# Parallel Simulation

Rainer Dömer, Guantao Liu, and Tim Schmidt

**Abstract**

The SystemC standard is widely used in industry and academia to model and simulate electronic system-level designs. However, despite the availability of multi-core processor hosts, the reference SystemC simulator is still based on sequential *Discrete Event Simulation (DES)* which executes only a single thread at any time.

In recent years, parallel SystemC simulators have been proposed which run multiple threads in parallel based on *Parallel Discrete Event Simulation (PDES)* semantics. While this can improve the simulator run time by an order of magnitude, synchronous PDES requires careful dependency analysis of the model and still limits the parallel execution to threads that run at the same simulation time.

In this chapter, we review the classic DES and PDES algorithms and then present a state-of-the-art approach called *Out-of-Order Parallel Discrete Event Simulation (OOO PDES)* which breaks the traditional time cycle barrier and executes threads in parallel and out of order (ahead of time) while maintaining the standard SystemC modeling semantics. Specifically, we present our *Recoding Infrastructure for SystemC (RISC)* that consists of a dedicated SystemC compiler and advanced parallel simulator. RISC provides an open-source reference implementation of OOO PDES and achieves fastest simulation speed for traditional SystemC models without any loss of accuracy.

## Contents

R. Dömer (✉) • G. Liu • T. Schmidt
Center for Embedded and Cyber-physical Systems, University of California, Irvine, CA, USA
e-mail: doemer@uci.edu; guantaol@uci.edu; schmidtt@uci.edu

© Springer Science+Business Media Dordrecht 2016
S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7358-4_19-1

## Acronyms

| | |
|---|---|
| **AST** | Abstract Syntax Tree |
| **DE** | Discrete Event |
| **DES** | Discrete Event Simulation |
| **DUT** | Design Under Test |
| **ESL** | Electronic System Level |
| **OOO PDES** | Out-of-Order Parallel Discrete Event Simulation |
| **PDES** | Parallel Discrete Event Simulation |
| **RISC** | Recoding Infrastructure for SystemC |
| **SG** | Segment Graph |
| **SLDL** | System-Level Description Language |

## 1 Introduction

Electronic System Level (ESL) design space exploration in general and the validation of ESL designs in particular typically rely on dynamic *model simulation* in order to study and optimize a desired system or electronic device before it is actually being built. For this simulation, an abstract model of the intended system is first specified in a System-Level Description Language (SLDL) such as SpecC [13] or SystemC [15]. This SLDL model is then compiled, executed, and evaluated on a host computer. In order to reflect causal ordering and provide abstract timing information, SLDL simulation algorithms are usually based on the classic *Discrete*

*Event (DE)* semantics which drive the simulation process forward by use of events and simulation time advances.

As an IEEE standard [18], the SystemC SLDL [15] is predominantly used in both industry and academia. Under the umbrella of the Accellera Systems Initiative [1], the SystemC Language Working Group [31] maintains not only the official SystemC language definition but also provides an open-source proof-of-concept library [16] that can be used free of charge to simulate SystemC design models within a standard C++ programming environment. However, this reference implementation follows the traditional scheme of sequential *Discrete Event Simulation (DES)* which is much easier to implement than truly parallel approaches. Here, the discrete time events generated during the simulation are processed strictly in sequential order. As such, the SystemC reference simulator runs fully sequentially and cannot utilize any parallel computing resources available on today's multi- and many-core processor hosts. This severely limits the simulation speed since a single processor core has to perform all the work.

## 1.1 Exploiting Parallelism for Higher Simulation Speed

In order to provide faster simulation performance, parallel execution is highly desirable, especially since the SLDL model itself already contains clearly exposed parallelism that is explicitly specified by the language constructs, such as `SC_METHOD`, `SC_THREAD`, and `SC_CTHREAD` in the case of SystemC SLDL. Here, the topic of *Parallel Discrete Event Simulation (PDES)* [12] has recently gained a lot of attraction again. Generally, the PDES approach issues multiple threads concurrently and runs these threads in parallel on the multiple available processor cores. In turn, the simulation speed increases significantly. The naive assumption of a linear speedup of factor $nx$ for $n$ available processor cores occurs very rarely due to the usually quite limited amount of parallelism exposed in the model and the application of Amdahl's law [2]. Typically, a speedup of one order of magnitude can be expected for embedded system applications with parallel modules [6].

