A New Parallel SystemC Kernel Leveraging Manycore Architectures

Nicolas Ventroux, Tanguy Sassolas
CEA, LIST,
Computing and Design Environment Laboratory
91191 Gif-sur-Yvette CEDEX, France
Email: nicolas.ventroux@cea.fr

***Abstract*—The complexity of system-level modeling is continuously increasing. Electronic System Level (ESL) design requires fast simulation techniques to control future SoC development cost and time-to-market. However, SystemC simulations are sequential and then limited by single-thread performance. In this paper, we present a new parallel SystemC kernel that efficiently leverages the multiple cores of a host machine, reaching high simulation performance without relaxing accuracy. It supports atomic parallel evaluation of SystemC processes and repeatable execution for HW/SW debugging. This new kernel is fully compliant with existing standards and easy to integrate in any existing SystemC model. Evaluations show a maximum acceleration of 34x compared to Accellera SystemC on a 64-core AMD Opteron machine.**

I. Introduction

Electronic System Level (ESL) design and verification require efficient modeling and simulation environments to take into account the complexity of modern ICs composed of billions of transistors. SystemC [1] is a hardware description language standard for system-level modeling established by major EDA vendors through the *Accellera Systems Initiative* and widely used in the IC industry. Together with TLM (Transaction Level Modeling) [2], it provides an effective framework for HW/SW co-design, system-level modeling and performance evaluation, design space exploration (DSE), high-level verification, as well as early development of system software.

SystemC is a specific HW description API and a Discrete Event Simulation (DES) kernel that emulates parallel execution through a cooperative sequential evaluation. As a result, the execution speed of SystemC models depends on a single thread execution performance. Therefore, with MPSoC modeling for instance, the simulation speed is directly divided by the number of simulated cores. Such simulation length increase is unacceptable for DSE or software development. To allow fast SystemC simulation using highly parallel modern hosts, parallel SystemC implementations are needed.

The contribution of this paper is a new SystemC kernel named *SCale* able to simulate in parallel any SystemC and TLM 2.0 model, while guaranteeing the atomic evaluation of SystemC processes.

This paper is organized as follows. Section 2 highlights the technical challenges. Section 3 presents the related work. Section 4 describes the *SCale* SystemC kernel. Section 5 explains how to use *SCale* with any SystemC simulation environment. Section 6 evaluates the kernel and compares its performance with a similar existing work. Finally, Section 7 concludes the paper.

II. Technical challenges

SystemC uses a cooperative multithreading technique to model concurrency, ensuring that all processes are sequentially evaluated without interruption between two *wait* statements. Even if the process evaluation order is unpredictable in SystemC, the sequentiality imposed by the kernel ensures deterministic and predictible simulations for early SW development and simulator-based debugging. When the design is simulated multiple times using the same stimulus and the same version of the simulator, the process ordering between different runs will not vary.

Thus, SystemC simulations hide most of concurrency errors and design problems. For this reason, many works propose techniques to detect non-determinism anomalies. There are three different approaches: automatic and static verification based on model checking [3], comparison of multiple simulations with different scheduling orders [4]–[6] or post-analysis of simulation traces [7].

On the contrary, parallel SystemC simulations can exhibit various types of dependent process behavior, such as race conditions. Apart from globally shared variables, which are often bad design practices, a typical example is the existence of shared variables between a module's processes. But TLM brings even more examples of process dependencies when, for instance, multiple TLM masters share a TLM slave and then its member variables. However, a parallel SystemC kernel compliant with the standard must ensure that a process evaluation behaves like an atomic section and provides a deterministic behavior.

III. Related Work

Most existing parallel SystemC frameworks do not deal with parallel process atomicity issues and leave it to the designer who must avoid shared variables or use exclusion mechanisms when necessary. Among these, some techniques consist in dividing a model into several subsystems distributed on multiple cores [8], [9]. However, this introduces a significant overhead mainly due to distant synchronizations usually performed through MPI communications. For this reason, relaxed synchronizations have been proposed in [10], [11] but introduce temporal deviation errors or simulation overheads to preserve full accuracy. All these techniques suffer from limited scalability due to frequent synchronizations, and simulation speed depends on the natural concurrency of simulated architectures. Many other solutions propose to modify the SystemC kernel to parallelize the evaluation of processes [12]–[14]. The main idea consists in using specific thread containers to support the parallel evaluation of independent processes. In addition to parallel evaluation, some existing works proposed to accelerate the execution of individual processes by using GP-GPU [15]. But, most of these solutions come with limitations and modeling constraints, or have a limited practical interest for ESL modeling since little data-level parallelism exists at TLM level. All of these solutions have been proved to increase simulation speed, but removing concurrent shared-memory communications in most SystemC/TLM models is either unfeasible or drastically limits simulation performances.

