Standard-compliant parallel SystemC simulation of loosely-timed transaction level models: From baremetal to Linux-based applications support

Gabriel Busnot, Tanguy Sassolas, Nicolas Ventroux, Matthieu Moy

Standard-compliant Parallel SystemC simulation of Loosely-Timed Transaction Level Models: From Baremetal to Linux-Based Applications Support

Gabriel Busnot[a], Tanguy Sassolas[a], Nicolas Ventroux[a], Matthieu Moy[b]

[a] *CEA, LIST, 91191 Gif-sur-Yvette CEDEX, France*
[b] *Univ Lyon, EnsL, UCBL, CNRS, Inria, F-69342, LYON Cedex 07, France*

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/TLM2.0 must get faster while maintaining accuracy. However, the ASI SystemC reference implementation remains sequential and cannot leverage the multiple cores of modern workstations. In this paper, we present SCale 2.0, a new implementation of a parallel and standard-compliant SystemC kernel, reaching unprecedented simulation speeds. By coupling a parallel SystemC kernel with shared resources access monitoring and process-level rollback, we can preserve SystemC atomic thread evaluation while leveraging the available host cores. We also generate process interaction traces that can be used to replay any simulation deterministically for debug purpose. Evaluation on baremetal applications shows ×15 speedup compared to the ASI SystemC kernel using 33 host cores reaching speeds above 2300 Million simulated Instructions Per Second (MIPS). Challenges related to parallel simulation of full software stack with modern operating systems are also addressed with speedup reaching ×13 during recording run and ×24 during the replay run.

Keywords: SystemC, TLM 2.0, parallel simulation, rollback

1. Introduction

Electronic System Level (ESL) design and verification is increasingly challenging due to the soaring complexity of SoCs and time to market constraints. Systems like manycore or complex Intellectual Properties (IPs) requiring sophisticated drivers increases the development time of software to the point where it has now become predominant. Just as programming or hardware description languages have evolved to help the user design more complex products, hardware (HW)/software (SW) integration tools involved in the SoC design process must support this trend. Among these tools, modelling 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. VP then allow for early software development, HW/SW co-design, system-level modelling 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 specifically interested in the Transaction-Level modelling (TLM) version 2.0 [2] for SystemC which enables higher level of abstractions for faster simulation, increased interoperability and model reuse. However, as specified in the IEEE SystemC standard, concurrency is emulated using the co-routine semantics, implemented by the reference Accellera kernel with cooperative sequential processes evaluation. It guarantees deterministic execution and protects against race conditions but also enforces single-threaded evaluation. As multicore SoCs are getting ubiquitous, the simulation speed decreases in inverse proportion to the number of processing units in the model. To tackle this issue, it is necessary to take advantage of multicore host platforms, but a parallel implementation of SystemC is needed.

We propose SCale 2.0, a standard-compliant parallel SystemC kernel which guarantees process evaluation atomicity and simulation reproducibility. We support any TLM model including loosely-timed coding style with the use of the Direct Memory Interface (DMI) protocol. Our technique based on lightweight process access monitoring has a limited overhead even when used with the fastest Instruction Set Simulators (ISS's) available. The contributions of this work are:

- A Finite State Machine (FSM)-based shared-resource access granting policy that prevents most atomicity violations;
- A fast process atomicity violation detection procedure

Email addresses: gabriel.busnot@cea.fr (Gabriel Busnot), tanguy.sassolas@cea.fr (Tanguy Sassolas), nicolas.ventroux@cea.fr (Nicolas Ventroux), matthieu.moy@univ-lyon1.fr (Matthieu Moy)

coupled with a roll-back error recovery mechanism;

- A highly scalable data-structure for FSM storage;
- A variable accuracy system coupled with adaptive parallel/sequential evaluation for simulation fast forwarding;
- CPU mode based unscheduling for extra conflict avoidance.

The rest of the paper is organized as follows. Section 2 explains the challenges exposed by SystemC parallelization. Section 3 presents the related work. Section 4 details the fundamentals of the contributions of this paper. Section 5 describes the experimental setup and analyzes our experimentation results on baremetal and Linux-based applications. Section 6 further analysis challenges brought by the simulation of complex guest OS and introduces some solutions. Finally, section 7 concludes the papers and exposes our perspectives.

2. Challenges of Parallel SystemC/TLM

2.1. General Introduction to SystemC

SystemC is a Hardware Description Language (HDL) originally designed as an alternative to VHDL or Verilog for instance. It is designed as a C++ library providing constructs to model components, signals, clocks, events or processes, just as most HDL. A SystemC model can then be simulated using Discrete Event Simulation (DES). In a broad outline, a DES relies on the alternation 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 and notification* phase — denoted as *kernel phase* in this work — where the kernel propagates the results of the previous evaluation phase in the model and determines the events to be triggered next in order to start a new evaluation phase.

The SystemC standard [1] states that processes must be evaluated with respect to the co-routine semantics. This is a strong incitation toward sequential evaluation using cooperative multithreading. This approach de facto limits to the use of a single Operating System (OS) thread and thus a single host core. As most Discrete Event Simulator (DES), the reference implementation of the SystemC simulation kernel (also referred to as the ASI kernel [3]), has chosen this approach, making it unable to take advantage of the numerous cores available on modern platforms.

The TLM2.0 library for SystemC has been incorporated to the standard to portable module interfaces at higher abstraction level. TLM was broadly adopted during the last decade. TLM designs differ from Register Transfer Level (RTL) designs in that data transfers modelling is done with Interface Method Calls (IMCs) instead of sc_signals, sc_channels and the event machinery. An IMC only incurs the same time penalty as a virtual function call. As a result, much faster simulation can be achieved by reducing the number of kernel phases and thus of context switches. TLM 2.0 enables two coding styles: TLM-AT and TLM-LT. The former allows to split transactions into several steps to achieve timing accuracy close to RTL models. However, the later, TLM-LT, is most commonly used by software developpers for its much higher speed. Indeed, TLM-LT enables DMI and allows to reduce the number of process synchronization to the bare minimum thanks to blocking transactions together with *temporal decoupling*.

Indeed, DMI grants an initiator the ability to replace an IMC by a direct access to the underlying memory buffer of a simulated memory component using a raw C++ pointer provided by the component itself through the DMI interface.

Temporal decoupling is a standard speed optimization which comes at the cost of less accurate timing. It allows a process to run *ahead of time*, that is to accumulate in a local counter the elapsed simulation time until it reaches a predefined limit called the *quantum*, at which point it must synchronize with the other processes. Time decoupling typically cuts the number of SystemC process context switches hundreds of times.

All these modelling techniques, while providing significant speedups in sequential simulation make parallelization much harder.

2.2. Parallelizing SystemC

The evaluation phase is usually the most compute intensive, which makes it the target of most parallelization approaches (including this work). But SystemC parallelization presents several challenges described in [4, 5]. The first of them is the co-routine semantics of SystemC. It requires that all processes scheduled during a given evaluation cycle be evaluated atomically. The SystemC standard however allows parallel evaluation as long as this co-routine semantics is preserved but it does not give hints about how to achieve it. Most solutions presented in section 3 try to preserve the co-routine semantics but at the cost of various restrictive assumptions. For instance, the communication mediums between processes or the event delivery policy are constrained in order to provide guaranties that can be exploited by the parallel simulator.

Another threat to parallelization especially present in TLM models is the lack of opportunities to run processes in parallel. Indeed, at first sight, only processes scheduled during the same evaluation cycle are good candidates for parallel evaluation. In [5], the authors show that most of the time, less than two processes can be scheduled simultaneously in a classic TLM model. This is mostly due to the fact that, in TLM models, processes are not synchronized by a central clock like it is often the case in RTL models, reducing the probability of several processes being scheduled at the same time.

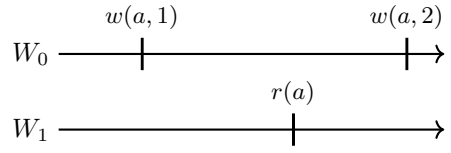

Figure 1: An example of process atomicity violation caused by concurrent access to a same memory location: a is initially 0, P_1 reads a as 1, which would be an impossible value in sequential evaluation.

TLM is also very challenging for parallel simulation as it replaces the channel-based communication between modules with interface methods calls, that is classic C++ function calls. While the various modules' states and processes tend to remain mostly isolated during the evaluation phase when using channels to communicate, TLM causes much more state sharing between processes throughout shared TLM slave accesses, threatening processes atomicity. When dealing with classic TLM models, even thread-safety is usually not guaranteed as these models are designed with a sequential mindset. The fact that all C++ constructs are allowed in SystemC makes it even harder to ensure even only thread safety, let alone processes atomicity. Protecting shared resources with critical section helps with thread-safety but has no effect on atomicity and severely hampers performance. In addition, all processes that use a same DMI pointer are susceptible to cause data-races if evaluated in parallel. To make matters worse, TLM-2.0 enables temporal decoupling, allowing processes to run for much longer periods of time, multiplying the risk of process atomicity violations like on fig. 1.

Finally, modern ISS's such as QEMU [6] have achieved great speedups in the recent years, reaching speeds above 1000 MIPS on a single-core host machine [7]. Also, temporal decoupling is mandatory in order to exploit the speed of such ISS's. Hence, the solutions used to allow parallel simulation must incur a very small overhead not to hamper the speed of a modern ISS.

Having highlighted the major challenges of SystemC parallelization, section 3 presents the existing work related to parallel SystemC simulation and shows that many of these issues remain unsolved.

3. Related Work

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 efficiently simulated in compliance with the SystemC semantics and, by extension, to the type of communications used in these models. Parallel SystemC solutions differ by many aspects: multi-thread, multi-process, distributed, hardware accelerated, static code-base analysis or even introduction of new semantics on top of SystemC. However, the most fundamental difference between two solutions lies in the way it manipulates the simulation time: synchronous or decoupled. Synchronous simulation (section 3.1) is well suited to RTL models as they tend to be synchronized with a central clock, thus offering a lot of parallelism in each evaluation phase. However, with TLM, time decoupling (section 3.2) is better suited in order to allow processes scheduled at close-enough time points to run in parallel. Similarly, temporal decoupling can also be exploited to restore process simultaneity but only [8] presented in section 3.3 deals with some TLM-specific issues like the tight coupling of processes.

3.1. Synchronous SystemC Parallelization

In the early days of SystemC parallelization, mostly cycle-accurate model simulation (e.g. RTL) was explored. A conservative approach is presented in [9]. The design to be simulated must be split into several subsystems chosen to be as independent as possible. These subsystems will be simulated in parallel. Processes communicate using regular channels, but a central process is in charge of computing the next simulation time after each evaluation phase. Shared variables and remote timed event notifications are not supported.

In [10], parSC, a centralized parallel SystemC scheduler uses multiple workers to run the evaluation phase. SystemC processes are mapped to these workers and the evaluation phase is bounded with barriers in order to synchronize the workers. The rest of the kernel logic remains sequential, including requests to the kernel that are buffered by each worker and processed sequentially after each evaluation phase. It is however the responsibility of the user to protect all resources shared between concurrently running processes. It is likely that in case of resource sharing, process atomicity is compromised together with simulation determinism.

LegaSCi [11], the sequel to parSC with better support for TLM, addresses this issue with the introduction of containment zones. All resources (data and processes) belong to a given zone. Only processes of a given zone can address resources of the same zone and processes of a zone run sequentially, thus preventing race conditions from happening. In case a process crosses a zone boundary through an IMC, it is migrated to the accessed zone and blocked until all processes of this zone are done. However, the user must ensure that a migrated process does not access data from its original zone after migration unless no other process of its original zone accesses this data again. Otherwise, data races could occur, compromising the whole simulation. Also, data sharing is only allowed after going through an IMC, unlike when using a DMI pointer to a shared memory for instance.

With raw simulation speed and ease of use in mind, [12] proposes a multiprocess and multi-kernel simulation engine designed around IPTLM, an inter-process adaptation of the TLM protocol. Interprocess communications are then strictly restricted to messages going through IPTLM sockets implemented using POSIX shared memories. In particular, shared variables and events are strongly discouraged. Determinism is not guaranteed anymore and

there are no references to time synchronization between processes. Also, interprocess communications must be avoided as much as possible to keep performance high.

Hardware acceleration has also been explored. For instance, [13] proposes a multicore shared memory SoC coupled with a SystemC kernel accelerator. While SystemC processes are evaluated in parallel by the numerous available cores, the update and notification phases are hardware accelerated. The update phase relies on a parity register telling which value is to be read or written on each channel. The notification phase uses a parallel search to find the processes sensitive to a given event. Communication between processes is obviously limited to signals as race and processes atomicity violations are not considered. Hence, this is a very RTL-oriented solution.

GP-GPU based acceleration has also been explored like with SAGA [14]. An RTL model is statically analyzed in order to construct the process dependency graph based on the model signals. Independent dataflows between processes are then extracted after possibly duplicating some processes to reduce dataflow coupling. Each dataflow is then executed on a different GPU warp in a sequential order respecting the dependency between processes. Warps synchronize after all processes of each dataflow have been evaluated, that is at least before every timed notification. A CPU/GP-GPU hybrid solution is detailed in [15]. It allows efficient simulation of mixed-abstraction models that include RTL and TLM modules. A toolchain enables automatic mapping of process on GPU or CPU depending on their nature in order to perform parallel process evaluation. While a GP-GPU can support great amounts of parallelism, this solution will only be efficient on sufficiently homogeneous designs. Also, not all C++ is compatible with CUDA kernels, thus limiting the expressivity of SystemC when simulated on a GP-GPU. In particular, TLM processes are unlikely to support GP-GPU evaluation. In case of a hybrid solution, CPU-GPU synchronization might introduce high latencies.

