Highly-Parallel Special-Purpose Multicore Architecture for SystemC/TLM Simulations

N. Ventroux*[†], J. Peeters*, T. Sassolas*
*CEA, LIST,
Embedded Computing Laboratory
91191 Gif-sur-Yvette CEDEX, France
Email: nicolas.ventroux@cea.fr

James C. Hoe[†]
[†]Computer Architecture Laboratory (CALCM)
Carnegie Mellon University
Pittsburgh, PA 15213-3890, USA
Email: jhoe@ece.cmu.edu

***Abstract*—The complexity of SystemC virtual prototyping is continuously increasing. Accelerating RTL/TLM SystemC simulations is essential to control future SoC development cost and time-to-market. In this paper, we present RAVES, a highly-parallel special-purpose multicore architecture that achieves simulation performance more efficiently by parallel execution of light-weight user-level threads on many small cores. We present a design study based on the virtual prototype of RAVES processors running a co-designed custom SystemC kernel. Our evaluation suggests that a 64-core RAVES processor can deliver up to 4.47x more simulation performance than a high-end x86 processor.**

I. Introduction

Electronic System Level (ESL) design has become essential in the design methodology of large digital systems. Based on the SystemC language [1], it provides an effective framework for HW/SW codesign, system-level and performance modeling, architecture exploration, high-level verification, as well as early development of system software. However, the increasing complexity of digital systems is reducing the benefit of using high-level simulations in the design flow. Slower performance can increase development costs and drastically impact the time to market. Thus, accelerating SystemC simulations is becoming essential to foster competitiveness.

Existing Approaches. A SystemC design is a cooperative and sequential multi-threaded program. It is highly irregular and control-oriented. Several works in the literature have attempted to optimize and parallelize SystemC simulations. For instance, some optimizations have been performed by removing unnecessary context switches and by statically scheduling SystemC processes for RTL modeling [2], [3] or TLM [4]. An acceleration can only be obtained mostly for RTL combinational logic simulations that can generate multiple unnecessary waking-ups.

Some works propose to divide the model into several subsystems which are distributed on multiple cores. These subsystems cooperate to carry out the complete simulation [5], [6]. However, it introduces a significant overhead mainly due to synchronizations. Performance depends on the natural concurrency and granularity of what is simulated.

Besides in SystemC, there is a natural thread-level parallelism in the same delta cycle during the evaluation of processes. Several prior works in the literature propose to exploit this parallelism. For instance in [7], [8], multiple CPUs are used to evaluate in parallel the processes. Unfortunately, the synchronizations still impact the performance mainly because a synchronization is necessary at each delta cycle. This is also the reason why some recent works relax synchronizations [9], [10]. However, optimistic techniques in large simulations is costly in terms of memory and performance, especially with the non-predictable behavior of MPSoCs.

To reduce the synchronization overhead, some works exploit the parallelism of a single multicore chip. In [11], processes are dispatched on the different secondary processors of the CELL architecture. Their proposal has several constraints; for instance, the processes are sensitive to a single signal, which is very rare in hardware modeling. Finally, the most recent works use GP-GPU platforms [12]–[15]. Nonetheless, the maximum number of concurrent and different threads that can be executed in parallel is limited by the number of threads groups in a GPU. In addition, most of the acceleration has been obtained by considering high data-parallel benchmarks or independent simulation instances, by modifying the SystemC kernel to reduce synchronizations, or by executing identical parallel threads.

Special Purpose Architecture. In this paper, we propose to study the performance that can be obtained with a special-purpose multicore processor, based on many power-efficient and smaller RISC cores. The underlying parallelization approach can also be generalized to enable a performance-oriented system based on high-performance cores.

Thus, we present a special-purpose HW/SW co-designed approach to exploit parallelism in any RTL and TLM SystemC simulations. A specific heterogeneous multicore architecture and a very lightweight system software has been designed to efficiently manage SystemC processes. Our RApid Virtual prototyping Emulation System (RAVES) is a hardware platform that can support a dynamic and parallel execution of SystemC processes. This platform comes with a hardware SystemC kernel accelerator to reduce control and synchronization overheads, as well as to increase the scalability of our approach. A lightweight and optimized parallel SystemC kernel manages the SystemC processes. The SystemC kernel accelerator allows the processes to be efficiently dispatched to manycores for parallel execution. A hardware virtual prototype of this platform has been developed, and is used in this study for the evaluations.

Contributions. The contributions of this paper are: (1) a SystemC kernel for parallel simulation, (2) a hardware SystemC kernel accelerator to reduce control and synchronization overheads, and (3) a multicore architecture that integrates the parallel SystemC kernel and the kernel accelerator to execute SystemC/TLM simulations.

The paper is organized as follows. The second section motivates our approach. Section 3 presents the RAVES SystemC kernel. Section 4 introduces the RAVES multicore architecture. Section 5 depicts the execution model to explain how SystemC simulations are executed on RAVES. Section 6 discusses the validation of our solution and compares our results to existing work. Finally, Section 7 concludes the paper and discusses possible future work.

II. Motivation

To implement parallel and concurrent mechanisms, the SystemC library can be compiled either with kernel threads (PosixThread) or with user-level threads (QuickThread).

With kernel threads, the operating system (OS) has a descriptor for each thread belonging to a process. It can automatically schedule them on several cores according to their availability. However, the SystemC kernel is already managing the execution of threads and uses a cooperative multithreading strategy. This prevents the OS from efficiently scheduling threads, generating unnecessary migrations of threads between cores, as shown Figure 1. By forcing the execution on one core, the total execution time is actually reduced by 2.88x.

When forced onto a single-core in sequential mode, kernel threads still induce a high overhead. On the other hand, the use of user-level threads brings an optimal performance in a sequential thread execution, like with cooperative multithreading. All threads are managed into one main kernel thread. Running a thread becomes almost equivalent to a simple function call. Figure 1 points out an 13.3x acceleration compared to the kernel thread version. Unfortunately, user-level threads do not support thread-level parallelism, nor can they exploit the host system multiple cores.

