Parallel Programming with SystemC for Loosely Timed Models: A Non-Intrusive Approach

Matthieu Moy

Parallel Programming with SystemC for Loosely Timed Models: A Non-Intrusive Approach

Matthieu Moy
Grenoble INP, Verimag (UMR CNRS 5104),
Grenoble, F-38041, France

***Abstract*—The SystemC/TLM technologies are widely accepted in the industry for fast system-level simulation. An important limitation of SystemC regarding performance is that the reference implementation is sequential, and the official semantics makes parallel executions difficult. As the number of cores in computers increase quickly, the ability to take advantage of the host parallelism during a simulation is becoming a major concern. Most existing work on parallelization of SystemC targets cycle-accurate simulation, and would be inefficient on loosely timed systems since they cannot run in parallel processes that do not execute simultaneously.**

We propose an approach that explicitly targets loosely timed systems, and offers the user a set of primitives to express tasks with *duration*, as opposed to the notion of time in SystemC which allows only instantaneous computations and time elapses without computation. Our tool exploits this notion of duration to run the simulation in parallel. It runs on top of any (unmodified) SystemC implementation, which lets legacy SystemC code continue running as-it-is. This allows the user to focus on the performance-critical parts of the program that need to be parallelized.

I. Introduction

Systems-on-a-Chip are complex systems involving dedicated hardware components and one or several processors. Because of time to market pressure, it is not possible to wait for the physical chip to develop the software, hence the developers need simulators, or *virtual prototypes*, earlier in the design flow. These prototypes need to be fast enough to make it possible to run complex applications, and must expose enough details of the hardware to allow non-portable software to run. The RTL level of abstraction is detailed enough, but far too slow, and available too late in the design cycle, hence a new level of abstraction called *Transaction Level Modeling* (TLM) has been introduced. We are interested in the SystemC [1] implementation of TLM.

There are several sub-levels of abstraction within TLM depending on the target usage: if the virtual prototype is meant for performance evaluation, it needs to be precise with respect to timing (either cycle-accurate or approximately timed). On the other hand, if only the functional correctness is important, timing can be relaxed considerably. TLM allows a purely untimed coding style, as well as a *loosely timed* (LT) coding style where the behavior is timed only if the timing is needed for the functionality. For example, a loosely timed model of a timer will usually be timed properly, but most computations can be considered instantaneous.

This paper has been partially supported by the French ANR project HELP (ANR-09-SEGI-006).

Although SystemC represents concurrent systems, it should be noted that timing and concurrency in SystemC are *simulated*, and the actual execution is usually sequential (The SystemC standard imposes co-routine semantics). Running the simulation in parallel is needed to take advantage of today's machines, and SystemC's sequentiality could become a major performance bottleneck in the future. Many approaches have been proposed for SystemC parallelization, but most of them are only able to run processes that run at the same simulation cycle, and hence apply only on cycle-accurate systems.

As opposed to this, the parallelization approach we propose targets loosely timed systems. We present a programming model in which processes can not only execute instantaneous tasks or wait statements, but also express the notion of duration, i.e. specify the start and end date of the computation and let the scheduler freely spread it across time. These tasks with duration are executed in parallel with the SystemC scheduler.

This approach is implemented in a library, that runs on top of an unmodified SystemC kernel (including proprietary ones that cannot be modified). Also, this means that legacy code will continue running with our approach, although they will not *automatically* benefit from parallelism. The user can identify performance-critical parts of the system, and focus on them to parallelize the program (adding synchronization as required).

The paper is structured as follows: Section II discusses related works. Section III presents the main contribution: the `during` tasks principle and implementation, for which Section IV give experimental results. Section V is the conclusion.

II. Related Work

As the official semantics for SystemC is to use co-routine semantics, the natural way to execute a SystemC program is to execute it sequentially. To take advantage of the parallelism of the host machine, it is tempting to run SystemC threads in parallel. The SystemC standard allows this, "provided that the behavior appears identical to the co-routine semantics" [1]. This implies two constraints on a parallel implementation:

1) It should not change the order in which processes are allowed to be executed. In particular, the simulated time imposes an order on the execution of processes.
2) It should not introduce new race conditions. For example, two SystemC processes may safely execute `x++` on a shared variable, but running two such processes in parallel cannot be allowed.

