Challenges for the Parallelization of Loosely Timed SystemC Programs

Denis Becker*† Denis.Becker@st.com

Matthieu Moy† Matthieu.Moy@imag.fr

Jérôme Cornet* Jerome.Cornet@st.com

*STMicroelectronics, F-38019 Grenoble, France
†Univ. Grenoble Alpes, VERIMAG, F-38000 Grenoble, France
CNRS, VERIMAG, F-38000 Grenoble, France

***Abstract*—SystemC/TLM models are commonly used in the industry to provide an early SoC simulation environment. The open source implementation of the SystemC simulator is sequential. The standard doesn't impose sequential executions, but makes this choice the easiest by imposing coroutine semantics. With the increasing size and complexity of models, and the multiplication of computation cores on recent machines, the parallelization of SystemC simulations is a major research concern.**

There have been several proposals for SystemC parallelization, but most of them are limited to cycle-accurate models. In this paper we give an overview of the practices in one industrial context. We explain why loosely timed models are the only viable option in this context. We also show that unfortunately, most of the existing approaches for SystemC parallelization can fundamentally not apply to these models. We support this claim with a set of measurements performed on a platform used in production at STMicroelectronics.

This paper both surveys existing techniques and identifies unsolved challenges in the parallelization of SystemC/TLM models.

I. Introduction

Transaction level modeling (TLM) allows to simulate systems-on-chip (SoC) orders of magnitude faster than register transfer level (RTL), which is a key advantage for the embedded software development. TLM models can be designed early in the design flow, where information about the detailed microarchitecture and precise timing may not be available. Within TLM, multiple timing styles exist, depending on the level of detail needed and/or available. The loosely timed (LT) style being the most abstract timed style of TLM is a relevant choice for the industrial development of embedded software, because it allows to take advantage of the early situation by abstracting and therefore speeding up the simulation.

SystemC offers an API with parallel semantics to model at TLM level. It is used to model hardware, itself intrisically parallel. A SystemC program consists of threads and methods that are executed by a scheduler, which guarantees that the order of their execution respects the constraints specified in the model. According to the SystemC standard [1], a scheduler must behave *as if* it was implementing coroutine semantics. It means that for each execution order, there must exist a sequential scheduling that reproduces the same case. The SystemC simulation kernel given by the Accellera Systems Initiative (ASI, formerly OSCI) is a sequential implementation. Thus, the execution of a SystemC simulation with it only uses the resources of one core. An advantage of a sequential implementation is that it makes the determinism of executions easier to implement and eases the reproducibility of errors. A drawback is that it cannot exploit the parallelism of the host machine. With the increasing size of models, the simulation time is the major bottleneck of complex hardware systems simulation. The parallelization of SystemC simulation is not straightforward, and is a major research concern.

For more than a decade now, there have been several proposals for SystemC parallelization. An approach chosen in [2], [3], [4] is to run multiple processes concurrently inside a delta cycle, with a synchronization barrier at the end of each one. Parallel discrete event simulation (PDES) has also been exploited. First, with a conservative approach [5], [6], [7], [8], where all the time contraints are strictly fulfilled. Then with a more optimistic approach, by using temporal decoupling [9], [10], which consists in relaxing the synchronization with a time quantum as a possible delay between two processes. An other work [11] combined different methods; the parallelization inside delta cycles with relaxed synchronizations. Optimistic (non-conservative) approaches may need a rollback mechanism in case the simulation went through an invalid path. To conclude with this panorama, `sc_during` allows specifying that some parts of the simulation can be run concurrently with the rest of the platform [12].

Each of these approaches have been proved experimentally to be efficient on some benchmarks, but the representativity of these benchmarks compared to industrial case-studies is questionable. Indeed, not much of the works above target LT simulations, while our experience is that such models are better suited for fast and early simulation.

One difficulty is that real case studies are often confidential, and hardly available for the research community working on parallel simulation. Conversely, most research tools are not publicly available, hence a fair comparison on case-studies is not possible. Our claim is that the challenges raised by the parallelization of LT SystemC models are fundamentally different from the ones in cycle-accurate or other fine-granularity models. As a consequence, many of the existing approaches cannot work on LT models. To support this claim, we present measurements performed on an industrial LT platform and we give an overview of some practices at STMicroelectronics (ST). By giving these measurements, we show that some approaches cannot work by construction on the LT model we want to parallelize, and thus that any of the implementations using the considered technique won't be efficient.