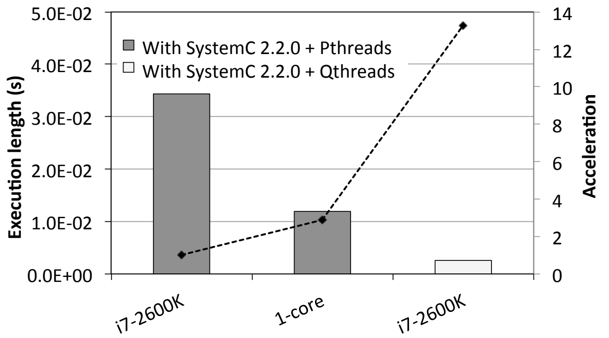


Figure 1. SystemC 2.2.0 analysis. Execution of the benchmark *TLM1* with N=256 (see section VI-A) on an Intel i7-2600K.

In order to have both the reactivity of user-level threads and the parallelism offered by kernel threads, mixed approaches have emerged. One kernel thread per core, named *worker*, is used as a container to locally execute user-level threads through *ucontext* primitives. Like in [7], [8], the whole simulation is placed into a kernel thread. At the beginning of every evaluation phase, this main thread can partially or completely offload processes onto its pool of worker kernel threads. Once all workers have completed the evaluation phase and reached the barrier, the main thread collects and processes the request queues from its workers.

Such solution has been implemented to study its performance, and enhanced with a software hierarchical process queue to take into account the high variability of SystemC processes, as well as to optimize computing resource workloads, which allows a dynamic dispatching of processes among multiple cores. This hierarchical queue is composed of a global queue and a local queue in each worker. These small local queues are initialized with a process to prevent all workers to simultaneously access the global queue at the beginning of the evaluation phase. Thus, a dynamic farming execution of processes can be done during the evaluation phase. As shown in Figure 2, our approach gives a similar performance to the optimized user-level version of SystemC, when executing on one core. However, with 7 workers, this implementation brings a rather high acceleration with a benchmark that needs few synchronizations (b) and no acceleration on the contrary (a). This also means that this approach can only work when synchronizations with the SystemC kernel remain low. This paper's objective is to evaluate a solution to reduce these overheads in order to reach an acceleration in any SystemC simulations.

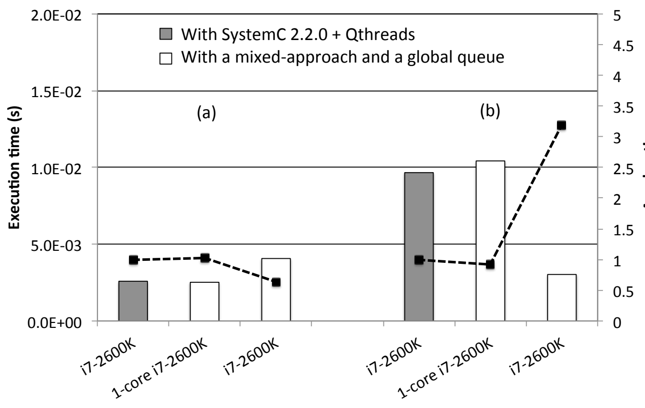


Figure 2. Workers-based kernel analysis on an Intel i7-2600K. (a) execution of the benchmark *TLM1* with N=256 ; and (b) execution of the benchmark *TLM2* with N=256 (see section VI-A).

In the rest of the paper, we present a complete system of custom SW kernel and co-designed processor hardware dedicated to SystemC to exploit parallelism onto multiple cores. The next section presents the software kernel architecture.

III. RAVES SW kernel

The RAVES SystemC kernel is fully compliant with SystemC 2.2.0 [1], with the exception that not all SystemC data types are available yet (but can be easily implemented). This does not prevent the modeling of RTL architectures since all data types in SystemC inherit from standard data types. The other limitation is that all SystemC objects must be dynamically allocated and debug facilities have not been implemented yet. The initialization and elaboration phases will not be taken into account into our results.

This kernel is highly inspired by our mixed implementation as it keeps the concept of the global process queue and the farming execution. Nonetheless, to reach better performance no linux kernel is used. In the RAVES kernel, the linux kernel thread *workers* are removed and replaced by a very lightweight system software. This micro-kernel has been optimized to support the execution of SystemC processes. It supports the creation of thread contexts, process allocation, process preemption and migration. With these modifications, we ensure an efficient implementation, as it can be done with QuickThreads, but with multicore processing.

As explained in the previous section, our scheduling strategy is based on a farming model of SystemC processes (*SC_METHODs* and *SC_THREADs*). Only the evaluation phase is executed in parallel on multiple cores. The other SystemC phases are still sequentially executed. Therefore, all processes are synchronized before the beginning of the update phase. The hierarchical queue is replaced by a hardware global queue for a higher performance. This process queue is used to store all to-be-evaluated SystemC processes for the evaluation phase. An API of about 30 functions has been developed for the debugging and for all communications between RAVES entities. Thanks to this API, all processes are able to trigger immediate or delta notifications, or for instance, to communicate with the process queue.

In addition, a simulation manager is used to launch and interleave different SystemC simulations. Multiple different or same simulation instances can be simultaneously executed on the RAVES architecture. Moreover, with our kernel, it is possible to instantiate a clock that can generate only positive edges, cutting down by 2 the time spent in the clock evaluation. This clock will be used in all our experiments using RAVES.

However, as shown in Figure 3, this full-software RAVES kernel implementation still comes with a significant overhead due to all sequential phases, which can reach 70 % of the total execution time. The useful execution time in a SystemC simulation is the evaluation phase only. In most cases, the delta notification and update phases mostly limit the performance and scalability. In particular, the two main overheads come from (1) the time spent to find all sensitive processes from a given event in the immediate and delta notification phases, and from (2) the update phase (up to 20% overhead).

