HAL
open science

SimSoC: A full system simulation software for embedded systems

Claude Helmstetter, Vania Joloboff, Hui Xiao

To cite this version:

Claude Helmstetter, Vania Joloboff, Hui Xiao. SimSoC: A full system simulation software for embedded systems. 2009 International Workshop on Open-source Software for Scientific Computation (OSSC-2009), LIAMA, Institute of Automation, CAS, Beijing, China Guizhou Normal University, China, Sep 2009, Guiyang, China. 7 p. inria-00435247

HAL Id: inria-00435247

https://inria.hal.science/inria-00435247v1

Submitted on 12 Jan 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

SimSoC: A full system simulation software for embedded systems

Claude Helmstetter
INRIA
Tsinghua University
Beijing, China
email: claude.helmstetter@inria.fr

Vania Joloboff
INRIA
Tsinghua University
Beijing, CHINA
email: vania.joloboff@inria.fr

Hui Xiao
INRIA
Tsinghua University
Beijing, CHINA
email: hui.xiao@formes.asia

***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 paper describes the overall architecture of the SimSoC full system simulator, a description of the ISS implementation and integration with some other components. A final section reports results obtained, in particular simulation of an existing System On Chip that can run the Linux operating system.**

I. Introduction

The development of embedded systems platforms requires increasingly large pieces of software running on complex System On Chips. A characteristics of embedded systems is that a new project (to design a new commercial product) combines new hardware with new application software. In order to save time to market, it is important that the software development can take place before the hardware development is completed. 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 product developers, the simulation environment is more valuable if it can achieve full system simulation, that is, it runs the exact binary software that will be shipped with the product, including the operating system and the embedded application. This simulation must be sufficiently fast so that software developers can run tests with reasonable response time to support an effective iterative development cycle. This precludes the use of cycle accurate simulation as used in hardware development as it is much too slow.

In many projects, the hardware is re-used from former project, but with evolutions or new components added. Hence, it is also necessary to support simulation of new hardware components with enough detail, possibly using simulation models coming from third party IP providers.

These requirements call for an integrated, modular, full simulation environment where already proven components, possibly coming from third-parties, 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 including 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

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

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, which is late in the project.

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 inter-process 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. 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.

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. Expressing the semantics of SystemC in a TLM context has been investigated in [7].

B. Instruction Set Simulation

An instruction-set simulator (ISS) is used to mimic the behavior of a target computer processor on a simulation host machine. The main task of an ISS is to carry out the computations that correspond to each instruction of the simulated program. There are several alternatives to achieve such simulation. In *interpretive simulation*, each instruction of the target program is fetched from memory, decoded, and executed, as shown in Figure 1. This method is flexible and easy to implement, but the simulation speed is slow as it wastes a lot of time in decoding. Interpretive simulation is used in Simplescalar [8].

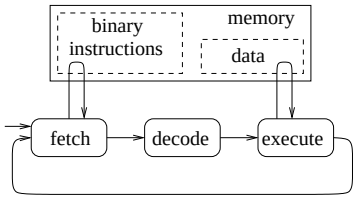

Fig. 1. Interpretive simulation

A second technique is compiled simulation (see Figure 2), also called *static translation*. The application program is decoded in a preliminary compilation phase and translated into a new program for the simulation host. The simulation speed is vastly improved [9], [10], but it is not as flexible as interpretive simulation. The application program must be entirely known at compile time, before simulation starts. This method is hence not suitable for application programs which will dynamically modify the code, or dynamically load new code at run-time, or applications like Java Virtual Machines that include a JIT compiler itself generating new code [11].