We believe this paper provides a better understanding of the potential bottlenecks of various parallelization approaches on such platforms. It should help both the design of efficient parallelization solutions and the design of representative benchmarks. We also propose a comprehensive survey of the existing solutions with a critical analysis. Section II give some background information. Then in Section III we present a panorama of the research works about the parallelization of SystemC simulations. Section IV presents the method we used to perform those measurements on this platform, and the results we obtained. We discuss these results in Section V.

II. Background

A. SystemC Scheduling

As a reminder, we first present in Figure 1 an abstract of how the SystemC simulator behaves, as stated in the SystemC standard [1]. In this paper, the locution *SystemC process* means indifferently `SC_THREAD` or `SC_METHOD` (`SC_CTHREAD` are not in the scope of this paper).

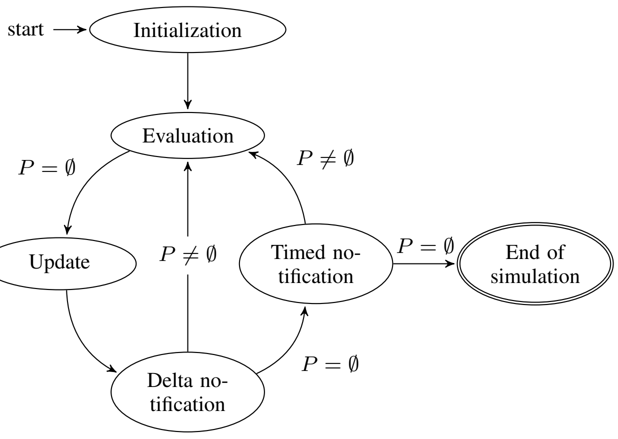


Figure 1. Behaviour of a SystemC simulator. P is the set of runnable processes.

The scheduler starts with an initialization phase that we don't detail here. In the evaluation phase, the runnable processes are executed with no particular order. The immediate notifications produced by these executions are then triggered. If there are runnable processes after the notifications, then the evaluation phase continues. Otherwise, the scheduler moves to the update phase, followed by the delta notification phase. The immediate notification loop is implicit on the figure, within the evaluation phase box. A *delta cycle* corresponds to the loop: *evaluation*, *update* and *delta notification*. At the end of a delta cycle, if there are no runnable processes, the scheduler checks timed events. If there are timed events, it picks the earliest one, sets the current simulation time to its time, and notifies the time change. A *timed cycle* corresponds to the loop: *evaluation*, *update*, *delta notification* and *time notification*. For concision we didn't represent the other sets that are involved in the scheduling algorithm.

B. Temporal Decoupling

In this subsection we remind what temporal decoupling is and we present some implementation aspects in ST. Before the standardization of TLM, temporal decoupling was already used in abstract SystemC models. It consists in defining a local time for each process. A process can increase its local time and then get ahead of the SystemC timestamp. This is called a *low cost timing annotation* because it only operates on a local variable and induces no SystemC operation. Then to keep time consistency, a *local time purge* is defined. To purge the local time means to perform a SystemC `wait` with the local time value, and then reset it. Two purge systems can be used:

1) Implicit, using a time quantum. If a local time counter gets ahead of the time quantum, then the process purges its local time.
2) Explicit, using a synchronization method. When this method is called during the execution of a process, the local time counter is purged.

Since the TLM-2 standard [13], temporal decoupling has been introduced in the TLM API. For the development of LT models, ST has made the choice to use temporal decoupling with explicit synchronizations, combined with time ranges, which are a typical LT construction. The pseudo-code of the *low cost timing annotation* and the *local time purge* respectively is as follows:

```
void annotate_LT(sc_time min, sc_time max) {
    local_min += min;
    local_max += max;
}
void synchronize() {
    wait(RAND_BETWEEN(local_min, local_max));
    local_min = local_max = 0;
}
```

This `synchronize` method (used at ST) picks a random value within the min and max local times. We will see in Section V that time ranges could be better used, but evenso the benefits for our test case are barely perceptible, which illustrates the claim we made in Section I.

C. Problematics

The parallelization of LT SystemC/TLM simulations implies to solve several challenges. We do not present classic problematics inherent to software parallelization, but focus on issues that are induced specifically by such models in an industrial context.

A SystemC parallelization solution must not introduce race conditions. In TLM/LT models, communications are made by transactions done through ports, exports and/or sockets. This causes intermodule function calls. For example, two initiators (e.g. CPUs) that concurrently access the same target (e.g. a RAM) will concurrently call the same function of this target. That makes the target component itself a shared resource, which introduces a race condition (if not protected).

In the industry, simulations are often heterogeneous. That means that parts of the models can be designed for high level synthesis (HLS) or at RTL level. A subset of the model can even be a real hardware component. An industrially compliant parallelization solution must be able to deal with this heterogeneity.