We next present a co-designed custom HW architecture to cut down the kernel overhead and enhance scalability and performance.

IV. RAVES HW ARCHITECTURE

The RAVES hardware architecture is a standard homogeneous shared-memory multicore architecture, which comes with a specific SystemC controller. A set of homogeneous cores has been used to support a dynamic dispatching of SystemC processes. In addition, all these cores use a shared memory since SystemC simulations require a lot of synchronizations. The SystemC controller is a standard processor, named SystemC Kernel Unit (SKU), with a hardware accelerator named SystemC Kernel Accelerator (SKA). This customized processor is used to execute the simulation manager and to interleave multiple SystemC simulations. The SKA reduces kernel and synhronizations overheads.

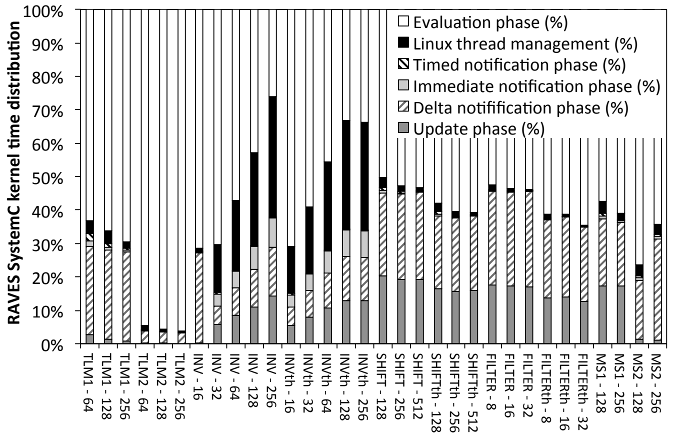


Figure 3. SW RAVES kernel analysis. Execution of the synthetic benchmarks presented in section VI-A on an Intel i7-2600K.

As depicted in Figure 4, this architecture is composed of: the SKU and the SKA to execute the SystemC kernel, a set of homogeneous Agent Processor Cores (APC) to evaluate SystemC processes, a Memory Management Unit (MMU), a multibus network, a unified shared L2 cache, and a DDR3 controller. The CPU represents the master processing unit that asks for SystemC simulations on RAVES. Depending on the chosen coupling level, the CPU could be a host-PC.

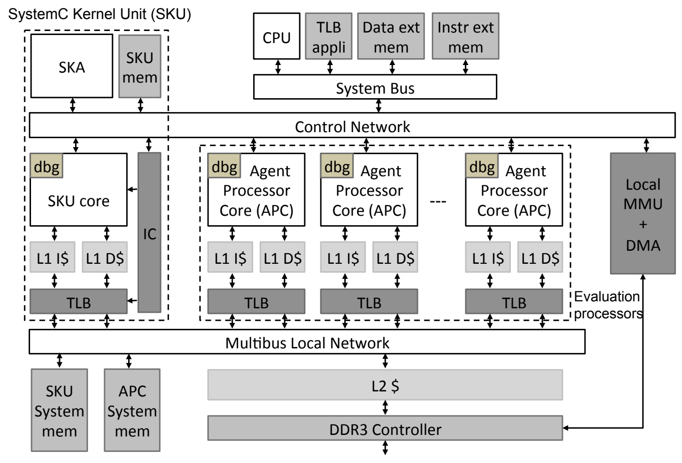


Figure 4. HW RAVES architecture.

The SKA is composed of three main hardware parts: (1) a fast process search engine, (2) a Finite-State-Machine (FSM) that manages the different SystemC phases accelerated in HW and the global process queue, and (3) an update phase accelerator. More details on the execution model are presented in Section V.

The fast process search engine implemented into the SKA helps to easily find all sensitive processes from a given event. As depicted Figure 5, for each event, a linked list of sensitive processes is stored into different complementary memories, and initialized during the elaboration phase. The event search engine is composed of different memory banks, with one comparator per bank. A counter is also used to address the same line of each bank. The event is presented in parallel to each comparator and the counter is incremented until the event is found. The output gives the address of the beginning of its sensitive process list. A hash function could also be implemented to manage a larger number of events and processes but this was not necessary for our study.

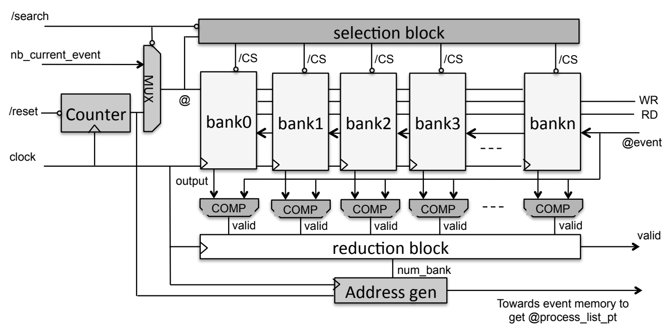


Figure 5. Fast process search engine implemented into the SKA.

As shown in Figure 6, the FSM manages the immediate notification and delta phases, as well as the activation of the evaluation and update phases. The global process queue is filled in by the software kernel at the beginning of a new simulation cycle, or during the immediate or delta phases. After each event notification or signal value modification, agents push an immediate or delta event in 2 distinct event queues. These queues are then used, during the immediate and delta notification phases, to find the next processes to evaluate.

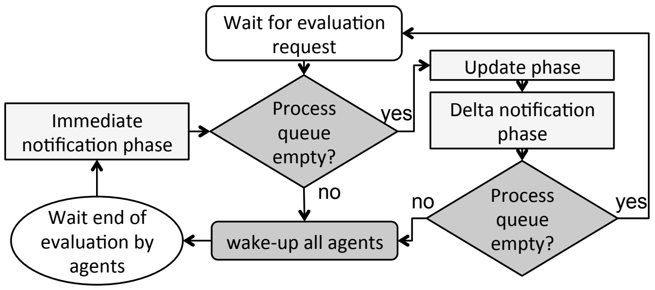


Figure 6. SystemC Kernel Accelerator FSM.

