Standard-compliant Parallel SystemC simulation of Loosely-Timed Transaction Level Models

Gabriel Busnot, Tanguy Sassolas, Nicolas Ventroux
CEA, LIST
Computing and Design Environment Laboratory
91191 Gif-sur-Yvette CEDEX, France
{*first*}.{*last*}@cea.fr

Matthieu Moy
Univ Lyon, EnsL, UCBL, CNRS, Inria
LIP
F-69342, LYON Cedex 07, France
matthieu.moy@univ-lyon1.fr

Abstract— To face the growing complexity of System-on-Chips (SoCs) and their tight time-to-market constraints, Virtual Prototyping (VP) tools based on SystemC/TLM must get faster while keeping accuracy. However, the Accellera SystemC reference implementation remains sequential and cannot leverage the multiple cores of modern workstations. In this paper, we present a new implementation of a parallel and standard-compliant SystemC kernel, reaching unprecedented performances. By coupling a parallel SystemC kernel and memory access monitoring, we are able to keep SystemC atomic thread evaluation while leveraging the available host cores. Evaluations show a ×19 speed-up compared to the Accellera SystemC kernel using 33 host cores reaching speeds above 2000 Million simulated Instructions Per Second (MIPS).

I. Introduction

Electronic System Level (ESL) design and verification is increasingly challenging due to the soaring complexity of System-on-Chips (SoCs) and time to market constraints. Thus, tools involved in the SoC design process have to support this trend. Among these tools, modeling and simulation environments are intensively used in early development stages to elaborate Virtual Prototypes (VPs). Virtual prototyping is a cost-effective technique which consists in the realization of a software model of the actual chip under design. VPs then allow for HW/SW co-design, system-level modeling at various levels of granularity, verification, performance evaluation or Design Space Exploration (DSE).

SystemC [1] is broadly used for VPs design in both industrial and academic communities. It is a C++ based HW description language supported by the *Accellera Systems Initiative*. We are interested in the Transaction-Level Modeling (TLM) [2] standard for SystemC which enables higher level of abstractions for faster simulation, increased interoperability and model reuse. However, as specified in the IEEE SystemC/TLM standard, concurrency is emulated using the co-routine semantics, implemented by the reference Accellera kernel with cooperative sequential processes evaluation. It guarantees deterministic execution but also enforces single-threaded evaluation. As multicore SoCs are getting ubiquitous, the simulation speed decreases in inverse proportion to the number of cores in the model. To tackle this issue, it is possible to take advantage of multicore host platforms, but a parallel implementation of SystemC is needed.

We propose a parallel and standard-compliant SystemC kernel which guarantees process evaluation atomicity and simulation reproducibility. It supports any TLM model including loosely-timed coding style together with the Direct Memory Interface (DMI) protocol. Our technique has a limited overhead even when used with the fastest Instruction Set Simulators (ISS's) available. We rely on a conflict-avoidance heuristic combined with conflict-detection and a fast rollback mechanism. Also, little source code modifications are required.

The rest of the paper is organized as follows. Section II explains the challenges exposed by SystemC parallelization. Section III presents the related works. Section IV details the contributions of this paper. Section V describes the experimental setup and analyses our experimentation results. Finally, section VI concludes the papers and exposes our perspectives.

II. challenges of Parallel SystemC/TLM

Parallelizing SystemC presents some fundamental obstacles which are further detailed in [3], [4]. Indeed, the reference SystemC implementation [5], as most Discrete Event Simulator (DES), models concurrency using the co-routine semantics implemented with cooperative multi-threading on a single core. In a broad outline, it results in the alternance of two main simulation phases:

1) The *evaluation* phase where the new state of the model is computed as a function of the current state by the various processes evaluated sequentially;
2) The *update* phase where the kernel propagates the results of the evaluation in the model.

The evaluation phase is usually the most compute-intensive, which makes it the target of most parallelization approaches (including this paper). Theses approaches rely on the fact that sequential evaluation of processes is not strictly enforced by the standard as long as the outcome follows co-routine semantics, i.e. is equivalent to a sequential evaluation of the processes. One way to enforce co-routine semantics is to avoid any interaction between processes during the evaluation phase. This is typically the case in Register Transfer Level (RTL) models, where processes interaction is deferred to the update phase, within controlled channels such as SystemC's `sc_signal`.

