Portable SystemC-on-a-Chip

Abstract

SystemC allows description of a digital system using traditional programming features as well as spatial connectivity features common in hardware description languages. We describe an approach for in-system emulation of circuits described in SystemC. The approach involves a new SystemC bytecode format that executes on an emulation engine running on the microprocessor and/or FPGA of a development platform. Portability is enhanced via a USB flash-drive approach to loading the bytecode format onto the platform. Performance is improved using emulation accelerators on an FPGA. We describe our SystemC-to-bytecode compiler, bytecode format, emulation engine, and emulation accelerators. We illustrate use of the approach on a variety of examples, showing easy porting of a single application across various platforms, and showing emulation speed on an FPGA that is comparable to SystemC execution on a PC.

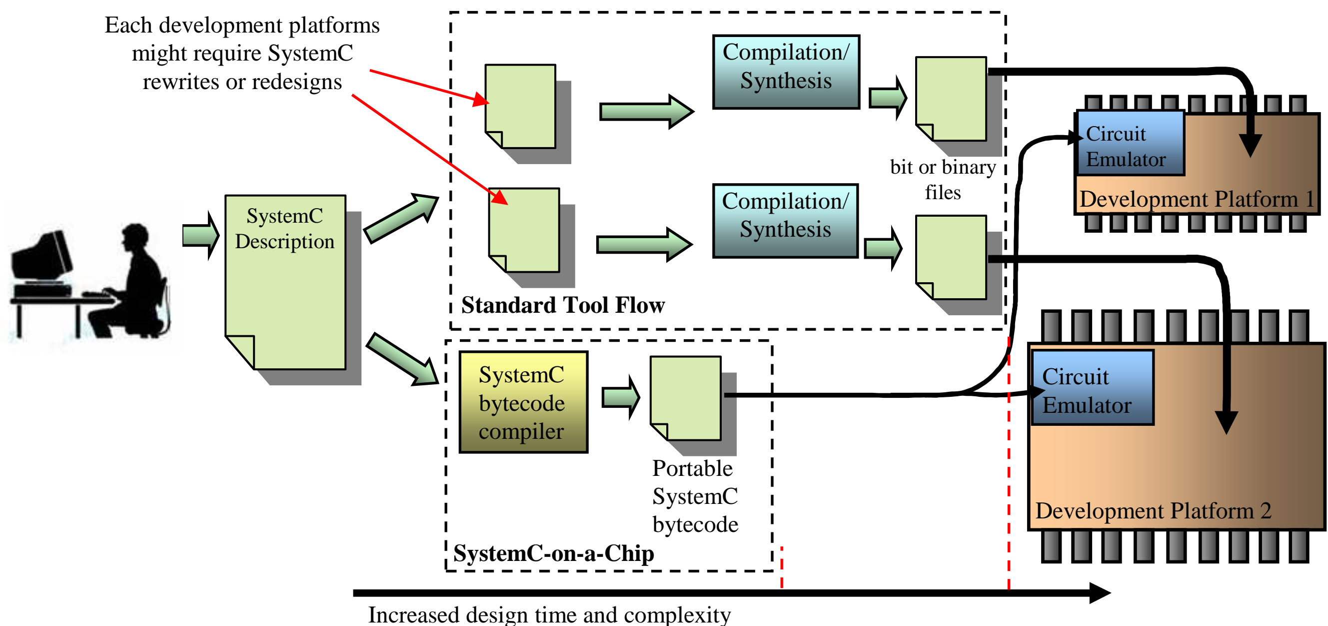
1. Introduction

SystemC [26] represents a digital system description approach based on C++. SystemC uses object-oriented features of C++ to enable descriptions that include features common in previous hardware description languages (HDLs), such as creation of components, instantiation and connection of components to form a circuit, and precisely-timed communication and execution among concurrently-executing components, all using existing C++ syntax. Regular C++ code can be included in descriptions, and SystemC also provides a thread library, thus supporting description of both the "software" (sequential instructions coupled with parallel threads) and "hardware" (circuit) parts of an entire system in a single description language.

While a SystemC description can be executed on a PC for simulation purposes before eventually synthesizing the description to an ASIC, FPGA, or board-level customized implementation, in-system SystemC emulation, wherein the executing description would interact with physical inputs and outputs (I/O), would also be useful. In-system emulation is common for embedded processors. Though slower than a custom implementation, emulation enables early prototyping, and benefits from real I/O rather than fabricated I/O in simulation, whose creation can difficult and time-consuming while still not matching the complexity and nuances of real I/O. Emulation can be especially useful for SystemC, as illustrated in Figure 1, due to the fact that synthesis tools can be expensive (compared to compilers), may only run on limited PC platforms and be challenging to install (especially on lower-end PCs), may be unpredictable with respect to circuit size/speed or tool runtime, often require long runtimes (such as hours or days), may not support particular target devices or platforms, and can only synthesize the parts of the code written for synthesis. The tradeoff is that the emulation engine must be present on a target platform, but this is a one-time task, which may be done by the platform's developers or by platform users (such as teaching assistants in an educational setting).

For education, where system execution speed may not be a top priority, emulation may be entirely sufficient, such as when describing a microprocessor system as is commonly done in computer architecture courses, where such descriptions may never be intended for synthesis, but execution on a physical platform is desired. In fact, for some systems (in education

Figure 1: SystemC-on-a-Chip allows a designer to emulate SystemC descriptions on various supported development platforms. Emulation enables early prototyping and interaction with real peripherals and I/O, while reducing the need for advanced compilation and synthesis.



settings or otherwise), emulation may be fast enough to serve as a final implementation, obviating the need for synthesis, akin to virtual machines sometimes being sufficient for executing processor bytecode such as Java bytecode. For example, a "reaction timer" system may involve several interacting components interfacing with buttons, LEDs, and LCDs, with emulation speed being fast enough to interact with all these items. In such cases, SystemC ultimately represents a parallel programming approach such as an approach using POSIX threads, with the added benefit of supporting circuit-style spatial connectivity, but the drawback of not (presently) supporting real-time scheduling as in a real-time operating system approach.