The update phase is a HW/SW implementation. The parts that cannot be parallelized in software are accelerated on hardware. The SW part has been integrated into the evaluation phase in order to distribute its complexity among all agents. Each *sc_signal* owns 2 write buffers *next_value_odd* and *next_value_even*, and a current buffer *m_value*. In addition, each *sc_signal* owns a HW current parity register and an HW update register into the SKA. The parity value represents the status of the update flag stored into the register. As depicted in Figure 7, each read or write starts by updating its current buffer from a write buffer, according to the current parity value returned by the SKA. During a write, if the new value is different from the current value, a delta event is pushed and a request is sent to the SKA to ask for a future update of the parity value. This request sets the update register to '1' into the SKA. The number of registers corresponds to the maximum number of *sc_signal* that can be instantiated. Then, the update phase only consists in doing a parallel bit-to-bit exclusive OR between the update registers and the current parity ones.

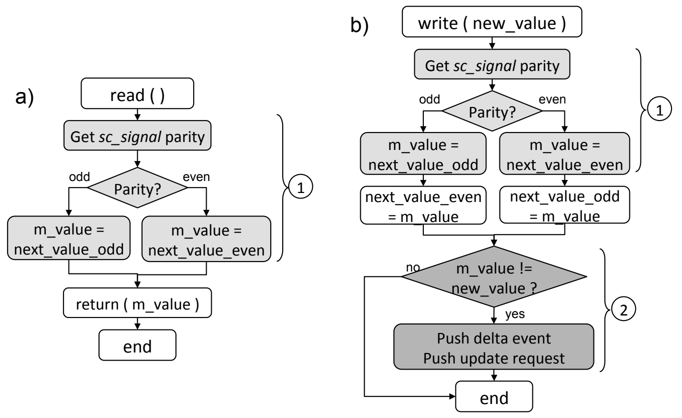


Figure 7. New *sc_signal* read (a) and write (b) method diagrams: (1) post-update phases and (2) pre-update phase.

V. RAVES EXECUTION MODEL

At reset, the SKU core loads the simulation manager code from the *SysC kernel* memory and waits for a CPU command, whereas APCs wait for evaluation requests after booting from the *Agent kernel* memory. A CPU-host processor can ask the SKU to execute new SystemC simulations by triggering an interruption through the system bus. Then, thanks to the MMU, the simulation manager loads and allocates the corresponding code into the DDR.

When ready, the SKU core preempts itself to execute the SystemC simulation. The SystemC kernel starts by binding and elaborating the simulator. The instantiation of each *SC_THREAD* asks for the creation of its context and stack by the MMU. In addition, a unique identification number, used for any communications with the SKA, is requested to the SKA for each new *sc_signal* and SystemC process. Concerning process sensitivity lists, all information are also sent to the SKA to initialize the event search engine.

After this initialization, all processes are asked to be evaluated (except those with the *dont_initialize* command), by pushing them in the SKA process queue. Finally, the SKU core preempts itself to return to the simulation manager. The simulation manager waits for the end of the different phases managed by the SKA, and is also ready to manage other requests coming from the CPU. As soon as the SKA process queue is empty after the delta and immediate notification phases, the simulation manager launches again the SystemC kernel. Then, the timed notification phase is performed and the simulation cycle is completed. The SystemC kernel is terminated if a user-provided maximum simulation time is reached, or if there is no more process to evaluate. Then, the SKU resumes to the simulation manager. Else, a new complete cycle is started again.

Each agent is waiting for an evaluation request from the SKA. When the interruption is received, an interruption routine pops a process from the SKA process queue. If it is a *SC_METHOD*, the agent initializes its context and executes the body of the corresponding function. On the contrary, if it is a *SC_THREAD*, the agent restores the previous context, gets the previous PC, executes the function until a *wait* is encountered, and finally saves its context. Because the process allocation is dynamic, a cache flush and invalidation is done after each evaluation.

VI. Evaluations

In this section we evaluate the performance of RAVES using a virtual prototype implemented in SystemC/TLM. Sections VI-A and VI-B present our experimental setup and benchmarks. Finally, section VI-C analyses RAVES performance and limitations.

A. Benchmarks

In order to evaluate RAVES performance, 10 synthetic benchmarks are considered. Each of them is highlighting a specific part of the SystemC kernel.

- The benchmark *TLM1* makes *N* masters initiate transactions towards N slaves at each clock cycle (TLM). Each master is implemented as a *SC_THREAD* and has an exclusive access to its slave. It represents a cycle-accurate implementation of a multicore TL model.
- The benchmark *TLM2* implements *N* Instruction Set Simulators (ISS) based on ArchC 2.0 [16] that communicate with their private memory. Each ISS successively executes 10 instructions and then synchronizes with the kernel. It represents an ideal implementation of a virtual MPSoC.
- The benchmarks *INV*, or *INVth*, is a chain of *N* inverters composed of *SC_METHODs*, or *SC_THREADs*. Only the first inverter is synchronized with a clock. The other processes are woken up through immediate notifications.
- The benchmarks *SHIFT*, or *SHIFTth*, is a shift register of *N* registers composed of *SC_METHODs*, or *SC_THREADs*. It represents a typical RTL implementation with pipelined parallelism. All processes are sensitive on a positive edge of the clock.
- The benchmarks *FILTER*, or *FILTERth*, is *N* 16-stage RTL Finite Impulse Response filters, composed of *SC_METHODs*, or *SC_THREADs*, which outputs are connected through a selection block that picks out the highest value. It represents a typical RTL implementation with structural parallelism. All processes are sensitive on a positive edge of the clock.
- The benchmark *MS1* is a master that communicates to *N* slaves through *N* SystemC ports. At each clock cycle, *N* events are triggered and each of the corresponding sensitive *SC_THREAD* process is executed.
- The last one *MS2* is a master that communicates to *N* slaves through 1 SystemC port. At each clock cycle, only 1 event is triggered and the *N* corresponding sensitive *SC_THREAD* process are executed.