3.2. Time Decoupling

At the root of all the approaches presented in this section is the Parallel Discrete Event Simulation (PDES) [16]. This technique essentially allows to run processes scheduled at different times while maintaining timing consistency. However, the amount of decoupling varies from one implementation to another. PDES can be either conservative or optimistic. The former guaranties that the local time of each part of the design will never be greater than any message it receives (i.e. timing violations never occur). The later takes the risk to sometimes cause timing violations but provides a rollback mechanism to recover from such error. Optimistic PDES is considered too hard to apply to SystemC due to the complex state of a simulation preventing a general approach to rollback and thus is never used.

A distributed SystemC simulation framework based on ArchSim [17], a distributed simulation platform for system level high performance computer design, is presented in [18]. It mainly requires that processes communicate only using channels and that the design can be partitioned into relatively independent subsystems. These subsystems are then mapped to several host computers using the ArchSim [17] parallel simulation framework. Each subsystem has its own SystemC scheduler. Communications between subsystems are achieved using ArchSim channels that wrap and multiplex the behavior of conventional SystemC channels in a distributed context. Time synchronization between processes is achieved by waiting on all remote input channels of each node to determine the next earliest timestamped message. According to this and to its own internal events, each subsystem can compute its next simulation time. However, remote timed event notification is not supported. Distributed simulation can scale up to hundreds of nodes but synchronization between these nodes relies on networking whose latency is order of magnitudes higher than shared memory synchronization. Also, the amount of parallelism will often be limited by the number of relatively independent parts in the simulated system. With the advent of Uniform Memory Access (UMA) chips reaching up to 64 cores to this day, distributed simulation might become less attractive.

While the previous approach is more oriented toward RTL simulations, [19] tackles TLM simulations with TLM-DT. This solution is explicitly targeted at shared memory SoC simulation. Thus, three types of components are defined (initiator, interconnect and target) and three types of communications (request from an initiator to a target going through an interconnect, the associated response or an interrupt from a target to an initiator). Here, each component of the design has its own local time and there is no more global simulation time. Initiators are free to run until they send a request to a target or they reach the lookahead time set by the user before the simulation. Interconnects wait for a packet to be present on all their inputs in order to choose the most recent one. It can then compute its new local time before forwarding the earliest request to the correct target. The target updates its local time in turn using the transaction timestamp before sending back the answer. The transaction delay is added to the request timestamp at each processing step so that the initiator can update its own local time at the end of the transaction. However, interrupts cannot be handled the same way as it would cause deadlock (e.g. an interrupt from a target to an initiator waiting for this same target). Thus, interrupt requests are polled by initiators and handled as soon as their local time is greater than the interrupt timestamp, causing small timing errors and non-determinism. The timing accuracy is comparable to a TLM-AT model. SystemC-SMP, a parallel simulator dedicated to these types of models is proposed in [20]. Processes are grouped in order to favor internal communications and then mapped to different CPUs, each running its own local scheduler.

The lookahead time is exploited differently in [21].

While it was only a limit to the amount of time a process can run without synchronizing in [19], the lookahead time t_{la} also defines the minimum amount of time that a remote event notification must be triggered in advance. For instance, if the local time of a process is t_p, then it must not notify events to process in other time zones before $t_p + t_{la}$ because it guaranties that the remote process will not see the event in the past. This is a direct application of conservative PDES. However, this constraint can be hard to honor in a real-world model, so the authors introduced flexible time decoupling. It consists in adjusting all remote event notification delays according to the chosen policy. The accurate policy forbids any adjustment and raise errors if the lookahead is not respected. The deterministic mode increases the notification delay just as much as required for the lookahead to be respected. Finally, the fast mode increases the delay to be just ahead of the targeted thread time, which sacrifices determinism. This solution can take advantage of temporal decoupling as the local time advance guarantees that events will be notified ahead of time of remote processes. However, remote transactions remain blocking, reducing the benefits of temporal decoupling.

SystemC-Link [22] brings an additional refinement defining delays for channels linking two time zones. This delay becomes is a kind of local lookahead time that is applied only to processes in time zones connected by the channel. Two scheduling policies are also provided: as-soon-as-possible and as-late-as-possible. The former makes each process yield whenever it wants to advance its local time while the later lets processes run until they reach the maximum lookahead allowed by the delays of the neighbor channels.

Imposing no constraint on the simulated design, [23] proposed an approach based on compiler-driven static analysis. A standard model can be analyzed by a SystemC semantics aware compiler in order to detect the dependencies between code *segments* (i.e. the code between two scheduling points). Based on this analysis, segments can run in parallel if they do not have dependencies like accessing a same variable. But also, if the compiler can prove that a given segment will not receive any event before its current next scheduling time, this segment can be issued in advance. This is called out-of-order parallel evaluation. Load balancing based on the compiler-estimated run time of each segment is added in [24]. A major limitation of this approach, however, is its lack of support for programmatically constructed platforms (e.g. CPUs instantiated in a for loop). This is addressed in [25] where the previously compile-time information can now be completed at run-time after platform elaboration. Also, closed source libraries can be manually annotated to be handled appropriately by the scheduling algorithm. In order to reduce false positives, the modules interconnections are considered in what is called port-call-path-sensitive analysis [26]: two segments of modules that are not connected are guaranteed not to have dependencies. Finally, [27] adds event delivery prediction so that processes can be scheduled even before any of their sensitive events is triggered. The major limitation of this approach, however, is the drastic pessimization caused by pointer dereferencing: two segments that dereference a pointer are systematically conflicting if the content of the pointer is not statically known. As a result, a Symmetric Multiprocessing (SMP) TLM-LT model will always run sequentially because of the dynamically defined address of transactions targeting the shared memory component.

Some sort of time decoupling is also provided by the `sc_during` semantics defined in [28]. While in classic DES, a task always run instantaneously before catching up by waiting for the amount of time it would have taken on the real system, `sc_during` allows to start a task with a duration associated. The explicit use of a duration helps determining which tasks are independent so that they can run in parallel: two tasks whose durations overlap are independent as they do not need the result from the other one to start. Starting a task with duration is as simple as spawning an OS thread to run the task, call `wait(<task_duration>)`, which does not suspend the task itself, and then join the task when the `wait` returns. Additional functionalities are provided in order to control `sc_during` tasks and interact with the simulation kernel from such task. Tasks with duration is implemented as an independent library and thus can be used with any standard compliant SystemC simulation kernel. This approach introduces parallelism in a very simple way. However, this is the responsibility of the user to guarantee that tasks are running in isolation, otherwise race conditions and non-determinism could occur.

3.3. SCale: Optimistic Parallel SystemC Simulation

As they require frequent enough synchronization most of the aforementioned standard-compliant approaches target at most TLM-AT models which are rather slow at a few MIPS or TLM-DT models offering sufficient architectural decoupling. General TLM-LT models can reach hundreds of MIPS thanks to temporal decoupling but present several additional obstacles to parallelization as they tend to make extensive use of shared host resources. It is especially true when considering the DMI interface which bypasses transactions all together and is not efficiently supported by any of the previously mentioned approaches in the case of an SMP model with a single shared-memory for instance. While interoperability relies on users implementing the standard interfaces, it is not uncommon to see model vendors rely on custom interfaces for the internals of their components. For instance, the interrupt lines of a CPU model could be exposed directly to an interrupt controller via a regular C++ method. This avoids the cost of an extra process responsible only for raising interrupts using a signal raised at times potentially not aligned on the quantum. Also, as the time between two synchronizations increases, so does the risk of atomicity violations. As a result, quantum-based temporal decoupling is an additional obstacle to the use of all previously described approaches.

SCale, the parallel SystemC kernel described in [8] relies on the same general principle as SCale 2.0, the work presented in this article: memory accesses monitoring. SCale 2.0 has already been introduced in [29] but new features related to Linux-based benchmark simulation (section 6) and a lot more insights (section 4) are given in this article.

In SCale, SystemC processes are grouped by the user in order to be assigned to workers evaluated in parallel by different pthreads. Each worker is responsible for evaluating sequentially its processes scheduled in the current evaluation phase (also referred as *quantum*). SCale then uses memory access monitoring as its central mechanism to prevent and detect process atomicity violations and generate a simulation trace used when replaying the simulation (e.g. for debugging). When a process tries to access some shared memory, its worker is unscheduled and resumed during the sequential evaluation phase that follows immediately the parallel one. While limited to evaluation phase level parallelism (as opposed to time-decoupled PDES approaches), temporal decoupling usually allows for most processes (e.g. ISS's) to be scheduled at synchronized time points, maintaining a very high level of parallelism in most evaluation phase.

Monitoring memory accesses also allows for building a worker dependency graph for each evaluation phase. If workers have been evaluated atomically, then processes also have been evaluated atomically. A dependency between workers W_a and W_b written as $W_a \rightarrow W_b$ is created when a pair of memory accesses implies that an equivalent sequential schedule of the current evaluation phase must evaluate W_a before W_b in order to yield the same result. Specifically, a dependency exists when one of the following pair of memory accesses involves two different workers:

- Read After Write (RAW): If W_b reads a value after W_a wrote it during a parallel evaluation, a sequential schedule where W_b comes before W_a would make W_b read a different value.
- Write After Read (WAR): The reason is similar to the RAW case.
- Write After Write (WAW): If W_b overwrites a value previously written by W_a, the final value differs from a sequential schedule where W_b comes before W_a.

From that point, we can check whether workers have been evaluated atomically by checking if there exists a sequential schedule that yields to the same state at the end of the evaluation phase, that is if and only if the dependency graph defines a partial order on the workers, that is if and only if it is acyclic. This assertion is checked after every evaluation phase using the Tarjan's strongly connected components algorithm (any cycle detection algorithm would do). If the graph is acyclic, then it can be used to define an equivalent sequential schedule of workers for simulation replay purpose (only the workers involved in dependencies need to be scheduled sequentially during the simulation replay). If there is a circular dependency, however, this is called a *conflict*.

SCale provides an annotation function that must be called before every memory access. The internal logic of this function has been greatly modified between SCale and SCale 2.0 and is detailed in section 4. What remains is that this function returns immediately if the calling worker can perform the instrumented access during the parallel phase or else the worker is paused until the sequential phase (i.e. it is *unscheduled*). However, SCale relies on user-provided annotations to tag some address ranges as shared and force sequential evaluation of workers that may otherwise violate the co-routine semantics by accessing shared memory concurrently with another worker. It is often very hard to predict which memory regions are going to be shared during the simulation, especially in the presence of dynamic memory allocation and memory virtualization which both prevent static code analysis. Also, shared memory regions often move during a simulation. Declaring all of them as shared from the beginning to the end of a simulation is very suboptimal. This makes SCale unfit for the simulation of architecture using modern guest OSs such as Linux.

Memory accesses must be recorded in the same order as they are performed in the simulation for the conflict checking to be correct. This constraint is not addressed in [8], leaving it to the user to implement an atomicity mechanism. 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 high worker counts.

The work presented in the present article is inspired from SCale [8] but tackles its main functional limitations while providing significant speed and scaling improvements. We no longer 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. Finally, the approach has been generalized to all types of workers interactions instead of only shared memory related ones. All these improvements drastically improve speed of baremetal applications and show promising results on some Linux-based applications while simplifying the instrumentation of shared resources accesses.

4. Proposed Parallel SystemC Kernel

4.1. Overview

For the sake of simplicity, it is assumed in this section that the only shared resource of the simulation is the model memory. We explain in section 4.4 how the presented system is easily generalized to any form of shared resources like peripherals, interrupt lines, etc. Assuming only memory is shared during the simulation, we also assume that the user has correctly instrumented all memory accesses through a call to the provided function `mem_instr` before each memory access in the platform model. Note that since

most shared memory accesses are issued by the ISS (e.g., QEMU), the instrumentation can be done once and for all in the ISS. In our case, it results in a single line of code for the whole model. The software code is run unmodified.

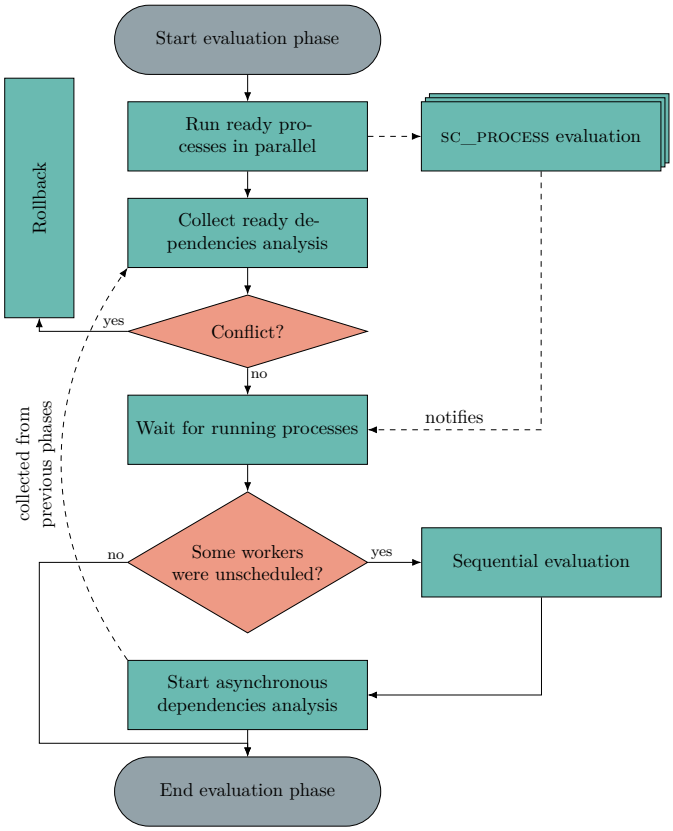

Figure 2: Evaluation phase flow chart of our parallel SystemC kernel.