An other huge challenge for SystemC parallelization is the adaptability on existing platforms. Indeed, as for every technology change, the migration has a cost. This cost must be put in perspective with the time saved if the parallelization solution was in production: a solution that requires an important effort may not be profitable even if it shows substantial performance benefits. This is also weighted by the lifecycle of the platform. In ST the platforms have a short lifespan (6-8 months in average) and are designed using components taken from ST libraries. In other industrial contexts the platforms have a longer life, that can reach several years, and then more time can be spent on the design.

III. Related Work

A. Parallelization Inside Delta Cycles

A conservative PDES with lookahead has been implemented in [14]. The solution works on top of the SystemC kernel, which remains unchanged. However, to apply this solution, the model must be written in compliance with some provided templates. It explicitly targets RTL-like platforms, because the solution is based on low-level SystemC features like `sc_signal`, which tends to be unused in TLM models. This proposal comes with a lot of future work propositions, and is not industrially applicable as-is. Some of the proposed ideas are addressed by more recent papers.

The works presented in [5], [2], [3] use the fact that within a delta cycle, the execution order of the processes is undefined. This assertion is exploited to concurrently run the runnable processes of the same delta cycle. A synchronization barrier is placed at the end of each delta cycle. The processes are partitionned into groups that will be executed by an instance of the SystemC simulator on a specific core. In [5] all the processes of a module have the same affinity. In [2] the authors have experimented different strategies to balance the load on the available cores. Those works have showed good speed-ups for specific cases. However we can notice that due to the synchronization barrier, all the SystemC simulator instances are always at the same delta cycle. It clearly appears that a necessary condition to have a significant speed-up is to have a sufficient number of concurrently runnable processes in every delta cycle. We will see that this is far from being the case when we consider LT models.

A high degree of parallelization can be achieved by exploiting GPUs. This has been studied by [4], [15], [16]. In [4], [16] the authors have chosen the CUDA programming model. The proposed tool in [4] is a source-to-source translator which produces a CUDA-style code from a RTL-synthesizable SystemC model. High speed-ups are achieved, however we can notice that all the test cases are pipelined platforms at RTL level. So there is independency of data between the pipeline stages and the computations are performed on clock edges (fixed and high number of runnable tasks in each cycle). In [15] the authors compared a CUDA and an OpenCL implementation. The work presented in [16] consists in exploiting both GPU and multi-CPU architectures. It supports mixed-abstraction simulations. The parallelization is performed within delta cycles, with support for immediately notified processes. They have benched their solution on a set-top box model, which was implemented in order to be used as a bench. This induces that some optimizations have been made, that may hide the real amout of refactoring needed to efficiently parallelize an existing industrial platform.

In [17] is presented the *RAVES* hardware platform. Each evaluation phase is parallelized (i.e. even after immediate notifications). Though, the new aspect here is that the authors have implemented a hardware architecture which implements their parallel SystemC kernel, and the actual embedded software running on it is the platform model. The results are promising, but again their test cases mostly model cycle-accurate MPSoC.

B. Dependency Analysis

The SystemC standard states that within a delta cycle, the execution order of the processes is undefined. However that doesn't mean that they can be run concurrently. Indeed, one must consider the case of shared memory and/or resources. Most of the work presented in the previous section have made strong hypothesis about the models, that avoids race conditions. On existing platforms, it happens that most of the time these hypothesis are not fulfilled. To prevent race conditions, some researchers have chosen to look into static dependency analysis.

In [18] a parallel SystemC simulator with static analysis of dependencies is presented. A static analysis tool first scans the model in order to produce a dependency scheme. Then this dependency scheme is provided as input to the modified SystemC simulator, which uses it to schedule in parallel different processes. The main idea is that a runnable process can be run if it is independant with each running process. As we presented in Subsection II-C the case where mutiple components perform transactions to the same target is equivalent to a shared resource access. However the static analysis as presented here doesn't include the resolving of transaction addresses.

In [19] the authors also present a simulator which performs static analysis to prevent data, timing or event conflicts. The technique is similar to branch prediction in hardware. The parallelization is performed within delta cycles.

The solution presented in [20] uses both static and dynamic analysis to propose an adaptative algorithm and tool flow for SystemC/RTL parallelization. Their current solution is based on RTL features, e.g. `sc_signal`, however they used sufficient abstractions in their algorithm to keep hope for future TLM support (planned as a future work by the authors).