With respect to PDES performance, it is very important to understand the dependencies among the discrete time events and also to distinguish between the dependencies present in the model and the (additional) dependencies imposed by the simulation algorithm. In general, SLDL models assume a *partial order* of events where certain events depend on others due to a causal relationship or occurrence in sequence along the time line, and other events are independent and may be performed in any order or in parallel. On top of this required partial ordering of events in the model, a sequential simulator (described in Sect. 2) imposes a much stronger *total order* on the event processing. This is acceptable as it observes the weaker SLDL execution semantics but results in slow performance due to the overly restrictive sequential execution.

In contrast, PDES follows a partial ordering of events. Here, we can distinguish between regular *synchronous PDES*, which advances the (global) simulation time in order and executes threads in lockstep parallel fashion, and aggressive *asynchronous PDES*, where simulation time is effectively localized and independent threads can execute out of order and run ahead of others. As we will explain in detail in Sect. 3, the strict in-order execution imposed by synchronous PDES still limits the opportunities for parallel execution. When a thread finishes its evaluation phase, it has to wait until all other threads have completed their evaluation phases as well. Threads finishing early must stop at the simulation cycle barrier, and available processor cores are left idle until all runnable threads reach the cycle barrier.

This problem is overcome by the latest state-of-the-art approaches which implement a novel technique called *Out-of-Order Parallel Discrete Event Simulation (OOO PDES)* [6, 8]. By internally localizing the simulation time to individual threads and carefully processing dependent events only at required times, an OOO PDES simulator can issue threads in parallel and ahead of time, exploiting the maximum available parallelism without loss of accuracy. Thus, the OOO PDES presented in Sect. 4 minimizes the idle time of the available parallel processor cores and results in highest simulation speed while fully maintaining the standard SLDL and unmodified model semantics.

We should note that parallel simulation in general, and synchronous PDES and OOO PDES in particular, can be implemented in different environments and are *not* language dependent. In fact, the parallel simulation techniques discussed in Sects. 3 and 4 have been originally designed using the SpecC language [10, 13, 35] and integrated into the system-on-chip environment (see Chap. 31, ▸ SCE: System-on-Chip Environment and in particular Sect. 4.1, "Simulation") but can be equally well applied to other SLDLs with explicitly exposed parallelism. Without loss of generality, and in order to present the state of the art, we describe in this chapter the evolution from sequential DES over synchronous PDES to asynchronous OOO PDES using the SystemC SLDL [15, 18] which is both the de facto and official standard for ESL design today. In particular, we describe in Sect. 5 our Recoding Infrastructure for SystemC (RISC) [21], which is open source [20] and consists of a dedicated SystemC compiler and corresponding out-of-order parallel simulator, and thus provides a reference implementation of OOO PDES for SystemC.

Finally, we conclude this chapter in Sect. 6 with experimental results for embedded application examples that demonstrate the significantly reduced simulator run times due to parallel simulation.

## 1.2 Related Work on Accelerated Simulation

Parallel simulation is a well-studied subject in the literature [4, 12, 24]. Two major synchronization paradigms can be distinguished, namely, conservative and optimistic [12]. *Conservative* PDES typically involves dependency analysis and ensures in-order execution for dependent threads. In contrast, the *optimistic* paradigm assumes that threads are safe to execute and rolls back when this proves incorrect. Often, the temporal barriers in the model prevent effective parallelism in conservative PDES while rollbacks in optimistic PDES are expensive in execution.

The OOO PDES presented in Sect. 4 is conservative and can be seen as an improvement over synchronous PDES (Sect. 3) approaches, such as [11, 27] for SystemC and [7] for SpecC.

Distributed parallel simulation, including [4, 17], is a natural extension of PDES where the system model is partitioned into sets of modules and distributed onto a set of networked host computers. Here, the necessary model partitioning may create extra work, and often the network speed becomes a bottleneck for synchronization and communication among the multiple host simulators.

In a wider perspective, simulation performance can also be improved by clever software modeling and utilizing specialized hardware. Software approaches include the overall concept of transaction-level modeling (TLM) [3] and temporal decoupling [32, 33], which abstract communication from a pin-accurate level to entire transactions with only approximate timing, and source-level [30] or host-compiled simulation [14], which model computation and scheduling at a higher abstraction level. Typically, these modeling techniques trade off higher simulation speed at the cost of reduced timing accuracy.