However, the absence of interaction between concurrent processes cannot be enforced at higher levels of abstraction such as TLM, especially with loosely-timed TLM. Indeed, TLM processes mostly interact during the evaluation phase. In addition, interactions between processes can take any form allowed by C++, in particular shared states and raw

978-1-7281-4123-7/20/$31.00 ©2020 IEEE

pointers. TLM even encourages these practices by providing the DMI protocol. DMI allows initiators to directly access the underlying memory of a component using raw pointers and is extensively used with memories.

Also, only processes that are scheduled at the same date are natural candidates to run in parallel. Again, it is a common situation in RTL models where processes are often triggered by a system clock. However, in TLM models, processes are not synchronized by a central clock anymore, reducing the probability of several processes being scheduled at the same date.

Fortunately, another common acceleration technique used in loosely-timed TLM is temporal decoupling. It allows a process to run ahead of simulation time by a given *quantum* of time, reducing the number of context switches and speeding up the simulation. While this can help parallelization by synchronizing the processes on the quantums, increasing the number of processes scheduled at a same date, temporal decoupling also increases the amount of time simulated between two synchronizations and therefore, the interaction between processes and the risk of race conditions.

Finally, modern ISS's such as QEMU [6] have achieved great speedups in the recent years, reaching speeds above 1000 Million simulated Instructions Per Second (MIPS) on a single-core host machine when used in standalone mode. Hence, the solutions used to allow parallel standard-compliant simulations must incur a very small overhead not to hamper the speed of a modern ISS.

Section III presents the existing work related to parallel SystemC simulation and shows that many of these issues remain unsolved.

III. Related work

A. Parallel SystemC approaches

To this day, all attempts to parallelize SystemC simulations have made some restrictive assumptions. They are usually related to the abstraction level of the models that can be simulated in compliance with the SystemC semantics and, by extension, to the type of communications used in these models.

In the early days of SystemC parallelization, mostly cycle-accurate model simulation was explored. In [7] processes scheduled during a same delta-cycle are allowed to run in parallel. In practice, such execution often yields the same result as a sequential schedule, but this property is not guaranteed. This approach is improved in [8] and [9] by studying active and passive load balancing techniques to use the available hardware more efficiently. Cycle-accurate SystemC simulations have also been performed on GPU [10] or on dedicated hardware such as [11] where a manycore chip embeds a SystemC kernel accelerator. With a different mindset, the author of [12] proposes to enhance SystemC semantics by associating a duration to processes so that they can be evaluated asynchronously. Yet, determinism is not guaranteed if the asynchronous processes do not run in complete isolation from the rest of the simulation like in a TLM simulation.

However, with the introduction of TLM, SystemC models now have a higher level of abstraction to benefit from the speed-accuracy trade-off. We target the TLM level of abstraction, for which techniques targetting cycle-accurate models are not efficient. To tackle this, Parallel Discrete Event Simulation (PDES) [13] can be used to relax the synchronization constraints between the processes of a simulation. In [14], several compute nodes are used to run a distributed simulation where synchronization occurs between neighbour nodes. Similarly, TLM-DT [15] splits the design into clusters but operate at the TLM level. Each cluster is simulated in a dedicated host thread which can run in parallel to the other clusters. Each cluster manages its own local time and synchronizes with the others through timestamped messages. Other similar approaches have been proposed such as [16] where the clusters are connected through channels with latencies. It allows a cluster to run ahead of its neighbours without risking to receive a message from the past. These solutions require all communications between clusters to use channels and are best suited to simulate platforms presenting different zones with mostly internal communication to reduce the amount of interactions, as opposed to Symmetric Multirocessing (SMP) platforms which usually rely on a single shared memory.