The evaluation phase flow chart of our parallel SystemC simulation kernel is summarized fig. 2. We start each evaluation phase with a *parallel phase* (section 4.2) where all the ready processes are evaluated in parallel by worker threads. During this parallel phase, worker accessing memory addresses detected as shared are unscheduled by the `mem_instr` function until the *sequential phase* (section 4.3) where they are resumed in a sequential order in order to prevent race conditions (section 4.3.1). Workers are unscheduled before performing the suspicious memory access, so that potentially problematic accesses are integrally performed in the sequential phase.

`mem_instr` takes three arguments: the accessed address, the number of bytes accessed and the type of access. This function is outlined in algorithm 1.

After each evaluation phase, it is checked that no conflict occurred, i.e., that no circular dependencies between workers exist (section 4.3.2). The conflict check is delegated to additional OS threads and the result collected when ready during a posterior evaluation phase. If no conflict occurred, the list of dependencies is recorded in order to generate the simulation trace that can be used for deterministic simulation replay (section 4.3.3). If a

Algorithm 1 `mem_instr` function outline

```
1:  procedure MEM_INSTR(addr, nBytes, isWrite)
2:      if not in sequential phase then
3:          w ← getCurrentWorkerId()
4:          a ← reducedAddr(addr)                    ▷ c.f. 4.2.2
5:          m ← FSMarray[a]                          ▷ c.f. 4.2.5
6:          go ← m.updateFSM(w, isWrite)             ▷ c.f. 4.2.2
7:          if not go then
8:              wait sequential phase                ▷ c.f. 4.3
9:          end if
10:     end if
11:     recordAccess(addr, nBytes, isWrite, w)       ▷ c.f. 4.3.2
12: end procedure
```

conflict did occur, we rely on host process level rollback (section 4.3.4) to reinstate the simulation in a valid anterior state and start over. The conflict can now be avoided thanks to a proper sequentialization defined by knowledge from previous run.

The instrumentation atomicity is guaranteed at no cost due to the *zero dependencies guaranty* (section 4.2.1) provided by the FSM-based memory access granting policy of SCale (section 4.2.3).

Finally, FSMs need to be periodically reset in order to reflect the most accurately possible the current sharing state of the associated address and not compromise the simulation speed with unnecessary unscheduling.

4.2. The Parallel Evaluation Phase

All evaluation phases start with a parallel phase where processes are evaluated in parallel by the worker they are attached to.

4.2.1. Advantages of Zero Dependencies Parallel Phase

During a simulation, most of the evaluation phases do not present any dependency. This is because shared variables tend to rarely be used in real-world applications in order to preserve execution speed in the real system. For that reason, we have made design choices that take advantage of this assessment. We enforce that no dependency occurs during the parallel evaluation phase, postponing all the complex logic to the sequential phase. In practice, because most evaluation phase cannot cause dependencies, this does not significantly limit performance.

Knowing that zero dependencies can occur during the parallel phase, several properties are obtained and detailed in the following sections:

1. "Instrumentation + memory access" needs not be atomic as long as instrumentation comes first.
2. Memory accesses during the parallel phase can be recorded in parallel as they never depend on each other, so a total order is not required.
3. If no workers have been unscheduled during the parallel phase, then no conflicts have occurred, and check is not required.

Hence, the "zero dependencies during parallel phase" property is a strict prerequisite to enable vital optimizations across the whole system. We provide it using an FSM-based granting policy detailed in the next section.

4.2.2. The Address FSM

Keeping in mind that zero dependencies must occur during the parallel phase, we must choose a memory access granting policy to determine which accesses can be performed during the parallel phase and which cannot.

The optimum strategy would consist in recording, for each address and for each evaluation phase, the list of accesses to check if the new accesses introduce a dependency (RAW, WAR or WAW). This would be, however, very expensive as it requires memory allocation and mutex synchronization at every access. Thus, we had to define a memory access policy with a much faster decision time.

In SCale [8], it was up to the user to define the shared and the read-only addresses in order to check if each access targets one of these addresses. However, in most applications, shared memory addresses are too complex or even impossible to statically enumerate mainly because of dynamic memory allocation and memory virtualization. Also, a memory region might be shared only during some parts of the simulated program. Declaring all regions *shared at some point of the simulation* as *shared for the entire simulation* would then result in numerous useless sequentializations and an important performance degradation.

To implement our memory access granting policy, addresses are first grouped into blocks of size S, defined by the user. A good size to choose is typically the largest number of bytes accessible with a single CPU instruction (e.g. 8 bytes on RISC-V 64 bits) as it includes the most common memory access. When calling `mem_instr` with address a as argument, the *reduced address* a' is computed first (algorithm 1 line 4) as $a' = a/S$. If the worker is not unscheduled, it can then access any address in the range $[a', a' + S)$. In case a memory access spreads among several S bytes aligned intervals because it is either unaligned or it accesses more than S bytes, a helper function `mem_instr_slow` that calls `mem_instr` several times in order to protect all the accessed memory must be used instead. We have chosen to have two functions so that memory accesses that require a single call to `mem_instr` are instrumented as fast as possible. Grouping addresses may lead to an approximation, but the approximation is conservative: we may unschedule a worker that could have been granted the access, but we never grant an unsafe access.

An FSM described fig. 3 is then associated to each address block using a data structure described in section 4.2.5. Each address block can independently be in one of these 4 states:

1. NO_ACCESS: After initialization or reset (c.f. section 4.2.4).

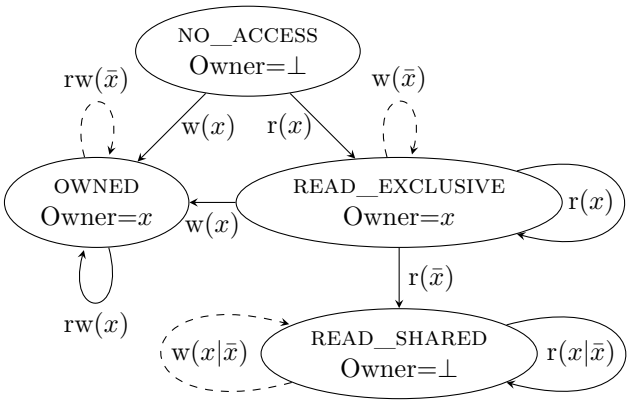

Figure 3: Memory access monitoring FSM. x is the worker identifier (WID) of the worker doing the access from NO_ACCESS; $\bar{x}$ designates any worker other than x; r and w stand for read and write. Workers are unscheduled on - - ➤ transitions.

2. OWNED: When an address has been accessed by *only one worker* and with *at least one write* since last reset. This worker is called the *owner* of the address.
3. READ_EXCLUSIVE: When an address has been *only read* by a *single worker* since last reset. This worker is also called the *owner* of the address.
4. READ_SHARED: When an address has been *only read* and by *at least two workers* since last reset.

The goal behind this FSM is to allow workers to access the memory they are not sharing freely (the OWNED state) and to allow read-only memory to be accessed concurrently (the READ_SHARED state) in order to unschedule as few workers as possible. However, these two states are not sufficient and the READ_EXCLUSIVE state is crucial to make the FSM efficient. Indeed, when the FSMs get reset, memory was being used by the various workers of the simulation. Let us assume that address a was owned by worker W (e.g. a is some stack memory in the simulated program). If the next access to a done by W is a read, it does not mean that a has become a read-only shared address and must go in the READ_SHARED state. It is very likely that a is still used by W only and should come back to the OWNED state. However, not all addresses that are read first are owned by the reader (e.g. read-only shared memory). The READ_EXCLUSIVE state allows to differ the decision until it is possible to choose between OWNED and READ_SHARED based on more than only reads from a single worker.

It can be noted that any access causing a dependency corresponds to a "- - ➤ " transition, causing the offending worker to be unscheduled. Thus, the zero dependencies guaranty is provided by this FSM.

The FSM state is composed of the following fields packed in 4 bytes to allow for fast Compare And Swap (CAS): the state id, the owner WID and a generation counter for fast reset (see section 4.2.4).

Upon memory access, algorithm 2 is used in algorithm 1

Algorithm 2 FSM update algorithm

```
1:  procedure UPDATEFSM(WID, accessType)
2:      S_old ← S                          ▷ S is the FSM 4-byte state
3:      S_new, go ← getNewS(S_old, WID, accessType)
4:      if S_new != S_old then
5:          S.CAS(expected=S_old, new=S_new)
6:          if CAS failed then
7:              return updateFSM(WID, accessType)
8:          end if
9:      end if
10:     return go
11: end procedure
```

at line 6 for fast and wait-free FSM update. The transition computation getNewS is defined by fig. 3 and consists in a 4-case switch statement with a couple of ternary expressions per case. getNewS returns the new state and a boolean which is true if and only if the transition is not a "- - ➛ ", that is if the access is granted.

It can be noted that the transition application requires an atomic CAS as two or more workers might attempt to update the FSM concurrently. However, doing the CAS is required only if changing the state (c.f. line 4). It is an optimization that happens to be correct for this specific FSM. It can be verified by assuming the CAS is always done. In the cases where it would have failed while being skipped by the line 4, the final state of the FSM and the value returned by UPDATEFSM are actually the same whether a second attempt is done on line 7 or not. Hence, the CAS is useless in those cases in the first place.

Keeping the state from one quantum to another makes this approach extremely fast but also causes unnecessary unscheduling. For instance, if the first access to an address in the READ_SHARED state is a write, it will lead to unscheduling despite causing no dependency. However, preserving the state across evaluation phases also help triggering the optimization on line 4 more often because addresses tend to be used in the same way for relatively long periods of time (c.f. temporal reference locality principle [30]). But just as an address is not used for a single purpose for the whole power-on time of a computer, keeping the same state for an address during the whole simulation is not efficient either. Choosing a good reset policy is a difficult question by itself and is discussed in section 4.2.4.

4.2.3. Correct access recording order

One crucial correctness condition is the memory accesses are recorded in the same order as they are performed. At least, they must be recorded in an order that exhibits the same dependencies as their real order.

Let us assume an alternate FSM that does not provide the zero-dependencies guaranty, applied in the example Figure 4. The instrumentation records the write of P_0 to address a before the read of P_1 while P_1 reads a before P_0 writes it. The dependency analysis would then consider

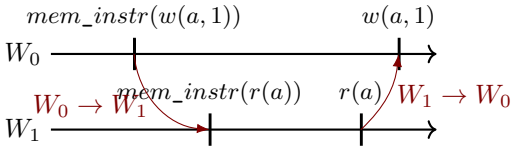

Figure 4: Example of bad memory access recording order. Axes represent the wall clock time.

$P_0 \rightarrow P_1$ because of a RAW on a while the real dependency is $P_1 \rightarrow P_0$ because of a WAR. Such error could lead to undetected conflicts which is not acceptable from a correctness standpoint. An obvious workaround would be to include the instrumentation and the access in a critical section prohibiting at least the other memory accesses to the same variable to happen simultaneously, but that would result in unacceptable performance degradation.

However, enforcing strictly identical order between accesses and their recording is not necessary as long as the correct dependencies are detected. In particular, if the recorded order of independent memory accesses differs from their real order, no dependencies are missed as theses accesses are independent by hypothesis. The zero dependencies guaranty provided by our FSM implies that no two dependent memory accesses can occur concurrently during the parallel phase. The scenario fig. 4 cannot happen as P_1 would be unscheduled before the read to a is both recorded and performed. In general, the order in which memory accesses are recorded during the parallel phase is not significant in our case. Indeed, any reordering of the recorded accesses would not produce any dependency as they are all independent.

Eventually, we do not guaranty strict memory access recording order. Instead, we provide a sufficient guaranty so that the instrumentation and the corresponding access need not be protected by any sort of additional synchronization for the recorded order to be correct. This has a major impact on performance and allows perfect algorithmic scaling with the number of workers running in parallel.

4.2.4. Efficient FSMs Reset

In the FSM depicted on fig. 3 no transitions leave the states OWNED and READ_SHARED: they are fixed points. Therefore, once reached, such state would last during the entire simulation. This accommodates programs whose memory accesses pattern is constant over its execution. For instance, if the simulated program only consists in multiplying two squared matrices of size n and storing the result in a third one: $C = A \times B$. Each one of the N threads is responsible for computing $\frac{n}{N}$ consecutive lines of C. A would stay in the READ_EXCLUSIVE state as each line is only read by a single worker. B would stay in the READ_SHARED state as all workers read the entire matrix. C would stay in the OWNED state as each line is written by a single worker. However, addresses are often used by a worker for a certain amount of time

and then by another like in the Deriche filter. In this algorithm, horizontal and vertical pass on an image are alternated. In a naive parallel implementation, the image is split horizontally for the horizontal pass and vertically for the vertical pass. Consequently, pixels are all in the OWNED state after the horizontal pass and need to change owner before the vertical pass in order not to unschedule most memory accesses. However, setting the owner of an address is only possible from the NO_ACCESS state.