Contrary to these approaches, more recent works proposed automated solutions to guarantee the atomicity of simulations. [16] statically analyzed data dependencies in SystemC processes to detect non-commutative actions. With this information, an execution process ordering is defined to guarantee the atomicity. Dependent processes are sequentially evaluated and independent ones are run in any order without respecting delta and time cycles. The use of static analysis methods is however limited by the model complexity. [17] introduced the use of specific primitives to associate a duration to each process action. Temporal independent processes can be executed in parallel, and this implementation is well-suited for temporal decoupling modeling. However, cycle-based simulations using SystemC channels generate frequent kernel synchronizations and using the Accellera SystemC, which is a non-parallel kernel, imposes a too significant overhead for a parallel simulation. [18] proposed the use of zones to gather modules. All processes of a zone are sequentially evaluated into a container thread named worker, ensuring determinism. A process and its context can be reassigned during the evaluation phase to another zone in case of inter-zone communications, but it can no longer access its initial zone afterwards. This limits its use with temporal decoupling models. In addition, atomicity is only guaranteed when using TLM sockets or primitive channels. In addition, in TLM shared-memory-oriented models, most processes are dependent, and the parallelism is then limited with this approach.

In the next section, we present a new parallel SystemC kernel designed to leverage the computing power of multiple host cores. This kernel can be easily and efficiently used in any SystemC / TLM simulator and guarantees atomicity of simulations.

IV. SCALE DESCRIPTION

We developed a new lightweight SystemC kernel named *SCale* able to parallelize the evaluation of SystemC processes on multiple host cores. It supports the parallel simulation of any SystemC 2.3.1 [1] and TLM 2.0 [2] model annotated with only few specific *SCale* primitives. *SCale* is very similar to the Accellera SystemC kernel. All existing simulation phases in the Accellera SystemC kernel are present in *SCale*. It brings no structural or usage limitations and is easy to integrate into existing SystemC designs.

A. Founding Principles

SCale objective is to maintain an atomic evaluation of processes and to warn the designer when a deviation happens. As *SCale* implements the evaluation-update phase, whenever a model uses only SystemC channels for inter-process communications, the behavior in parallel is guaranteed to be fully accurate and deterministic. Otherwise, processes can compete for shared resources and memory spaces through any kind of communication layers (TLM, DMI, global variables) and some *SCale* primitives can be used to prevent the occurrence of ordering errors. Fortunately, such dependency between processes are rarely exhibited during simulation. This is due to the fact that a well-designed parallel architecture maximizes parallelism between entities and therefore limits conflicting communications. Using this property, *SCale* was designed to efficiently monitor shared-resource accesses during simulation, assert that no ordering conflict occurs between processes and sequentialize them otherwise.

Another challenge addressed by *SCale* is that SystemC simulation must be repeatable. This is key to allow hardware model and software debugging by being able to reproduce bugs during multiple executions of the same design. Nevertheless, having a fully deterministic SystemC scheduler can hide race conditions and lead to design errors. In addition, because the simulation host is undeterministic, ensuring a repeatable execution order between the threads is hard. As a result, SCale eagerly keeps a certain level of non-determinism between successive executions to better simulate real HW parallel systems and to emphasize concurrent errors. Still, *SCale* can also produce the global process ordering from any simulation and use it as an input for future *SCale* executions to guarantee determinism if needed.

B. Parallel Evaluation

To efficiently implement the parallel evaluation phase, SystemC processes are executed inside multiple resident and independent kernel-level threads (POSIX). Each thread is statically allocated on a different physical host core. The first thread executes the kernel while the other ones, named *workers*, perform *SC_METHOD* and *SC_THREAD* evaluations. A busy-wait is performed by the kernel thread until the end of the evaluation phase to minimize the worker synchronization costs. *SC_METHOD* processes are functions called by the worker, and *SC_THREAD* processes are user-level threads using *ucontext* primitives executed inside a worker's context (Fig. 1).