C. Distributed Time/Relaxing Synchronizations

In [6] a programmation paradigm for many-cores and many-clusters modeling in SystemC is presented. The work is based on the PDES principle and proposes a conservative synchronization with a lookahead time. Multiple instances of a SystemC simulator are run. To avoid deadlocks, the sending of null messages (which only contains a timestamp) is introduced. The handling of interrupts is proposed by the addition of a timestamp to it, with a polling in the beginning of each CPU loop. This removes the asynchronous characteristic of interruptions, but guarantees that they will be noticed in a meaningful time. Their test bench consists in comparing a cycle accurate bit accurate (CABA) simulation to a TLM one, in order to balance the time saving with the loss of precision. The conclusion of their work is that for a very acceptable loss of precision, they can simulate models at TLM speed, so approximately 50 times faster than for CABA simulations. However we can identify two major drawbacks in their bench. The first one is that the software task is very basic: wait for an interrupt and then display a message on the terminal, in a loop. The second one is that the best speedup is reached for the platform with the biggest number of simulated cores (i.e. 39). On our industrial platform, the number of simulated cores is low (no more than 7) and obviously the software is more complex. The next work completes this one by proposing an implementation of this simulator: TLM-DT (Distributed Time).

TLM-DT is presented in [7], [21]. It is compliant with the TLM2 standard, however it needs to shift from a global time to a distributed time, which induces to modify all the timing information in the model. This solution reaches good performances on MPSoC and NoC platforms. However, those platforms are composed of many instances of similar if not identical components. This regularity in the architecture induces a different profile from platforms with lots of hardware Intellectual Properties (IPs).

Based on TLM-DT, the authors of [22], [8] have developped their own parallel implementation of a SystemC/TLM simulator. It is based on a PDES algorithm and designed for clustered platforms. Both implicit and explicit synchronizations have been implemented to bound temporal errors. Their analysis is common to SystemC and SpecC languages. They particularly focus on protecting the shared parts of the simulation, i.e. the simulation kernel and the communications, and have included locks in their scheduling algorithm. They present promising results on hardware-parallelized versions of video decoding and image encoding models. A drawback of such an approach is that if there is no parallelization in the description of the critical part of the platform, e.g. here it is the video/image decoding/encoding, the given algorithm won't achieve a significant speed-up.

An optimistic approach of PDES has been studied in [9]. The optimistic characteristic does not come from a rollback mechanism (contrary to what *optimistic* means in other research papers) but from a weak synchronization mechanism. The platform is divided into groups of modules, which are simulated by a specific instance of a SystemC simulator. Synchronizations are performed when a time quantum has been overlapped. It is also possible to define transaction-specific time behaviour. This approach makes strong hypothesis about shared variables: it considers that shared variables (except the ones from the SystemC kernel) has been either well protected or purposely not protected.

In [10] is proposed a parallel SystemC simulator implementation called *SCope*. They run multiple instances of a sequential SystemC kernel, each one on a different worker thread. Each instance has access to some objects of the simulation (i.e. modules, ports, threads, etc). Simulator instances are allowed to run at different simulation times: a lookahead time is defined. Their results showed a good speed-up on a four-cores host, for the model of a specific hardware structure called *EURETILE* which is similar to a network-on-chip.

D. Tasks with Duration

In [23] the authors present jTLM. This is a Java experimentation framework for TLM simulation. The interest is that it can exploit multi-core architectures, by introducing a different timing approach: tasks with duration, as alternative to instantaneous computations followed by a time elapse. This notion has been extended in a SystemC framework called `sc_during` [12]. It consists in defining some parts of the simulation to be independant with the rest of the platform during a specified amount of time. This new information is then used to run tasks from different modules in parallel with each other. This solution needs some refactoring in the code of the platform to actually parallelize it, but it has the advantage to let legacy SystemC code running as-is, sequentially. This solution explicitly targets LT models; the notion of tasks with duration is not meaningful for clock sensitive processes.

E. Other Works

The authors of [24], [25] present a parallel TLM simulation kernel not based on a SystemC layer. That removes the modeling level genericity offered by SystemC, but allows fine optimizations. The authors have implemented a dynamic load balancing strategy to compute the number of simulator instances to run, and the amount of tasks to give to each one.

In [26] the authors present an analysis of the execution semantics of both SystemC and SpecC. They have implemented an extension to the SpecC simulation kernel to support multicore simulation. However, the semantics of SystemC and SpecC are different, notably in their definition of signals. In SpecC, signals are defined as monitors: each of their accessors/modificators are in mutual exclusion.

A parallel implementation of SystemCASS, a cycle accurate version of SystemC, is presented in [27]. The authors have implemented a SystemCASS simulator that executes all the transitions (Mealy or Moore) of the same cycle in parallel (it can be seen as an equivalent to parallelization inside delta cycle techniques).