Another approach exposed in [17] or [18] relies on static compiler-driven analysis to identify data and event dependencies between processes. Based on that, a parallel schedule can be issued. However, data and event dependencies are hard to spot at higher levels of abstraction resulting in too much execution sequentialization. This would hamper performances. In particular, in a typical TLM model, the RAM is a variable potentially shared between all initiator components on the bus, and static analysis cannot guarantee the absence of concurrent accesses to a particular memory location without heavyweight alias analysis.

All the aforementioned approaches target at most approximately timed models which are rather slow at a few tens of MIPS. Loosely-timed models present a number of additional obstacles to parallelization as they tend to make extensive use of shared variables. Also, as the time between two synchronizations increases, so does the risk of atomicity violation while running a process. With raw simulation speed and ease of use in sight, [19] proposes a multiprocess simulation engine. Interprocess communications are then strictly restricted to messages implemented using POSIX shared memories. While determinism is not guaranteed anymore, interprocess communications must be avoided as much as possible to keep performance high. In addition, DMI cannot be used between processes, limiting the simulation speed. A contrasting approach is taken in [20]. Our work is strongly influenced by this work so subsection III-B details it further.

B. SCale, advantages and limitations

SCale, the parallel SystemC kernel described in [20] uses memory access monitoring as its central mechanism to detect and prevent process atomicity violations. Monitoring memory accesses allows for building a process dependency graph used to guarantee SystemC standard compliance and simulation reproducibility. To that extent, SCale provides an annotation function that must be called before every

memory access. However, SCale relies on user-provided annotations to tag some address ranges as shared and serialize SystemC processes that would otherwise violate the SystemC semantics.

Also, every access must be atomic with the corresponding instrumentation. That is, once a process P_0 instruments an access to a variable a_0, no other process can access a_0 before P_0 has performed its access to a_0. Otherwise, the recorded order of memory accesses could be different from the real order, inducing incorrect dependency analysis. This constraint is not addressed in [20], leaving to the user to implement. The simplest solution consists in putting the instrumentation and the access together inside a critical section protected by a mutex. But this is very costly and does not accommodate higher core counts.

The work presented in this paper, while inspired from [20], tackles its main functional limitations while providing significant speed and scaling improvements. In particular, we do not require any manual annotation of address ranges and detect shared addresses at runtime. Also, instrumentation and accesses are atomically performed without requiring any additional synchronization.

IV. Proposed parallel SystemC kernel

Similarly to [20], we start the evaluation with a *parallel phase* where all scheduled processes are launched concurrently. During this parallel phase, processes accessing potentially conflicting addresses are descheduled. We enhance this solution by removing the need for static declaration of shared and read-only addresses through dynamic shared address detection. Descheduled processes are blocked until the parallel phase is over, and are rescheduled to continue their execution in a *sequential phase*. In addition, process atomicity violation checking is now performed asynchronously and process level rollback is used if violation occurred, ensuring SystemC standard compliance. The next subsections present the main implemented mechanisms.

A. Shared Addresses Detection and Conflict Avoidance

In modern applications, shared memory addresses are too complex or even impossible to statically enumerate. Dynamic memory allocation and memory virtualization are two of the main reasons. Also, memory regions might be shared only during some phases of the simulated program. Declaring all of them as shared for the entire program might result in numerous useless sequentializations. To provide flexibility and reduce the risk of falsely shared memory regions, we rely on dynamic detection of shared addresses. It requires the instrumentation of all simulated memory accesses and is based on the ID of the process making the access and the type of the access (read or write).

To detect shared addresses a Finite State Machine (FSM) is associated to each address. Each address can independently be in one of the four states illustrated in Fig. 1:

1) NO_ACCESS: After initialization or reset.
2) OWNED: When an address has been accessed by only one process and with at least a write since last reset. This process is called the *owner* of the address.

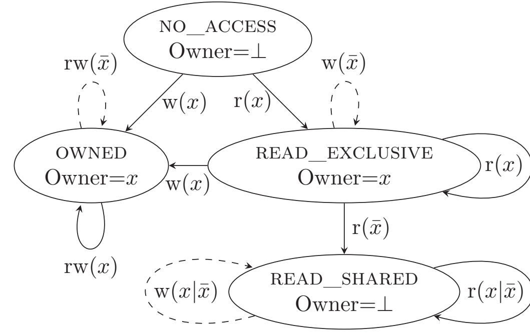

Fig. 1: Memory access monitoring FSM. x is the process doing the access from NO_ACCESS; $\bar{x}$ designates any process other than x; r and w designates read and write. Processes are descheduled on - - ➝ transitions.