Simulation acceleration can also be achieved by special purpose hardware, such as field-programmable gate arrays (FPGA) and graphics processing units (GPU). For example, [29] emulates SystemC code on FPGA boards and [23] proposes a SystemC multi-threading model on GPUs. As a hybrid approach, [28] parallelizes SystemC simulation across multiple CPUs and GPUs. Usually, special hardware approaches pose limitations on the model partitioning across the heterogeneous simulator units.

Parallel simulation can also be organized as multiple simulators which run independently in parallel and synchronize regularly or when needed (i.e., when timing differences grow beyond a given threshold). For instance, the Wisconsin Wind Tunnel [22] uses a conservative time bucket synchronization scheme to coordinate multiple simulators at predefined intervals. Another example [34] introduces a simulation backplane to handle the synchronization between wrapped simulators and analyzes the system to optimize the period of synchronization message transfers.

From this discussion, we can see that parallel simulation is an important topic that has been addressed and optimized with a multitude of approaches. For the remainder of this book chapter, we will focus on the SystemC language and present two general parallel simulation techniques, namely, synchronous PDES in Sect. 3 and an advanced out-of-order PDES algorithm in Sect. 4. Both approaches are general in the sense that they do not require any special setup or hardware, pose no limitations on the simulation model, and do not sacrifice any timing accuracy in the result.

## 2 Discrete Event Simulation (DES)

Before we address the details of modern parallel simulation, we review in this section the classic sequential simulation of models with discrete time. After the description of this traditional DES algorithm, we then extend it incrementally to synchronous PDES in Sect. 3 and finally to out-of-order PDES in Sect. 4.

## 2.1 Discrete Time and Discrete Event Model

SLDL simulation is driven by discrete events and simulation time advances. We will use the term *simulation time* for the simulated time maintained by the simulator. This must not be confused with *simulator run time* which is the actual wall-clock time that measures how long it takes the simulator to perform the simulation on the host. Simulation time consists of a tuple $(t, \delta)$ where $t$ represents an integral amount of simulated time since the simulation start $t = 0$ and $\delta$ is a positive integer that counts iterations at the same simulated time $t$ due to event notifications.

Formally, time tuples, often referred to as time stamps, form a partial order and can be compared as follows [6]:

equal: $(t_1, \delta_1) = (t_2, \delta_2)$ iff $t_1 = t_2, \delta_1 = \delta_2$
before: $(t_1, \delta_1) < (t_2, \delta_2)$ iff $t_1 < t_2$, or $t_1 = t_2, \delta_1 < \delta_2$
after: $(t_1, \delta_1) > (t_2, \delta_2)$ iff $t_1 > t_2$, or $t_1 = t_2, \delta_1 > \delta_2$

Events serve the purpose of synchronization among communicating or dependent threads. Threads can *notify* events or *wait* for events. An event notification at time $(t, \delta)$ reaches a thread waiting for the event at the same time, or it expires without any effect if no thread is waiting for the event. If a thread wakes up due to a notified event, it resumes its execution at the next delta increment $(t, \delta + 1)$ or immediately at $(t, \delta)$ in case of a SystemC immediate notification.

In other words, the SLDL semantics use an outer cycle in the simulation process to model time advances so that thread execution can reflect estimated duration or delays by adding to time $t$. In addition, there is an inner cycle in the simulator, called *delta cycle*, that is used for event notifications which may in turn wake waiting threads, in which case time is incremented by one $\delta$ count. In the case of SystemC, yet another innermost cycle is available where so-called immediate event notifications may take place without any time advance. Since immediate notifications can easily lead to nondeterministic models with potential deadlocks or lost events, these should generally be avoided.

## 2.2 Scheduling Queues

SLDLs use a set of parallel threads to execute the functionality in the model. In SystemC, such threads are explicitly specified as `SC_METHOD`, `SC_THREAD`, or `SC_CTHREAD`. These threads are then managed by a scheduler in the simulator kernel which decides when threads are actually dispatched to run.