IV. Measures

One important issue when one tries to parallelize software is to find which approach will be the most efficient, depending on the profile of the application. We have an industrial platform at our disposal. It models a set-top box including video and audio encoding and decoding. The software running on it is a modified Linux kernel. In this section, we will describe some characteristics of this platform, in order to provide an example of an industrial LT platform.

We have modified the ASI implementation of the SystemC kernel to add track generation, in order to get interesting measurements. The following parts will present the results on the industrial platform we had at our disposal.

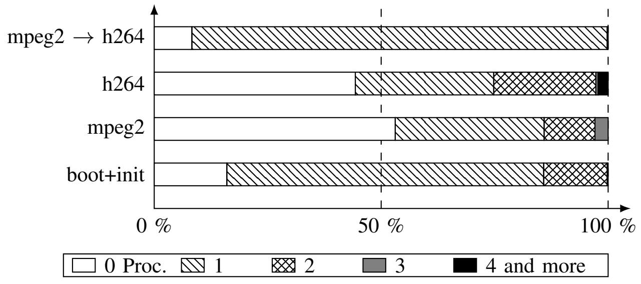
A. Overview

This platform is composed of $\approx 900,000$ lines of code including $\approx 750,000$ lines of C++ code (given by `cloc`). It contains 850 modules hierarchically organized. Counting only the leaf modules leads to the number of 750 modules. There are 1068 registered `SC_THREAD` and 163 `SC_METHOD`.

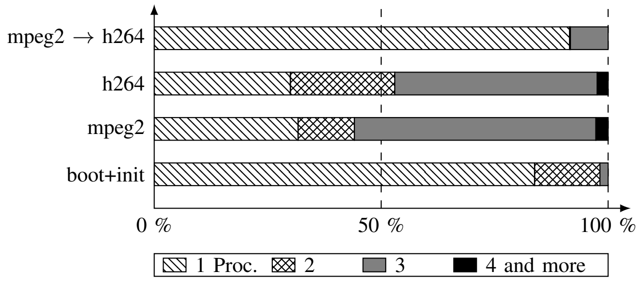
B. Number of Runnable Processes

To measure the theoretical efficiency of *parallelization inside delta cycles* (e.g. [14], [5], [2], [3], [4], [17]) we measure the number of runnable processes at the beginning of each delta cycle. This measurement gives an upper bound of the degree of parallelization achievable with such techniques. Due to shared resources, the real degree of parallelization may be fewer than this upper bound. Indeed a runnable process may share a dependency with an other one, making the concurrent run of those two processes not consistent with coroutine semantics.

The measurement consists in counting the number of runnable processes by adding a count in the SystemC kernel in each delta cycle. Figure 2 shows that most of the time, the number of runnable processes is low, almost always less than or equal to 3. This result indicates that the approaches which need the hypothesis of a great amount of runnable processes in each delta cycle cannot be efficient on this model. Note that the case where zero `SC_THREAD` are runnable represents the case of `SC_METHOD`-only cycles.



(a) `SC_THREAD` only



(b) `SC_THREAD` and `SC_METHOD`

Figure 2. Partitionning of delta cycles, by number of runnable processes, for four different test cases.

C. Wall-Clock Duration

We are also interested in identifying the most wall-clock time consuming processes, to highlight the critical parts of the simulation. Our goal being to parallelize the simulation, this helps to have correct hypothesis about the profile of process executions in the SystemC simulator. In order to do this, we measure the wall-clock time elapsed during the execution of each SystemC process. To perform this measurement, we use the `clock_gettime()` function provided by the `<ctime>` header and the Linux library `librt`. We use it to only get the real computation time, which does not include the sleeping periods of the OS process. Figure 3 presents the repartition of wall-clock time between different categories:

- processes from simulated CPU cores (Cores)
- processes from simulated hardware modules (IPs)
- SystemC kernel (Kernel)

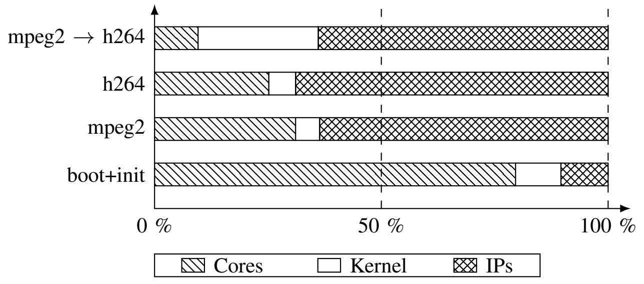


Figure 3. Partitionning of the wall-clock time elapsed, by category of processes, for four different test cases.