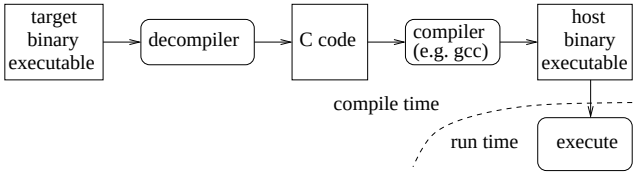

Fig. 2. Compiled simulation

A third technique to implement ISS is *dynamic translation* [12]–[14]. With dynamic translation (see Figure 3), the target instructions are fetched from memory at runtime, like in interpretive simulation. They are decoded on the first execution and the simulator translates these instructions into another representation which is stored into a cache. On further execution of the same instructions, the translated cached version is used. If the code is modified during run-time, the simulator invalidates the cached representation. Dynamic translation combines the advantage of interpretive simulation and compiled simulation as it supports the simulation of programs that have either dynamic loading or self-modifying code,

In the past decade, dynamic translation technology has been favored, such as [15]–[17] . The target code to be executed

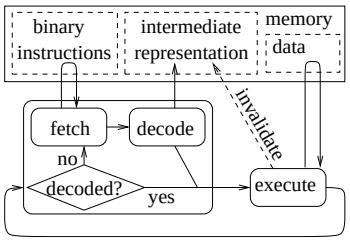

Fig. 3. Dynamic translation

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.

C. Virtual Machine

Full system simulation is also achieved in so called Virtual Machines such as QEMU [18] and GXemul [19] 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 4. 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.

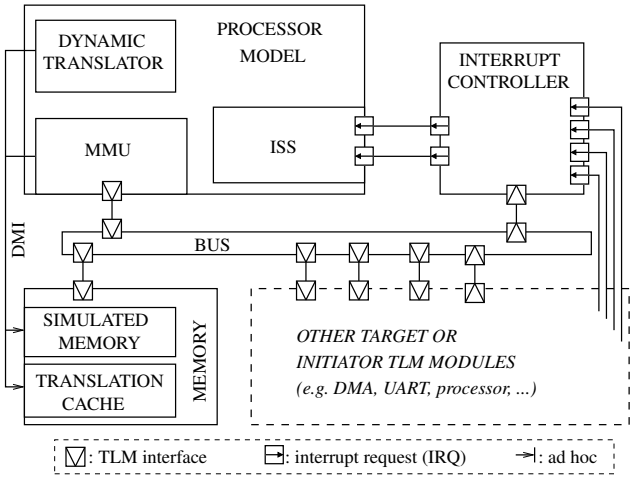

Fig. 4. SimSoC architecture

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. SimSoC ISS

In order to compare different techniques, and to provide different levels of trade-offs of accuracy vs. speed, we have implemented three kinds of instruction simulation corresponding to three modes that the simulator can run.

The first mode, named M0, is interpretive simulation. This is the basis from which we can compare performance. The second mode (M1) is dynamic translation with no specialization. This mode shows the performance improvement obtained with dynamic translation compared to interpretive simulation. The third mode (M2) is dynamic translation with specialized pseudo instructions as described below. This mode shows the performance improvement obtained with specialization over standard dynamic translation as in M1 mode.

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 [19]. 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.

Partial evaluation is a compiling optimization technique, also known as specialization [20]. 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 specifically for this data. Specialization can be advantageously used in processor simulation, because data can often be computed at decoding time, and a specialized version of the generic instruction can be used to execute it.

The simulation code then uses fewer tests, fewer memory accesses and more immediate instructions. This technique has been used to some extent in the IC-CS simulator [21].

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 SimSoC binary decoder can be 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.

As 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, which can be generated by a code generator. This code 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. 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 [22], 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 a few 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-2 standard [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-2 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-2 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. MMU Simulation

The Memory Management Unit (MMU) of a microprocessor is the hardware component that controls memory access and enforces the policy set by the application software, typically the operating system. On every memory access, the MMU checks whether it is a valid access, otherwise routes the instruction to an exception mechanism. Additionally, on some processors, such as ARM and PowerPC, the MMU performs the translation of virtual addresses to physical addresses. Because it is involved on each memory access, it is a critical element in the overall performance of a microprocessor and of a simulator.