Approaches for parallel execution of SystemC programs can be classified depending on the way they deal with these constraints: conservative ensure them by construction, while optimistic approaches relax them, ideally with a verification and rollback mechanism to correct them after the fact.

A. Semantics-Preserving Parallelization

An ideal conservative approach would preserve the official semantics completely. To solve problem 1 above, one can introduce parallelism within a δ-cycle (i.e. within one iteration inside a simulation instant), but keep a global synchronization barrier between each δ-cycle. Since the order of execution within an evaluation phase is unspecified in SystemC, it is correct to run processes in any order.

To solve problem 2, one needs to ensure that two processes accessing the same variable will never be run in parallel. This can be done structurally, under the assumption that a process will only access variables declared within its own module. It is then sufficient to ensure that at most one process is running within each module. SystemCASS [2] is an example tool using this assumption to perform a dependency analysis (for static scheduling). Unfortunately, communication using TLM break this assumption: TLM communication channels transport transaction through function calls, hence the code processing a transaction in a target module is executed in the context of the initiator module. In particular, the natural implementation of a RAM makes it a shared array between all components accessing it.

Another option is to statically analyze the code of the processes to detect accesses to shared variables. [3] proposes an approach to parallelization of SystemC based on a dependency analysis performed on a SystemC-transition (i.e. piece of code executed between two `wait` statements) basis. [4] applies similar static analysis for SpecC. Scoot [5] proposes a finer analysis using model-checking (applied to scheduling and model-checking, not to parallelization). These approaches are heavyweight since they require a static analysis of arbitrary C++ code, and require the complete source code of the model to be available. The efficiency of these approaches is disappointing on TLM models for which a few target components create a synchronization bottleneck (back to our RAM example, two processes that may access the RAM never be executed in parallel).

B. Parallelism within δ-cycles

A second category of approaches is to be conservative with respect to problem 1, but to leave it up to the user to deal with problem 2, by avoiding shared variables, and using proper synchronization and locking when needed.

[6] actually distributes the simulation over several machines, running one SystemC scheduler on each node. The first version enforced a global synchronization at each δ-cycle, and further optimizations [7] allowed relaxing this synchronization by computing the next simulated time locally on each nodes.

Another approach is to modify the SystemC kernel to allow a parallel simulation [8]. The scheduler maintains a set of runnable processes, and several threads run an evaluation phase that pick processes from this set and run them, finishing the evaluation phase with a global synchronization barrier. This work is currently focused on performance, but leaves the responsibility of protecting the atomicity of accesses to shared data-structure to the SystemC processes. Similar work propose some performance optimizations using various load-balancing techniques like work-stealing, work-sharing or manual grouping [9], [10] to improve the cache usage. To avoid having to use a modified SystemC kernel, [11] allows running a distributed simulation on top of SystemC. It runs the δ-cycles one by one, calling `sc_start(SC_ZERO_TIME)` repeatedly in the main loop of the distributed engine.

C. Parallelism Across Simulated Instants

All the works presented above have in common the fact that they parallelize the content of δ-cycle, but cannot run in parallel processes that should be executed at different simulated times. In other words, they require a simultaneity between processes to take advantage of parallelism. This is appropriate for cycle-accurate simulation, where clock ticks usually wakes up many processes at the same time. But higher levels of abstraction tend to eliminate clock, and deal with quantitative time instead. Figure 1 shows an example modeling a system performing two computations in parallel. Computations that take some time are usually modeled by instantaneous computations followed by a SystemC wait. For example, TLM-2's temporal decoupling [1] produces this effect (the call to `wait` is done when synchronizing a process's local clock with the simulation time). In simulation, this program will run without ever having two processes runnable at the same time. In the presence of loose timing annotations (e.g. `wait(X±50%)`), the situation is even worse since the argument to `wait` becomes a random value.

```
void P::compute() {
  P1(); wait(25, SC_MS);
  P2(); wait(12, SC_MS);
}
```

```
void Q::compute() {
  wait(10, SC_MS);
  Q1(); wait(13, SC_MS);
  Q2(); wait(11, SC_MS);
}
```