This is a coarse-grain view of the wall clock time partition. From these results, we deduce that for this platform, most of the wall clock time is elapsed in IP modules, describing hardware components. We can also notice that the time elapsed in the SystemC kernel is far from being negligible. This is mainly due to a high rate of immediate event notifications, especially in video encoding and/or decoding cases which involves a lot of computations in hardware IP modules. The fact that most of the computation time is spent in IP models instead of cores brings its set of problems. Indeed, the behaviour of CPU models is often more predictible because the CPU have a more systematic behaviour.

An other interesting result is that $\approx 50\%$ of the computation time is spent in only 4 or 5 processes depending on the case. For the *boot+init* and the transcoding (*mpeg2 → h264*) cases, it is even $\approx 80\%$. The rest of the computation time consists in small amounts in different processes.

V. Discussion

A. Parallelization Inside Delta Cycles

As we have seen in Section IV, the number of runnable processes at the beginning of each delta cycle is too low to expect an interesting speed-up with *parallelization inside delta cycles* for such platforms. In Subsection II-B, we have presented the LT time decoupling mechanisms used at ST. We have thought of an optimization in the time picking, that could increase the number of runnable processes in each delta cycle. Indeed, when a synchronization point is reached, the current ST implementation picks a random value within the time range, and sends it to the SystemC kernel with `wait`. So the kernel doesn't have the range bounds: it can't exploit the range information.

One can think that if the SystemC kernel could exploit this information, we could optimize the number of runnable processes at each cycle. So we have implemented such a strategy, and we will see that the improvements are quite low. To exploit ranged waits, we have added this function to the SystemC kernel:

```
void wait(sc_time min, sc_time max);
```

This function has the following semantics: *yield to the kernel, and wake up the current thread on a time included in the given range*. To illustrate how we choose the best value in order to maximize the number of runnable processes in the next delta cycle, we present an example. Let us consider the following code in two distinct SC_THREAD that we consider independant with each other:

```
void mod1::compute() {
    // ...
    wait(sc_time(1, SC_NS), sc_time(5, SC_NS));
    // ...
}
void mod2::compute() {
    // ...
    wait(sc_time(3, SC_NS), sc_time(7, SC_NS));
    // ...
}
```

A graphical representation of executions of this model is shown on Figure 4. We can see the processes (on the left side) and rectangles (on the middle part) which indicates the time interval in which the execution of the corresponding process is valid. The times actually simulated by the simulator are represented by dashes.

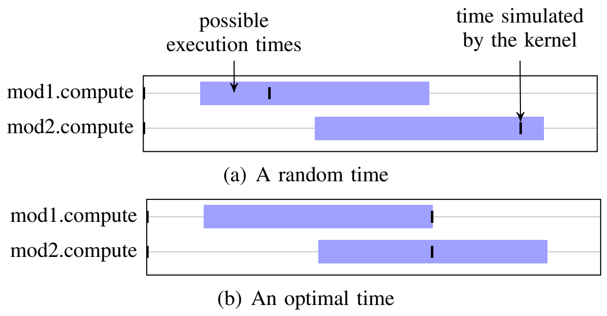


Figure 4. Execution diagrams of two processes using wait with a time range, for two different time choice policies: a random value (a) and a value which maximizes the number of eligible processes (b) in the next cycle.

With this representation, it clearly appears that depending on the times that are simulated, we have a different number of runnable processes on a different number of cycles:

- In the case of Figure 4(a), there are two simulated times, each with one runnable process.
- In the case of Figure 4(b), there is only one simulated time, with two runnable processes. Thus this case induces a higher degree of parallelization for an approach *inside delta cycles*.

To implement such a strategy, we had to bind the ST annotate/synchronize API (see Section II) with this newly added wait. The computation of the simulated time is no longer performed in the synchronize method, but is dedicated to the modified SystemC kernel. Finally, the last part is to implement a policy to choose the next time to simulate, in the kernel. To choose the next simulated time value, we choose the minimum value of all the upper bounds, for all the registered time events. By chosing the minimum value of all the upper bounds, we guarantee that the next time is not too late (i.e. that we don't skip any registered timed event). Moreover, it is the farthest valid time, so it will trigger the biggest number of processes, for the next time cycle. Then each timed event whose range includes this value is popped and triggered. Note that with this strategy, we are still compatible with the old wait by defining a time range with both bounds equal to the same time value.