Efficient simulation of the Deriche filter (and of most algorithms), hence requires to be able to reset the FSMs at well-chosen instants. We will discuss here two aspects of the problem:

1. When to reset the FSMs;
2. How to reset the FSMs efficiently.

Ideally, an FSM should be reset whenever its state does not reflect the real state of the associated address (e.g. an owner change in the Deriche filter). This is very hard to detect accurately but an incorrectly classified address will cause quasi systematic unscheduling, which is easy to measure. A first approach could be to reset all addresses that caused some unscheduling during the previous quantum. However, a worker can only be unscheduled once per quantum so no more addresses than there are workers can be reset in each quantum with this approach. This is inefficient as large amounts of addresses usually should change owner simultaneously (e.g. the Deriche filter again). Setting apart the addresses that need to be reset from those that do not is impossible as it depends on the future of the application.

As a result, we have chosen to reset all addresses every time a worker is unscheduled. That way, all addresses can transit to their most appropriate state from that point. Addresses that did not need to be reset will only incur at most a couple of additional CASs that are relatively inexpensive and performed in parallel by all workers. Resetting all addresses is a good strategy to avoid further workers unscheduling. This reset policy also relies on the observation that truly shared addresses such as mutexes are seldom used in parallel programs relatively to the computational workload due to their performance cost.

Other reset policies have been tested but gave no performance improvement at best:

- Reset every N quantum
- Reset every N unscheduling
- Reset after one worker is unscheduled N times

Note that the special case "reset every $N = 1$ quantum" with addresses grouped byte by byte makes our access granting policy optimal in the sense that it only unschedules all workers introducing actual dependencies and only those. However, systematic reset together with undersized address groups leads to important instrumentation overhead. This is mostly caused by the multiple calls to `mem_instr` required for instance for typical word-sized memory accesses together with many additional FSM non-fixed-point transitions.

Algorithm 3 FSM update algorithm with lazy reset

```
1:  procedure updateFSM(WID, accessType, gen)
2:      S_old ← S
3:      curGen ← S_old.gen
4:      if curGen ≠ gen then                          ▷ reset
5:          S_tmp.state ← NO_ACCESS
6:          S_tmp.owner ← ⊥
7:          S_tmp.gen ← gen
8:      else S_tmp ← S_old
9:      end if
10:     S_new, go ← getNewS(S_tmp, WID, accessType)
11:     if S_new != S_old then
12:         S.CAS(expected=S_old, new=S_new)
13:         if CAS failed then
14:             return updateFSM(WID, accessType, gen)
15:         end if
16:     end if
17:     return go
18: end procedure
```

Choosing to reset either none or all FSMs also enables $O(1)$ reset implementation using a *generation-based reset*. As mentioned in section 4.2.2, a generation counter is part of the state of the FSM. To virtually reset all FSM, a single counter called `fsm_gen` needs to be incremented. The value of `fsm_gen` is then passed to updateFSM as third argument in algorithm 2 which is updated as in algorithm 3 in order to perform *lazy reset*. That way, FSMs are only reset if they are accessed and directly by the very first worker accessing them. The zero dependencies guaranty is maintained as the reset of an FSM can only occur at the very first access of the quantum. Thus, no dependency can be introduced.

4.2.5. Ultra-Fast Scalable FSM Storage

On one hand, for memory accesses instrumentation to be fast and scalable with the number of workers, they must be stored in a container that supports concurrent accesses while requiring little to no synchronization. However, we do not require deletion or iteration. On the other hand, the memory map of the simulated platform can be shaped in arbitrary ways. It can span over huge memory ranges, be sparse or even runtime defined (e.g. the PCI-e protocol). The memory usage of our map must also remain contained as it affects rollback performance (c.f. section 4.3.4).

Let us first briefly discuss the most common containers and why they do not fulfill our requirements:

- A statically allocated contiguous array of FSMs would potentially guzzle huge amounts of memory in order to cover all the whole address map;
- Splitting it into a list of arrays in order to cover only the used memory ranges is still quite memory

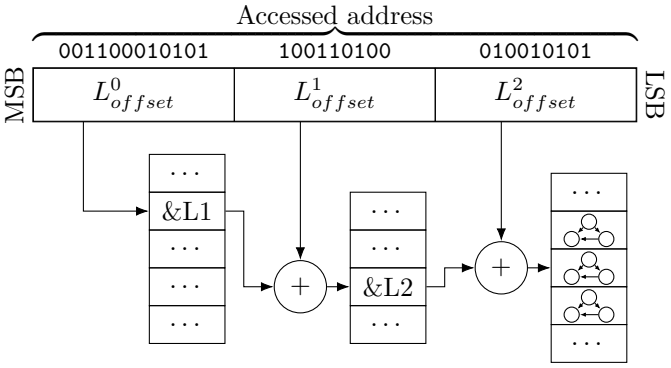

Figure 5: Multi-level FSM table. Only tables hit by the access are drawn.

intensive, requires additional target dependent parameterization from the user, is algorithmically inefficient if there are many holes and does not support dynamically defined memory maps;

- `std::map` and `std::unordered_map` both require external synchronization for concurrent accesses, which would compromise performances.

Instead, we designed a custom container inspired from multilevel page tables as illustrated fig. 5. Accessing an element is done by using the successive fields of the address to compute offsets in successive levels of nested tables. If the table is defined with N levels, the $N-1$ first levels (the *intermediate* levels) contains pointers to the beginning of the next level tables. The last level contains the element themselves (i.e. FSMs in our case). When an intermediate table is allocated, all pointers are set to `nullptr`. When a worker W reaches a `nullptr` on the path to an FSM, the next levels of the table need to be allocated. To that extent, W first allocates the next level of the table before attempting an atomic CAS on the null pointer in order to make it point to the freshly allocated next level of the table. If the CAS fails, it means that another worker has concurrently allocated the next level so W can free the allocated next level and proceed. When allocating the last level of the table, all FSMs must be initialized to the NO_ACCESS state. Non-null pointers in the intermediate levels of the table being constant, no synchronization is required for concurrent access once initialized.

This container supports constant time random access and concurrent wait-free reads and allocation. We are also compatible out of the box with a 64-bit physical memory map with a memory usage dedicated to FSM storage close to the real memory usage of the simulated software. Also, the locality of reference of the simulated application is reflected in the container layout, thus benefiting from the host caching system. A software caching system could also be implemented in order to memorize the lastly accessed page of FSM, but the additional logic cancels the theoretical benefit.

We have implemented a template-parameterized page table allowing us to experiment several configurations. Such structure presents a speed-memory trade-off as increasing the number of levels tends to improve the allocation granularity but increases the number of pointer indirections required to access an element at the same time. The best configuration can depend slightly on the simulated platform but usually lies between 3 and 4 levels. Using a larger first level (allocated only once) and successive levels of decreasing size seems to give the best compromise on the experiments conducted but the impact is small. Also stepping down to 2 levels in case of a 32-bit target platform does not show any significant speedup ($\sim 1\%$). We eventually chose a 3-level table with respective sizes $(2^{23}, 2^{21}, 2^{20})$ in order to be able to cover all individual bytes in a 64-bit address space.

4.3. The Sequential Evaluation Phase

When one or more workers have been unscheduled during the parallel phase, they must complete their execution during the sequential phase. First, we need to choose an order to resume them (section 4.3.1). Also, dependency can appear during the sequential phase, so we perform dependency analysis at the end of each sequential phase (section 4.3.2). In case no conflict is detected but dependencies exist, they are recorded to be used during simulation replay (section 4.3.3). However, if a conflict did occur, then the simulation is no longer valid, so we use rollback to start over from the last checkpointed valid state (section 4.3.4).

4.3.1. Choosing the Sequential Evaluation Order

A priori, any worker order is valid for the sequential phase, but some orders are more likely to trigger conflicts than others. Indeed, the dependencies formed during the sequential phase depend on the order in which workers are resumed. Figure 6 illustrates how the sequential scheduling can cause a conflict or avoid it. In this example, we assume P_0 and P_1 are unscheduled because of an access to an already OWNED address not part of the illustration. Independently from this unscheduling access, they both access the shared variable x during the evaluation phase. If P_1 is scheduled before P_0 like in case ⓐ a conflict is formed as P_1 reads between the two writes of P_0. However, if P_0 is scheduled before P_1 like in case ⓑ, both writes happen before the read from P_1, avoiding the conflict.

This simple example illustrates the need for an efficient sequential scheduling policy. Having no existing dependency at the beginning of sequential phase, these cannot be used to constrain the scheduling order. However, when a worker W_0 is unscheduled due to an access to an address a that has W_1 registered as its owner in the FSM, it is likely that W_1 has recently accessed a. As a result, it is likely that when W_0 will resume and do the access to a, the dependency $W_1 \to W_0$ will instantly form, whatever the chosen sequential scheduling. A reasonable choice, then, is to schedule W_1 before W_0 to minimize the risk for a dependency $W_0 \to W_1$ to form.

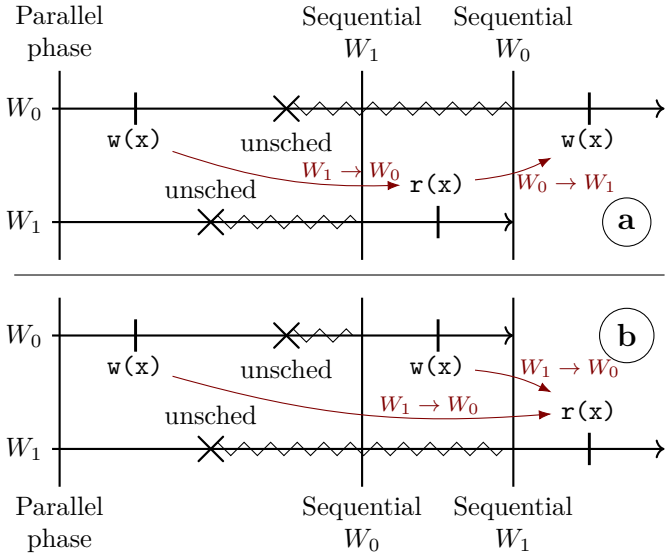

Figure 6: Two sequential worker scheduling leading to different dependencies.

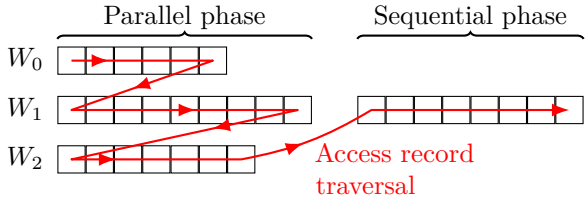

Figure 7: Structure used to record memory accesses with a valid traversal order used for dependency analysis.

To achieve this, every time a worker is unscheduled, we register in a temporary potential dependency graph G a dependency with the owner of the considered address, if it exists. At the end of the parallel phase, we attempt a topological sort on G to obtain an optimized scheduling order. We also add the unscheduled worker that are not in G at the end of the sequential schedule. In case G is cyclic, we schedule all workers in ascending WID order for simplicity.

One thing to keep in mind, however, is that the sequential phase might or might not issue a conflict, independently from the shape of G. G is only a heuristic which has proven to give significantly better results than random scheduling on most applications. For that reason, a strict dependency analysis is always required as explained in the next section.

4.3.2. Asynchronous Dependencies Analysis

We have demonstrated in Section 4.2 that dependencies cannot occur during the parallel phase thanks to our memory access granting policy and worker unscheduling. In case there is no sequential phase because no workers have been unscheduled — which is the case more than 99% of the time in most applications — we can simply proceed to the next evaluation phase without any extra precaution.

To that extent, during both the parallel and the sequential phases, all memory accesses are recorded in order to construct the worker dependency graph at the end, if need be (line 11 of algorithm 1). The required information is the accessed address, the number of bytes, the type of access and the WID of the accessing worker.

Recording all accesses during the parallel phase in a conventional container such as an `std::vector` would create huge contention. Fortunately, thanks again to the zero dependencies guaranty, the recording order of memory accesses during the parallel phase does not change the final dependencies. We only need that all accesses recorded during the sequential phase are recorded before the accesses of the sequential phase which, in turn, must be recorded in correct order.

Recording the sequential phase in the real order is achieved using a simple `std::vector` shared by all workers in the sequential phase. For the parallel phase where the order does not matter, we use one vector per worker to guarantee maximum decoupling of memory accesses recording. Accesses can then be enumerated one vector at a time finishing with the vector of the sequential phase like on fig. 7.

At the end of the sequential evaluation phase, the recorded accesses are analyzed using another OS thread while the simulation continues. We use the same graph cycle detection algorithm to check for conflicts as in [8] (i.e. Tarjan's algorithm) but with a thinner resolution. Indeed, accesses are analyzed at the byte level, meaning that accesses that hit the same block of memory addresses but different bytes inside this block will not cause dependencies. Because this analysis seldom takes place and is performed asynchronously and can use spare host cores, its impact on simulation speed is reduced to the bare minimum.

An important optimization consists in recycling the memory access records once they are checked. Indeed, growing newly constructed vectors causes numerous memory allocations and initializations that are expensive compared to the rest of the memory accesses monitoring procedure. For that reason, we only empty out these vectors in order to reuse their existing buffers.

The dependencies analysis results are gathered by the kernel thread during a subsequent parallel phase while it waits for workers to finish as shown on fig. 2. Two results are returned. First, the analysis tells if there was a conflict during the analyzed evaluation phase. If there is, rollback is used to recover from the error (c.f. section 4.3.4). If there is no conflict, a linear ordering of the workers involved in the dependencies is saved in the *output replay vector* alongside the analyzed phase identifier to later be able to replay the simulation (c.f. section 4.3.3). The rollback functionality will need to replay the simulation from the restoration point up to the problematic evaluation phase, which uses the same mechanism as full simulation replay for debugging purpose. We therefore present the replay mechanism first.