Fig. 1. Parallelism Without Simultaneity

For such systems, semantics-preserving approaches are even harder. The scheme proposed in [4] performs a static analysis for processes that may be executed at different instants, but in SystemC/TLM programs with a shared bus, target components will still create conflicts in many cases, and this approach requires the availability of the whole source code. The alternative is to use optimistic approaches, i.e. one processes should be allowed to advance its local time without waiting for other computations to terminate. Maintaining the same execution order as the reference semantics becomes intractable, hence optimistic simulation either (a) requires a violation-detection together with a rollback mechanism or (b) deliberately accepts some order violations, even though it can control them with some logical clock mechanism in addition to the quantitative simulated time. Approaches in category

(a) have been successfully applied to VHDL simulations, but the case of SystemC is much harder to deal with: since the user can call arbitrary C++ code and system calls, a rollback mechanism is not implementable in general.

One solution in category (b) is proposed in [12], introducing the TLM with *Distributed Time* (TLM-DT) abstraction level and the associated parallel simulator. In TLM-DT, processes do not rely on the SystemC simulated time, but manage a local clock, that is synchronized with other components regularly (on every communication, and using null messages when a predefined time quantum elapsed without actual communication). This way, ordering violations can happen only with processes running with a difference of at most the time quantum. This approach does not need simultaneity for parallelization, and relaxes the synchronization barrier, but still enforces it every time quantum (through null messages). It requires a particular modeling style and forbids some SystemC primitives, hence is not applicable on legacy code. It is designed to be efficient on approximately timed model, but not on loosely timed ones.

Another approach relying on the notion of time quantum is presented in [13]. It performs a distributed simulation similarly to [6], but instead of performing a synchronization barrier for every δ-cycle, the authors propose to force processes to execute only on a multiple of the quantum.

Tasks and Duration: An experiment of transaction-level modeling outside the SystemC world is jTLM [14]. The management of simulated time is different in that jTLM provides a notion of "task with duration" [15]. Instead of modeling such task as an instantaneous computation followed by a wait, the user can specify the duration of the task, and let the scheduler spread the computation over the specified period of time. The example of Figure 1 becomes the one in Figure 2.

```
void P_compute() {
  consumesTime(25){P1();}
  consumesTime(12){P2();}
}
```

```
void Q_compute() {
  awaitTime(10);
  consumesTime(13){Q1();}
  consumesTime(11){Q2();}
}
```

Fig. 2. Parallelism With Duration in jTLM

This way, tasks `P1()` and `Q1()`, although not simultaneous, have an overlap, hence can run in parallel.

This notion of duration and overlap is interesting for parallel execution of models with a coarse timing granularity. The coarsest the task are, the less likely it is for their start and end points to be simultaneous, but the most likely it is for them to overlap. Unlike cycle-based approaches, the efficiency will therefore increase with the size of tasks. An obvious limitation of this approach is that jTLM is not compatible with SystemC, hence unable to use legacy SystemC code.

III. Tasks with Duration in SystemC

None of the approaches mentioned above can parallelize the execution of SystemC programs across instants safely and without changing the code. It does not seem realistic to target fully automatic parallelization for real-life TLM programs. Instead, our contribution is a new programming model that complements the one of SystemC and allows the programmer to solve the parallelization problem by using the new primitives on performance-critical parts of the program. The semantics is inspired from the one of jTLM, but the implementation is made as a library called `sc-during` (while jTLM implements tasks with a dedicated scheduler). It runs on top of SystemC, without modifying it. The library can be freely downloaded from http://sc-during.forge.imag.fr/.