The number of simulated cycles is constant and independent of *N*. Table I summarizes their complexity by the total number of SystemC processes and by the number of evaluation phases per simulation cycle that also represents the synchronization demand with the SystemC kernel.

Benchmarks	# SystemC processes	# evaluation phases per simulation cycle
TLM1, TLM2	$1+N$	$1+(2)_{clk}$
INV, INVth	$2+N$	$N+1+(2)_{clk}$
SHIFT, SHIFTth	$3+N$	$1+(2)_{clk}$
FILTER, FILTERth	$4+43\cdot N$	$1+(2)_{clk}$
MS1, MS2	$1+N$	$2+(2)_{clk}$

Table I. Complexity analysis of benchmarks depending on the parameter N. $(x)_{clk}$ means x additional evaluation phases per simulation cycle due to the clock. This value is divided by 2 on RAVES when using the positive-edge-only clock.

B. Experimental setup

We designed a virtual prototype of RAVES within the SESAM environment [17]. SESAM is a SystemC/TLM simulation framework that eases the architecture exploration of multicore platforms. Within this framework, all communications and modules are timed. In addition, all memory and network-on-chip contentions are modeled. According to [18], accuracy is higher than 90 %.

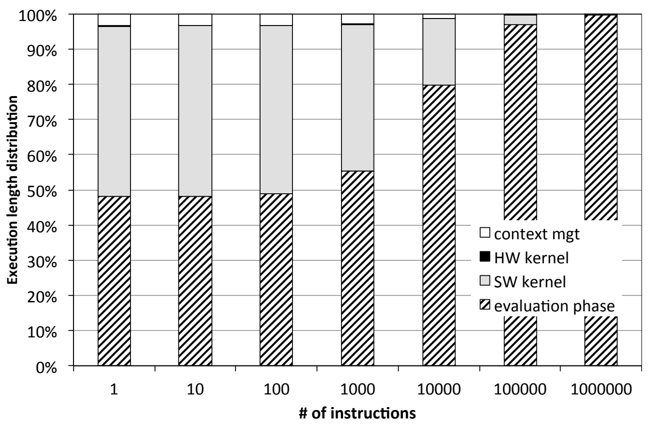
The SKU is a RISC processor and comes with an Interrupt Controller (IC), 32KB L1 instruction and data caches, a local memory and its SystemC Kernel Accelerator (SKA). APCs are also RISC cores with 32KB L1 instruction and data caches; 1 to 64 cores will be considered. All L1 caches use LRU write-back and write-allocate policies, as well as 16B blocks. All cores can have an access to 2 local shared memories, which contain the system code, and an interleaved shared 8-bank 4MB L2 cache. The multibus is a 64-bit 6-cycle-latency network-on-chip. On the contrary, the control network is a simple 32-bit 6-cycle-latency bus. The MMU is used for dynamic allocation and to transfer SystemC simulation code from external memory to the DDR.

This paper will consider 3 different kinds of RISC cores. It demonstates the benefits of integrating more complex cores. The first one, *core A*, is based on the Mips24K core [19] provided by SESAM as a functional Instruction Set Simulator (ISS). Based on the study in [20], the IPC for this 8-stage pipelined RISC is 0.78. The second one, *core B*, is based on a dual-issue superscalar ARM Cortex A9 core (r0p1) from the *CoreTile Express A9x4* board [21]. The third one, *core C*, is based on a 3-way VLIW core from the Tilera TilePro64 chip [22]. Only *core A* is simulated. With the 2 other cores, results from real platforms are injected. By executing all benchmarks with the RAVES SW kernel on the 3 cores, we get an acceleration factor between them. This factor is applied on *core A* results to mimics the performance that could be expected by integrating *cores B* and *C* into RAVES.

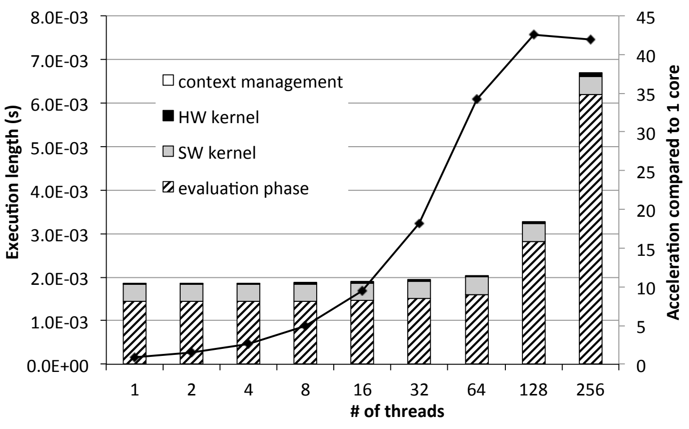
RAVES performance is compared to an Intel i7-2600K at 3.7 GHz running SystemC 2.2.0 on a RHEL6 Linux 2.6.32 kernel. RAVES frequency is assumed to be 1 GHz with the cores A and C, and 2 GHz with the core B, except for the networks-on-chips, the memories and the SKA which are running at a frequency two-times slower. Measurements are performed after the binding and elaboration phases. All compilations have been done at level O3 with the same options with GNU gcc 4.4.7.