3) READ_EXCLUSIVE: When an address has been only read by a single process since last reset. This process is also called the *owner* of the address.
4) READ_SHARED: When an address has been only read and by at least two processes since last reset.

The READ_EXCLUSIVE state is crucial to make the FSM efficient. Indeed, after a reset followed by one or more reads from a single process at address a_0, it is impossible to know whether a_0 is going to be read by another process, making it READ_SHARED, or if it is going to be written by the current reader process, making it OWNED. The READ_EXCLUSIVE state allows to wait for one of these scenarios. Indeed, identifying a read address as systematically READ_SHARED would result in forced sequentialization for addresses read and written by a single process (e.g. stack accesses), drastically hampering performance.

We use this FSM to prevent process atomicity violations by preventing dependencies between processes. During a parallel evaluation, accesses to shared memory causes dependencies between processes. In general, in a given simulation quantum, a process P_0 that writes to an address a_0 *depends on* all processes that accessed a_0 before P_0 and all processes that will access a_0 after will depend on P_0. Our goal is to prevent the appearance of circular dependencies which correspond to atomicity violations. In our case, any access that would introduce a dependency during the parallel phase leads to the process being descheduled before the access, that is if a process:

- tries to write to an address already owned by another process;
- or tries to read an address in the OWNED state whose owner is another process.

This ensures that no process dependency can be introduced during the parallel phase, avoinding most common conflict situations by construction. However, conflicts involving multiple addresses might occur during the sequential phase. Because of that, exhaustive conflict analysis must be performed at the end of every sequential phase, as explained in subsection IV-C.

B. FSM reset policy

In the FSM depicted on Fig. 1 no transition leave the states OWNED and READ_SHARED, meaning that

once reached, such state would last during the entire simulation. This accommodates programs whose memory accesses pattern is constant over its execution. However, addresses are often used by a thread and then by another in dataflow processing for instance. In that case, such addresses would be detected as shared, which is true at the full program level but not at the dataflow stage level. Because accesses to shared addresses cause a time-consuming process sequentialization, it is important to avoid false positives.

We achieve this by reseting all FSMs whenever a process got descheduled during the previous quantum. This reset policy relies on the observation that truly shared addresses such as mutexes are seldom used in parallel programs due to their performance cost. In practice, most of the addresses detected as shared were data being passed from a process to another. With our reset policy, the process receiving some data from another process is sequentialized only during a single quantum when it accesses this data for the first time. Then, the state of all addresses would be reset so that the process can become the new owner of the data it received. Other addresses' state would naturally be reinstated due to steady program access patterns. Other reset policies like resetting only addresses that caused a deschedule have been explored and lead to significantly worse results. Indeed, a first conflicting address is often followed by others as data is often passed as a block. Resetting all addresses is a good strategy to avoid further conflicts.

C. *Conflict Check and Recovery*

We presented in the previous sections how per address conflicts can be prevented during the parallel phase using process descheduling. However SystemC consistency conflicts involving accesses to several addresses and spread accross both the parallel and the sequential phase can still occur.

To detect such situation, it must be checked that no conflict occurred at the end of an evaluation phase despite the shared memory addresses detection. The evaluation phase flow chart of our SystemC kernel showing the parallel and the sequential phase is represented Fig. 2. At the end of the parallel phase, if there was no descheduled processes, it means that no dependency exists between processes, hence no conflict either. But in case there are descheduled processes, an exhaustive analysis must take place. To that extent, during both the parallel and the sequential phases, all memory accesses are recorded to perform the final graph analysis if needed. At the end of the sequential evaluation phase, the recorded accesses are asynchronously checked for conflicts using the same graph analysis algorithms as in [20] while the simulation continues. Because this analysis seldom takes place and is performed asynchronously using spare host cores, its impact on simulation speed is reduced to the bare minimum.