4.3.3. Replay Trace Generation

SCale 2.0 allows for simulation replay. That is, the first run of a simulation, which we call the *recording run* generates an execution trace that can be used during subsequent runs of the same simulation to reproduce the same behavior. We call such run a *replay run*. Notice that the recording run, while being non-deterministic in the sense that the equivalent sequential schedule of each phase is not predetermined, respects the co-routine semantics and is already standard compliant. Replay is an additional convenience offered for debugging purpose.

Simulation replay is useful for debugging purpose in order to reproduce a faulty behavior and fix it. However, replaying a simulation requires determinism, which is our first motivation in achieving standard-compliant parallel simulation, that is having each evaluation phase be equivalent to *a* sequential evaluation. In order to be able to replay a simulation, we need to know which sequential schedule we are equivalent to.

This is achieved after each sequential phase of the recording run using worker dependencies analysis yielding an equivalent partial sequential schedule after each sequential phase. Only the workers involved in dependencies appear in this sequential schedule. For instance, if there are 5 workers with WIDs ranging from 0 to 4 and the dependency graph after evaluation phase E is

$$2 \rightarrow 0, 2 \rightarrow 3$$

then the equivalent sequential schedule provided by the dependencies analysis can be $2 \rightarrow 0 \rightarrow 3$ or $2 \rightarrow 3 \rightarrow 0$. As 1 and 4 are not part of it, it implicitly means that they can be scheduled at any time in the equivalent sequential schedule of E, during the parallel phase of the replay run.

To enable simulation replay, the recording run only need to store in a file all the equivalent partial sequential schedule returned by the dependencies analysis associated to the identifier of the evaluation phase they relate to. This identifier is an integer incremented before each evaluation phase.

When replay is activated by passing the trace file to the simulator, an *input replay vector* is initialized with the replay file content ordered by decreasing order. The scheduler logic in replay mode is then described in algorithm 4. In particular, line 3 is responsible for testing if the next evaluation phase has a constrained order by checking the phase identifier of the last element of the input replay vector; line 4 retrieves this order; line 5 pops the last entry in the input replay vector; line 10 performs the constrained evaluation. The evaluation can be conducted in several ways. The sequential workers being independent from the ones in the parallel phase, they can be evaluated in parallel with the parallel phase (i.e. the sequential phase runs in parallel with the parallel phase). However, maintaining the sequential phase after the parallel phase allows safe replay verification using the same memory accesses instrumentation as during the recording run. We have selected the first option for our experiments but the second one helped for model instrumentation debugging.

Algorithm 4 Scheduler logic outline. *irv* is the input replay vector.

```
1: procedure EVALUATENEXTPHASEREPLAY(irv)
2:     ++phaseID
3:     if phaseID == irv.back().phaseID then
4:         seq ← irv.back().orderedworkers
5:         irv.pop_back()
6:     else
7:         seq ← ∅
8:     end if
9:     par ← {pid | pid is ready and pid ∉ seq}
10:    parAndSeqEval(par, seq)
11: end procedure
```

Memory accesses instrumentation could be disabled in replay mode if the model is guaranteed to be correctly instrumented. The most efficient way to do it would be to have a replay-specific binary with all the dependency analysis related functions replaced by empty functions instead of repeatedly checking if replay is active in numerous places of the simulation. However, due to the relatively low overhead of instrumentation and the debugging help it provides in case of bugged model instrumentation, we have chosen to keep instrumentation active in replay mode.

Replay is also used in case of rollback as it is developed in the next section.

4.3.4. Conflict Recovery With Rollback

In case of conflict, the simulation is no longer valid, and it must rollback to the last valid checkpointed state. While rollback has never been considered for optimistic PDES of SystemC models due to its cost and complexity, it has been used for other purposes. For instance, [31] proposes a Checkpoint/Restore (C/R) framework for SystemC virtual platform that enables resuming a regular SystemC simulation at different points for debugging purpose. In a completely different approach, [32] proposes to rely on POSIX's `fork()` to recover from timing errors caused by temporal decoupling. While `fork()` cannot be used for multithreaded process checkpointing because threads do not survive forking, process level checkpointing like in [31] supports it. However, we make a different use of it. A complimentary and non SystemC-specific approach can be found in the *rr-project* [33]. *rr* —for *record and replay*— runs a process in a virtually sequential environment while recording all its inputs (e.g. data returned by system calls). It allows to replay execution of large-scale processes such as a full featured browser but enforces sequential execution where SCale 2.0 aims at the opposite: adding parallelism to an originally sequential and deterministic simulator. These two approaches to recording and replay are not to be confused but could be combined in order to increase SCale 2.0 compatibility with simulation that rely on external outputs.

For simulation rollback, we rely on process C/R using CRIU [34]. This tool can perform full OS process state checkpoint to drive and restore a process from these generated files. Also, CRIU supports incremental checkpoints, that is it can checkpoint only the memory that has been modified since the last process checkpoint. This speeds up drastically the checkpointing operation allowing to increase their frequency. Together with OS automatic file caching or the use of a RAM disk, process checkpoint overhead is limited. Typically, a simulation using 1 GB of physical memory can be checkpointed in less than 500 ms while incremental checkpoints take about 100 ms.

Figure 8 illustrates the overall rollback-based conflict recovery logic. An initial dump is performed before the first evaluation phase. Then, the simulation runs until a conflict arises. If so, the simulation is restored to the last valid checkpointed state and runs again in replay mode until the conflicting quantum is reached. This quantum is sequentially evaluated to prevent the conflict from occurring again. A new snapshot is made right after this quantum to be used as the next restore point in case another conflict arises later. Checkpointing once the conflicting quantum finishes guaranties faster simulation progress in case another conflict arises briefly after.

Notice that it is required to use simulation replay between the restored checkpoint and the conflicting phase as a conflict could occur at a different place otherwise, causing simulation to rollback to the same checkpoint several times in a raw, preventing progress. It is also mandatory to wait for all pending evaluation phase check to complete in order to ensure the next checkpoint contains a valid state of the simulation.

Checkpoint frequency impacts could benefit from more in-depth investigation, but we have sufficient insights that allows to choose an appropriate parameter. To this day, we perform regularly spaced dumps at between 1- and 2-secons intervals depending on the simulated application profile. It appeared that the impact on final performance is minor as conflicts tend to be densely distributed in some areas, making the checkpoint frequency determined by the rollback frequency, while they are seldom in other areas of the simulation, making the checkpoints spaced at the chose frequency. Excessively reducing the checkpoint frequency will also incur a longer replay phase after each extended conflict-free period, canceling the small benefit of a large checkpoint interval. We settled at a 1.5 second checkpoint interval as a good compromise.

Performing a rollback is useful only if able to memorize the information required to avoid the conflict the next time. In our case, we want to transfer the replay instructions and the conflicting phase identifier. Figure 9 illustrates the principles of the rollback system designed toward that goal. It is packaged as an external library as rollback-based error recovery could be applied to other applications than SystemC simulation. While the simulation can self-checkpoint sending a request to the CRIU service, it cannot self-restore. Hence, for a C/R cycle to complete autonomously, we need

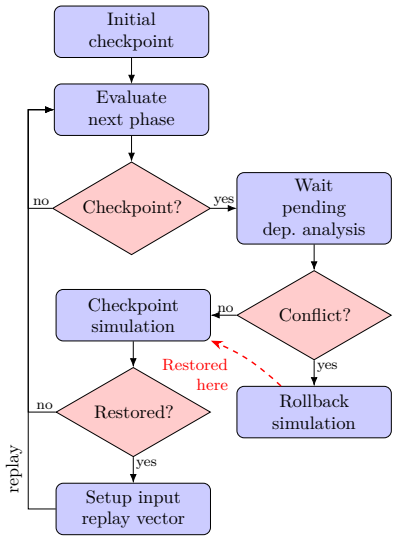

Figure 8: Rollback-based conflict recovery.

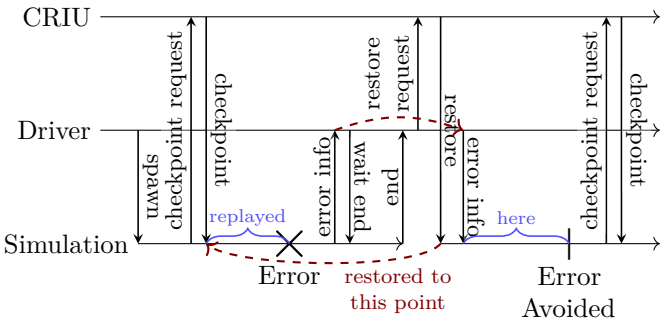

Figure 9: Protocol of interaction between CRIU, driver and simulation. In SCale 2.0, an error always corresponds to a conflict during an evaluation phase.

two processes: the driver and the simulation. The simulation is the actual workload that can encounter errors and requires rollback. The driver is an idle process that only spawns the simulation and wakes up to serve rollback requests from the simulation. The simulation and the driver communicate together with named pipes as the link must survive one of the two processes dying. The QEMU service process listen to requests on a UNIX-domain socket.

When the simulation needs to be restored, it sends a request to the driver together with some serialized data before aborting. Once the simulation process is done, the driver sends to CRIU a restore request. Finally, once the simulation is restored, it retrieves the serialized data that the driver immediately sends back and starts simulating again.

However, while being a fast and powerful C/R tool, CRIU has not been designed for error recovery using rollback. It presents certain limitations that require some additional engineering on our side to circumvent. The major limitation is that a process can only be checkpoint and restore with the same OS process identifier. However, if

such identifier has been recycled for another process between simulation abort and restore, CRIU will not restore the simulation until the process identifier is freed. For rollback to be usable, we had to run simulations in a process identifier namespace [35]. Other technicalities outside of the scope of this paper also required to use a program as *dumb-init* [36] for zombie process reaping as CRIU restores processes as daemons.

Additional minor limitations are CRIU's flat refusal to restore processes with handles to files whose size have changed. For instance, if a file contained $35kB$ at checkpoint time but is $42kB$ at restore time, CRIU refuses to proceed for safety reasons. This could be fixed as a patch on CRIU side in order to ignore such check (thus overwriting the content written after the last checkpoint) or by truncating all files to their checkpoint size before sending the restore request. The simplest fix is to flush streams to files only right before each checkpoint, when the simulation has been fully checked up to this point and it is guaranteed it will never need to revert to an earlier checkpoint. Also, CRIU accesses some kernel facilities restricted to `root`, thus requiring starting its service manually or to run the whole simulation as `root`. Lastly, incremental memory dump is not supported for the first checkpoint after a rollback, requiring a full and slower dump. As a result, even if memory can be checkpointed and restored at close to $3GB/s$, freeing unnecessary memory before checkpoints can be beneficial to both restore and non-incremental checkpoint speed. We do so by freeing all memory access records used as part of the dependencies analysis system. The cost of reallocating access records buffers every 1.5 seconds or so is negligible. This limitation could however also be relatively easily fixed on CRIU side and is discussed on their side.

4.4. Generalization to Any Shared Resources

As explained in the section 4.1, we have assumed up to this point that the only worker interactions come from shared model memory accesses. However, interactions could happen at many other locations of a SystemC model. For instance, the interrupt management system of an ISS such a QEMU is a complex set of variables read and written from both inside and outside the ISS. All these *shared resources* must be handled carefully.

We distinguish 2 types of shared resources based on the type of interactions they cause:

1. *with external side effects*: changing the order of interaction changes the behavior of the simulation processes. For instance, model memory accesses, interrupt raising, or timer component access often have consequences on the rest of the simulation.
2. *without external side effects*: changing the order of interaction does not changes the behavior of the processes. For instance, incrementing an atomic access counter on a component, allocating system memory or reading a constant shared variable does not influence the rest of the simulation. However, the value of an atomic counter must never be used as a condition for anything or it would have side effects.

It is up to the user to detect all potential interactions between worker, just as when checking interactions between threads in a regular multithreaded program. A number of these interactions cause data races but end up being without external side effects once protected either using atomic operations or mutexes. The rest of the interactions (i.e. with external side effects) must be protected using our system.

To that extent, the user first must define each shared resource perimeter. Just as memory bytes are grouped to increase monitoring efficiency, a shared resource such as the interrupt management system of an ISS can either be considered as a set of shared resources (each one of its variables) or as a unique resource. Similarly, a timer module likely has some internal logic that is shared between processes and only making it thread safe could change the event delivery order. This makes the timer a resource with external side effects, thus requiring monitoring of its accesses. The question is again whether the timer should be considered as a single resource or as a set of resources. After our experience, considering such aggregated resources as a unique resource is often the safest and faster choice.

At this point, resources are delimited but we have only explained how we can protect a full 64-bit memory map. We generalize this approach by classifying all operations on a resource as reads or writes. The former are operations that do not modify the accessed resource while the later do. Then, we define identifiers for all these resources and associate an FSM to each one of them. The identifiers can be contiguous integers, enabling the use of a preallocated vector to store the FSMs. Finally, the user just needs to insert calls to the provided `generic_instr` function that does the same as the `mem_instr` function but for the other resources designated as *generic resources*. In case it is not clear whether some code, for instance hidden inside an ISS, performs reads or writes, it is a conservative choice to declare the whole as a write.

All the required elements for standard compliant parallel simulation of time-decoupled TLM-LT models have been exposed. Section 6 will discuss some extra functionalities brought in order to answer some Linux-specific issues we have encountered.

5. Evaluation

5.1. Experimental Setup

Experiments have been conducted on a 36-core bi-Xeon Gold 6154 clocked at 3.5GHz with frequency scaling disabled.