We introduce an approach to SystemC emulation, involving several parts. We created a compiler to convert SystemC to a new bytecode format that we developed, which possesses MIPS-like instructions supplemented with new SystemC-specific instructions. We developed an emulation engine that can run on a microprocessor on a development platform and that executes the SystemC bytecode while interacting with I/O and (optional) peripherals (frame buffers, UART, etc.). Because portability is important in the approach, we introduce a USB flash-drive method for programming, wherein the compiler-generated textual bytecode file is copied to a USB flash-drive, which is then read by the development platform and just-in-time translated to the machine-level bytecode used by the emulation engine. For the common situation where the emulation engine is implemented on (or with access to) an FPGA, we developed FPGA-based custom emulation accelerators that substantially increase the emulation speed, enabling SystemC execution speeds comparable to middle-to-high-end PCs.

The rest of the paper is organized as follows. Section 2 describes related work. Section 3 describes the SystemC-on-a-Chip emulation framework. Section 4 summarizes experiments and results. Section 5 concludes.

2. Related Work

There has been previous work in capturing applications and circuits to increase portability. Andrews [1][2] focuses on creating operating system and middleware abstractions that extend across the hardware/software boundary, enabling a designer to create applications for hybrid platforms with one executable. Levine [14] describes hybrid architectures with a single, transformable executable. They argue that an executable described for a queue machine (converse of a stack machine) makes runtime optimizations to a specialized FPGA fabric feasible. Moore [16] describes writing applications that dynamically bind at runtime to reconfigurable hardware for the purposes of portability. Similar to Andrews [1][2], the authors develop hardware/software abstractions by writing middleware layers that allow application software to utilize reconfigurable DSP cores. Vuletic [31] proposes a system-level virtualization layer and a hardware-agnostic programming paradigm to hide platform details from the application designer and lead to more portable circuit applications. Sirowy [23] shows that while many manually created published circuits can be captured in a temporal language like C for portability benefits, there are still circuits that require explicit spatial constructs, and that can't readily be captured in temporal languages.

There has also been some research in the field of hardware emulation for verification and testing, including the BEE reconfigurable platform [6], and network-on-chip emulation platforms [11]. Nakamura [18] describes a hardware/software verification platform that uses shared register communication between a processor simulator and FPGA emulator. Benini [3] describes virtual in-circuit emulation of SystemC circuits for co-verification and timing accurate prototyping. Rissa [21] evaluates the emulation speeds of several SystemC models compared to standard HDL models.

Much research has involved virtualization [15][24], with several commercial products developed in response to the need for portable virtual machines. VMware [30] and the open source product Xen [32] concentrate on developing virtual machines that allow the end-user to run multiple operating systems concurrently. The Java Virtual Machine [25] allows the programmer to write operating system independent code, and tools like DOS Box and console emulators allow the user to run legacy applications in modern operating systems. Fornaciari [10] extends virtualization to FPGA platforms, giving the application designer a virtual view of an FPGA that is then physically mapped via operating system functionality. Some work has focused on accelerating Java bytecode through the design of custom bytecode accelerators [12][19]. Virtualization has also been used to abstract complex microcontroller details from the beginning embedded systems student [22].

There are a number of models of computation and circuit capture methods. Brown [4] shows that a parallel model of computation requires machine primitive units, control constructs, communication mechanisms, and synchronization mechanisms. Circuits are usually captured in a hardware description language (HDL), like SystemC [26], Verilog [28], or VHDL [29], although circuits can also be captured using schematics. Our work focuses on the *synthesizable subset* of SystemC [27], involving the language constructs allowed in SystemC synthesis tools like Cadence [5] and Coware [7].

SystemC-on-a-Chip uses virtualization techniques to achieve portable SystemC applications on any development platform that can support an in-system emulation engine. Our portability approach doesn't require O/S support, and relies on explicit parallel constructs like signals and spatial connectivity.

3. SystemC-on-a-Chip Platform

The SystemC-on-a-Chip platform consists of five parts, including a SystemC bytecode compiler, a new intermediate SystemC bytecode format, a portable USB flash drive download interface, an emulation engine, and FPGA-based emulation acceleration units.

3.1 SystemC Bytecode Compiler

We considered several options to achieve in-system emulation of SystemC descriptions. One approach was to port the publicly available SystemC libraries to each development platform, and add support for I/O and peripheral interaction. Such an approach would allow the same SystemC binary to run on any supported development platform, including standard PCs. Also, the SystemC circuit would run natively on the development platform's microprocessor. However, the SystemC libraries are large and require OS support, thus limiting the number of platforms that could support the SystemC-on-a-Chip framework. Furthermore, the SystemC libraries build a simulation kernel into the circuit executable, increasing the size of the executable and

Figure 2: SystemC Bytecode Compiler: **(a)** The SystemC bytecode compiler builds on PINAPA, a SystemC front-end tool, and uses a custom SystemC bytecode backend; **(b)** Sample code generation during the first phase of the SystemC bytecode back end.

