g. bits F and I of the CPSR for ARM ISS), and notifies a SystemC event if required.

C. Parallelism and Scheduling

Each instance of ISS contains a SystemC process, such as most of the device models. A SystemC process must release control to the scheduler (e.g. through the wait() primitive), otherwise it keeps control and prevents other processes from executing. For example, the code "`while(!irq_pending){}`" is wrong since it would block the simulation if executed: since the other processes are not executed, they cannot generate an interruption.

Concerning our ISS, we could simulate very faithfully the parallelism by executing a wait after each instruction, followed by an interrupt test. Unfortunately, the wait instruction is very time costly (at most a few millions per second with the QuickThread library used by SystemC). We evaluate in section IV two solutions, that can be combined: 1) executing a wait instruction every N instructions; 2) placement of wait instructions based on the identification of logical *System Synchronization Points* as explained in [16].

IV. Experiments with ARM ISS

All experiments below are run on a Intel Quad@2.66GHz; the whole simulator is compiled with `g++-4.2 -O3`. The embedded software is cross-compiled with `arm-elf-gcc` version 4.1.1.

A. Application benchmark

We have developed a cryptographic benchmark using an open source library from the XYSSL project [17]. This benchmark encrypts and decrypts some data with the algorithms implemented by this library. Results are given by table I, for arm32 mode and thumb mode (16-bit instructions), for optimized and non-optimized embedded code. We have run GXemul [14] on the same benchmark.

These experiments show that the dynamic translation can accelerate the simulation by a factor of 10. When using DMI, SimSoC is more efficient than GXemul, which uses a similar dynamic translation technique, even though it uses SystemC/TLM interfaces and synchronization. In thumb mode, the same source program compiles to more instructions, hence a longer simulation duration whereas the speed expressed in Mips is similar to arm32 mode.

TABLE I

RESULTS FOR THE `crypto` BENCHMARK

	no dynamic transl.		with dynamic transl.		
	no DMI	DMI	no DMI[a]	DMI	GXemul
arm32 -O0	479 s 7.2 Mips	291 s 11.8 Mips	108 s 32 Mips	28.1 s 123 Mips	58.1 s 59.4 Mips
arm32 -O3	123 s 7.8 Mips	86.5 s 11.1 Mips	12.8 s 75 Mips	6.85 s 140 Mips	18.7 s 51.2 Mips
thumb -O0	1699 s 5.9 Mips	929 s 10.8 Mips	164 s 61 Mips	81 s 123 Mips	thumb mode
thumb -O3	275 s 5.9 Mips	161 s 10 Mips	21.6 s 75 Mips	14.7 s 110 Mips	not available

[a]excepted for the dynamic code translator

B. Transmission benchmark

We consider now a system composed of two subsystems linked by a model of null-modem cable; each subsystem contains an ARM processor, a bus, a memory and a model of UART, all described at the TLM level of abstraction. This system is represented on figure 2. The embedded software transmits data from one subsystem to the other, using software flow control based on CTS and RTS signals.

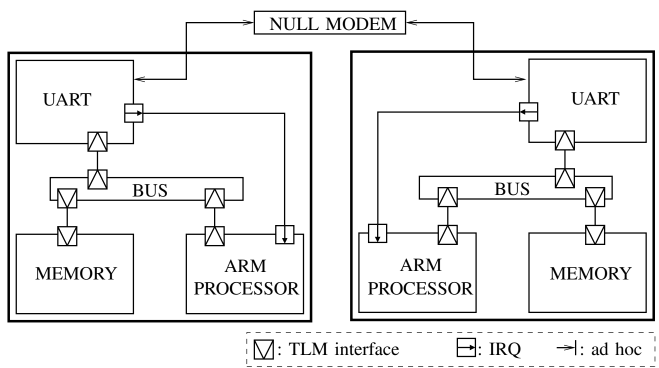

Fig. 2. Architecture of the transmission benchmark

The results displayed in table II show the influence of SystemC synchronization. Using a `wait` after every simulated instruction (most of these synchronization points are then useless), the speed transfer between the two UARTS reaches a maximum of 49 Kb/s. The speed reaches 1.46 Mb/s when synchronizing upon every 128 instructions. However a better result of 2.18 Mb/s can be obtained by detecting idle loops in the binary code to replace them with synchronization points and issuing the `wait` calls at appropriate places in transaction operations. With only one wait instruction every 256 instructions, we observe a wrong behavior, meaning that the simulation is not faithfull enough.