To coordinate the execution of the threads according to the SLDL semantics, the simulation scheduler typically maintains sorted lists or *queues* of threads where each thread is a member of one queue at any time. For ease of understanding, we use a simplified formal model here and ignore special cases in the SystemC language, such as suspended processes. Formally we define the following scheduling sets [9]:

$\mathsf{THREADS} = \mathsf{READY} \cup \mathsf{RUN} \cup \mathsf{WAIT} \cup \mathsf{WAITTIME}$
$\mathsf{READY} = \{ \, th \mid th \text{ is ready to run } \}$
$\mathsf{RUN} = \{ \, th \mid th \text{ is currently running } \}$
$\mathsf{WAIT} = \{ \, th \mid th \text{ is waiting for one or more events } \}$
$\mathsf{WAITTIME} = \{ \, th \mid th \text{ is waiting for time advance } \}$

At the beginning of simulation at time $(0,0)$, all threads are placed into the READY queue and the other sets are empty:

$\mathsf{THREADS} = \mathsf{READY}$
$\mathsf{RUN} = \mathsf{WAIT} = \mathsf{WAITTIME} = \varnothing$

During simulation, the scheduler moves threads between the queues and suspends or resumes their execution according to their state. In order to describe the discrete event scheduling algorithms below, we formally define the following scheduling operations on threads *th* maintained in queues A and B:

**Run**(*th*): thread *th* begins or resumes its execution
**Stop**(): the current thread stops running or suspends its execution
**Yield**(*th*): the current thread stops running and yields execution to thread *th*
*th* = **Pick**(A): pick one thread *th* out of set A
**Move**(*th*, A, B): move thread *th* from set A to set B

For inter-thread synchronization and communication, the SystemC language provides events and channel primitives with special **update** semantics. Without going into the details of event notifications and channel updates, we denote the set of instantiated primitive channels in the model as CHANNEL.

## 2.3 Sequential Discrete Event Scheduler

Figure 1 shows the traditional DES scheduling algorithm as it is implemented by the Accellera SystemC reference simulator [16]. Most notably, this algorithm is fully sequential in the sense that only a single thread is made runnable at all times. In SystemC specifically, the choice of the next thread to run is nondeterministic by definition, so one thread is randomly picked from the READY queue and placed into the RUN queue for execution. When the thread returns to the scheduler due to execution of a `wait` statement, it yields control back to the scheduler which in turn picks the next thread to run.

When the READY queue is empty, the scheduler performs requested channel updates and event notifications which typically fills the READY queue again with threads that wake up due to events they were waiting for. These are taken out of the WAIT queue and a new delta cycle begins.

If no threads become ready after the update and notification phase, the current time cycle is complete. Then the scheduler advances the simulation time, moves all threads with the earliest next time stamp from the WAITTIME queue into the READY queue, and resumes execution with the next time cycle.

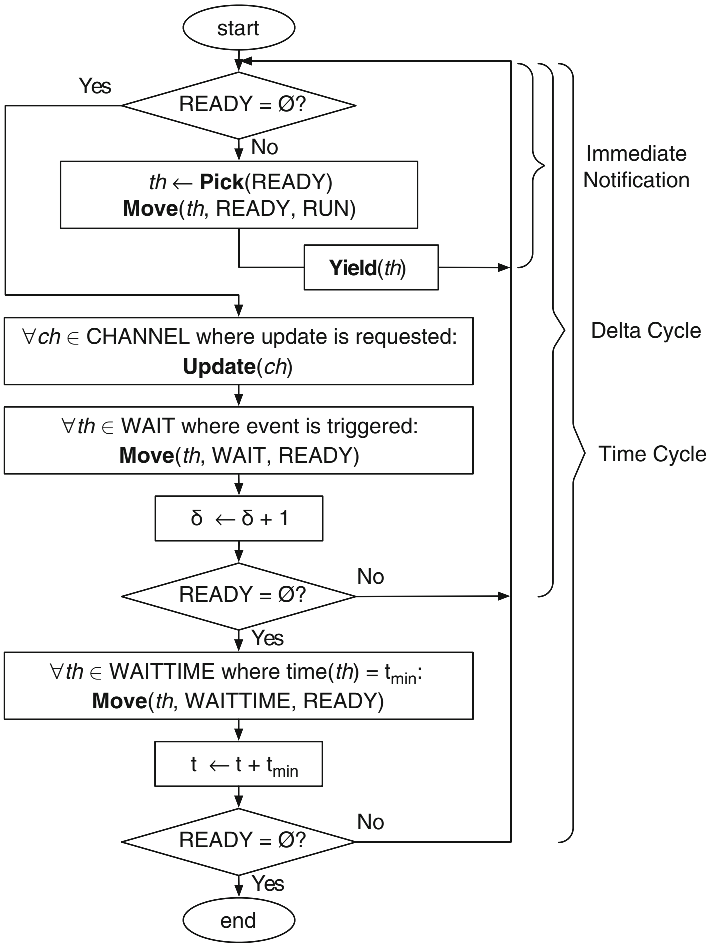


**Fig. 1** Traditional discrete event simulation (DES) scheduler for SystemC

Finally, when both the READY and WAITTIME queues are empty, the simulation terminates.