C. Performance analysis

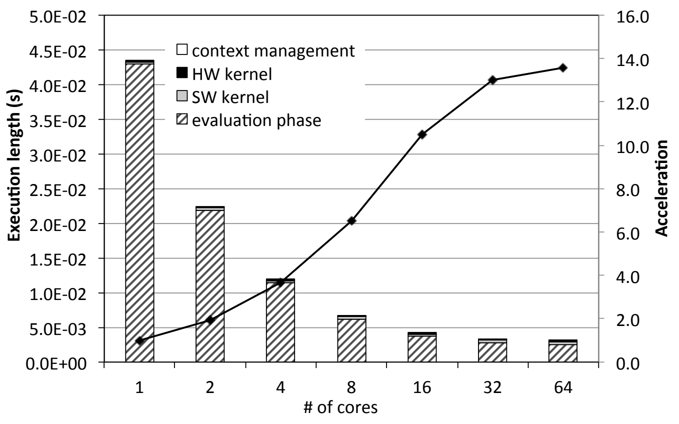
Figure 8-a represents the results obtained by executing a synthetic benchmark executing 256 threads (SystemC processes) on a 1-core RAVES architecture. Each of the threads is composed of a given number of nop instructions. It shows that the average thread length has a direct impact on the kernel overhead and then on performance. With 10K-length



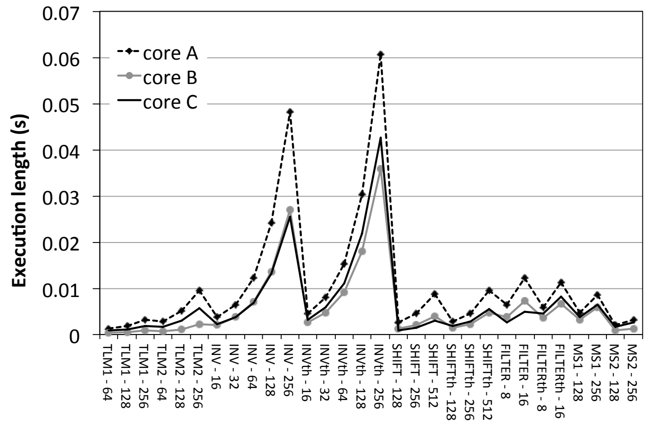
(a)



(b)



(c)



(d)

Figure 8. RAVES results. (a) execution of 256 threads composed of different numbers of nop instruction on 1 A-core RAVES architecture to analyze the impact of the thread size on RAVES performance; (b) execution of different numbers of 10K-length instruction threads on 64 A-core RAVES architecture to analyze the number of threads impact on RAVES performance; (c) study of the number of agents of A-core RAVES architecture with the benchmark TLM1 and N=256; and (d) RISC core study for agents on a 64 core RAVES architecture with all benchmarks.

instruction threads, this overhead is reduced to 20 %. By keeping this value and increasing the number of threads on a 64-core RAVES architecture, we show in Figure 8-b that the maximum theoretical acceleration is around 43x with 128 or 256 threads. The execution time increases when the number of threads is greater than the number of cores. We observe a slowing down at 64 cores mainly because of our shared L2 memory, which hit rate drops to 68.5 %.

Figure 8-c represents the result obtained with *TLM1* and $N = 256$. The maximum acceleration is about 14x. With this benchmark, threads are smaller ($\sim$ 1K). The kernel overhead impacts more performance. Even if the L2 becomes a bottleneck when the number of cores increases, the speed-up remains the more significant on 64 cores. For RTL benchmarks like *FILTER* and *FILTERth*, it remains around 11x. On the contrary, the benchmark *TLM2* reaches an acceleration of 20.6x on 64 cores, thanks to the high concurrency brought by ISSes. With RAVES, *SC_METHODs* and *SC_THREADs* are very similar and only differs by one context saving. Indeed, the initialization of the context of an *SC_METHOD* and the loading of the previous context of an *SC_THREAD* before their evaluation have the same cost. Therefore, performance is equivalent with our approach.

Figure 8-d depicts the execution length of all benchmarks executed on a 64-core RAVES architecture, when using different RISC cores. It is interesting to notice that both *cores B* and *C* have similar results. Indeed, our SystemC benchmarks are very irregular and few data-parallelism can be exploited. Thus, the internal parallelism of *core C* does not provide a significant improvement. A higher frequency and a more performant misprediction unit can get a higher performance. This is the reason why, *core B* demonstrates a significant improvement, especially with complex RTL benchmarks, compared to *core A*. A dual-issue superscalar architecture is our best candidate among available and selected RISC cores.

Figures 9-a and 9-b highlights the acceleration provided by our SKA. As expected, when the simulation needs to frequently interact with the kernel, as with the benchmark *FILTER* on Figure 9-a, the SKA has a high impact on performance. The maximum acceleration obtained on a 64 B-core RAVES architecture compared with a software-only RAVES version is around 3.2x. The acceleration of the SystemC kernel drastically cuts down the control overhead. Even if we add in the evaluation phase a part of the update phase (that explains why the evaluation phase time increases with the HW acceleration), the dynamic dispatching of threads, coupled to