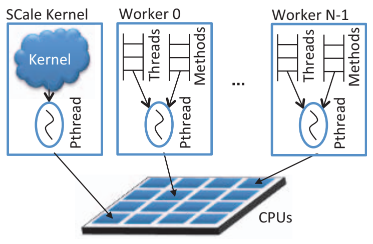

Figure 1: *SCale* SW architecture

Every SystemC process is statically allocated by the user to a given worker using a specific primitive $worker\_affinity <<$ which can be used at process instantiation. For dynamic processes created with *sc_spawn*, the option member *set_worker_affinity(uint32_t worker_id)* achieves similar behavior. A process without a specified affinity is attached to worker 0. If all processes are on worker 0, the evaluation phase is sequential like with Accellera SystemC. The behavior of *SCale* on only 1 CPU has been validated as being identical to the Accellera kernel. Indeed, since in both kernels the elaboration phase and the process scheduling are similar, simulations are exactly the same when using 1 CPU.

Processes are associated to a given worker and cannot switch to another one. In the example below, the process *do_count* will be executed on worker *ID*.

```
SC_THREAD (do_count)
sc_sensitive << clock.pos();
worker_affinity << ID;
```

Another major difference with the Accellera kernel is that all SystemC primitives are thread-safe. Kernel shared-resources, like *immediate*, *delta*, *timed*, and *update* events queues, are vectorized to support parallel write accesses.

C. Leveraging SCale with temporal decoupling

As it parallelizes the evaluation phase but not the kernel processing itself, *SCale* yields better speed-up for model with compute intensive SystemC processes. Thus, *SCale* proposes the use of a specific primitive to implement temporal decoupling for TLM simulations. This primitive named *sc_global_quantum_sync* performs a global quantum synchronization. Using this primitive ensures optimum performances when executing TLM models with *SCale*. Indeed, the timed notification phase looks for the nearest timed event to wake-up all the sensitive processes. If the processes do not wait for the same timed event, they cannot be executed on workers in parallel (Fig. 2-a). On the contrary, synchronizing all *SC_THREAD* processes on regular synchronization times guarantees a full parallel evaluation of processes (Fig. 2-b).

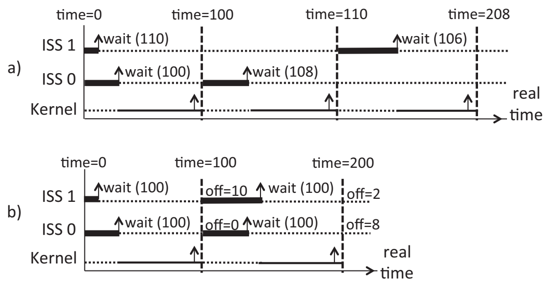

Figure 2: Temporal decoupling implementation.

D. Shared resources management

As stated earlier, keeping the atomicity of SystemC processes is key to maintain the simulation correctness. With *SCale*, the user must annotate all potentially shared resources with specific monitoring primitives. While this annotation is manual, it could be automated using dependency analysis tools. 3 types of primitives are available.

The *thread-safe* primitives *multi_lock* and *multi_unlock* must be used to ensure exclusive accesses to shared resources. Based on POSIX mutexes, these functions allow the reservation and the release of multiple resources. Indeed, multiple locks can be taken when using, for instance, hierarchical NoCs. To prevent the risk of deadlock, the reservation is automatically freed upon a call to the SystemC *wait* function and resumed upon process wake-up. These primitives are implemented by keeping a taken mutex list per worker. The user must be aware of potential deadlocks when nested shared resource accesses are not performed in the same order by all threads. If the case arises, using the same mutex for all shared resources shall solve the issue while reducing parallelism and performance.

The *atomicity-check* primitive *sc_process_conflict_check* is used to monitor all process accesses to shared resources and verify that access order complies with an atomic evaluation of all processes. As it is intended for easy integration within memory-mapped TLM slaves, this function takes as input the access base address, its size, and a boolean to differentiate read and write accesses. Because multiple workers can call it in parallel, this function is thread safe. During the parallel evaluation phase, all read accesses to a monitored address are stored in a compact manner. If at least a write access occurs on a monitored address, a dependency graph is dynamically built and continuously updated for every worker accessing the address. At the end of the parallel evaluation phase, every generated dependency graph is analyzed (using Tarjan's strongly connected components algorithm) to verify it is acyclic. Otherwise, multiple workers' accesses were interleaved, breaking the atomicity rule, and information is issued to the end-user. After performing a per-address check, all graphs are merged in a unique global dependency graph, and the same method is used to detect an atomicity violation. The resulting global dependency graph represents one sequential process evaluation order that would have resulted in the same model state.