## 3 Parallel Discrete Event Simulation (PDES)

The sequential DES algorithm can be easily extended to support synchronous parallel simulation. Instead of a single thread in DES, regular PDES manages multiple threads at the same time in the RUN queue. These threads can then execute truly in parallel on the parallel processors of the host.

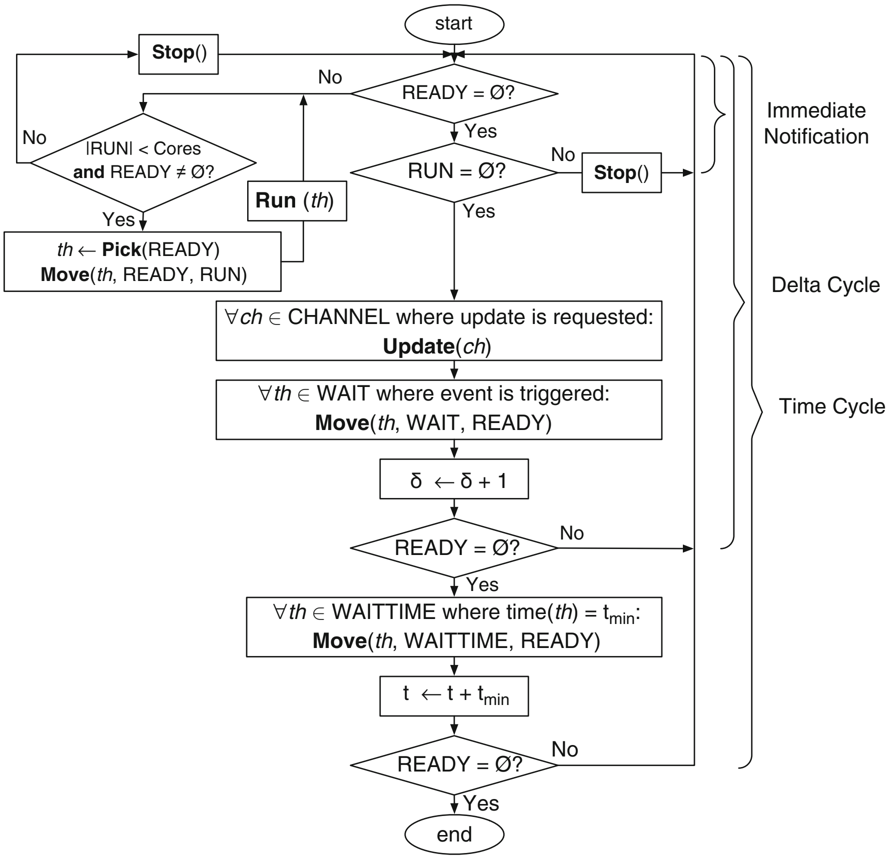

start
Stop()
No
READY = Ø?
Yes
No
|RUN| < Cores and READY ≠ Ø?
Yes
Run (th)
RUN = Ø?
No
Stop()
Yes
th ← Pick(READY)
Move(th, READY, RUN)
∀ch ∈ CHANNEL where update is requested: Update(ch)
∀th ∈ WAIT where event is triggered: Move(th, WAIT, READY)
δ ← δ + 1
READY = Ø?
No
Yes
∀th ∈ WAITTIME where time(th) = t_min: Move(th, WAITTIME, READY)
t ← t + t_min
READY = Ø?
No
Yes
end
Immediate Notification
Delta Cycle
Time Cycle

**Fig. 2** Synchronous parallel discrete event simulation (PDES) scheduler for SystemC

## 3.1 Parallel Discrete Event Scheduler

The PDES scheduling algorithm, as shown in Fig. 2, operates the same way as the traditional scheduler in Fig. 1, but with one exception: the synchronous parallel scheduler picks multiple threads from the READY queue and runs them in parallel on the available processor cores.