The conflict analysis results are gathered by the kernel thread during the parallel phase. Two results are returned. First, the analysis tells if there was a conflict during the checked evaluation phase. If there is no conflict, a linear ordering of the processes involved in the dependencies is saved. It is used to ensure reproducible simulation in a

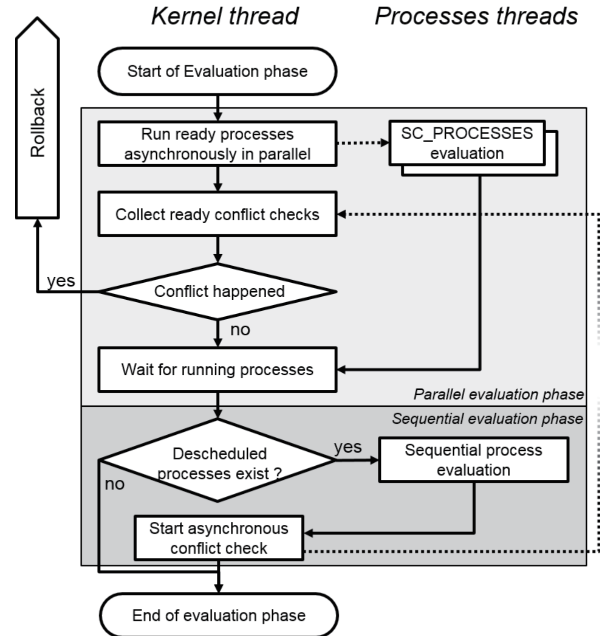

Fig. 2: Evaluation phase flow chart of our parallel SystemC kernel.

replay mode where dependent processes evaluation order is constrained. This mode allows for strict simulation reproduction for debug and analysis purposes as required by the SystemC standard. But in case of conflict, the simulation is no longer valid and it must rollback to the last saved state.

Rollback is achieved at OS process level using CRIU [21]. This software is able to perform full OS process state dump to disk and restore a process from these files. In particular, it can restore threaded OS processes, which is not possible with `fork()`-based checkpoint/restore approaches like [22]. Also, CRIU supports incremental dumps, speeding up drastically the checkpointing operation. Together with OS automatic file caching or RAM disk, this makes the process dump overhead negligible.

Concretely, an initial dump is performed before the simulation starts. Then, the simulation runs until a potential conflict arises. If so, the simulation is rolled back and run again until the conflicting quantum is reached. This quantum is sequentially evaluated to prevent the conflict from occurring again. A new snapshot is made after this quantum to be used as the next restore point if another conflict arises later on. This checkpoint/restore policy is simple but efficient as conflicts tend to be very rare in practice. Rollback is a fallback plan, should all previous mechanisms fail to prevent conflict from occurring. Preliminary experiments show that conflicts do occur when running an operating system such as Linux, and efficiently dealing with them is part of our future works

D. *Atomic Instrumentation*

One crucial property of parallel simulation correctness lies in the atomicity of the memory accesses instrumentation. Indeed, it must be guaranteed that the memory accesses are recorded in the same order as they are actually performed.

As explained in subsection IV-A, thanks to our conflict avoidance mechanism, no dependency can be introduced

during the parallel phase. It means that during the parallel evaluation phase, only concurrent reads are allowed at a given address. Because the order of concurrent reads has no influence on a process atomicity, the recorded order of concurrent reads can differ from the real order without changing the conflict analysis outcome. In case of a write, if the address is NO_ACCESS or the writer is already owner, it becomes or remains OWNED and no other process can access it during the parallel phase. If the address is READ_SHARED, then the writer is descheduled.

As a result, as long as an access is instrumented before it is performed, there is no need for extra synchronization to ensure atomicity of instrumentation together with accesses. This crucial property actually comes at no performance and complexity costs thanks to our conflict avoidance policy.

V. Evaluation

A. Experimental Setup

Experiments have been conducted on a 36-core bi-Xeon Gold 6154 clocked at 3.5GHz with frequency scaling disabled. All measures are done 3 times. The average is reported together with an error bar on the graphs.