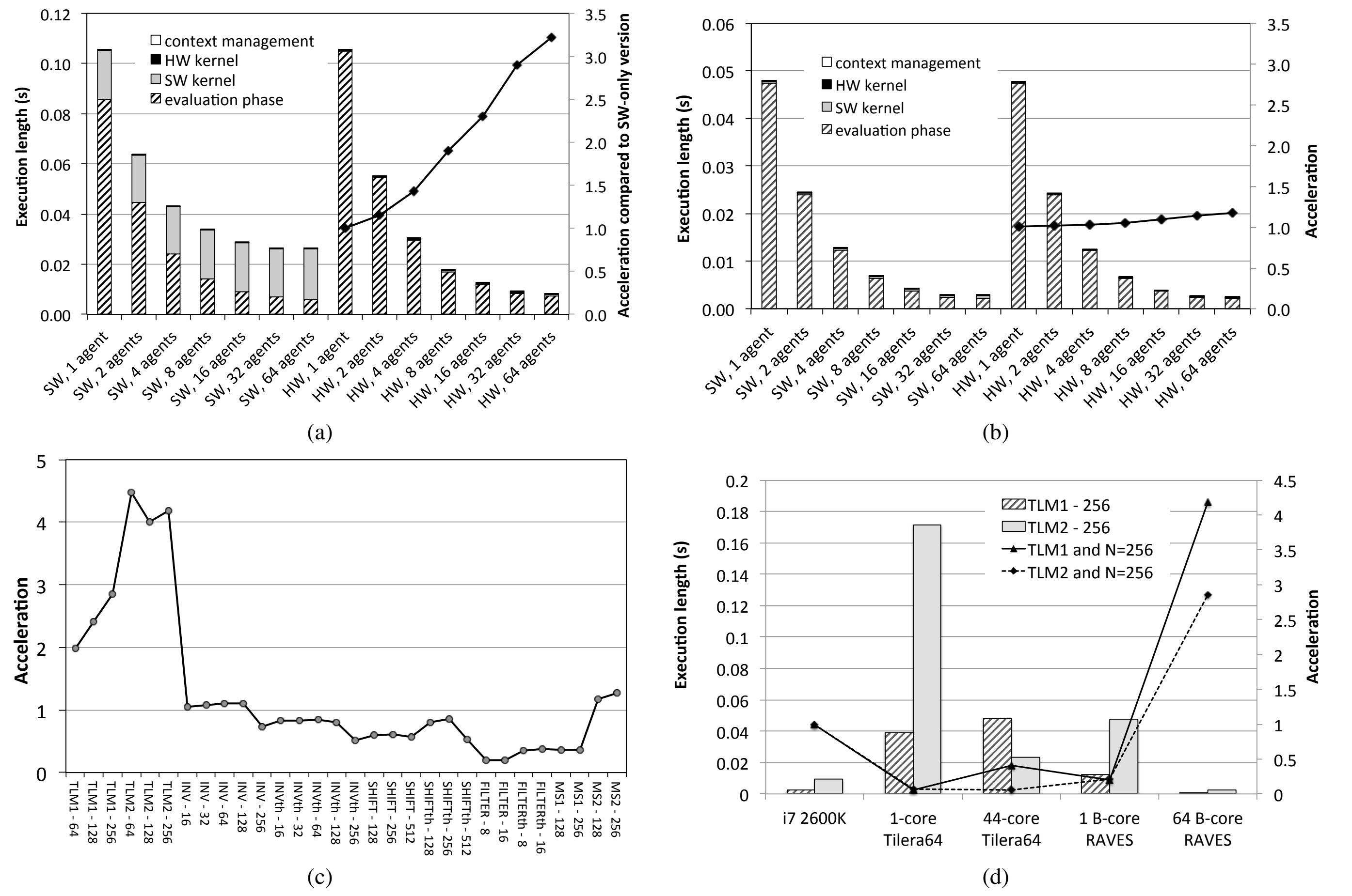


Figure 9. RAVES results. study of the SystemC kernel hardware acceleration on a 64 B-core RAVES architecture by comparing a SW-only version of RAVES with the HW/SW RAVES implementation with (a) the benchmark FILTER and N=16, and (b) the benchmark TLM2 and N=256; (c) acceleration study for all benchmarks of 64 B-core RAVES architecture compared to a QThread SystemC 2.2.0 implementation running on an Intel i7-2600K; and (d) comparison study between the Intel i7-2600K, the Tilera TilePro64 and a 64 B-core RAVES architecture. The acceleration is calculated using the user-level thread SystemC execution time on the Intel i7-2600K as reference for each benchmark.

a low kernel overhead, delivers a high scalability to RAVES. On the contrary, the acceleration remains low when simulating ISSes or highly concurrent SystemC processes, like with the benchmark *TLM2* on Figure 9-b. Indeed, most of the execution time is spent by agents and the SKU has a little impact on performance.

Figure 9-c points out the accelerations obtained with a 64 B-core RAVES architecture compared to the Intel i7-2600K running SystemC 2.2.0 with user-level threads. Accelerations reach up 1.99x to 4.47x for the TLM benchmarks *TLM1* and *TLM2*. Whereas Figure 2 shows that no acceleration can be obtained with a software-only approach on i7 with *TLM1*, the same benchmark scales on the RAVES architecture thanks to our optimized parallel SystemC implementation and our hardware SystemC kernel accelerator. Concerning, RTL benchmarks (all except *TLM1* and *TLM2*), the maximum acceleration remains around 1.12x. The frequent need for kernel synchronizations reduces the performance. Even with our approach, accelerations remain low or negative compared to a sequential execution using user-level threads on a high-end x86 processor.

Finally, Figure 9-d depicts a comparison study with the Intel i7-2600K with user-level threads, a 64 B-core RAVES architecture and the Tilera TilePro64 running at 700 MHz. The results on the TilePro64 architecture highlight that only highly parallel SystemC simulations can scale with the number of cores. However, the 44 cores remain unsufficient to compete with the i7. On the contrary, the RAVES architecture is 2.85x to 4.47x more performant than the i7.

RAVES is the only solution to support RTL/TLM simulations that does not bring any strong structural or usage limitations. Moreover, this study demonstrates that significant accelerations can be obtained even with a many-core platform, as long as a very efficient SystemC synchronizations and thread management can be performed. In addition, our approach can be easily extended. By using more complex and performance-oriented cores, RAVES could outperform the presented results.

Among recent notable work, it is with the SCGPSIM tool [13] that we found the most accurate information to compare our results. With a JPEG benchmark they obtained a 6.7x acceleration with their GP-GPU solution compared to a user-level thread SystemC implementation on an Intel i7 at 2.8 GHz. Thus, at equivalent frequency the acceleration is similar to what RAVES can reach. However, their solution is only efficient with SystemC models with highly independent processes and data-level parallelism. We believe that it is rarely the case in practical use-cases, and that SystemC processes usually need a strong interaction with the kernel.

An i7-2600K is about $216\,mm^2$ (Intel 32 nm) and 95 W, whereas a 64 B-core RAVES platform is about $229\,mm^2$ (TSMC 40 nm) and consumes about 64 W (values estimated by adding cores and memories/caches information obtained from [23] and [24]). As a result, this RAVES platform can get a better performance with a similar complexity and a reduced energy consumption.