Finally, some *elaboration* primitives can be used during the elaboration phase to reduce the *atomicity-check* cost or solve atomicity-conflicts. *sc_process_conflict_readonly_mem* specifies that a memory range is read-only. As a result, no atomicity issue can occur on these addresses during evaluation. This is later used to automatically bypass *sc_process_conflict_check* processing and reduce monitoring cost. On the other hand, *sc_process_conflict_shared_mem* specifies that a memory region is shared in read/write mode between processes and therefore prone to atomicity conflict. When an access to an address in a shared region is detected by *sc_process_conflict_check*, any other worker trying to access to any address of this shared region is yielded before the access occurs. The preempted workers are evaluated after the parallel evaluation phase in a specific sequential phase. Two other functions can be used to specify the granularity of the atomicity checking mechanism: *sc_process_conflict_resolution_set* and *sc_process_conflict_page_size_set*. The former reduces the size of the monitoring structure by considering multiple addresses as one. However, it may lead to false conflict detection if different workers access different successive addresses. The later specifies a splitting size for the memory conflict structure to speed-up the memory address search and to allow effective parallel calls to *sc_process_conflict_check*.

All shared resource features implemented in *SCale* use a per worker basis. Since all processes on a worker are sequentially evaluated, the process atomicity problem is equivalent to analyzing atomicity issues at the worker level.

E. Repeatable simulation

To enable efficient debugging and compatibility with the SystemC standard, simulations performed using *SCale* are repeatable, yielding the exact same results. As for the Accellera SystemC kernel, repeatability is only ensured if the model remains unchanged. To implement this feature, the worker execution order obtained from the global dependency graph for each evaluation phase can be saved in a file. For each evaluation phase with at least one dependency issue, the file contains the evaluation phase timestamp, a list of independent workers, and an ordered list of dependent workers. The ordered list is obtained by using a standard topological sorting algorithm of the global dependency graph.

Then, the *SCale* scheduler can take into account the file information to maintain the process execution order. If no information is available in the file for the current evaluation phase timestamp, the scheduler executes all workers in parallel. Otherwise, it schedules the parallel evaluation of the independent workers, then schedules the sequential execution of dependent workers according to the ordered list. No atomicity conflict can arise during the repeated simulation. Indeed, for a conflict to happen during the parallel execution phase, a dependency between at least two processes must exist and would have been detected as such in the first execution. Therefore, the processes would have been sequentially scheduled during the repeated run. *sc_set_scheduling_order_backup* and *sc_set_scheduling_order_replay* functions can be used to activate the scheduling order backup and replay.

F. SCale *Kernel loop*

Given all previously described features, the *SCale* scheduling loop differs from the Accellera one. Still, it maintains all classical SystemC phases as illustrated in Fig.3.

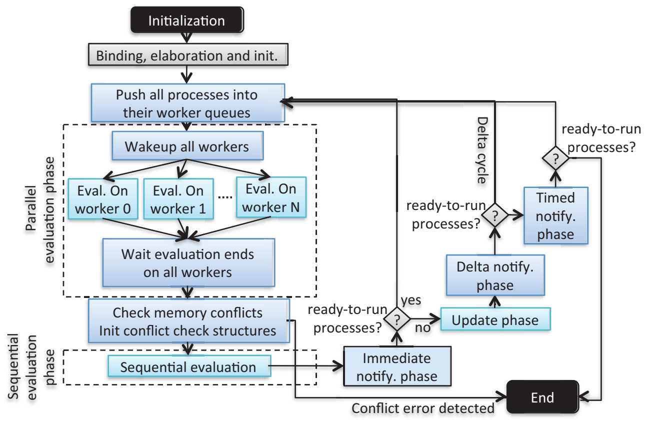

Figure 3: *SCale* SW kernel diagram.