TABLE II

RESULTS FOR THE TRANSMISSION BENCHMARK

wait every N instructions				wait on send and idle loop
N=1	N=16	N=64	N=128	
42.1 s 49 Kb/s	3.78 s 550 Kb/s	1.89 s 1.10 Mb/s	1.42 s 1.46 Mb/s	0.950 s 2.18 Mb/s

V. CONCLUSION

We have presented in this paper the SimSoC simulation framework in order to run full system simulation, with a focus on the ISS technology. The SimSoC framework integrates into a single simulation engine SystemC/TLM hardware models with a dynamic translation ISS designed as a TLM model, remaining fully SystemC compliant, requiring no further synchronization with additional outside components.

A SimSoC ISS performs dynamic translation of the target code into an internal representation, using specialized functions to optimize performance. Our current developments of the technology are experimenting with further improvements of the simulation speed, in particular the idea of generating host machine code from the intermediate representation in a parallell thread. SimSoC is planned to be distributed as open source software.

REFERENCES

[1] M. Meerwein, C. Baumgartner, T. Wieja, and W. Glauert, "Embedded systems verification with fgpa-enhanced in-circuit emulator," in *ISSS '00: Proceedings of the 13th international symposium on System synthesis*. Washington, DC, USA: IEEE Computer Society, 2000, pp. 143–148.

[2] P. Gerin, S. Yoo, G. Nicolescu, and A. Jerraya, "Scalable and flexible cosimulation of soc designs with heterogeneous multi-processor target architectures," in *ASP-DAC '01: Proceedings of the 2001 conference on Asia South Pacific design automation*. New York, NY, USA: ACM, 2001, pp. 63–68.

[3] F. Fummi, G. Perbellini, M. Loghi, and M. Poncino, "Iss-centric modular hw/sw co-simulation," in *GLSVLSI '06: Proceedings of the 16th ACM Great Lakes symposium on VLSI*. New York, NY, USA: ACM, 2006, pp. 31–36.

[4] *SystemC v2.1 Language Reference Manual (IEEE Std 1666-2005)*, Open SystemC Initiative, 2005, http://www.systemc.org/.

[5] F. Ghenassia, Ed., *Transaction-Level Modeling with SystemC. TLM Concepts and Applications for Embedded Systems*. Springer, June 2005, iSBN 0-387-26232-6.

[6] *OSCI SystemC TLM 2.0, Draft 2 for Public Review*, Open SystemC Initiative, 2006, http://www.systemc.org/.

[7] M. Reshadi, N. Bansal, P. Mishra, and N. Dutt, "An efficient retargetable framework for instruction-set simulation," in *CODES+ISSS '03: Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. New York, NY, USA: ACM, 2003, pp. 13–18.

[8] M.-K. Chung and C.-M. Kyung, "Improvement of compiled instruction set simulator by increasing flexibility and reducing compile time," in *RSP '04: Proceedings of the 15th IEEE International Workshop on Rapid System Prototyping*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 38–44.

[9] B. Delsart, V. Joloboff, and E. Paire, "JCOD: A lightweight modular compilation technology for embedded Java," *Lectures Notes in Computer Science*, vol. 2491, pp. 197–212, 2002.

[10] M. Reshadi, P. Mishra, and N. Dutt, "Instruction set compiled simulation: a technique for fast and flexible instruction set simulation," in *Design Automation Conference, 2003. Proceedings*, 2003, pp. 758–763.

[11] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa, "Retargetable and reconfigurable software dynamic translation," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO03)*, 2003.

[12] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: a transparent dynamic optimization system," *SIGPLAN Not.*, vol. 35, no. 5, pp. 1–12, 2000.

[13] F. Bellard, "Qemu, a fast and portable dynamic translator," in *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2005, pp. 41–41.

[14] A. Garave, "Gxemul," 2007. [Online]. Available: http://www.gavare.se/gxemul/gxemul-stable/doc/index.html/

[15] H. Hongwei, S. Jiajia, C. Helmstetter, and V. Joloboff, "Generation of executable representation for processor simulation with dynamic translation," in *Proceedings of the International Conference on Computer Science and Software Engineering*. Wuhan, China: IEEE, 2008.

[16] J. Cornet, F. Maraninchi, and L. Maillet-Contoz, "A method for the efficient development of timed and untimed transaction-level models of systems-on-chip," in *DATE: Design, Automation and Test in Europe*, 2008.

[17] X. Community, "Xyssl cryptographic open source code," 2007. [Online]. Available: "http://xyssl.org/code/"