A. Principle of Tasks with Duration

An example code using `sc-during` is shown in Figure 3. To use the API, a module derives from the class `sc_during`. The main primitive is the method `during(`d`, `f`)` that takes a duration d and a function f to be executed as argument. This function can only be called from an `SC_THREAD` (i.e. not from an `SC_METHOD`), as it calls `wait` internally. The function f is passed using `boost::function` and can be either a pointer to function, or a method using `boost::bind`. f may not interact directly with SystemC (i.e. call `wait` or `notify`), but section III-B1 will present a way to perform such interactions.

```
1  extern void P1(), P2(), Q1(), Q2();
2  struct P : sc_module, sc_during {
3    void compute() {
4      during(sc_time(25, SC_MS), P1);
5      during(sc_time(12, SC_MS), P2);
6    }
7    SC_CTOR(P) {SC_THREAD(compute);}
8  };
9  struct Q : sc_module, sc_during {
10   void compute() {
11     wait(10, SC_MS);
12     during(sc_time(13, SC_MS), Q1);
13     during(sc_time(11, SC_MS), Q2);
14   }
15   SC_CTOR(Q) {SC_THREAD(compute);}
16 };
17
18 int sc_main(int, char **) {
19   P p("p"); Q q("q"); sc_start(); return 1; }
```

Fig. 3. Parallelism With Duration in SystemC

The execution of this program is illustrated in Figure 4. It starts the normal way, using the SystemC scheduler and global simulated time. Once a call to `during` is encountered, the execution of the function passed as argument (e.g. `P1`, at $t = 0$ line 4) can be delegated to another operating system thread. The simulation will continue with other processes on the SystemC side, in parallel with the execution of `P1`. SystemC and `P1` will join at $t = 25$ ms, i.e. either the SystemC simulation reaches 25 and waits for `P1` to complete, or `P1` terminates at $t < 25$.

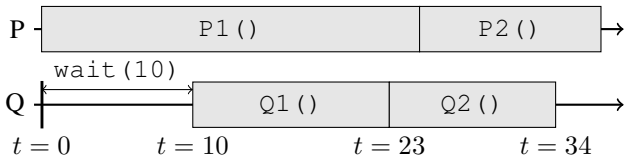


Fig. 4. Execution of a Program with During Tasks

While executing `P1` as a separate task, the SystemC scheduler continues its execution, executes `wait(10, SC_MS);` and reaches $t = 10$. Then, it encounters another call to `during` and launches `Q1` in a separate task, just like it did for `P1`. At this point, the simulation can contain 3 OS threads: one for each task, and one running the SystemC kernel. As `P1` and `Q1` have an overlap in time, they can be executed in any order, or with any interleaving. On the other hand, the end of task `Q1` predates the start of `P2`, so the scheduler ensures that `Q1` is completed before starting `Q2`.

It is of course possible to have instantaneous computations too. Any code within a SystemC process and outside a call to `during` will run instantaneously, following the SystemC semantics and independently from the `sc-during` library (another way to create instantaneous tasks is `during(SC_ZERO_TIME, f)` which creates a separate task that must complete before the end of the next δ-cycle).

The `sc-during` library contains 4 different implementation of the notion of task with duration. We first illustrate the idea with the one called THREAD, which is straightforward and unoptimized. Figure 5 shows a simplified version of this implementation (more implementation details are needed to introduce other primitives in section III-B).

```
1 void during(sc_core::sc_time duration,
2        boost::function<void()> routine) {
3   boost::thread t(routine); // create thread
4   sc_core::wait(time); // let SystemC execute
5   t.join(); // wait for thread completion
6 }
```

Fig. 5. Simplified Implementation of `during()`

B. Synchronization and Time Modeling

The ideas presented up to now allow running a piece of code in parallel with a SystemC simulation, but up to now, the task can hardly communicate with the SystemC thread. In particular, it cannot call SystemC primitives like `wait()` and `notify()`, and shared variables between the SystemC thread would create race-conditions. In order to provide a complete programming model to the user, we need to introduce additional synchronization primitives.

1) Full Synchronization with SystemC: `sc_call()`: A `during` task may want to execute a piece of code within the context of the SystemC thread. This way, this piece of code will be allowed to execute SystemC primitives (e.g. `wait`, `notify` ...) the normal way, and to access variables shared with SystemC without specific locking or race condition (like two SystemC processes access shared variables).