The reference VP used for the evaluation of our contributions is a RISC-V SMP platform illustrated fig. 10. Each core is modeled by an instance of QEMU encapsulated in a SystemC wrapper implemented after [7]. The platform is composed of 1 to 32 cores for baremetal benchmarks, thus

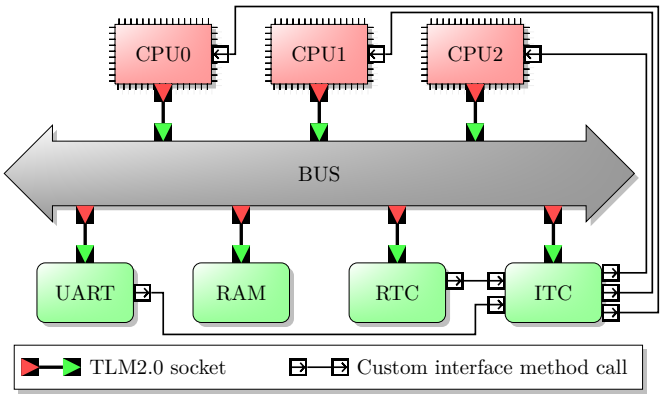

Figure 10: Architecture of the simulated platform here with 3 cores

using 2 to 33 cores on the host machine because of the kernel thread. For Linux benchmarks, only the 32-core version of the platform is used. The cores are connected through a bus to a RAM, an UART and an interrupt controller. DMI is used to access the memory for faster simulation. Our goal here is to shorten the memory accesses simulation in order to evaluate the relative overhead of instrumentation in worst-case conditions. In our case, DMI reduces a memory access to checking that it targets the RAM and doing an access of the size required by QEMU using a raw pointer to the underlying memory of the RAM component.

We have selected five benchmarks to evaluate the performance of the proposed approach. Three of them are implemented both in baremetal, that is to run without the support and complexity of a guest OS, and on Linux 4.10:

1. Matmul: 10 parallel multiplications of two square matrices of size 512. Each thread computes a horizontal block of the result. Threads only synchronize between each of the 10 multiplications.
2. Deriche [37]: A 10-pass Deriche filtering is applied in place to a 4 megapixels image. This benchmark is composed of a horizontal followed by a vertical filtering, making the whole image shared by all the threads. Each core processes a contiguous block of the image, either horizontal or vertical depending on the filtering stage.
3. MobileNet [38]: a 31-layer classification convolutional neural network analyzing a triple channel 160×160 images. The parallelism potential varies depending on the computed layer and much more synchronizations occur than in the first two benchmarks. Each convolution of each layer is attributed to a different thread.

All these three applications were ported to Linux using POSIX's threads. Synchronization is achieved in both cases using a spinlock-based barrier.

Two other benchmarks from the PARSEC3.0 suite [39] were only used on Linux as they make extensive use of OS supported functionalities (e.g. dynamic memory allocation or filesystem accesses):

1. Blackscholes: An Intel RMS benchmark computing options pricing using the Black-Scholes partial differential equation;
2. Swaptions: Another Intel RMS benchmark computing options pricing but using the Heath-Jarrow-Morton framework.

Despite looking similar from the description standpoint, these two applications have drastically different behavior.

Please also notice that the results obtained in the following experiments cannot be compared to other approaches conveniently. First, most of the other contributions have no public release. Then, none of the other approaches tackles parallel simulation of LT-TLM models of SMP platforms. Only SCale 1.0 does and a comparison is done in section 5.3.1.

5.2. Experimental functional validation

In addition to performance evaluations, functional validity and robustness of simulation replay have been experimentally asserted using a synthetic baremetal benchmark. It consists in a master thread that sends software interrupts to a set of slave threads upon reception of a timer interrupt scheduled at regular intervals. Between interrupts, slave threads hash a local variable h repeatability in an infinite loop. When they receive the interrupt, they all pause and the value of h for each thread is collected by the master thread and hashed in a *total* variable. This cycle is repeated up to 1024 times varying the timer interrupt interval. The output of the program is the final value of *total* which shows pseudo random variations from an execution to the next when **not** using deterministic replay as interrupts are raised and handled at non-deterministic times. To the contrary, the value of *total* at the end of the replay runs is always the same as at the end of the corresponding recording run.

We also checked on Blackscholes and PARSEC and on many other complex Linux applications from the PARSEC suite (e.g. ferret, fluidanimate, freqmine, etc.) that replay with monitoring enabled exhibits no conflict nor deadlocks caused by atomicity violations for instance. Considering some of these applications can cause hundreds of rollbacks and generate replay traces with tens of thousands of scheduling constraints, we consider this as an additional experimental proof of validity.

The remaining of this section now focuses on the performance of SCale 2.0.

5.3. Performance Evaluation

As the baremetal and Linux-based applications behave differently with SCale 2.0, they are analyzed in separate sections, respectively sections 5.3.1 and 5.3.2.

5.3.1. Baremetal Performance Evaluation

Baremetal applications present the advantage to be very predictable and to offer the highest simulation speed in most simulators. We use them as semi-synthetic use cases

to characterize the impact of our approach on simulation speed (mainly the speedup from parallelization and instrumentation overhead) and as a comparison against SCale in its original version.

Figure 11a illustrates the impact of quantum size on simulation speed. As expected, increasing the quantum size results in a significant speedup reaching up to 2,300 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. Relying on shared variables, each synchronization leads to worker sequentializations and FSM reset. When the quantum increases, the amount of time spend in sequential phases because of worker unscheduling increases to a point where it is no longer compensated by the speedup in the parallel phase. For the rest of the baremetal evaluations, we use a quantum of 30,000 instructions with 1GHz simulated processors (i.e., 30,000 ns quantum) as a performance compromise between the three benchmarks.

To evaluate the influence of memory accesses instrumentation and worker sequentialization, four versions of the kernel are compared on fig. 11b. The overhead of instrumentation and sequentialization compared to fully parallel simulation ranges from 34 to 48%. Part of this speed reduction is due to sequentialization as the overhead of instrumentation. The rest is caused by instrumentation overhead which is to put in perspective with the raw speed of the free parallel simulation: from 1,600 to 3,200 MIPS. Sequentialization overhead is hardly compressible as it results from strict co-routine semantics enforcement. Instrumentation overhead is to compare to the already extremely low memory access simulation cost: a pair of function calls and a raw pointer-based memory access. Also, the increase in speed compared to [8] is significant ranging from $\times 60$ to $\times 110$. It is mostly due to the much faster instrumentation technique together with the asynchronous conflict checking.

Figure 11c illustrates how our simulation kernel scales with the number of host cores used to simulate a 32-core platform. Overall, speedups using 32 host cores (plus the kernel core) range between $\times 17$ and $\times 21$ compared to using a single host core on SCale 2.0. The exact cause of sublinear speedups in unknown but is also observed when running a parallel simulation without memory accesses instrumentation. As a result, either SCale 2.0, the model, the simulated software or the host inherently do not scale up to 32 workers. A combination of these factors is probably in play.

fig. 11d shows the impact of simulated platform complexity (number of simulated cores) on speed when always using one host core per simulated core. While using the Accellera kernel is faster to simulate a single core platform due to a simpler scheduler and the absence of instrumentation, the speedup is already significant on a dual-core platform simulated in parallel as shown on fig. 11d. It reaches up to $\times 15$ on a 32-core simulated platform running Matmul.

In all baremetal benchmarks, all conflicts were avoided, and rollback was unnecessary. In general, conflicts are systematically avoided if processes always synchronize using a same shared variable before accessing data previously owned by another worker. It is the case in our baremetal benchmarks as they all implement a fork join pattern with barriers every time the working set of the threads changes (e.g. between the horizontal and vertical passes in Deriche). This barrier being a single shared variable, all but one worker are unscheduled when accessing it (c.f. zero dependencies guaranty), resulting in a fully sequential evaluation at barrier crossing. Before and after the barrier, working sets are disjoined in order to avoid data races, as in most lock-based parallel algorithms.

However, whenever lock-less programming patterns are used in the simulated software, shared variables can be used in unpredictable order and conflicts arise. This is typically the case in Linux based application as detailed in section 5.3.2.

5.3.2. Linux Performance Evaluation

Figure 12 shows that, without further measures to avoid conflicts, while the speedup ranges from $\times 2$ to $\times 8.5$ during replay runs, it caps at $\times 3$ during the recording run and is sometimes lower than $\times 1$, going down to $\times 0.5$. This section will further analyze the root causes of these lower performance and provide solutions to some of them.

Lower speedups can have many causes, either internal or external to Scale 2.0. For instance, bad parallelism in the simulated platform causing unbalanced load between workers. This is not the case here as each worker evaluates the same number of processes all simulating symmetrical cores.

Bad parallelism in the simulated software can also cause bad load balancing among workers. For instance, simulating an idle core is almost instantaneous in our model resulting in some workers waiting for those simulating non idle cores. This is the case during the Linux boot and poweroff which are mostly sequential procedures and during some parts of each benchmark (e.g. setup, result aggregation, etc.). For instance, the parallel section of the Blackscholes benchmark accounts for less than 50% of the total simulation duration when simulated on the ASI kernel and the rest of the simulation being mostly single threaded. According to Amdahl's law, the maximum speedup achievable through parallelization is less than $\times 2$. In general, when simulating a full software stack that includes numerous sequential sections, the maximum achievable speedup cannot be proportional to the number of workers, but the upper limit is very hard to determine accurately. In order to eliminate this unknown from the causes of sub-linear speedup, all subsequent measures ignore the boot, the poweroff and the benchmark loading procedure from the measure to focus on the parallel workload only.

However, OS provided functionalities such as virtual memory or filesystem management can introduce massive

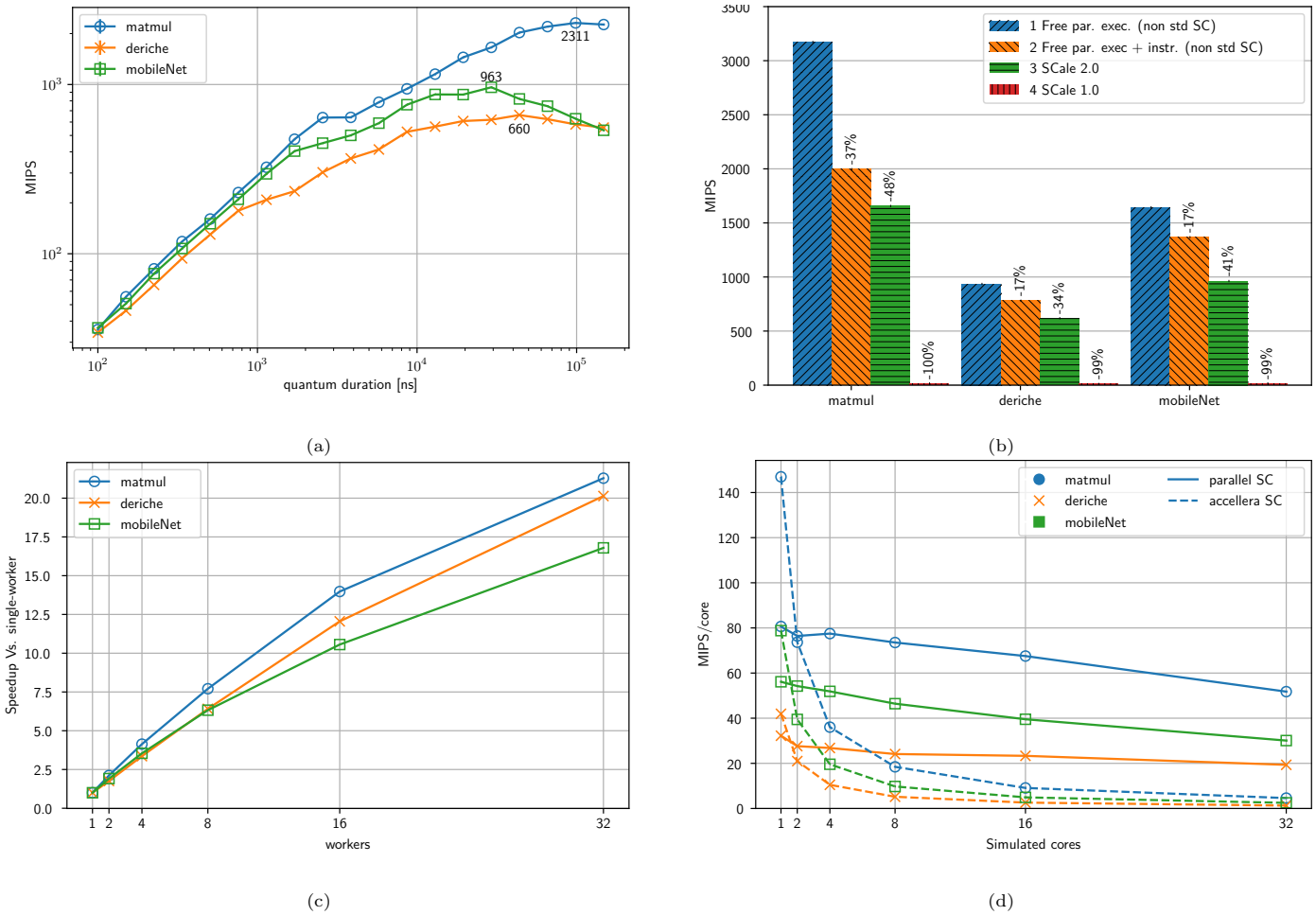

Figure 11: (a) Simulation speed analysis depending on the simulation quantum size. The highest point of each curve is annotated. (b) Impact of instrumentation and worker unscheduling compared to free parallel execution when simulating 32 cores. Version 1 consists in a parallel simulation without enforcing processes atomicity. It includes the required synchronization for peripheral accesses of atomic instruction simulation for instance. Version 2 shows the overhead of instrumentation and conflicts detection alone without worker sequentialization. Version 3 implements all the contributions of this paper and is standard compliant. Version 4 is the same model but linked with version 1.0 of SCale from [8] without the ensuring correct memory order recording as explained in section 4.2.3 as systematic locking is unfair to this solution. (c) Scaling of the simulation speed with the number of used host cores (excluding the kernel core). (d) Simulation speed per simulated core with parallel (1 host core per simulated core) and sequential simulation (Accellera kernel).

amounts of synchronizations compared to the benchmark workload itself. The resulting numerous worker interdependencies lead to a lot of sequential worker evaluations that can be a major cause of sub-linear speedup in some portions of the simulation. Still, sequential worker evaluation account for less than 5% of the total simulation time in all benchmarks as shown on fig. 13, making it a secondary cause of reduced speedup. Frequent checkpointing cost is also to be considered. While incremental checkpointing drastically reduces its cost, it accounts for up to 10% of the total simulation time.