The main task of the MMU is to find whether or not a virtual address is mapped into real memory, and check its access rights. It does this by constructing an associative table named the Table LookAhead Buffer (TLB) associating for each entry an address to the real memory location and the access rights defined by bits in the TLB entry.

Initially, the MMU is inactive. When the software starts-up the MMU, it must have prepared the page tables and the TLB so that the MMU will find the destination address in

the TLB and table walk. The hardware associative search is done simultaneously for all entries in the table. If the table look up fails, an exception is raised named *TLB miss*. On some architectures, when the TLB search fails, the hardware performs itself a page table walk. Then the MMU also has a pointer on the page table location in memory so that it can search for the requested entry. If that table walk also fails, then an exception is raised, allowing the software to terminate the program or provide a new page table. If the location is found it is added to the TLB. The TLB fills up progressively. When the TLB is full and a new memory access is required some older reference is trashed out.

An issue with regards to the virtual to physical memory mapping is the MMU simulation speed. The MMU hardware associative TLB search is a constant time operation, but a software table search is dependent upon the TLB size. Whereas hardware performance increases with a larger TLB size, the simulator performance may decrease with larger TLB size...

However there are two aspects in the MMU simulation depending on the simulation goals. If the goal is to run the application software to verify its functional properties, measuring the TLB misses is not important. In that case, it is not necessary to simulate the TLB with its exact size.

A constant time performance can be achieved using a large TLB mapping the entire address space. Indeed, the virtual memory space is very large but finite. The number of 4K bytes pages on a 32 bits architecture is at most 2, therefore using a TLB of 2 entries (4 Megabytes of memory) makes it possible to implement a MMU TLB lookup in constant time. This in principle does not scale up well for 64 bits architecture. But in fact, in embedded systems applications, even if the address space is using 64 bits, the amount of memory actually used by the application remains limited. Therefore it is possible to build a two level sparse table with segments (of 4 Gigabytes each) mapped by the first level table, and still achieve constant time lookup.

SimSoc implement both a configurable fixed-size TLB for the applications that want to measure the TLB misses, and the wholly mapped address space with constant time lookup.

Another aspect of memory management in simulation using dynamic translation is the coherency between the translation cache and simulated memory. If a program is modified, for example because the operating system is paging out this code and paging in new code. The translated code becomes obsolete, and presumably new translated code will be generated when the new code is activated. The simulator must hence detect memory write operations that overwrite previously translated code and the corresponding translated cached code must be invalidated.

It is not efficient to check for every memory store instruction if the destination address corresponds to one of the cached instructions. The technique used in SimSoC consists in using the simulation host MMU to perform this. Whenever, some code is translated, the page storing the code on the simulation host machine is marked as read-only. Whenever a simulated instruction is attempting to write into such location, the host operating system raises an exception. The exception handler can then flush the translation cache, modify the access bit to let the write operation terminate.

D. 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 [23].

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 [24]. 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 [19] on the same benchmark.

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

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.

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 5. The embedded software transmits data from one subsystem to the other, using software flow control based on CTS and RTS signals.

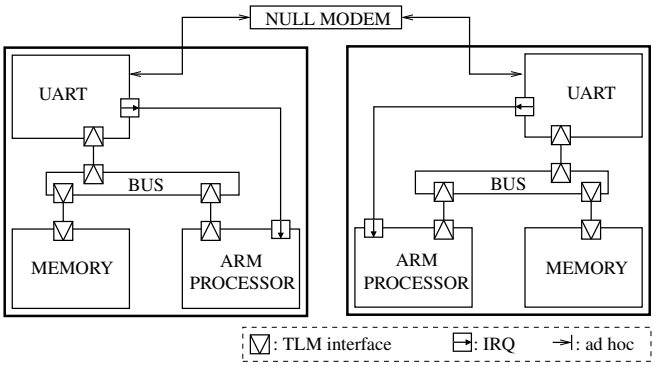

Fig. 5. Architecture of the transmission benchmark

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 faithful 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	3.78 s	1.89 s	1.42 s	0.950 s
49 Kb/s	550 Kb/s	1.10 Mb/s	1.46 Mb/s	2.18 Mb/s