The `sc_call()` API function does this: it is callable from a `during` task and takes a function as argument. The current task is blocked, and the function is scheduled to be executed next time SystemC allows it. The `during` task is unlocked once the function has completed its execution.

We provide two implementations for the scheduling of the function to be executed: In SystemC 2.2, we simply record the function to be called, and call it when the call to `sc_core::wait` associated to the task terminates. In SystemC 2.3, we use `async_request_update()` to schedule the execution during the next δ-cycle.

2) Controlling Time Elapse and Duration: `extra_time()` and `catch_up()`: Up to now, the only way to specify the duration of a task is to provide it as argument to the `during()` method. This is not always possible, since it would require knowing how long the computation done within the task would take, before executing it.

To solve this issue, the `sc-during` library provides the function `extra_time(sc_time t)`, that can be called from a `during` task. This function increases the duration of the current task by t. In the implementation of the library, this means the `wait` statement line 4 in Fig 5 is actually inside a loop along the lines of `while (remaining_time) wait(remaining_time);`, and `extra_time` increments `remaining_time` with proper synchronization. An example is shown in Figure 6.

```
1  void trace(string s) {
2    cout << sc_time_stamp() << ": " << s << endl; }
3  SC_MODULE(A), sc_during {
4    void P() {
5      wait(5, SC_MS);
6      during(5, SC_MS, boost::bind(&A::f, this));
7      trace("done");
8    }
9    void f(void) {
10     trace("before extra_time");
11     extra_time(7, SC_MS);
12     trace("after extra_time");
13   };
14   SC_CTOR(A) { SC_THREAD(P); }
15 };
```

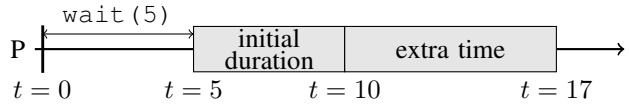


Fig. 6. Example Code Using `extra_time()`, and Execution Trace

This program will produce an output of the form (output is generated by the `trace` function defined line 1 which displays the current simulated time and a string):

X `ms: before extra_time`
Y `ms: after extra_time`
`17 ms: done`

The semantics ensures that the duration of the task is the initial duration plus the sum of times accumulated with `extra_time()`, hence the last line `17 ms: done`. Before the call to `extra_time`, the `during` task spans only over $[5, 10]$, hence $X \in [5, 10]$ is ensured. After calling `extra_time`, the task spans over $[5, 17]$, hence $Y \in [X, 17]$. In general, the semantics of `during` tasks with `extra_time` is that a piece of code within the task can be executed at a simulated time $t \in [s, s + d + e]$ where s is the starting time of the task, d is the initial duration, and e is the sum of the extra-time accumulated up to this point.

By default, time may flow freely during the execution of

a during task. This may be problematic if the program relies on some fairness between a during task and the SystemC simulation. Take the example of a polling loop like **while** (!x) {}. In SystemC, such a loop which doesn't let time elapse will create an infinite loop, and will have to be rewritten **while** (!x) {wait(t);} (for some time constant t). Within a during task, it is possible to achieve a similar effect with **while** (!x) {extra_time(t);}. Unfortunately, while this piece of code creates a during task with an unbounded duration, it does not *ensure* that the SystemC simulated time will actually increase. The catch_up function may be used as follows to ensure fairness:

```
1 while(!x) {
2   extra_time(10, SC_MS);
3   trace("between");
4   catch_up();
5   trace("after");
6 }
```

If this code is executed starting at $t = 0$, in a during task whose duration is initially 0, it will produce an output of the form:

X_1 ms: between
10 ms: after
X_2 ms: between
20 ms: after

with $X_1 \in [0, 10]$ and $X_2 \in [10, 20]$. The simulated time after returning from catch_up is fixed to the duration of the task at this point.

Alternatively, the user can use the relaxed form catch_up(t), providing as argument a time t and that only ensures that the simulated time will advance until the difference between the current time and the end of the task is at most t.