However, the sources of slow down are dominated by the cost of conflicts. Rollback and resimulation alone account for 40% to 50% of the simulation. Adding the cost of the non-incremental checkpoint required after each rollback, conflicts account for closer to 60% of the total simulation time.

As a result, reducing the number of conflicts to a minimum is essential in order to preserve overall simulation speed. The various options explored are discussed in section 6.

6. Full Software Stack Simulation

6.1. Obstacles to Conflicts Suppression

The first improvement one could think of would be to improve the conflict avoidance measures. We have identified two direct improvement tracks:

1. Issuing better schedules in the sequential phase as not all of them always lead to conflicts;
2. Unscheduling workers earlier using a more refined access granting policy in order to increase the amount of sequential execution as a trade of with rollback and resimulation.

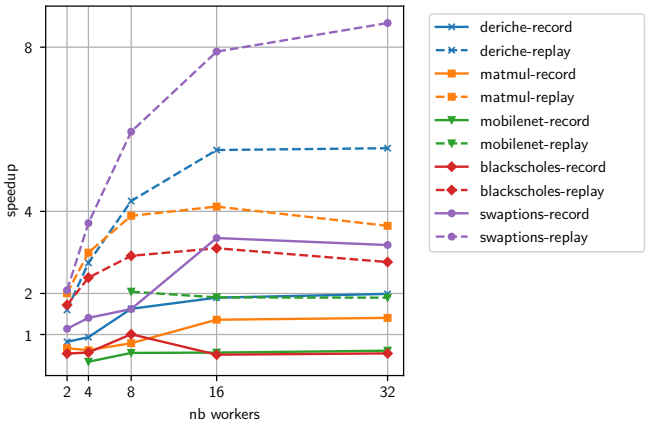

Figure 12: Speedup of SCale 2.0 compared to the ASI reference SystemC kernel depending on the number of workers in recording or replay mode. Results in this graph include the Linux boot and poweroff that are mostly sequential. (Due to instabilities under investigation using 2 and 4 workers with the MobileNet benchmark, some values are absent from the graph.)

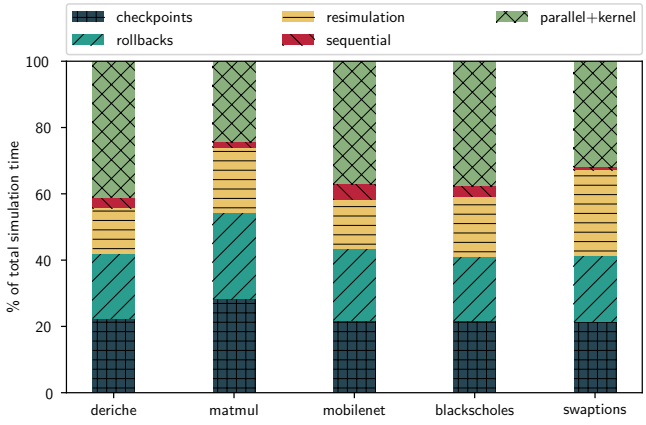

Figure 13: Analysis of the time spent in parallel and sequential simulation, in checkpointing and rollback procedures as well as the overhead caused by re-simulation after rollback, when using 8 workers (the trend is the same with all numbers of workers). Note that non-parallel simulated code might not result in sequential worker evaluation. The *sequential* time only reflects the amount of time spent in sequential evaluation phases.

On the one hand, rapidly issuing a good sequential schedule is difficult as no dependencies yet exist at the end of the parallel evaluation phase. Also, there is no guarantee that there exists a sequential schedule that would prevent a given conflict from occurring. Determining, after a conflict has occurred, if a different sequential schedule would have prevented *any* conflict from happening is impossible without running this very same sequential phase in a different order. We plan on doing extensive testing soon taking advantage of our rollback mechanism in order to determine the room for improvement on this track.

On the other hand, a better access granting policy often results in a compromise between the time required to determine if an access can proceed or lead to a worker being unscheduled, the amount of additional unscheduling done while no conflict would have occurred anyway (i.e. pessimization) and the amount of conflicts let through. While we plan on performing in-depth analysis of conflicts causes in order to quantify the room for improvement, we have resorted to other strategies detailed in the next sections in order to significantly reduce conflicts occurrences.

6.2. Adaptive parallel/sequential evaluation and variable accuracy

While characterizing the memory access patterns responsible for the conflicts is a difficult problem, the code sections responsible for conflicts are easier to identify. The first obvious code sections that cause many conflicts are the Linux boot and poweroff procedures. At the same time, these code sections are of little interest to the simulator user who usually wants to analyze the behavior of its platform in the application under development, not the OS setup and tear down code.

We define the Region of Interest (ROI) as the part of the code where the user needs more accurate information about the simulated platform behavior such as timing or cache behavior. On the opposite, we assume that the user only needs the platform behavior to be functionally correct outside of the ROI. It generally requires that the instruction set and peripheral accesses are consistently simulated in order to guarantee progress and valid model state. In particular, timing information is of little interests outside of the ROI. However, determinism must be preserved both inside and outside the ROI to ensure determinism.

In order to maximize simulation speed outside of the ROI, parallel execution is not always the fastest option if timing is optional. QEMU on its own can reach very high simulation speeds at the instruction set simulation level thanks to dynamic binary translation. Though, as detailed in [7], integrating QEMU to a timed SystemC/TLM model requires that memory accesses (i.e. load, stores and fetches) are instrumented in order to hand over to the SystemC model each time such access occurs. This instrumentation, while being a pair of indirect function calls followed by a DMI memory access in our case significantly slows down QEMU but still maintains top level performances in the class of SystemC/TLM models.

In order to take advantage of the huge raw speed of QEMU outside of the ROI, while preserving modelling accuracy inside, we have setup *dynamic accuracy* also described in [7]. It allows to switch during the simulation from what we call *fast mode*, that is limiting memory access instrumentation to peripheral accesses and instruction counting for rough timing estimation, to *accurate mode* enabling full memory access instrumentation in QEMU for SystemC modelling. However, the fast mode does not support parallel execution as `mem_instr`, the SCale 2.0 instrumentation function, cannot be called if memory accesses are not instrumented in QEMU. As a result, standard-compliant parallel execution cannot be ensured anymore but sequen-

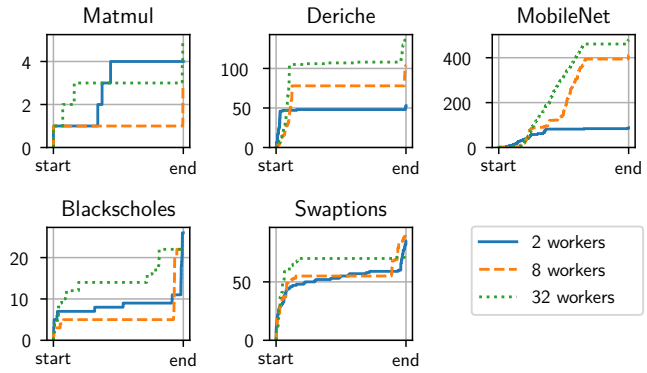

Figure 14: Temporal repartition of conflicts in ROI (cumulative curves).

tial evaluation remains valid.

Because we have measured that our model in fast mode is faster than parallel execution in accurate mode during the Linux Boot and poweroff, we have integrated a variable parallel/sequential evaluation feature to SCale 2.0 allowing to switch during the simulation from sequential to parallel scheduling. Together with dynamic accuracy, the variable parallel/sequential evaluation allows to choose the faster and lighter yet deterministic sequential fast mode outside of the ROI while using the parallel accurate mode during the ROI.

Remains the question of the performance inside the ROI. Each benchmark behaves differently, some giving very good speedups in parallel execution while others cause so many conflicts that the ROI is even slower than using the sequential ASI SystemC kernel. We further investigate this issue in the next section.

6.3. CPU Mode Based Unscheduling

We have studied the conflicts distribution in each benchmark ROI using 2, 8 and 32 workers on fig. 14. Independently from the benchmark, conflicts tend to be grouped at the beginning or the end of the ROI. In-depth analysis showed us that the conflicts frequency is much higher during the first pass on the benchmark dataset. For instance, in Deriche, conflicts occur when the input image is copied into the working buffer while in MobileNet conflicts occur mostly during the first iteration (out of three on the same input) during which the input and network coefficients are loaded into memory. Similarly, the final conflicts can either be caused by the writing of the result into the result buffer or by the threads joining.

In order to validate those assumptions, we edited four versions of the Matmul, Deriche and MobileNet benchmarks:

1. *native*: No modification
2. *no join*: The end of the ROI is triggered before the benchmark threads exit.
3. *warmup*: All data buffers used during the benchmark are accessed once before the ROI starts.

Table 1: Number of conflicts depending on benchmark variant and number of workers

Benchmark	Variant	Conflicts (nb. workers) 2	8	32
Matmul	original	4	3	5
	no join	10	1	1
	warmup	10	4	4
	warmup no join	5	0	0
Deriche	original	53	104	136
	no join	33	109	121
	warmup	5	6	5
	warmup no join	0	0	0
MobileNet	original	88	411	479
	no join	84	386	418
	warmup	18	16	14
	warmup no join	9	2	0

4. *warmup no join*: Both 2. and 3.

The number of conflicts in each one of these variations is shown in table 1. Set aside some minor variations especially on Matmul which natively presents few conflicts, each variant in the list is an improvement over the previous one, with the warmup-enabled variants being strongly better than the others.

When running under an OS such as Linux, even a simple variable initialization can cause a lot of extra code to be executed like a page fault handling procedure for instance. This translates into a strong correlation between the conflicts frequency and the amount of *system code* being executed in the benchmark.

Then, in order to reduce the number of conflicts, we have chosen to run all the system code in the sequential phase. System code can be identified by looking at the CPU mode [40]. If it is in machine or supervisor mode, then we are facing system code and if it is in user mode, then we are facing user code.

We added a callback to QEMU in order to catch CPU mode changes. In addition, a `force_sequential(bool)` function has been added to SCale 2.0. This function allows a worker to self unschedule and be executed sequentially when calling `force_sequential(true)` until it calls `force_sequential(false)`. In order to run all system code sequentially,

```
force_sequential(new_mode != user_mode)
```

is called upon every CPU mode changes.

The number of conflicts when executing the system code sequentially as well as the speedup in the ROI compared to the ASI kernel are shown on table 2 and fig. 15. It can be observed that the number of conflicts is drastically reduced to between 0 and 3 for the whole simulation (this number varies slightly from one recording run to the other). It leads to a speedup greater than 1 in all benchmarks using

Table 2: Number of conflicts during the ROI using `force_sequential()` with benchmarks Matmul, Deriche, Blackscholes and Swaptions, enabling 2, 4, 8, 16, and 32 workers

Nb. workers	2	4	8	16	32
Deriche	1	1	0	1	2
Matmul	1	1	1	1	0
Blackscholes	1	3	0	0	3
Swaptions	0	1	0	0	3

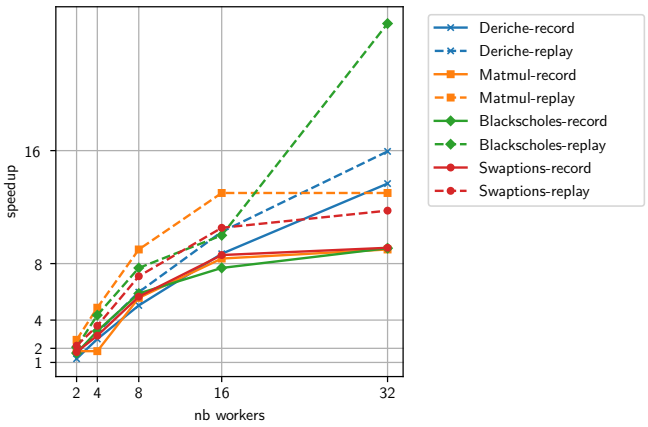

Figure 15: Speedup Vs. ASI kernel during the ROI using `force_sequential()` with benchmarks Matmul, Deriche, Blackscholes and Swaptions[1], enabling 2, 4, 8, 16, and 32 workers

2 to 32 workers. Specifically, depending on the benchmark, the recording run speedup ranges from ×9 to ×13 and the replay run speedup ranges from ×11.5 to ×24. It should also be noted that going from 16 to 32 workers leads to little improvements. This is partly due to the ROI not being fully parallel and getting too short to show a significant speedup when using 32 cores.