During the kernel initialization, and after the binding and the elaboration, all workers are created and attached to different logical host cores. After their initializations, the workers wait for a *ready* semaphore to begin the evaluation phase. Then, the allocation of the main SystemC kernel thread is forced on the first physical core, and all registered processes are pushed in their respective worker queues. The *ready* semaphore is set to allow the workers to start in parallel the evaluation of their process queues. Each worker starts by sequentially evaluating all its *SC_METHOD* processes, and then cooperatively executes all its *SC_THREAD* ones. During the parallel evaluation of all SystemC processes by workers, the kernel does a low-latency polling through the call of atomic functions. This active barrier guarantees that the host OS will never yield the kernel thread, which could generate a low kernel reactivity and, therefore, a significant overhead. At the end of the parallel evaluation phase, ordering conflicts are checked, and dependent workers, yielded during the parallel evaluation phase, are successively woken-up to execute all preempted processes. The other steps of the SCale kernel are similar to the Accellera SystemC ones. The *immediate*, *delta*, and *timed notifications* are successively executed, modeling *delta-cycles* and simulation steps. When the user wants to repeat the previous simulation, all scheduling order information are stored in a specific internal structure, and processes are evaluated by following this given order in both the parallel and sequential evaluation phases.

V. Using SCale

Using the SCale kernel is almost transparent to the user. If the simulator is known by the designer, only few minutes of work are necessary to adapt an existing Accellera SystemC simulation environment to SCale (experienced with SESAM [19] and SimSoC [20]). First of all, in the *main* function, the number of maximum workers must be set with the primitive *sc_set_nb_worker_max(uint32_t val)*. Then, each SystemC process must be associated to a worker. Finally, all presented primitives can be inserted to implement temporal decoupling, manage shared resources, or activate the repeatable mode. SCale has been fully instrumented to measure its potential overheads: the shared resources access time, the kernel phase duration, or the workers activity rates are one among many monitored parameters. This instrumentation helps the designer understand where optimizations can be applied on the model and the application.

VI. Evaluations

In this section, we evaluate the performance of *SCale* using two simulation environments named SESAM and SimSoC. Sections VI-A and VI-B present the experimental setup and benchmarks. Because *SCale* benefits from architecture and application parallelism, evaluations are performed on a massively parallel architecture running parallel applications. Finally, section VI-C analyzes *SCale* performance and limitations.

A. *Manycore architecture model*

In order to evaluate our new parallel SystemC kernel, two different simulation environments modeling a 2D-mesh manycore architecture are considered. This architecture uses a physically distributed shared memory (see Fig. 5).

The first environment, named *SESAM* [19], is a SystemC/TLM simulation framework for the exploration of multi and manycore architectures. SESAM uses functional ISSes based on the ArchC HDL [21] and provides a set of timed/untimed/functional or cycle-accurate IPs (NoC, memory controllers, etc.). Our evaluations use a 2D-mesh parallel NoC with multiple shared-resource locks in slave wrappers to protect memory banks. With SESAM, the model is composed of a central control processor (CCP) and up to 62 processing units (PU), totaling a maximum number of 63 processing cores. All processing cores are MIPS32 R2 in SESAM.

The second environment, named *SimSoC* [20], is a SystemC/TLM 2.0 simulation framework using ISSes with dynamic binary translation (DBT). ISSes directly communicate using DMI with a single shared memory instead of multiple memory banks. With *SimSoC*, the CCP does not exist and PUs are able to boot by themselves. Then, the maximum number of processing cores is 62. DBT with no specialization (mode M1) is used to ensure a more regular execution time between workers and then, better parallelization. With SimSoC, PowerPC processing cores are chosen.

In both environments, the total number of SystemC processes is equal to the number of ISS in the model. All private modules composing the processing unit (PU) are allocated on the ISS's worker. Internal PU communications are sequential and remain inside the worker. All other SystemC processes are gathered in another single worker. In addition, both environments use a temporal-decoupling technique based on a system global quantum to model timings and limit synchronizations with the SystemC kernel.