C. Temporal Decoupling and Duration

Temporal decoupling is a technique standardized by TLM-2 to reduce the number of wait statements executed. The idea is to allow processes to maintain a local lock t_{local} to model the fact that the process is "in advance" with respect to the SystemC time. Instead of using wait(d), processes increase t_{local} by d (function annotate in Figure 7), and periodically call a function to actually perform the wait() statement (function synchronize in Figure 7).

```
1 void annotate(sc_time d) {
2   t_local = t_local + d;
3 }
```

```
1 void synchronize() {
2   wait(t_local);
3   t_local = 0;
4 }
```

Fig. 7. Simple Temporal Decoupling API

```
1 void annotate(sc_time d) {
2   extra_time(d);
3 }
```

```
1 void synchronize() {
2   catch_up();
3 }
```

Fig. 8. Temporal Decoupling API within during Tasks

This synchronize function can be called whenever t_{local} becomes greater than a predefined time quantum (a call can be inserted in annotate for instance), or before synchronization points [16]. The sc-during library can benefit from temporal decoupling. The primitives of Figure 7 can be re-written as Figure 8 when used inside (possibly unbounded) during tasks. This preserves the property that important actions (i.e. the ones after calling synchronize) are executed at the right SystemC time, and the computations calling annotate are modeled with the right duration.

D. Implementation

We gave in section III-A a sketch of the THREAD implementation for during tasks. The actual implementation is more complex since it needs to manage sc_call, extra_time and catch_up. We also provide several implementation of parallelism: POOL pre-allocates a set of threads and works in a producer-consumer scheme, ONDEMAND creates new threads as needed and re-uses them instead of deleting and re-creating them, and SEQ is a reference sequential implementation.

IV. Experimental Results

A. Experimental Setup and Platform Synchronization

We experimented our approach on a SystemC model of an existing platform containing a CPU, a timer, a GPIO, a RAM and an interrupt controller (ITC), running on an FPGA, and executing an embedded software running Conway's Game of Life (the same C code is running on the FPGA and on the TLM platform). To illustrate parallelization of simulation, we added multi-processing capabilities to the system, adding N additional CPUs and ITCs. CPU0 delegates parts of the computation (slices of the image to compute) to other CPUs. It does so by setting a flag in RAM, and triggering an interrupt using the ITC associated to the CPU. Bus-communication is abstracted with a TLM-2 channel. Transactions (either read or write) are initiated by a component, routed through a Bus module towards a target module.

The most performance-consuming components are the CPUs, on which we are now going to focus. We use a simple ISS, but our approach would work equally well with any other SystemC module provided the approach has some notion of timing (possibly approximate). One ISS step takes a constant amount of time (PERIOD), hence we can basically count the number N of steps, and wait for N*PERIOD when synchronizing with SystemC time. We tried several strategies to choose when and how to synchronize:

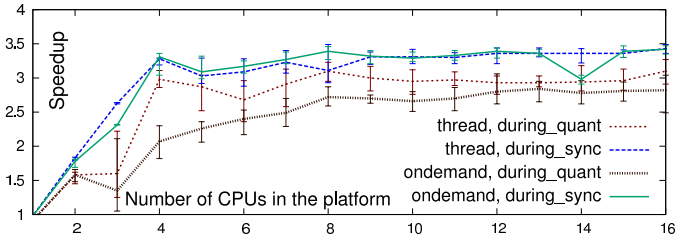
The first one is called during_quant and is directly inspired from the notion of quantum. "Packs" of 5000 instructions are executed in a during task of duration $d = 5000*$PERIOD. The second one, called during_sync, requires the user to explicitly set the synchronization points of the model (writing to some magic values in the addressmap), and is an application of Section III-C: we start a during task initially empty, call extra_time(5000*PERIOD) every 5000 ISS steps, and call extra_time(N*PERIOD); N = 0; catch_up() at every synchronization point (i.e. before sending or waiting for an interrupt, within polling loops, ...).