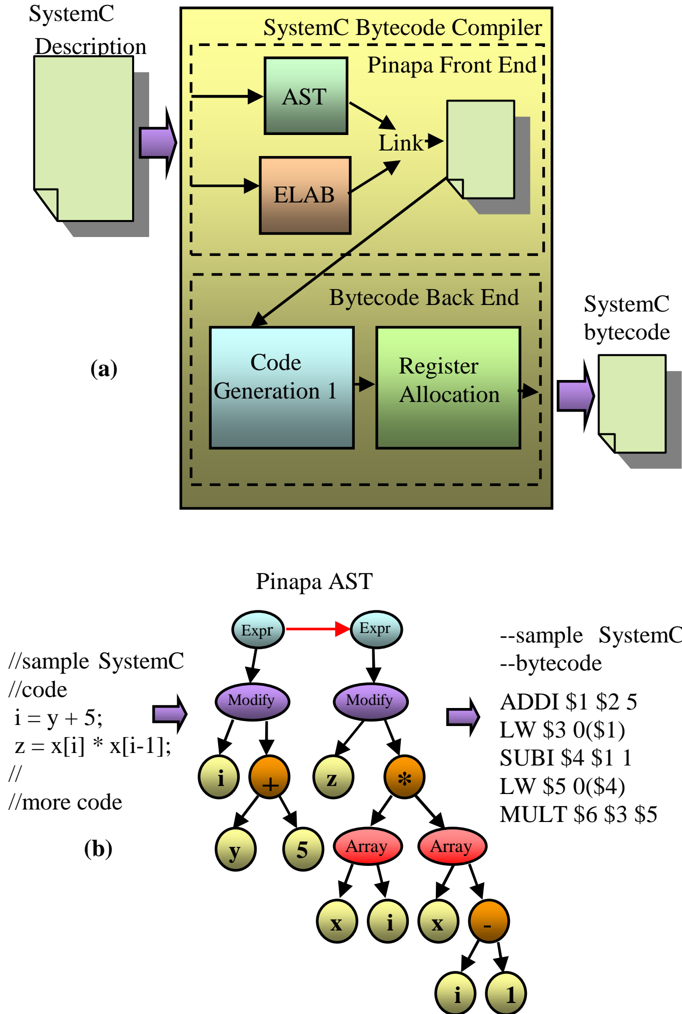


Figure 3: SystemC Bytecode Format. Each process is described by a number of MIPS-like instructions, with additional instructions added for SystemC specifics, like reading signals, extracting bit ranges, etc.

circuit: signals processes
signals: signal or
signals signal
processes : process or
processes process
signal : SIGNAL NAME COLON NUMBER
process : PROCESS sensitivity_list code
sensitivity_list: NAME or
sensitivity_list NAME COMMA
code: instruction or
code instruction
instruction:

SRL *or* SLL *or* SLLV *or* SRLV
or MULT *or* MFLO *or* ADD
or SUB *or* AND *or* OR *or* ADDI
or ANDI *or* ORI *or* XORI
or SUBI or LW *or* SW
} Computation instructions

or J *or* JR *or* BEQ *or* BNE
or BLE *or* BGT *or* BLT *or* BGE
} Control instructions

or BIT *or* RANGE *or* READ
or WRITE *or* CONCAT *or* WAIT
or END
} SystemC-specific instructions

making testing multiple SystemC descriptions quickly more difficult.

Another option was to decompile the SystemC executable, extract the circuit, and retarget that circuit for a custom emulation framework. The decompilation approach separates the circuit from the simulation kernel, allows testing multiple circuits quickly, and potentially a smaller circuit executable. A custom emulation framework also allows smaller development platforms to take advantage of in-system SystemC emulation. However, decompilation is difficult, and solutions that operate at the source SystemC level seemed more feasible.

The option that we chose was to directly operate from SystemC source code to produce bytecode, as shown in Figure 2. Our SystemC compiler builds upon the *PINAPA* tool [17]. Originally intended as a front-end for circuit verification tools, PINAPA provides a *gcc* compiler front-end to SystemC circuits that extracts a circuit's spatial and architectural features from the SystemC description.

The PINAPA front-end performs two operations on the SystemC program. PINAPA uses a modified version of the *gcc* compiler to extract behavioral information about each process and component in the circuit to generate the corresponding abstract syntax trees (AST), and uses a modified SystemC kernel to extract the circuit's architectural features, like ports, signals, and spatial connectivity. Finally, PINAPA links the architectural description (ELAB) to each component's AST to form the intermediate output.

We created a custom two-pass back-end to the PINAPA compiler that accepts PINAPA's AST+ELAB output and generates SystemC bytecode. The first pass traverses each ELAB component's AST. The first pass inlines auxiliary functions, flattens hierarchical descriptions, and generates initial SystemC bytecode assuming an infinite amount of available registers, shown in Figure 2(b). The second pass performs a linear scan register allocation [20] on the first pass output to constrain the intermediate code to a fixed number of registers. The output of the register allocation pass is a readable text file of the SystemC description in SystemC bytecode.

3.2 SystemC Bytecode Format

The SystemC-on-a-Chip platform accepts a bytecode version of the SystemC description, and not a traditional SystemC binary, nor the SystemC source code. A traditional SystemC binary includes much more information than is actually required to

emulate the application, including constructs to support object-oriented C++ programming, and the simulation kernel. SystemC source code separates the circuit from the simulation kernel, but requires compiler support on each development platform. Similar to Java bytecode and a Java Virtual Machine, an intermediate SystemC bytecode format separates the SystemC description behavior from the simulation kernel, doesn't require a platform compiler, and can run on any development platform that supports the SystemC bytecode format.

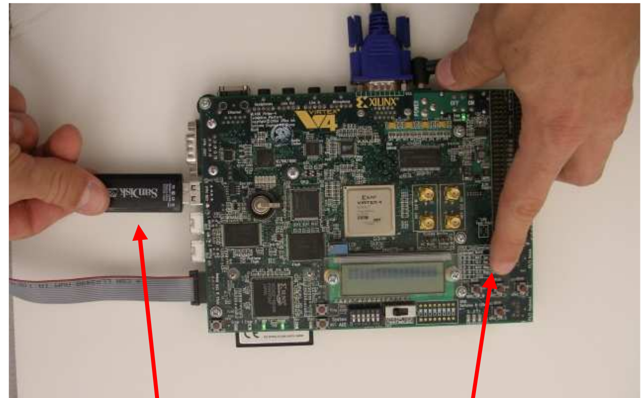
The format of the SystemC bytecode is shown in Figure 3. The SystemC bytecode is a flattened version of the original SystemC description. The SystemC bytecode compiler flattens the SystemC description to more efficiently emulate the SystemC bytecode. A SystemC *circuit* is composed of a list of signals and a list of processes. A *signal* is a wire or set of wires that connects independently running processes, and is defined by a signal name and bit width. A *process* is a behavioral description of a circuit entity. A process is defined by a *sensitivity list,* a list of signals the process is sensitive to, and a list of sequential instructions which define the process's behavior.