The reference VP used for the evaluation of our contributions is a RISC-V SMP platform. Each core is modeled by an instance of QEMU encapsulated in a SystemC wrapper. The platform is composed of 1 to 32 cores (64 cores cannot run in parallel on our test machine). These cores are connected through a bus to a ROM, a RAM and a UART. DMI is used to access the memories for faster simulation. Our goal is to shorten the memory accesses simulation to the maximum to evaluate the relative overhead of instrumentation in worst-case conditions.

We have selected three benchmarks to evaluate the performance of the proposed approach:

1) Matmul: 32 classic parallel multiplications of two square matrices of size 512. Each thread computes an horizontal block of the result. Threads only synchronize before ending.
2) Deriche [23]: A 10-pass Deriche filtering is applied in place to a 4 megapixels image. This benchmark is composed of an horizontal followed by a vertical filtering, making the whole image shared by all the threads.
3) MobileNet [24]: a 31-layer classification convolutional neural network analyzing 3 triple channel 160x160 images. The parallelism potential varies depending on the computed layer and much more synchronizations occur than in the first two benchmarks.

A last validation application has been designed to verify that the *replay* mode effectively allows for reproducible simulations. It is composed of 32 synchronization barriers. The ID of the thread which unlocks each barrier is stored until all barriers have been unlocked. At that point, in *replay* mode, we expect the list of thread IDs to be persistent from a simulation to another guaranteeing practically, if not yet formally, the repeatability from run to run of the approach, required by the SystemC standard.

B. Results

Fig. 3a illustrates the impact of quantum size on simulation speed. As expected, increasing the quantum size results in a significant speedup reaching up to 2000 MIPS with Matmul. However, when the quantum gets too large, speed decreases for Deriche and MobileNet. This is due to the much higher number of synchronizations in a single quantum. Each synchronization leads to process sequentializations as they rely on shared variables. When the quantum increases, the amount of time sequentially simulated increases to a point where it is no longer compensated by the speedup in the parallel phase. For the rest of the evaluations, we use a quantum of 30,000 instructions as a performance compromise between the three benchmarks.

To evaluate the influence of memory accesses instrumentation and processes sequentialization four versions of the kernel are compared on Fig. 3b. The overhead of instrumentation and sequentialization compared to fully parallel simulation ranges from 12 to 40%. Significant part of this speed reduction is due to sequentialization as the overhead of instrumentation alone is under 9%. Sequentialization overhead is non-compressible as it results from strict co-routine semantics enforcement. Also, the increase in speed compared to [20] is significant ranging from $\times 25$ to $\times 50$. It is mostly due to the much faster instrumentation techniques together with the asynchronous conflict checking.

Fig. 3c illustrates how our simulation kernel scales with the number of host cores used to simulate a 32-core platform and Fig. 3d shows the impact of simulated platform complexity (number of simulated cores) on speed when always using one host core per simulated core. Overall, speedups using 32 host cores (plus the kernel core) range between $\times 16$ and $\times 23$ compared to using a single host core. Also, while using the Accellera kernel is faster to simulate a single core platform due to a simpler scheduler and the lack of instrumentation, the speedup is already significant on a dual-core platform simulated in parallel as shown on Fig. 3d. It reaches up to $\times 19$ on a 32-core simulated platform running Matmul.

In all benchmarks, all conflicts were avoided and rollback was unnecessary. In general, conflicts are systematically avoided if processes always synchronize using a shared variable before using data produced by another process, as in our benchmarks. Indeed, all processes get correctly ordered relatively to this single variable before accessing the new data.

Finally, our simulation kernel exposed the expected behaviour on the validation application. Without *replay* enabled, the process ID sequence varies randomly from one execution to another. However, when *replay* is enabled, the sequence remains the same across successive executions.