The Linux-based benchmark results are showed in simulation time instead of MIPS. MIPS are irrelevant in these benchmarks because of the Wait For Interrupt (WFI) instruction. Indeed, in our model, WFI is simulated by immediately waiting for the rest of the quantum time until an interrupt is raised. It results in zero simulated instruction the WFI instruction and the next interrupt, that is during all processors idle time. Because benchmark parallelism is not perfect, processors idle for a non-negligible amount of time, resulting in relatively low MIPS values. On the opposite, WFI is not used in baremetal benchmarks, hence the use of MIPS in that case.

As a result, we have achieved fast parallel and standard compliant simulation of TLM-LT models running Linux. We mainly rely on memory-access-monitoring-based conflict avoidance but exclude notoriously conflicting code sections of the parallel evaluation phase based on the CPU mode provided by the ISS. The extra sequential evaluations are largely compensated by the conflict occurrences reduction. We now plan on doing in-depth analysis of the conflicting sections in order to identify the root cause of the conflicts. If regular access patterns are identified, it might be possible to improve the memory access granting policy at little performance cost. Calls to `force_sequential()` could then be removed and even system code would be parallelized properly.

[1] Due to inconsistencies still under investigation in the MobileNet benchmark when using `force_sequential()`, this benchmark is not included in these graphics. The trend is however the same as for the other benchmarks.

7. Conclusion

This article details a new standard-compliant approach for SystemC parallelization of TLM-LT models using temporal decoupling and fast ISS's. It relies on lightweight resource access monitoring, conflict avoidance heuristic, dependency analysis, error recovery through rollback and fast replay. It is proven efficient on both baremetal and Linux-based applications with speedups reaching ×15 compared to the ASI reference implementation on baremetal applications and on Linux-based applications, ×13 during recording run and ×24 during the replay run.

However, some specific scenarios defeat the resource access granting policy, triggering numerous conflicts and making the cost of rollback cancel the parallelization speedup. In general, SCale 2.0 is the most efficient when simulating code with few interactions in order to minimize the sequential evaluations and the risk of conflict. Variable parallel/sequential scheduling enables sequential execution of code sections like the conflict-prone boot, which allows to use of a very fast but untimed mode in our ISS. Also, forced sequentialization of privileged code eliminates most of the conflicts that remain in the ROI.

In addition, SCale 2.0 solves numerous issues related to user software execution on top of an operating system that SCale 1.0 could not deal with like virtualized memory. Also, all interactions between processes like interrupts are now now supported using the same mechanism as for memory accesses.

We now plan on performing trace-based analysis of memory accesses and interrupts in order to refine the resource access granting policy while maintaining its overhead minimal. Also, reducing the quantum size could be beneficial to reduce the number of conflicts, but it requires a faster SystemC scheduler. Finally, resolving conflicts at the process level instead of the worker level could reduce further the number of conflicts and increase overall parallelism.

References

[1] IEEE Standard for Standard SystemC ® Language Reference Manual IEEE Computer Society, Vol. 2011, 2012. doi:10.1109/IEEESTD.2012.6134619.

[2] J. Aynsley, OSCI TLM-2.0 language reference manual, Open SystemC Initiative (OSCI), Tech. Rep (2009).

[3] ScRefImpl, SystemC reference implementation. URL https://www.accellera.org/downloads/standards/systemc

[4] R. Dömer, Seven Obstacles in the Way of Standard-Compliant Parallel SystemC Simulation, IEEE Embedded Systems Letters 8 (4) (2016) 81–84. doi:10.1109/LES.2016.2617284.
URL http://ieeexplore.ieee.org/document/7589063/

[5] D. Becker, M. Moy, J. Cornet, Challenges for the parallelization of loosely timed SystemC programs, in: Proceedings - IEEE International Symposium on Rapid System Prototyping, RSP, Vol. 2016-Febru, IEEE, 2016, pp. 54–60. doi:10.1109/RSP.2015.7416547.
URL http://ieeexplore.ieee.org/document/7416547/

[6] F. Bellard, QEMU, a Fast and Portable Dynamic Translator, USENIX Annual Technical Conference, FREENIX ... (2005).

[7] A. Charif, G. Busnot, R. Mameesh, T. Sassolas, N. Ventroux, Fast virtual prototyping for embedded computing systems design and exploration, in: ACM International Conference Proceeding Series, Vol. Part F1483, ACM Press, New York, New York, USA, 2019, pp. 1–8. doi:10.1145/3300189.3300192.
URL http://dl.acm.org/citation.cfm?doid=3300189.3300192

[8] N. Ventroux, T. Sassolas, A new parallel SystemC kernel leveraging manycore architectures, Proceedings of the 2016 Design, Automation and Test in Europe Conference and Exhibition, DATE 2016 (2016) 487–492doi:10.3850/9783981537079_0325.
URL http://ieeexplore.ieee.org/abstract/document/7459359/

[9] B. Chopard, P. Combes, J. Zory, A conservative approach to SystemC parallelization, in: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Vol. 3994 LNCS, Springer Verlag, 2006, pp. 653–660. doi:10.1007/11758549_89.

[10] C. Schumacher, R. Leupers, D. Petras, A. Hoffmann, parSC: Synchronous parallel SystemC simulation on multi-core host architectures, Embedded Systems Week 2010 - Proceedings of the 8th IEEE/ACM/IFIP International Conference on Compilers, Architecture and Synthesis for Embedded Systems, CODES+ISSS'2010 (2010) 241–246doi:10.1145/1878961.1879005.

[11] C. Schumacher, J. H. Weinstock, R. Leupers, G. Ascheid, L. Tosoratto, A. Lonardo, D. Petras, A. Hoffmann, LegaSCi: Legacy systemc model integration into parallel simulators, ACM Transactions on Embedded Computing Systems 13 (5s) (2014) 1–24. doi:10.1145/2678018.
URL https://dl.acm.org/doi/10.1145/2678018

[12] J. Virtanen, P. Sjövall, M. Viitanen, T. D. Hämäläinen, J. Vanne, Distributed systemc simulation on manycore servers, in: NORCAS 2016 - 2nd IEEE NORCAS Conference, IEEE, 2016, pp. 1–6. doi:10.1109/NORCHIP.2016.7792920.
URL http://ieeexplore.ieee.org/document/7792920/

[13] N. Ventroux, J. Peeters, T. Sassolas, J. C. Hoe, Highly-parallel special-purpose multicore architecture for SystemC/TLM simulations, in: Proceedings - International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, SAMOS 2014, 2014, pp. 250–257. doi:10.1109/SAMOS.2014.6893218.

[14] S. Vinco, D. Chatterjee, V. Bertacco, F. Fummi, SAGA: SystemC acceleration on GPU architectures, Proceedings - Design Automation Conference (2012) 115–120doi:10.1145/2228360.2228382.
URL http://dl.acm.org/citation.cfm?doid=2228360.2228382

[15] R. Sinha, A. Prakash, H. D. Patel, Parallel simulation of mixed-abstraction SystemC models on GPUs and multicore CPUs, in: Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC, 2012, pp. 455–460. doi:10.1109/ASPDAC.2012.6164991.

[16] R. M. Fujimoto, Parallel discrete event simulation, Communications of the ACM 33 (10) (1990) 30–53. doi:10.1145/84537.84545.
URL http://portal.acm.org/citation.cfm?doid=84537.84545

[17] Y. Q. Huang, H. L. Li, X. H. Xie, L. Qian, Z. Y. Hao, F. Guo, K. Zhang, ArchSim: A System-Level Parallel Simulation Platform for the Architecture Design of High Performance Computer, Journal of Computer Science and Technology 24 (5) (2009) 901–912. doi:10.1007/s11390-009-9281-9.

[18] Z. Hao, Q. Lei, L. Hongliang, X. Xianghui, Z. Kun, A parallel SystemC environment: ArchSC, in: Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS, 2009, pp. 617–623. doi:10.1109/ICPADS.2009.28.

[19] E. Viaud, F. Pêcheux, A. Greiner, An efficient TLM/T modeling and simulation environment based on conservative parallel discrete event principles, in: Proceedings -Design, Automation and Test in Europe, DATE, Vol. 1, 2006. doi:10.1109/date.2006.244003.

[20] A. Mello, I. Maia, A. Greiner, F. Pêcheux, Parallel simulation of systemC TLM 2.0 compliant MPSoC on SMP workstations, Proceedings -Design, Automation and Test in Europe, DATE (2010) 606–609doi:10.1109/date.2010.5457136.
URL http://ieeexplore.ieee.org/document/5457136/http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5457136

[21] J. H. Weinstock, L. G. Murillo, R. Leupers, G. Ascheid, Parallel SystemC Simulation for ESL Design, ACM Transactions on Embedded Computing Systems 16 (1) (2016) 1–25. doi:10.1145/2987374.
URL http://dl.acm.org/citation.cfm?doid=3008024.2987374

[22] J. H. Weinstock, R. Leupers, G. Ascheid, D. Petras, A. Hoffmann, SystemC-link: Parallel SystemC simulation using time-decoupled segments, in: Proceedings of the 2016 Design, Automation and Test in Europe Conference and Exhibition, DATE 2016, 2016, pp. 493–498. doi:10.3850/9783981537079_0114.

[23] W. Chen, X. Han, C. W. Chang, G. Liu, R. Dömer, Out-of-order parallel discrete event simulation for transaction level models, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 33 (12) (2014) 1859–1872. doi:10.1109/TCAD.2014.2356469.
URL http://ieeexplore.ieee.org/document/6951878/

[24] G. Liu, T. Schmidt, R. Dömer, A segment-aware multi-core scheduler for system C PDES, in: 2016 IEEE International High Level Design Validation and Test Workshop, HLDVT 2016, IEEE, 2016, pp. 100–107. doi:10.1109/HLDVT.2016.7748262.
URL http://ieeexplore.ieee.org/document/7748262/

[25] T. Schmidt, G. Liu, R. Dömer, Hybrid analysis of SystemC models for fast and accurate parallel simulation, in: Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC, IEEE, 2017, pp. 226–231. doi:10.1109/ASPDAC.2017.7858324.
URL http://ieeexplore.ieee.org/document/7858324/

[26] T. Schmidt, Z. Cheng, R. Dömer, Port call path sensitive conflict analysis for instance-Aware parallel SystemC simulation, in: Proceedings of the 2018 Design, Automation and Test in Europe Conference and Exhibition, DATE 2018, Vol. 2018-Janua, IEEE, 2018, pp. 349–354. doi:10.23919/DATE.2018.8342034.
URL http://ieeexplore.ieee.org/document/8342034/

[27] Z. Cheng, E. Arasteh, R. Dömer, Event Delivery using Prediction for Faster Parallel SystemC Simulation, Institute of Electrical and Electronics Engineers (IEEE), 2020, pp. 357–362. doi:10.1109/asp-dac47756.2020.9045492.

[28] M. Moy, Parallel programming with SystemC for loosely timed models: A non-intrusive approach, Proceedings -Design, Automation and Test in Europe, DATE (2013) 9–14doi:10.7873/date.2013.017.
URL http://dl.acm.org/citation.cfm?id=2485288.2485294

[29] G. Busnot, T. Sassolas, N. Ventroux, M. Moy, Standard-compliant Parallel SystemC simulation of Loosely-Timed Transaction Level Models, in: Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC, Vol. 2020-Janua, Institute of Electrical and Electronics Engineers (IEEE), 2020, pp. 363–368. doi:10.1109/ASP-DAC47756.2020.9045568.

[30] P. J. Denning, The locality principle (jul 2005). doi:10.1145/1070838.1070856.
URL http://portal.acm.org/citation.cfm?doid=1070838.

1070856

[31] S. Kraemer, R. Leupers, D. Petras, T. Philipp, A checkpoint/restore framework for systemC-based virtual platforms, in: 2009 International Symposium on System-on-Chip - Proceedings, SoC 2009, IEEE, 2009, pp. 161–167. doi:10.1109/SOCC.2009.5335656.
URL http://ieeexplore.ieee.org/document/5335656/

[32] M. Jung, F. Schnicke, M. Damm, T. Kuhn, N. Wehn, Speculative Temporal Decoupling Using fork(), in: Proceedings of the 2019 Design, Automation and Test in Europe Conference and Exhibition, DATE 2019, IEEE, 2019, pp. 1721–1726. doi:10.23919/DATE.2019.8714823.
URL https://ieeexplore.ieee.org/document/8714823/

[33] rr: lightweight recording & deterministic debugging.
URL https://rr-project.org/

[34] Criu, Criu.
URL https://criu.org/Main{_}Page

[35] pid_namespaces(7) - Linux manual page.
URL http://man7.org/linux/man-pages/man7/pid{_}namespaces.7.html

[36] Yelp/dumb-init: A minimal init system for Linux containers.
URL https://github.com/Yelp/dumb-init

[37] R. Deriche, Using Canny's criteria to derive a recursively implemented optimal edge detector, International Journal of Computer Vision 1 (2) (1987) 167–187. doi:10.1007/BF00123164.

[38] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, H. Adam, MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications (apr 2017). arXiv:1704.04861.
URL http://arxiv.org/abs/1704.04861

[39] C. Bienia, S. Kumar, J. P. Singh, K. Li, The PARSEC benchmark suite: Characterization and architectural implications, in: Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT, 2008. doi:10.1145/1454115.1454128.

[40] ISA Specification - RISC-V International.
URL https://riscv.org/specifications/privileged-isa/