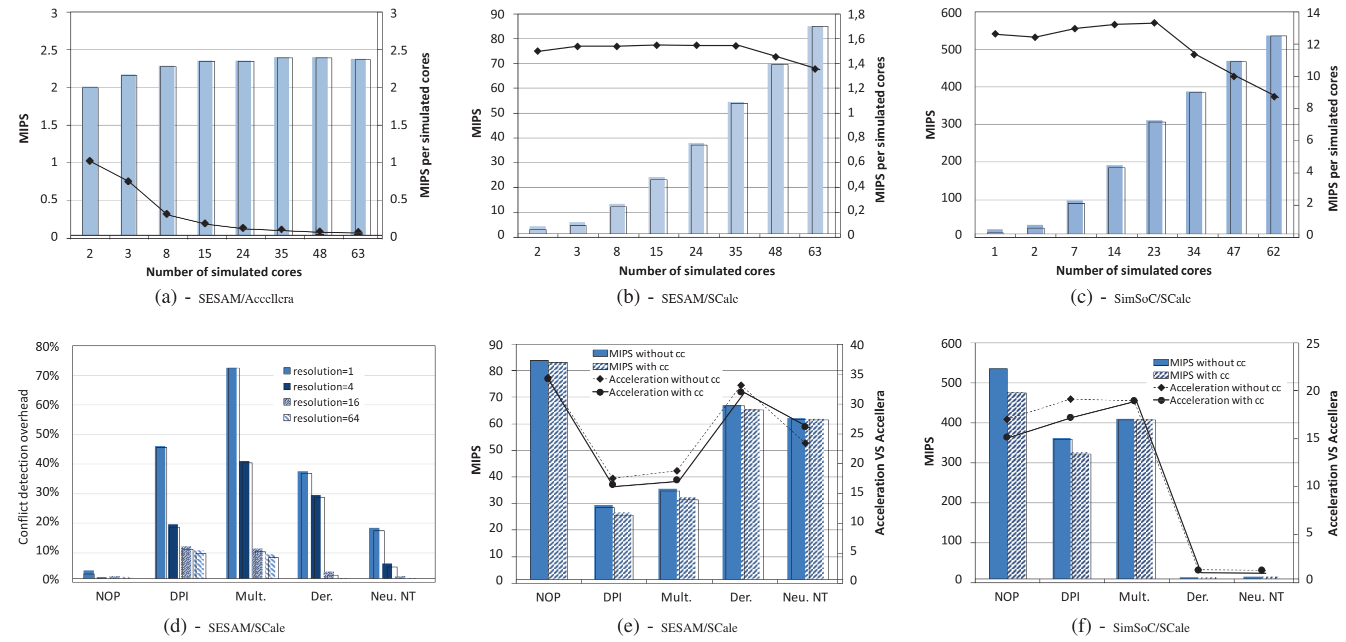

Figure 4: (a) Performance analysis of SESAM with Accellera SystemC by increasing the number of simulated cores (NOP); (b) performance analysis of SESAM with *SCale* on 63 workers by increasing the number of simulated cores (NOP); (c) performance analysis of SimSoC on 62 workers by increasing the number of simulated cores (NOP); (d) overhead study of monitoring shared-memory accesses with SESAM; and analysis of SESAM (e) and SimSoC (f) performance with *SCale* with and without conflict checking (cc) evaluation and, of the resulting accelleration compared to Accellera (resolution of 16).

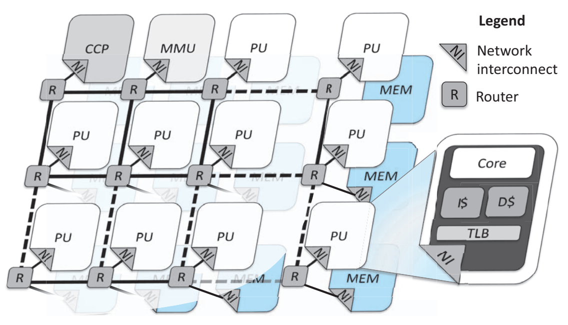

Figure 5: Manycore architecture. A variable number of processing units (PU) are connected together through a square 2D-mesh NoC and access a physically-distributed shared memory. A Central Control Processor (CCP) and a Memory Management Unit (MMU) can be used for task management and memory allocation.

B. Benchmarks

In order to evaluate *SCale* performance, five 62-task parallel shared-memory applications of approximately 1 billion instructions each are considered:

– *NOP* is based on multiple loops of nop operations and has been chosen to highlight the maximum acceleration that can be reached.
– *DPI* is the lightweight deep packet inspection (DPI) application distributed by Packetwerk [22], which consists in analyzing multi-protocol Ethernet packets.
– *Mult.* is a parallel matrix multiply application.
– *Der.* is a parallel Deriche image processing application based on a fast 2D-Gaussian convolution IIR approximation.
– $Neu.NT$ is a road sign detection application based on Convolutional Neural Network (CNN) [23].

C. Performance analysis

All results have been obtained on an AMD Opteron 6276 at 2.3 GHz, composed of 4 sockets of 8 multithreaded cores (total of 64 logical cores), running *SCale* or Accellera SystemC QT 2.3.1 on a Debian 6.0. In the following experiments, a quantum of 10K cycles is considered for ISS synchronizations.