The ISS triggers reads and writes on the bus, which triggers execution of handlers in the target modules. When these

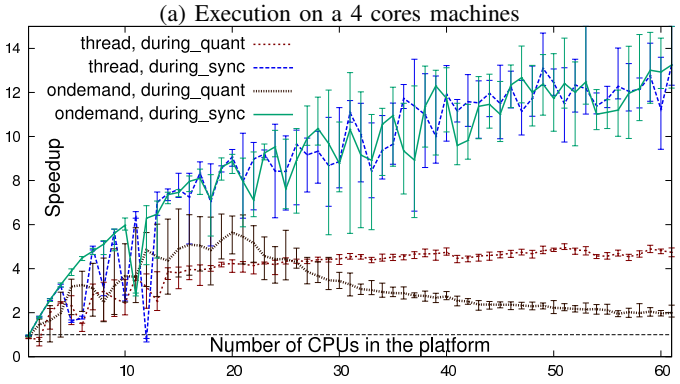
handlers do not call SystemC primitives, and if the data-access they perform are safe, we can let them run in the context of the `during` task. But for example, writing on `ITC` component may trigger an `e.notify()`, which cannot be run from a `during` task. When triggering a write transaction to such modules, the ISS wraps the bus access in an `sc_call`, so that the transaction be executed in the context of the `SC_THREAD`, like in the sequential version. We also wrap wait for interrupts in `sc_calls`. Most accesses to the bus are performed without synchronization: the bus component itself is thread-safe (no state variable except the addressmap, which is read-only during simulation), and memory access that do not require particular synchronization in the actual system are performed without synchronization in simulation.

B. Speedup and Scalability

We tested the scalability of the approach on two machines. The first one (Figure 9.(a)) has 4 Intel Xeon cores. The second (Figure 9.(b)) has 4 AMD Opteron-6176 processors, each containing 12 cores (total of 48 cores).



(a) Execution on a 4 cores machines



(b) Execution on a 48 cores machine

Fig. 9. Speedup Compared to SEQ for Different Implementation (average, min and max over 10 runs)

Both graphs show the benefit of defining explicit synchronization points, as the strategy `during_sync` always gives a better speedup than the quantum-based one. This also illustrate the fact that `sc-during` targets loosely timed systems, since coarser granularities models give better performance (unlike most alternative approaches). For low numbers of CPU in the model, the speedup is almost optimal, it does not decrease significantly when parallelizing the model too much (i.e. instantiation more CPU in the model than physical CPUs). With a large number of processors, even on the 48 cores machines, we can notice that the speedup is smaller than the number of processors. While disappointing, this result is not so surprising since the `during` task need some synchronization with SystemC (some bus accesses being protected with `sc_call`). We can further optimize the platform by making the ITC component itself thread-safe (by using `async_request_update()`), and get 13% additional speedup in ONDEMAND implementation and 23% in THREAD implementation, on machine (b) with 60 threads.

It should be noted that the suboptimal speedups are not due to extra computation, but essentially to latency of synchronization. Our experiments show that the CPU time spent in the computation increase by less than 35% even for very loaded simulation (60 CPUs). Unlike approaches using busy-waiting, this means that our parallel simulation will remain efficient on multi-users machines.

The `during_quant` strategy has a slightly lower speedup when the parallelism is low, but stagnates (and even *decreases* for the ONDEMAND implementation) with a large number of threads, because OS threads need frequent synchronization with SystemC, which becomes the synchronization bottleneck. This problem can be solved by increasing the quantum. For example, by setting the quantum to 50,000 instructions (i.e. 1 ms) instead of 5000, we speed up the simulation in THREAD implementation by a factor of 3.7. With a quantum of 10 ms, we get back to the same performance as the `during_sync` version. This illustrates the potential difficulty of choosing the right quantum, and the advantage of explicit synchronizations.

V. Conclusion

We presented an approach for parallel execution of SystemC models. A deliberate choice of the approach is that we require the user to use new primitives, as the alternatives would either be inefficient or would break the SystemC semantics for existing code. We showed that our approach allows a significant speedup with a reasonable modeling effort. By providing primitives to express the notion of duration, we allow the user to create more parallelism in the model than cycle-based approach, hence can efficiently parallelize loosely timed systems.