C. System On Chip simulation

As a proof of concept, we have developed a simulator to simulate a commercially available System-On-Chip, namely the SPEAr Plus600 circuit from ST Microelectronics. This SoC contains among other components two ARM926 subsystems (dual core), together with ARM UART and interrupt controllers, and many additional components, there are over 40 components in the SoC. We have developed functionally accurate simulators for all components directly necessary to boot Linux, in particular the interrupt controller, the UART, the Ethernet controller, the memory controller, the NAND flash memory controller and the serial memory controller. For other components that are not directly used by the Linux operating system, we have built *stub* simulation components such that the Linux drivers don't crash, although the simulation is not accurate.

The Linux operating system for this SoC is available from ST Microelectronics. Therefore it is possible to test the simulator by running this Linux kernel binary software on the simulator.

The SPEAr Plus simulator based on SimSoC simulates the serial memory that contains the compressed Linux boot, as the real device. The bootloader reads the Linux kernel from serial memory, uncompresses it and finally starts Linux.

The Linux operating system then boots in a few seconds and networking commands such as `ping` can be used effectively.

A simulated SoC can be connected to another simulator using TCP/IP protocol, this simulator running on the same machine or on a remote machine, thanks to tunneling Ethernet packets to the remote simulator.

D. Other Properties

The SimSoC simulator can be built entirely with open source software. It can be built with the open source SystemC library, and uses the GNU tools to build. We expect to release SimSoC as open source software when the code has reached stability.

The simulator can be connected to any debugger using the GDB protocol to debug simulated programs, although it does not support yet Linux kernel debugging.

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.

A complete simulator has been developed and tested for the ARM5 instruction set. Two more ISS'es are under development for the PowerPC and MIPS, for which the M0 and M1 mode have been developed, but not yet the M2 mode. Actually we are investigating an M3 mode that will be much faster using native code compilation technique.

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 v2.2 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 User Manual*, Open SystemC Initiative, 2008, http://www.systemc.org/.

[7] F. Maraninchi, M. Moy, J. Cornet, L. Maillet Contoz, C. Helmstetter, and C. Traulsen, "SystemC/TLM Semantics for Heterogeneous System-on-Chip Validation," in *2008 Joint IEEE-NEWCAS and TAISA Conference*, IEEE, Ed., Montréal Canada, 06 2008, p. unknown. [Online]. Available: http://hal.archives-ouvertes.fr/hal-00311011/en/

[8] SimpleScalar LLC, "SimpleScalar," 2004, http://www.simplescalar.com/.

[9] 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.

[10] 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.

[11] 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.

[12] B. Cmelik and D. Keppel, "Shade: a fast instruction-set simulator for execution profiling," in *SIGMETRICS'94*. New York, NY, USA: ACM, 1994, pp. 128–137.

[13] E. Witchel and M. Rosenblum, "Embra: fast and flexible machine simulation," *SIGMETRICS Perform. Eval. Rev.*, vol. 24, no. 1, pp. 68–79, 1996.

[14] E. Schnarr and J. R. Larus, "Fast out-of-order processor simulation using memoization," *SIGOPS Oper. Syst. Rev.*, vol. 32, no. 5, pp. 283–294, 1998.

[15] 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.

[16] 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.

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

[18] 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.

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

[20] Y. Futamura, "Partial evaluation of computation process—an approach to a compiler-compiler," *Higher Order Symbolic Computation*, vol. 12, no. 4, pp. 381–391, 1999.

[21] A. Nohl, G. Braun, O. Schliebusch, R. Leupers, H. Meyr, and A. Hoffmann, "A universal technique for fast and flexible instruction-set architecture simulation," in *DAC'02*. New York, NY, USA: ACM, 2002, pp. 22–27.

[22] 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.

[23] 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.

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