A process is captured as a sequence of sequential instructions. The SystemC bytecode instructions are a derivative subset of the MIPS RISC register machine instruction set [13], shown in the bottom half of Figure 3. We also considered targeting virtual stack or queue machines. The Java Virtual Machine [25] executes bytecode instructions intended for a stack machine, and [14] executes bytecode instructions for a queue machine. Proponents of stack and queue based bytecode formats argue that the stack/queue bytecode can more efficiently run on a virtual machine because the operands are implied. Other studies [8] have shown that the advantages of stack machines aren't as clear. The authors show the bytecode targeted towards a register machine can be competitive with stack machine code, and usually results in more compact code. An additional advantage is that register bytecode is more readable, potentially allowing a student to write bytecode in the absence of a SystemC bytecode compiler.

The SystemC bytecode format supports three different types of instructions: computation/memory instructions, control instructions, and SystemC-specific instructions. The computation and control instructions are derived from the MIPS instruction set [13]. We chose the RISC MIPS instruction set because the SystemC bytecode is easy to generate, because a RISC-based emulator can be efficient [8], and because the code is understandable to the beginning student. We also chose a representative subset of the MIPS instructions that would allow specifying all circuits described in the synthesizable subset of SystemC[27].

We added a number of SystemC-specific instructions to the base MIPS instruction set, including the *BIT, RANGE, READ, WRITE, and WAIT* instructions. The *BIT* and *RANGE* instructions extract either one bit or a range of bits from a given register. The *READ* and *WRITE* instructions allow a process to read and write signals, much as the process can load or store values to memory. We added the SystemC-specific instructions to more efficiently execute frequently occurring SystemC primitives and function calls. Most of the SystemC-specific instructions could have been implemented as a sequence of the basic computation instructions except for the WAIT instruction. The *WAIT* instruction allows a SystemC description to wait a

Figure 4: USB interface. The user copies SystemC bytecode to a USB flash drive, plugs the drive into a platform and pushes a button—the platform then begins emulating the SystemC description.



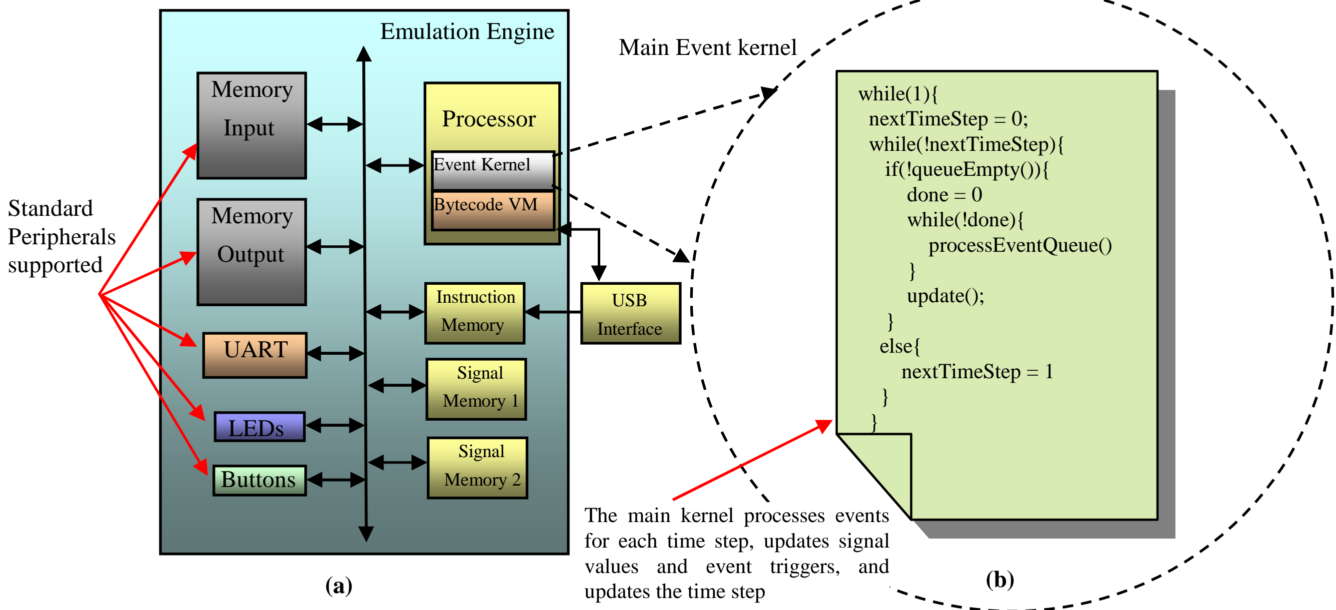
fixed number of simulated time steps. The WAIT statement is the only supported feature that does not follow the synthesizable SystemC guidelines, but allows designers to test their SystemC applications with custom bytecode test benches. The *END* instruction instructs the emulation engine that a process is done executing.

3.3 USB Download Interface

Our SystemC-on-a-Chip platform supports USB programming via a USB flash drive, rather than a traditional hardware programmer or USB cable. A traditional hardware programmer requires non-volatile memory and a removable chip, greatly limiting the number of supportable development platforms. An alternative programming approach is to program a device in-system using a USB cable. While eliminating the need for a programming device, such an approach still requires a PC every time a designer wishes to load a new SystemC description.

Instead, we chose a USB flash drive programming approach, illustrated in Figure 4. A user (such as a student) copies the desired SystemC description (in bytecode format) onto a USB drive as a file, plugs the drive into the SystemC-on-a-Chip platform, and presses a button on the platform that downloads the program from the flash drive to the internal emulation engine. The approach eliminates the need for non-volatile memory in the development platform. The approach enables loading and changing circuits by inserting and swapping flash drives, enabling more mobility and portability. The approach also matches current usage schemes for popular electronic devices, allowing a beginning student to start programming with minimal effort, and using a familiar paradigm. The cost is that the SystemC-on-a-Chip platform must contain an internal USB flash drive reader.

Figure 5: Basic Emulation Engine. The emulation engine consists of a hybrid event/time driven kernel to allow a variety of different circuits to be implemented. Circuits can also take advantage of a range of standard peripherals, including lights, buttons, a UART, and input and output memories.



3.4 Emulation Engine

The basic emulation engine supports SystemC bytecode written or generated for the synthesizable subset of SystemC. We currently do not support higher level features of SystemC like transaction level and system level modeling because we are presently targeting SystemC descriptions that could eventually run natively on an FPGA. Figure 5(a) shows the architecture of the basic emulation engine.