Our approach works with any SystemC scheduler, it is a simple library whose primitives are called directly from the `SC_THREAD`, hence intrusion in the user code is minimal. We could in theory run our library on top of a parallel implementation of SystemC, although such setup has not been experimented. Running code in a separate OS thread limits the interactions with SystemC, hence the approach applies best to parallelize computation that do not interact with the rest of the simulation, but we also provide primitives to allow synchronizing with SystemC, hence communication with other processes (`sc_call`, `extra_time` and `catch_up`).

Further work include providing more tools for synchronization to the user. For example, we manually wrapped in `sc_call` some transactions, but the `Bus` component could do it automatically for user-defined address ranges. This approach can be compared with the SpecC approach, where the language

reference allows parallelism within a component, but the communication channels act as monitors [17].

REFERENCES

[1] *IEEE 1666 Standard: SystemC Language Reference Manual*, Open SystemC Initiative, 2011. [Online]. Available: http://www.accellera.org/

[2] "SystemCASS," SoClib project.

[3] Y. Bouzouzou, "Accélération des simulations de systèmes sur puce au niveau transactionnel," Diplôme de Recherche Technologique, UJF Grenoble, 2007.

[4] W. Chen, X. Han, and R. Dömer, "Out-of-order parallel simulation for ESL design," in *DATE*, W. Rosenstiel and L. Thiele, Eds. IEEE, 2012, pp. 141–146.

[5] N. Blanc and D. Kroening, "Race analysis for SystemC using model checking," *ACM TODAES*, vol. 15, no. 3, p. 21, 2010.

[6] C. B, C. P, and Z. J, "A conservative approach to SystemC parallelization," in *International Conference on Computational Science*, 2006.

[7] P. Combes, E. Caron, F. Desprez, B. Chopard, and J. Zory, "Relaxing synchronization in a parallel SystemC kernel," in *International Symposium on Parallel and Distributed Processing with Applications*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 180–187.

[8] C. Schumacher, R. Leupers, D. Petras, and A. Hoffmann, "ParSC: synchronous parallel SystemC simulation on multi-core host architectures," ser. CODES/ISSS '10. New York, NY, USA: ACM, 2010, pp. 241–246. [Online]. Available: http://doi.acm.org/10.1145/1878961.1879005

[9] E. P, P. Chandran, J. Chandra, B. P. Simon, and D. Ravi, "Parallelizing SystemC kernel for fast hardware simulation on SMP machines," ser. PADS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 80–87. [Online]. Available: http://dx.doi.org/10.1109/PADS.2009.25

[10] R. S. Khaligh and M. Radetzki, "A dynamic load balancing method for parallel simulation of accuracy adaptive TLMs," in *FDL*, 2010.

[11] K. Huang, I. Bacivarov, F. Hugelshofer, and L. Thiele, "Scalably distributed SystemC simulation for embedded applications," *International Symposium on Industrial Embedded Systems*, pp. 271–274, Jun. 2008.

[12] A. Mello, I. Maia, A. Greiner, and F. Pecheux, "Parallel simulation of SystemC TLM 2.0 compliant MPSoC on SMP workstations," in *DATE Conference Exhibition*, March 2010, pp. 606 –609.

[13] R. Khaligh and M. Radetzki, "Efficient parallel transaction level simulation by exploiting temporal decoupling," in *IESS 2009*, vol. 310. Springer-Verlag New York Inc, 2009, p. 149.

[14] 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.

[15] ——, "Modeling of time in discrete-event simulation of systems-on-chip," in *MEMOCODE*, July 2011.

[16] J. Cornet, "Separation of functional and non-functional aspects in transactional level models of systems-on-chip," Ph.D. dissertation, Institut National Polytechnique de Grenoble, 2008.

[17] R. Domer, W. Chen, X. Han, and A. Gerstlauer, "Multi-core parallel simulation of system-level description languages," in *ASP-DAC*. IEEE, 2011, pp. 311–316.