In the evaluation phase, as long as the READY queue is not empty and an idle core is available, the PDES scheduler issues a new thread from the READY queue in a loop. When a thread finishes earlier than the other threads in the same cycle, a new ready thread is picked and assigned to the free processor core. Thus, within the same delta cycle, PDES keeps as many processor cores as busy as possible.

However, we notice that only threads at the same time $(t, \delta)$ run in parallel. Synchronous PDES implies an absolute barrier at the end of each delta and time cycle. All threads need to wait at the barrier until all other runnable threads finish their current evaluation phase. Only then the synchronous scheduler performs the channel update and event notification phase for the next delta or advances simulation time for the next time cycle.

## 3.2 Protection of the Parallel Simulation Kernel

The benefit of PDES running more than a single thread at the same time comes at a price. Explicit synchronization among the parallel threads becomes necessary in critical sections. In particular, shared data structures in the simulation kernel, including the global simulation time, event lists, and thread queues, need to be properly protected for mutual exclusive access by the concurrent threads.

In order to protect the central scheduling resources, locks (binary semaphores) and condition variables need to be introduced for proper thread synchronization. For example, our RISC prototype implementation (see Sect. 5 below) uses one dedicated kernel lock to protect the scheduling resources. This lock is acquired by the threads at every kernel entry and released upon kernel exit. Each thread also owns a condition variable $c$ that is used in combination with the lock to put threads to sleep (**Stop**() calls `wait`($c$)) or wake them up (**Run**($th$) calls `signal`($c$)), as determined by the scheduling kernel.

## 3.3 Preserving SystemC Execution Semantics in PDES

In contrast to the SpecC language, which allows preemptive parallel execution of the threads in the model [10], the SystemC language poses strict rules on standard-compliant simulation. This is a very important aspect to consider when applying PDES to SystemC. For semantics-compliant SystemC simulation, complex inter-dependency analysis over all threads and variables in the model is a prerequisite to parallel execution [9]. The IEEE standard SystemC Language Reference Manual (LRM) [18] clearly states that "process instances execute without interruption" and presumably is meant to simplify the writing of SystemC models. Here, the need to prevent parallel access conflicts to shared variables and to avoid potential race conditions among the parallel threads becomes a burden for the simulation environment (rather than for the model designer).

This requirement is also known as *cooperative (or coroutine) multitasking* which is explicitly assumed by the SystemC execution semantics. As detailed in [9], the particular problem of parallel simulation is also explicitly addressed in the SystemC LRM [18]:

> An implementation running on a machine that provides hardware support for concurrent processes may permit two or more processes to run concurrently, provided that the behavior appears identical to the coroutine semantics defined in this subclause. In other words, the implementation would be obliged to analyze any dependencies between processes and constrain their execution to match the coroutine semantics.

Consequently, a standard-compliant PDES environment for SystemC must identify and resolve any dependencies among the threads in the model. We will describe this required dependency analysis in detail in Sect. 5.3 because it is needed for both synchronous and out-of-order PDES.

## 4 Out-of-Order Parallel Discrete Event Simulation (OOO PDES)

In OOO PDES [6], we break the synchronous barrier in the simulator so that independent threads can also run in parallel when they are at a different simulation times (different $t$ or different $\delta$). In other words, threads are allowed to run ahead in time and thus can execute out-of-order, unless a causal relationship prohibits it.

### 4.1 Thread-Local Simulation Time

For OOO PDES, we replace the global simulation time $(t, \delta)$ with local time stamps for each thread. Thus, each thread $th$ maintains its own time $(t_{th}, \delta_{th})$.

Events get assigned their own time, too. Since events in the simulation model can occur multiple times and at different simulation times, we note an event $e$ notified at time $(t, \delta)$ as a triple $(id_e, t_e, \delta_e)$. Thus, every event is managed with its own ID and notification time attached.