Fig. 4-a represents the simulation performance obtained with Accellera SystemC when increasing the number of simulated cores. As expected, the number of MIPS remains approximately the same, and simulation speed is divided by the number of simulated cores. Accellera SystemC does not scale with model complexity and is limited by the performance of a single host core.

On the contrary, fig. 4-b and 4-c show the theoretical speed that could be obtained with SESAM and SimSoC using *SCale*. These figures depict the maximum numbers of MIPS obtained with 63 workers with SESAM and SimSoC when increasing the number of simulated cores. Contrary to Accellera, the number of MIPS per simulated cores remains very high and rather regular, whatever the simulation environment is. The numbers of MIPS per ISS are about 1.5 with SESAM and 12 with SimSoC. The theoretical maximum number of MIPS obtained with SESAM and SimSoC is respectively 85 MIPS and 536 MIPS.

Fig. 4-d shows the overhead obtained by monitoring all memory accesses and detecting non-atomic behaviors with SESAM. The resolution is expressed in number of bytes. As an example, a resolution of 4 can monitor 32-bit memory accesses. With our use-case applications, the maximum measured overhead is about 72 % with the matrix multiply application, which generates many memory accesses compared to processing time. However, the overhead is limited to approximately 10 % with a resolution of 16. This corresponds in SESAM to the 4-byte cache line sizes and is then sufficient to detect ordering anomalies.

Finally, figures 4-e and 4-f depict the simulation performance obtained with *SCale* compared to Accellera SystemC, with and without the activation of conflict checking. In Fig. 4-e and SESAM, the maximum acceleration is about 34x on 64 host cores compared to Accellera SystemC while keeping an exact accuracy and reaching 84 MIPS. The conflict checking mechanism has almost no impact on simulation speed since the programming model of the architecture implemented in SESAM guarantees the atomicity of memory accesses. However, during this evaluation, *SCale* found out different application bugs initially hidden by Accellera SystemC. We were then able to fix the race conditions due to non-aligned memory problems. The acceleration of benchmarks with lots of memory contentions (*DPI* and *Mult*) is only about 16x. The application parallelism level has a direct impact on *SCale* performances.

Concerning SimSoC in Fig. 4-f, the maximum acceleration is about 19x on 64 host cores, and simulation speed is equal to 534 MIPS. Accuracy is guaranteed by using the *SCale* conflict checking mechanism. The conflict checking mechanism overheads remain limited to 12.5 % depending on the cache hit rates of simulated ISSes. Contrary to SESAM, SimSoC applications use lock variables to synchronize threads. Such variables are declared as fully-shared to allow *SCale* to solve race condition issues and guarantee the atomicity of processes. As with SESAM, the acceleration of *DPI* and *Mult* are impacted by the memory contentions they generate. Concerning *Der* and *Neu. NT*, the unique shared-resource protection drastically limits the acceleration. A more adapted memory hierarchy could be implemented to solve this issue. Thus, in SESAM the multiple memory banks do not exhibit the same contentions. Contrary to SimSoC, a smart MMU is used in SESAM to distribute instructions and data in different memory banks to increase parallel accesses. A non-parallel system shows no acceleration with *SCale*. Therefore, using *SCale* could highlight parallelism issues on the architecture or the application.

Among previous works, only [17] uses similar experimental conditions. An acceleration of 15x is obtained on a 48-core AMD Opteron 6176 when simulating a temporal decoupled TLM MPSoC model. The application is the Conway's Game of Life and the image processing is sliced on the multiple CPUs of the platform. Considering the same application and a similar simulator (SESAM) with *SCale*, the acceleration reaches 21.7x on 48 cores compared to Accellera SystemC. This represents an improvement of +44 %. This result demonstrates the high-scaling potential of *SCale* to leverage the multiple cores of the host machine when simulating MPSoC architectures.

VII. Conclusion

This paper presented *SCale*, a lightweight parallel SystemC kernel able to leverage the multiple host cores of a host machine to accelerate SystemC / TLM 2.0 simulations. Even if DES parallelization is a difficult and well-known problem, we succeeded in implementing a scalable, efficient and user-friendly solution. Any standard SystemC model can be simulated by *SCale* with minimal instrumentation. Because it guarantees the atomicity of process evaluation, *SCale* produces exact simulation results. It also allows repeatable simulation for fast debugging of hardware and software. SCale brings a unique capacity to accelerate TLM simulations of multi and manycore architectures. Modeling a manycore architecture using different simulation frameworks shows a maximum acceleration of 34x compared to the Accellera SystemC on a 64-core AMD Opteron machine. In the future, we plan on studying the automatic insertion of *SCale* primitives that guarantee the atomic evaluation of SystemC processes.