VI. Conclusion

This paper introduced a new technique to parallelize loosely-timed SystemC TLM models in a standard-compliant fashion using memory access monitoring, dynamic detection of shared addresses and error recovery through process level rollback. It improves on previous

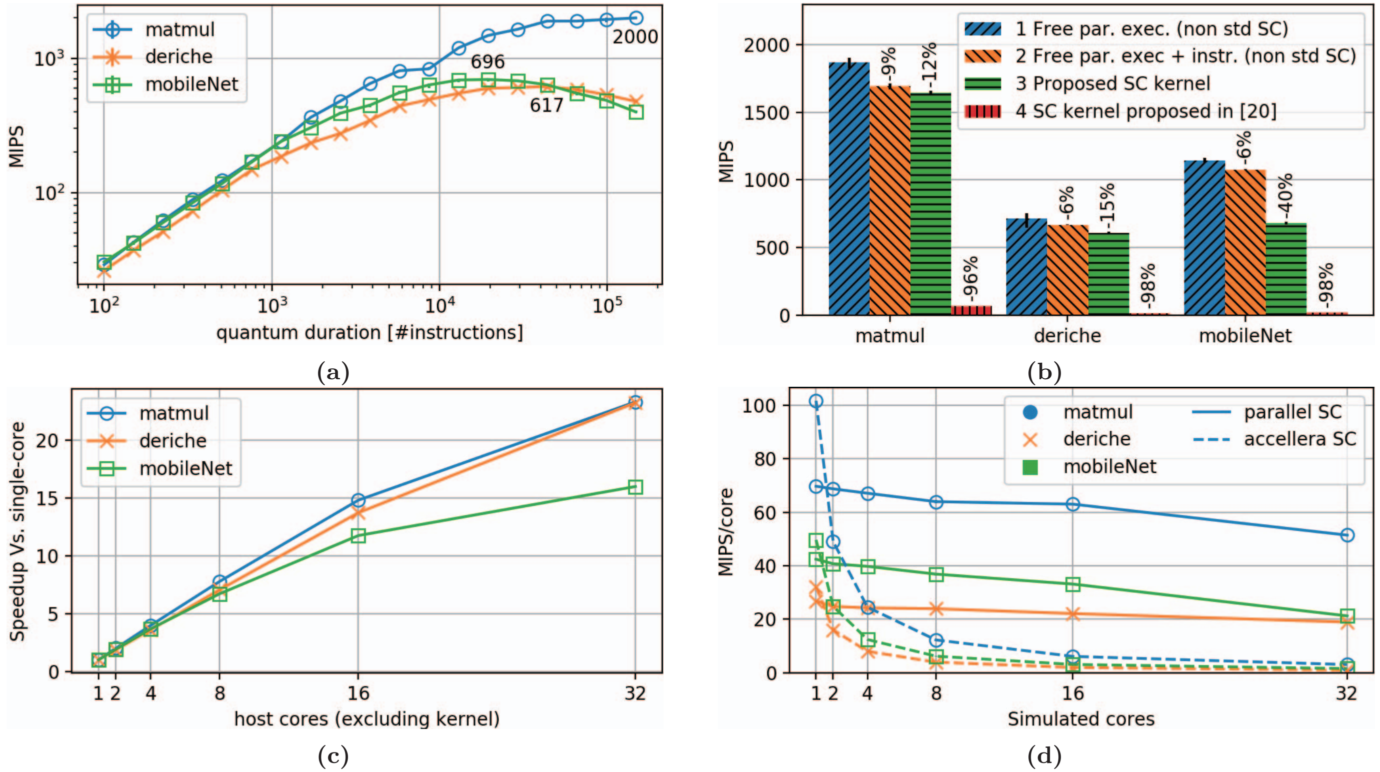

Fig. 3: (a) Simulation speed analysis depending on the simulation quantum size. The highest point of each curve is annotated. (b) Impact of instrumentation and processes descheduling compared to free parallel execution. Version 1 consists in a parallel simulation without enforcing processes atomicity. Version 2 shows the overhead of instrumentation and conflicts detection alone without process sequentialization. Version 3 implements all the contributions of this paper and is standard compliant. Version 4 is the kernel from [20]. (c) Scaling of the simulation speed with the number of used host cores. (d) Simulation speed per simulated core with parallel (1 host core per simulated core) and sequential simulation (Accellera kernel).

methods by avoiding manual declaration of shared addresses making it practical for any shared-memory applications. Also, the small overhead induced by our solution accommodates the fastest ISS's and reaches 2000 MIPS on 33 host cores when simulating a 32-core platform. Our solution scales well with the number of available host cores as it shows speedups up to ×25 between using 1 and 32 host cores (plus the kernel core). Finally, it demonstrates to be a clear improvement over the Accellera simulation kernel with up to ×19 speedup. We now plan to work on interrupts instrumentation to support simulation of models running OS like Linux for instance.