By implementing this policy, we increase the number of runnable processes at the beginning of each cycle for an execution. The final purpose is to determine if this increased value is high enough to justify the use of *parallelization inside delta cycles* techniques. We have run again the simulations of our industrial platform for the same test cases to measure the same metrics with this improvement. In Figure 5, we see that in most cases, the number of processes is still very low, even if in we have more occurences of cycles with *3* and *4 or more* runnable processes.

Those results illustrate our point: *parallelization inside delta cycles* will not efficiently parallelize the simulation of such models, even after performing a non-trivial optimization. Indeed, the maximum speed-up achievable (if we exclude all the race condition problematics) is no more than 4.

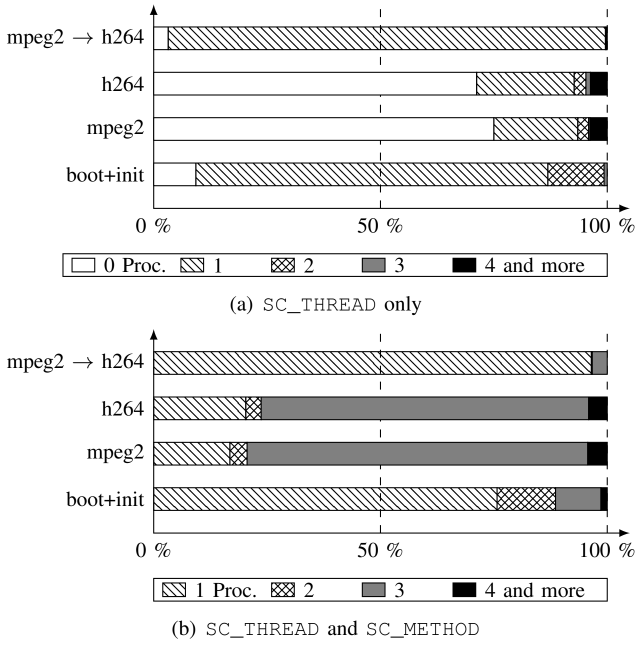


Figure 5. Partitionning of delta cycles, by number of runnable processes, for four different test cases, with optimized time picking within time ranges.

B. Usability on Existing Platforms

We have seen that *parallelization inside delta cycles* techniques are hardly efficient on our TLM/LT model. Then we can ask the question about the main other existing approaches.

A solution working on existing platforms with no refactoring will be easily accepted in the industry, where code refactoring of huge models is not conceivable. We can take the example of TLM-DT. To apply this solution, one must first adapt the platform code to fit TLM-DT API. Concerning TLM-DT, we can also point out that it targets MPSoC or NoC models. This provides the hypothesis that most of the simulation time is spent on the numerous similar CPU/core models. However, we have seen that most of the time is spent on IP modules.

In the case of sc_during the amount of refactoring needed is debatable. Indeed, this approach works on top of the SystemC kernel, so existing platforms will continue to work just like before the addition of the library, with no optimizations. To introduce parallelization in a model, one must specify tasks with duration. That makes this solution relevant for platforms where wall-clock time consuming parts can be clearly identified, and are present in sufficient number to justify the parallelization.

VI. Conclusion

Research on the parallelization of SystemC simulations has already produced different tools, and many of them allow important performance and scalability improvements. However, in this paper we showed that the case of non-MPSoC LT SystemC models with many hardware IPs raises a lot of different challenges that are not addressed in previous work. LT models are the only option for very fast simulation at an early stage of the design flow: they provide very good sequential performance by abstracting details that would slow down the simulation, require lightweight modeling effort and do not require information about detailed microarchitecture that are not yet available at this stage.

We showed that LT models exhibit characteristics that prevent most parallelization approaches from working. Approaches that run processes in parallel within delta cycles cannot get a speed-up greater than the number of processes runnable in this cycle, which hardly reaches 4 as a maximum in our case. Also, many approaches consider that the SystemC processes must not share variables, which is not true for TLM/LT because of the communication through function calls. As a consequence, most existing approaches are fundamentally limited when it comes to LT models.

We believe that this paper provides a better understanding of the problem, and by providing some measurements on an industrial platform, we even quantified the issue. We hope that these experiments will help new approaches to emerge and to be experimented on representative benchmarks. In future works `sc_during` can be tested on such industrial platforms to evaluate the benefits and balance it with the amount of refactoring. Temporal decoupling can also be exploited better to reduce the synchronizations between components.

References

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

[2] P. Ezudheen, P. Chandran, J. Chandra, B. Simon, and D. Ravi, "Parallelizing SystemC Kernel for Fast Hardware Simulation on SMP Machines," in *Principles of Advanced and Distributed Simulation, PADS ACM/IEEE/SCS 23rd Workshop on*, 2009, pp. 80–87.