The basic emulation is driven by a processing core that runs a lean, event-driven simulation kernel [9]. Figure 5(b) shows the pseudocode for the event-driven kernel. For each time step, the event-driven kernel processes a queue of ready-to-run *events*. An *event* is placed on the queue when a signal value is updated and that signal is on the sensitivity list of a process. Each time step might consist of multiple *delta* time steps, in which a process may execute multiple times during a time step. After each delta step, the event kernel updates the signal values, and places any new sensitive processes onto the event queue. The signals values are located on the system bus in *Signal Memory 1* and *Signal Memory 2*. Processes and peripherals write to *Signal Memory 1*, and read from *Signal Memory 2*. After each delta step, the event kernel copies the contents of *Signal Memory 1* to *Signal Memory 2*. The advantage of putting the signal memories on the bus is that peripherals have easy access to the signal values, and gives access to emulation accelerators. The disadvantage is that multiple peripherals might try to access the signal memories at the same time as the event kernel, blocking the bus, and degrading emulation efficiency.

The event-driven kernel calls a bytecode virtual machine to execute each event in the event queue. The *bytecode virtual machine* supports the SystemC bytecode instruction set described in Section 3.2. Each process is allocated an instruction memory, register file, and local data memory. The virtual machine also contains proper hooks to communicate with the standard peripheral and I/O set. We designed the bytecode virtual machine using standard techniques from [24] to increase the efficiency of execution.

The emulation engine supports platform I/O and peripheral access. The set of peripherals includes buttons, LEDs, UART, and input and output memories. We chose the peripherals to be a representative subset of peripherals that most development platforms could support. For development platforms with a larger set of peripherals, emulation designers could easily add extra support. The basic emulation engine supports SystemC descriptions that implement the interface shown in Figure 6. The description writer does not have to follow the standard interface, but the standard interface provides a convenient mapping between description's signals and the available peripherals.

3.5 Emulation Engine Accelerators

For the common situation where the SystemC-on-a-Chip platform is implemented on an FPGA, we've developed

Figure 6: SystemC-on-a-Chip Circuit Interface. The emulation engine supports access to multiple peripherals, including buttons, LEDs, and memory.

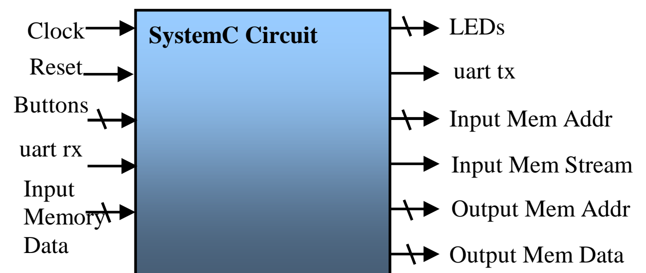
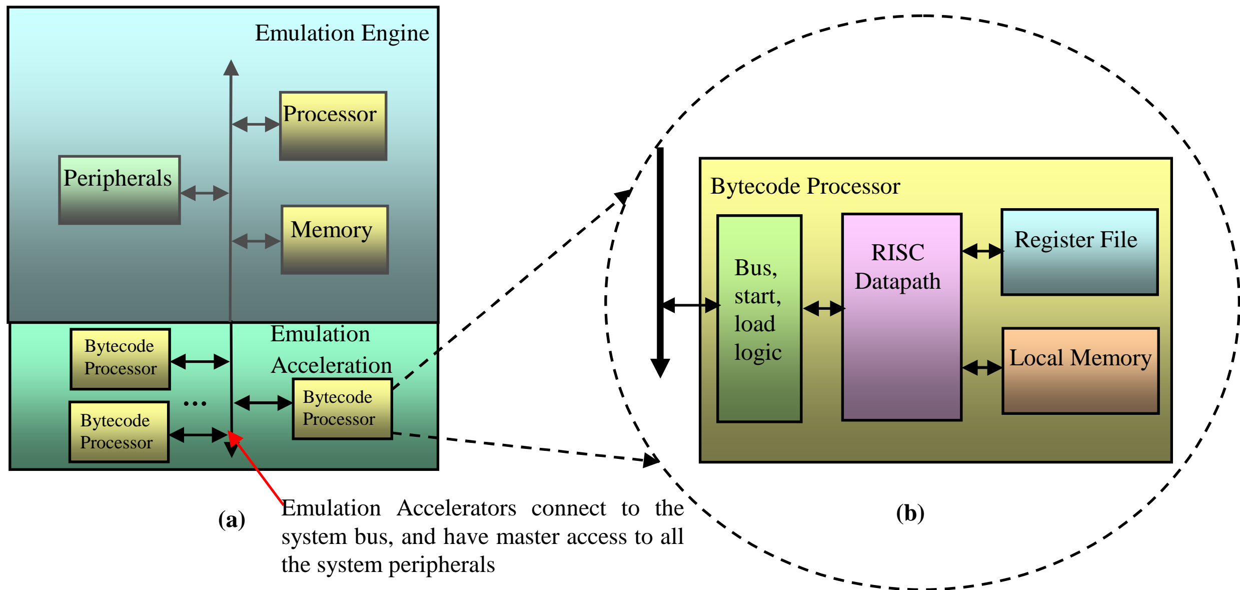


Figure 7: Emulation Accelerators. The emulation accelerator consists of a multicycle MIPs-like datapath than can execute one instruction in about 3-4 cycles, almost 100X faster than executing the same instructions in the base emulator.



emulation accelerators that substantially increase the SystemC emulation speed. Figure 7(a) shows multiple emulation accelerators connected to the basic emulation engine. Each emulation accelerator runs in parallel to the other emulation accelerators and the main emulation processor. Figure 7(b) shows the internals of one of the emulation accelerators. The emulation accelerator consists of a small SystemC bytecode processor and bus steering logic. The bytecode processor is a modified multi-cycle MIPS datapath, with connections to a register file and local data memory. The emulation accelerator can complete most instructions in 3-4 cycles, with the exception of the READ instruction, which has a nondeterministic execution time since the accelerator must read data from the system bus. The emulation accelerator is configured as a master on the system bus to allow the accelerator to read and write the emulation engine's signal memories independent from the emulation processor, and as a slave to allow the emulation processor to command the start of its execution.