References

[1] Accellera Systems Initiative, "SystemC 2.3.1," http://www.accellera.org.

[2] ——, "TLM 2.0," http://www.accellera.org.

[3] N. Blanc and D. Kroening, "Race Analysis for SystemC using model checking," in *ICCAD*, San Jose, USA, November 2008.

[4] C. Helmstetter, F. Maraninchi, L. Maillet-Contoz, and M. Moy, "Automatic generation of schedulings for improving the test coverage of systems-on-a-chip," in *FMCAD*, San Jose, USA, November 2006.

[5] H. Le and R. Drechsler, "Towards Verifying Determinism of SystemC Designs," in *DATE*, Dresden, Germany, March 2014.

[6] C. Schumacher, J. Weinstock, R. Leupers, and G. Ascheid, "SCANDAL: SystemC analysis for nondeterministic anomalies," in *FDL*, Vienna, Austria, September 2012.

[7] A. Sen, V. Ogale, and M. Abadir, "Predictive run-time verification of multiprocessor SoCs in SystemC," in *DAC*, Anaheim, USA, June 2008.

[8] H. Ziyu et al., "A Parallel SystemC Environment: ArchSC," in *ICPADS*, Shenzhen, China, December 2009.

[9] J. Peeters, N. Ventroux, T. Sassolas, and L. Lacassagne, "A SystemC TLM Framework for Distributed Simulation of Complex Systems with Unpredictable Communication," in *DASIP*, Tampere, Finland, November 2011.

[10] P. Combes, E. Caron, and B. Chopard, "Relaxing Synchronization in a Parallel SystemC Kernel," in *ISPA*, Sydney, Australia, December 2008.

[11] I. Pessoa, A. Mello, A. Greiner, and F. Pecheux, "Parallel TLM Simulation of MPSoC on SMP Workstation: Influence of Communication Locality," in *ICM*, Cairo, Egypt, December 2010.

[12] P. Ezudheen et al., "Parallelizing SystemC kernel for fast hardware simulation on SMP machines," in *PADS*, Lake Placid, New York, USA, June 2009.

[13] C. Schumacher, R. Leupers, D. Petras, and A. Hoffmann, "parSC: Synchronous Parallel SystemC Simulation on Multi-Core Host Architectures," in *CODES+ISSS*, Scottsdale, Arizona, USA, October 2010.

[14] N. Ventroux, J. Peeters, T. Sassolas, and J. Hoe, "Highly-Parallel Special-Purpose Multicore Architecture for SystemC/TLM Simulations," in *SAMOS*, Samos, Greece, July 2014.

[15] S. Vinco, D. Chatterjee, V. Bertacco, and F. Fummi, "SAGA: SystemC Acceleration on GPU Architectures," in *DAC*, San Francisco, California, USA, June 2012.

[16] W. Chen, X. Han, and R. Dômer, "Out-of-order Parallel Simulation for ESL Design," in *DATE*, Dresden, Germany, March 2012.

[17] M. Moy, "Parallel Programming with SystemC for Loosely Timed Models: A Non-Intrusive Approach," in *DATE*, Grenoble, France, March 2013.

[18] C. Schumacher et al., "legaSCi: Legacy SystemC Model Integration into Parallel SystemC Simulators," in *IPDPSW*, Boston, USA, May 2013.

[19] N. Ventroux et al., "SESAM: an MPSoC Simulation Environment for Dynamic Application Processing," in *ICESS*, Bradford, UK, July 2010.

[20] C. Helmstetter, V. Joloboff, and H. Xiao, "SimSoC: A full system simulation software for embedded systems," in *International Workshop on OSSC*, Guiyang, China, September 2009.

[21] S. Rigo, G. Araujo, M. Bartholomeu, and R. Azevedo, "ArchC: a systemC-based architecture description language," in *SBAC-PAD*, Paris, France, October 2004.

[22] Packetwerk. (2014, September) Lightweight Inspection (DPI) for libpeak. [Online]. Available: https://github.com/packetwerk/libpeak

[23] H. Trinh, M. Duranton, and M. Paindavoine, "Efficient Data Encoding for Convolutional Neural Network application," *TACO*, vol. 11, no. 4, January 2015.