[3] C. Schumacher, R. Leupers, D. Petras, and A. Hoffmann, "parSC: Synchronous Parallel SystemC Simulation on Multi-core Host Architectures," in *Hardware/Software Codesign and System Synthesis, IEEE/ACM/IFIP International Conference on*, 2010, pp. 241–246.

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

[5] B. Chopard, P. Combes, and J. Zory, "A Conservative Approach to SystemC Parallelization," in *Computational Science, ICCS*, 2006, vol. 3994, pp. 653–660.

[6] E. Viaud, F. Pêcheux, and A. Greiner, "An Efficient TLM/T Modeling and Simulation Environment Based on Conservative Parallel Discrete Event Principles," in *Design, Automation and Test in Europe (DATE)*, vol. 1, 2006, pp. 1–6.

[7] A. Vieira De Mello, I. Maia Pessoa, A. Greiner, and F. Pêcheux, "Parallel Simulation of SystemC TLM 2.0 Compliant MPSoC on SMP Workstations," in *Design, Automation and Test in Europe (DATE)*, 2010, pp. 606–609.

[8] R. Dömer, W. Chen, and X. Han, "Parallel Discrete Event Simulation of Transaction Level Models," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2012, pp. 227–231.

[9] S. Jones, "Optimistic Parallelisation of SystemC," Master's thesis, 2011.

[10] J. H. Weinstock, C. Schumacher, R. Leupers, G. Ascheid, and L. Tosoratto, "Time-Decoupled Parallel SystemC Simulation," in *Design, Automation and Test in Europe (DATE)*, 2014, pp. 191:1–191:4.

[11] P. Combes, E. Caron, F. Desprez, B. Chopard, and J. Zory, "Relaxing Synchronization in a Parallel SystemC Kernel," in *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, 2008.

[12] M. Moy, "Parallel Programming with SystemC for Loosely Timed Models: A Non-Intrusive Approach," in *Design, Automation and Test in Europe (DATE)*, 2013.

[13] *OSCI TLM-2.0 Language Reference Manual*, Std., 2009.

[14] M. Trams, "Conservative Distributed Discrete Event Simulation with SystemC using Explicit Lookahead," *Digital Force White Paper*, 2004.

[15] N. Bombieri, S. Vinco, V. Bertacco, and D. Chatterjee, "SystemC Simulation on GP-GPUs: CUDA vs OpenCL," in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2012.

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

[17] N. Ventroux, J. Peeters, T. Sassolas, and J. Hoe, "Highly-Parallel Special-Purpose Multicore Architecture for SystemC/TLM Simulations," in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), International Conference on*, 2014, pp. 250–257.

[18] Y. Bouzouzou, "Accélération des Simulations de Systèmes sur Puce au Niveau Transactionnel," Master's thesis, 2007.

[19] W. Chen and R. Dömer, "Optimized Out-of-order Parallel Discrete Event Simulation Using Predictions," in *Design, Automation and Test in Europe (DATE)*, 2013, pp. 3–8.

[20] S. Reder, C. Roth, H. Bucher, O. Sander, and J. Becker, "Adaptive Algorithm and Tool Flow for Accelerating SystemC on Many-Core Architectures," *Microprocessors and Microsystems*, pp. –, 2015.

[21] I. Maia Pessoa, A. Vieira De Mello, F. Pêcheux, and A. Greiner, "Parallel TLM Simulation of MPSoC on SMP Workstations: Influence of Communication Locality," in *International Conference on Microelectronics (ICM)*, 2010, pp. 359–362.

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

[23] G. Funchal and M. Moy, "jTLM: an Experimentation Framework for the Simulation of Transaction-Level Models of Systems-on-Chip," in *Design, Automation and Test in Europe (DATE)*, 2011.

[24] R. Khaligh and M. Radetzki, "Modeling Constructs and Kernel for Parallel Simulation of Accuracy Adaptive TLMs," in *Design, Automation and Test in Europe (DATE)*, 2010, pp. 1183–1188.

[25] ——, "A Dynamic Load Balancing Method for Parallel Simulation of Accuracy Adaptive TLMs," in *Specification Design Languages (FDL), Forum on*, 2010, pp. 1–6.

[26] R. Dömer, W. Chen, X. Han, and A. Gerstlauer, "Multi-Core Parallel Simulation of System-Level Description Languages," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2011.

[27] L. Ainey, A. Efrati, and S. Weiss, "Parallel Cycle-Accurate SystemC Kernel," in *Electrical Electronics Engineers in Israel (IEEEI), IEEE 28th Convention of*, 2014, pp. 1–5.