The emulation accelerators are statically mapped to execute one process during the course of a SystemC description execution. The emulation processor maps processes to emulation accelerators using a simple instruction-size priority scheme. The advantage of the static priority approach is the emulation software is kept simple and doesn't need to stop execution to reconfigure the emulation accelerators. The disadvantage is there may be situations where dynamically swapping out processes onto the emulation accelerators might result in faster emulation execution times. Also, since the size of the emulation accelerator's instruction and data memories are fixed, the main scheduling/mapping heuristic accounts for the limited memory sizes by only mapping processes to accelerators that are guaranteed to fit. The main event-driven kernel is modified to either simulate a process in software, or to make the corresponding call to the correct emulation accelerator.

The number of emulation accelerators can substantially increase the performance of the SystemC emulation since each emulation accelerator runs in parallel. The emulation accelerators do contend for the signal memories, but typical SystemC behavioral descriptions only read/write signals at the start and end of their descriptions. The advantages of emulation accelerators increase as the size of the SystemC processes increase since the emulation accelerator can execute bytecode instructions orders of magnitude faster than the basic emulation engine can. There are tradeoffs though. Assuming circuit emulation doesn't require fast execution, the FPGA area required to implement emulation accelerators could be allocated for other circuitry, including more advanced peripherals or I/O. Also, smaller process descriptions may not benefit much from emulation acceleration, or other SystemC execution times might be perfectly acceptable in without acceleration

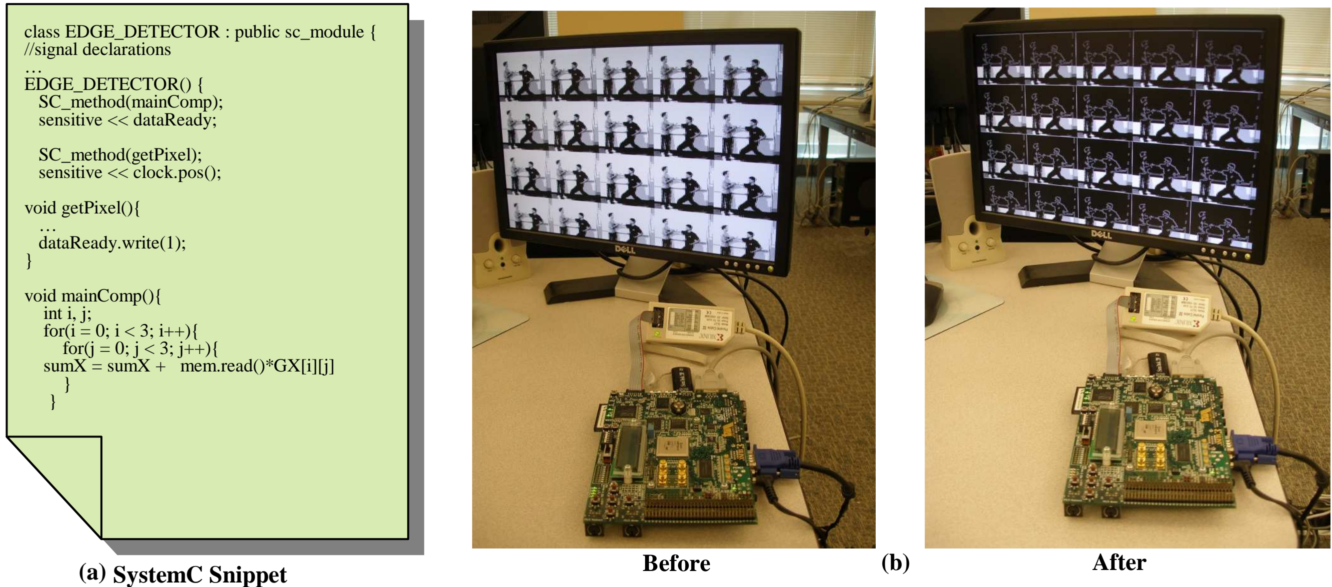
4. Experiments

We built two complete SystemC-on-a-Chip platforms, and implemented dozens of SystemC descriptions to demonstrate the applicability of in-system SystemC emulation. The systems we

Figure 8: SystemC-on-a-Chip Prototypes. Each system differed in size, processor, memory, and number of emulation accelerators, but each could run the same SystemC bytecode for a given SystemC description.

Development Platform	Main Processor	Bus Platform	Memory Location	# of emulation accelerators
Xilinx Virtex4 Ml403 FPGA	PowerPC	PLB	SRAM	2
Xilinx Spartan3E FPGA	Microblaze	OPB	BRAM	1

Figure 9: SystemC Experiments. (a) SystemC code for Image Edge Detection. The code took only minutes to create and compile before being put on a Virtex4. (b) Edge Detection running on a Virtex4. We connected the memory output to a frame buffer to see the results on a VGA screen.