References

[1] *IEEE Standard for Standard SystemC® Language Reference Manual.* 2012.

[2] J. Aynsley, "OSCI TLM-2.0 language reference manual," in *OSCI, Tech. Rep*, 2009.

[3] D. Becker, M. Moy, and J. Cornet, "Challenges for the parallelization of loosely timed SystemC programs," in *RSP*, 2016.

[4] R. Dömer, "Seven Obstacles in the Way of Standard-Compliant Parallel SystemC Simulation," in *IEEE Embedded Systems Letters*, 2016.

[5] https://www.accellera.org.

[6] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *USENIX*, 2005.

[7] Z. Hao, Q. Lei, L. Hongliang, X. Xianghui, and Z. Kun, "A parallel SystemC environment: ArchSC," in *ICPADS*, 2009.

[8] P. Ezudheen, P. Chandran, J. Chandra, B. P. Simon, and D. Ravi, "Parallelizing systemC kernel for fast hardware simulation on SMP machines," in *PADS*, 2009.

[9] C. Schumacher, R. Leupers, D. Petras, and A. Hoffmann, "parSC: Synchronous parallel SystemC simulation on multi-core host architectures," in *CODES+ISSS*, 2010.

[10] S. Vinco, D. Chatterjee, V. Bertacco, and F. Fummi, "SAGA: SystemC Acceleration on GPU Architectures," in *ASP-DAC*, 2012.

[11] N. Ventroux, J. Peeters, T. Sassolas, and J. C. Hoe, "Highly-parallel special-purpose multicore architecture for SystemC/TLM simulations," in *SAMOS*, 2014.

[12] M. Moy, "Parallel programming with SystemC for loosely timed models: A non-intrusive approach," in *DATE*, 2013.

[13] R. M. Fujimoto, "Parallel discrete event simulation," in *Communications of the ACM*, 1990.

[14] P. Combes, E. Caron, F. Desprez, B. Chopard, and J. Zory, "Relaxing synchronization in a parallel SystemC kernel," in *ISPA*, 2008.

[15] A. Mello, I. Maia, A. Greiner, F. Pecheux, I. M. aind A. Greiner, and F. Pecheux, "Parallel Simulation of SystemC TLM 2.0 Compliant MPSoC on SMP Workstations," in *DATE*, 2010.

[16] J. H. Weinstock, R. Leupers, G. Ascheid, D. Petras, and A. Hoffmann, "SystemC-Link : Parallel SystemC Simulation using Time-Decoupled Segments," in *DATE*, 2016.

[17] Weiwei Chen, Xu Han, and R. Dömer, "Out-of-order parallel simulation for ESL design," in *DATE*, 2012.

[18] T. Schmidt, Z. Cheng, and R. Dömer, "Port call path sensitive conflict analysis for instance-aware parallel SystemC simulation," in *DATE*, 2018.

[19] J. Virtanen, P. Sjövall, M. Viitanen, T. D. Hämäläinen, and J. Vanne, "Distributed systemc simulation on manycore servers," in *NORCAS*, 2016.

[20] N. Ventroux and T. Sassolas, "A new parallel SystemC kernel leveraging manycore architectures," in *DATE*, 2016.

[21] https://criu.org/Main_Page.

[22] M. Jung, F. Schnicke, M. Damm, T. Kuhn, and N. Wehn, "Speculative Temporal Decoupling Using fork()," in *DATE*, 2019.

[23] R. Deriche, "Using Canny's criteria to derive a recursively implemented optimal edge detector," *International Journal of Computer Vision*, 1987.

[24] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, *et al.*, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," in *arXiv*, 2017.