VII. Conclusion

This paper presented RAVES, a highly-parallel special-purpose multicore architecture dedicated to SystemC simulations. Our light-weight user-level threads and co-designed custom SystemC kernel bring to RAVES a unique capacity to accelerate RTL and TLM simulations on many cores. Our evaluation suggests that a 64-core RAVES processor can deliver better simulation performance than a high-end x86 processor. It can reach up an acceleration of 4.7x compared to a user-level thread SystemC implementation running on an i7-2600K with small RISC cores. We believe that these results could be improved by using more powerful cores. This is the reason why our future work will focus on the design of a hardware RAVES prototype with more performance-oriented cores. We will also work on a distributed RAVES system to increase again our performance.

References

[1] Accellera Systems Initiative, "SystemC 2.2.0, http://www.accellera.org."

[2] Y. N. Naguib and R. S. Guindi, "Speeding Up SystemC Simulation through Process Splitting," in *IEEE conference on Design, Automation and Test in Europe (DATE)*, Nice, France, April 2007, pp. 111 – 116.

[3] R. Buchmann and A. Greiner, "A Fully Static Scheduling Approach for Fast Cycle Accurate SystemC Simulation of MPSoCs," in *International Conference on Microelectronics (ICM)*, Cairo, Egypt, December 2007, pp. 105 – 108.

[4] K. Lun, D. Mller-Gritschneder, and U. Schlichtmann, "Removal of Unnecessary Context Switches from the SystemC Simulation Kernel for Fast VP Simulation," in *IEEE International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2011*, Samos, Greece, July 2011, pp. 150 – 156.

[5] H. Ziyu, Q. Lei, L. Hongliang, X. Xianghui, and Z. Kun, "A Parallel SystemC Environment: ArchSC," in *IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, Shenzhen, China, December 2009, pp. 617 – 623.

[6] J. Peeters, N. Ventroux, T. Sassolas, and L. Lacassagne, "A SystemC TLM Framework for Distributed Simulation of Complex Systems with Unpredictable Communication," in *IEEE Conference on Design and Architectures for Signal and Image Processing (DASIP)*, Tampere, Finland, November 2011, pp. 1 – 8.

[7] P. Ezudheen, P. Chandran, J. Chandra, B. Simon, and D. Ravi, "Parallelizing SystemC kernel for fast hardware simulation on SMP machines," in *ACM/IEEE Workshop on Principles of Advanced and Distributed Simulation (PADS)*, Lake Placid, New York, USA, June 2009, pp. 80 – 87.

[8] C. Schumacher, R. Leupers, D. Petras, and A. Hoffmann, "parSC: Synchronous Parallel SystemC Simulation on Multi-Core Host Architectures," in *ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Scottsdale, Arizona, USA, October 2010, pp. 241 – 246.

[9] P. Combes, E. Caron, and B. Chopard, "Relaxing Synchronization in a Parallel SystemC Kernel," in *International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, Sydney, Australia, December 2008, pp. 180–187.

[10] I. Pessoa, A. Mello, A. Greiner, and F. Pêcheux, "Parallel TLM Simulation of MPSoC on SMP Workstations: Influence of Communication Locality," in *International Conference on Microelectronics (ICM)*, no. 359-362, Cairo, Egypt, December 2010.

[11] L. Kaouane, D. Houzet, and S. Huet, "SysCellC: SystemC on Cell," in *International Conference on Computational Sciences and Its Applications (ICCSA)*, Perugia, Italy, July 2009, pp. 234 – 244.

[12] R. Sinha, A. Prakash, and H. Patel, "Parallel Simulation of Mixed-abstraction SystemC Models on GPUs and Multicore CPUs," in *IEEE Asia and South Pacific Design Automation Conference (ASP-DAC)*, Sidney, Australia, March 2012, pp. 455 – 460.

[13] S. Vinco, D. Chatterjee, V. Bertacco, and F. Fummi, "SAGA: SystemC Acceleration on GPU Architectures," in *IEEE Design Automation Conference (DAC)*, San Francisco, California, USA, June 2012, pp. 115 – 120.

[14] G. Kunz, D. Schemmel, J. Gross, and K. Wehrle, "Multi-level Parallelism for Time- and Cost-efficient Parallel Discrete Event Simulation on GPUs," in *ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation (PADS)*, July 2012, pp. 23–32.

[15] M. Nanjundappa, H. Patel, B. A. Jose, and S. Shukla, "SCGPSim: a fast SystemC simulator on GPUs," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, Yokohama, Japan, January 2010, pp. 149 – 154.

[16] R. Azevedo, S. Rigo, M. Bartholomeu, G. Araujo, C. Araujo, and E. Barros, "The ArchC Architecture Description Language and Tools," *International Journal of Parallel Programming*, vol. 33, no. 5, pp. 453–484, October 2005.

[17] N. Ventroux, A. Guerre, T. Sassolas, L. Moutaoukil, C. Bechara, and R. David, "SESAM: an MPSoC Simulation Environment for Dynamic Application Processing," in *IEEE International Conference on Embedded Software and Systems (ICESS)*, Bradford, UK, July 2010.

[18] N. Ventroux, T. Sassolas, A. Guerre, and C. Andriamisaina, *Multicore Technology: Architecture, Reconfiguration and Modeling*. CRC Press, June 2013, ch. SESAM Prototyping Solution, pp. 63 – 104.

[19] Mips Technologies, "http://www.mips.com."

[20] C. Bechara, N. Ventroux, and D. Etiemble, "Towards a Parameterizable Cycle-Accurate ISS in ArchC," in *ACS/IEEE International Conference on Computer Systems and Applications (AICCSA)*, Hammamet, Tunisia, May 2010.

[21] ARM, "CoreTile Express A9x4 Technical Reference Manual Cortex-A9 MPCore (V2P-CA9)," 2011.

[22] Tilera TilePro64, "http://www.tilera.com."

[23] ARM, "http://www.arm.com."

[24] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "CACTI 6.5, www.hpl.hp.com/research/cacti/."