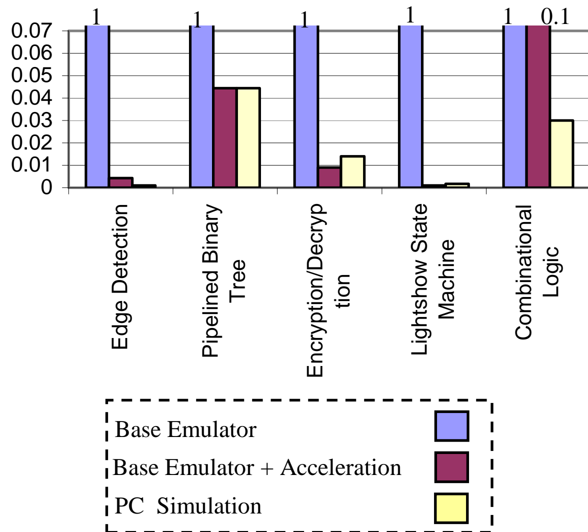


built are summarized in Figure 8. One platform used the Virtex4 Ml403 FPGA development board, and the other used a Spartan 3E FPGA development board. On the Virtex4 ML403 FPGA, we built the emulation engine on a PowerPC processor and used the PLB bus framework to access I/O and peripherals. On the Spartan 3E FPGA, we built the emulation engine on a Microblaze soft-core processor, using the OPB bus framework to access peripherals and I/O. The instruction memory, stack, and heap for the PowerPC based basic emulation engine were all stored in SRAM. In contrast, the instruction memory, stack, and heap for the Microblaze-based system were all located in on-chip BRAM. Due to limited BRAM resources, some SystemC descriptions would not run on the Microblaze-based platform. No SRAM existed on the Spartan 3E platform. The Virtex4 platform could support two emulation accelerators, and the Spartan 3E platform could support one emulation accelerator. The emulation accelerators required about 30% of the logic resources on each FPGA. The Spartan 3E only fit one accelerator because the rest of the FPGA resources were used to build the Microblaze softcore processor. The Virtex4 has a hardcore PowerPC, allowing more room for emulation accelerators. We built both systems using Xilinx EDK 9.2 and Xilinx ISE 9.2. For most descriptions we tested, both SystemC-on-a-Chip platforms systems could support the same SystemC circuit without altering the SystemC source code.

We implemented a number of different circuits in SystemC, including an edge detector, encryption/decryption applications, various state machines, and several smaller combinational logic components to exercise the entire SystemC bytecode set. Figure 9(a) shows a snippet of the SystemC source code for the edge detection circuit. The edge detection circuit was written with two processes, one process to gather pixel data from the input memory, and one process to perform the edge detection and output to the output memory. We configured each platform to use the output memory as a frame buffer, allowing visual inspection of the output on a VGA screen (Figure 9(b)). The edge detection circuit could process a 128x128 image in approximately 30 seconds on the base emulation engine (without acceleration support). While slow, in an early prototyping scenario, or in a classroom setting, such times might be acceptable. The edge detection circuit completed the same image in about 0.3 seconds when one emulation accelerator was connected. The basic emulation engine requires hundreds of cycles to execute one bytecode instruction because of a slow bus platform, instruction memory placement, and bytecode virtual machine abstraction. The emulation accelerator requires only a few clock cycles per instruction, resulting in several of orders of magnitude speedup. We also compared the edge detection circuit running on the SystemC-on-a-Chip platforms to the same SystemC circuit description running on an Intel-based PC running at 2 GHz. The SystemC edge detection circuit took 0.5 seconds to complete the same 128x128 image. The SystemC-on-a-Chip platform emulated the SystemC circuit faster than the PC simulated the SystemC description because the accelerator executed each bytecode instruction in only a few cycles. In contrast, the PC SystemC simulation requires tens of cycles to execute an instruction because the PC simulation runs through a more complex, templated, event-driven kernel that supports SystemC's more advanced modeling features.

Figure 10 compares a variety of SystemC descriptions on a base SystemC-on-a-Chip platform, on a base platform with emulation acceleration, and PC simulation. The figure shows results compared to the Virtex4 Ml403 development platform. The results for the Spartan 3E platform were comparable. On the Spartan 3E development platform, the Microblaze system clock was half the speed of the PowerPC on the Virtex4, but fetched memory more efficiently since the Microblaze had a dedicated bus to the BRAM instruction memory. The results show that SystemC-on-a-Chip execution *can* be comparable to SystemC simulation on a PC. Of course, the execution times are

Figure 10:. Performance of SystemC Emulator (normalize to the base emulator). In some examples, the emulator with acceleration was able to perform faster than the same SystemC circuit running on a PC.



comparable only until the number of emulation accelerators are exhausted. Also, the PC *can* simulate combinational logic more efficiently than in-system SystemC emulation because the PC event kernel can skip large sections of simulated time, although skipping large amounts of time is dependent on the test bench specification. The SystemC-on-a-Chip platform does not skip time steps, instead relying on peripheral interaction for testing. In many cases though, especially during early testing scenarios, high performance isn't critical and functional interaction with real peripherals and I/O is more important. In all cases, the basic emulation engine executed the SystemC descriptions ~100X slower than the emulation engine with acceleration or the SystemC PC simulation. The basic emulation engine has the advantage that many smaller development platforms could still support its software, enabling in-system SystemC emulation for less capable systems, or for systems without FPGA resources.

5. Conclusions

SystemC allows description of a digital system using traditional programming features as well as spatial connectivity features common in hardware description languages. We described an approach for in-system emulation of SystemC descriptions. The approach centers around a new SystemC bytecode format that executes on an emulation engine running on a microprocessor and/or FPGA on a development board. We described a full SystemC-on-a-chip framework that includes a SystemC bytecode compiler, the SystemC bytecode format, emulation engine, and emulation accelerators. We showed that a number of examples could be written once in SystemC, and then run unaltered on several prototype platforms from a USB flash drive, with execution times comparable to SystemC execution on a PC.

References

[1] ANDERSON, E., AGRON, J., PECK, W., STEVENS, J., BAIJOT, F., KOMP, E., SASS, R., AND ANDREWS, D. 2006. Enabling a Uniform Programming Model Across the Software/Hardware Boundary. In *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines* (April 24 - 26, 2006). FCCM

[2] ANDREWS, D., SASS, R., ANDERSON, E., AGRON, J., PECK, W., STEVENS, J., BAIJOT, F., AND KOMP, E. 2008. Achieving programming model abstractions for reconfigurable computing. *IEEE Trans. Very Large Scale Integr. Syst.* 16, 1 (Jan. 2008), 34-44.

[3] BENINI, L., BRUNI, D., DRAGO, N., FUMMI, F., AND PONCINO, M. "Virtual in-circuit emulation for timing accurate system prototyping," in *Proc. IEEE Int. Conf. ASIC/- SoC*, 2002

[4] BROWN, J.C Parallel Architectures for Computer Systems. IEEE Computer vol 37, no. 5. pp83-87 1989.

[5] CADENCE DESIGN SYSTEMS. http://www.cadence.com/us/pages/default.aspx

[6] CHANG, C., KUUSILINNA, K., RICHARDS, B., AND BRODERSEN, R. W. 2003. Implementation of BEE: a real-time large-scale hardware emulation engine. In *Proceedings of the 2003 ACM/SIGDA Eleventh international Symposium on Field Programmable Gate Arrays* (Monterey, California, USA, February 23 - 25, 2003). FPGA '03. ACM, New York, NY, 91-99

[7] COWARE. http://www.coware.com/

[8] DAVIS, B., BEATTY, A., CASEY, K., GREGG, D., AND WALDRON, J. 2003. The case for virtual register machines. In *Proceedings of the 2003 Workshop on interpreters, Virtual Machines and Emulators* (San Diego, California, June 12 - 12, 2003). IVME '03. ACM, New York, NY, 41-49

[9] FRENCH, R. S., LAM, M. S., LEVITT, J. R., AND OLUKOTUN, K. 1995. A general method for compiling event-driven simulations. In *Proceedings of the 32nd ACM/IEEE Conference on Design Automation* (San Francisco, California, United States, June 12 - 16, 1995). DAC '95. ACM, New York, NY, 151-156

[10] FORNACIARI, W. AND PIURI, V. Virtual FPGAs: Some Steps Behind the Physical Barriers. In Parallel and Distributed Processing (IPPS/SPDP'98 Workshop Proceedings), LNCS. 1998.

[11] GENKO, N., ATIENZA, D., MICHELI, G. D., MENDIAS, J. M., HERMIDA, R., AND CATTHOOR, F. 2005. A Complete Network-On-Chip Emulation Framework. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1* (March 07 - 11, 2005). Design, Automation, and Test in Europe. IEEE Computer Society, Washington, DC, 246-251

[12] GRUIAN, F. AND WESTMIJZE, M. 2007. BlueJEP: a flexible and high-performance Java embedded processor. In *Proceedings of the 5th international Workshop on Java Technologies For Real-Time and Embedded Systems* (Vienna, Austria, September 26 - 28, 2007). JTRES '07, vol. 231. ACM, New York, NY, 222-229

[13] HENNESSY, J. AND PATTERSON, D. Computer Architecture – A Quantitative Approach. Morgan Kaufman Publishers. 3rd edition. 1996

[14] LEVINE, B. A. AND SCHMIT, H. H. 2003. Efficient Application Representation for HASTE: Hybrid Architectures with a Single, Transformable Executable. In *Proceedings of the 11th Annual IEEE Symposium on Field-*

Programmable Custom Computing Machines (April 09 - 11, 2003). FCCM. IEEE Computer Society, Washington, DC, 101

[15] LEVIS, P. AND CULLER, D. 2002. Maté: a tiny virtual machine for sensor networks. *SIGOPS Oper. Syst. Rev.* 36, 5 (Dec. 2002), 85-95

[16] MOORE, N., CONTI, A., LEESER, M., AND KING, L. S. 2007. Writing Portable Applications that Dynamically Bind at Run Time to Reconfigurable Hardware. In *Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines* (April 23 - 25, 2007). FCCM. IEEE Computer Society, Washington, DC, 229-238

[17] MOY, M., MARANINCHI, F., AND MAILLET-CONTOZ, L. 2005. Pinapa: an extraction tool for SystemC descriptions of systems-on-a-chip. In *Proceedings of the 5th ACM international Conference on Embedded Software* (Jersey City, NJ, USA, September 18 - 22, 2005). EMSOFT '05. ACM, New York, NY, 317-324

[18] NAKAMURA, Y., HOSOKAWA, K., KURODA, I., YOSHIKAWA, K., AND YOSHIMURA, T. 2004. A fast hardware/software co-verification method for system-on-a-chip by using a C/C++ simulator and FPGA emulator with shared register communication. In *Proceedings of the 41st Annual Conference on Design Automation* (San Diego, CA, USA, June 07 - 11, 2004). DAC '04

[19] PARNIS, J. AND LEE, G. 2004. Exploiting FPGA concurrency to enhance JVM performance. In *Proceedings of the 27th Australasian Conference on Computer Science - Volume 26* (Dunedin, New Zealand). Estivill-Castro, Ed. ACSC, vol. 56. Australian Computer Society, Darlinghurst, Australia, 223-232

[20] POLETTO, M. AND SARKAR, V. 1999. Linear scan register allocation. *ACM Trans. Program. Lang. Syst.* 21, 5 (Sep. 1999), 895-913.

[21] RISSA, T., DONLIN, A., AND LUK, W. 2005. Evaluation of SystemC Modelling of Reconfigurable Embedded Systems. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 3* (March 07 - 11, 2005). Design, Automation, and Test in Europe. IEEE Computer Society, Washington, DC, 253-258

[22] SIROWY, S. SHELDON, D., GIVARGIS, T. AND VAHID. F. Virtual Microcontrollers. Workshop on Embedded System Education. June 2009.

[23] SIROWY, S., STITT, G., AND VAHID, F. 2008. C is for circuits: capturing FPGA circuits as sequential code for portability. In *Proceedings of the 16th international ACM/SIGDA Symposium on Field Programmable Gate Arrays* (Monterey, California, USA, February 24 - 26, 2008). FPGA '08

[24] SMITH, J. AND NAIR, R. VIRTUAL MACHINES: Versatile Platforms for Systems and Processes. Morgan-Kaufman Publishers. 2005

[25] STARK, R., SCHMID, J, AND BORGER, E. Java and the Virtual Machine- Definition, Verification, and Validation. 2001

[26] SYSTEMC. http://www.systemc.org

[27] SYSTEMC SYNTHESIZABLE SUBSET. http://www.systemc.org

[28] VERILOG SPECIFICATION. http://www.verilog.com/VerilogBNF.html

[29] VHDL SPECIFICATION STANDARD. http://www.vhdl.org/

[30] VMWARE. http://www.vmware.com

[31] VULETIC, M., POZZI, L., AND IENNE, P. 2005. Seamless Hardware-Software Integration in Reconfigurable Computing Systems. *IEEE Des. Test* 22, 2 (Mar. 2005), 102-113

[32] XEN. http://www.xen.org