Finally, we distinguish the sets of events that have been notified at a given time. Formally, we define:

$\mathsf{EVENTS} = \cup \mathsf{EVENTS}_{t,\delta}$
$\mathsf{EVENTS}_{t,\delta} = \{(id_e, t_e, \delta_e) \mid t_e = t, \delta_e = \delta)\}$

### 4.2 Dynamically Evolving Scheduling Queues

Rather than the static DES queues which exist at all times, we define for OOO PDES multiple sets ordered by their local time stamps and dynamically create and delete these sets as needed. For efficiency reasons, these sets are typically implemented as true queues where the threads are ordered by increasing time stamps. Formally, we define the following queues:

$\mathsf{QUEUES} = \{\mathsf{READY}, \mathsf{RUN}, \mathsf{WAIT}, \mathsf{WAITTIME}\}$
$\mathsf{READY} = \cup \mathsf{READY}_{t,\delta}$, where $\mathsf{READY}_{t,\delta} = \{th \mid th$ is ready to run at $(t, \delta)\}$
$\mathsf{RUN} = \cup \mathsf{RUN}_{t,\delta}$, where $\mathsf{RUN}_{t,\delta} = \{th \mid th$ is running at $(t, \delta)\}$
$\mathsf{WAIT} = \cup \mathsf{WAIT}_{t,\delta}$, where $\mathsf{WAIT}_{t,\delta} = \{th \mid th$ is waiting since $(t, \delta)$ for events $(id_e, t_e, \delta_e)$, $(t_e, \delta_e) \geq (t, \delta)\}$
$\mathsf{WAITTIME} = \cup \mathsf{WAITTIME}_{t,\delta}$, where $\delta = 0$, $\mathsf{WAITTIME}_{t,\delta} = \{th \mid th$ is waiting for simulation time advance to $(t, 0)\}$

As in the regular DES case, the simulation starts at time $(0, 0)$ with all threads in the $\mathsf{READY}_{0,0}$ queue. Then again the threads change state by transitioning between the queues, as determined by the scheduler:

**Move**$(th, \mathsf{READY}_{t,\delta}, \mathsf{RUN}_{t,\delta})$: thread *th* is issued and becomes runnable
**Move**$(th, \mathsf{RUN}_{t,\delta}, \mathsf{WAIT}_{t,\delta})$: thread *th* calls `wait(`$e$`)` for an event $e$
**Move**$(th, \mathsf{RUN}_{t,\delta}, \mathsf{WAITTIME}_{t',0})$, where $t < t' = t + d$: thread *th* calls `wait(`$d$`)` to wait for a time delay $d$
**Move**$(th, \mathsf{WAIT}_{t,\delta}, \mathsf{READY}_{t',\delta''})$, where $(t, \delta) \leq (t', \delta'')$: thread *th* is waiting since time $(t, \delta)$ for event $e = (id_e, t'_e, \delta'_e)$ which is notified at time $(t', \delta')$; in turn, thread *th* becomes ready to run at $(t', \delta'')$ where $\delta'' = \delta'$ (immediate notification) or $\delta'' = \delta' + 1$ (regular delta cycle notification)
**Move**$(th, \mathsf{WAITTIME}_{t,\delta}, \mathsf{READY}_{t,\delta})$, where $\delta = 0$: simulation time advances to time $(t, 0)$, making one or more threads *th* ready to run; the local time for these threads *th* is set to $(t_{th}, \delta_{th})$ where $t_{th} = t$ and $\delta_{th} = 0$

Whenever the sets $\mathsf{READY}_{t,\delta}$ and $\mathsf{RUN}_{t,\delta}$ become empty and there are no $\mathsf{WAIT}_{t',\delta'}$ or $\mathsf{WAITTIME}_{t',\delta'}$ queues with earlier time stamps $(t', \delta') \leq (t, \delta)$, then the scheduler can delete these sets as well as any expired events $\mathsf{EVENTS}_{t,\delta}$.

## 4.3 Out-of-Order Parallel Discrete Event Scheduler

Figure 3 shows the OOO PDES scheduling algorithm. Since each thread maintains its own local time, the scheduler can relax the nested loops structure of synchronous PDES and deliver events and update simulation times individually, providing more flexibility for threads to run in parallel. Overall, this results in a higher degree of parallelism and thus higher simulation speed.

Note that the prior loops for explicit delta cycles and time cycles in the scheduler control flow do not exist any more for OOO PDES. Instead, we only have one main loop where both the notification phase and time updates are processed. The READY queue is consequently filled with more threads which, however, are now subject to possible conflicts. These conflicts are then taken into account when threads are picked from the READY queue and issued for execution into the RUN set.

Note also that the WAITTIME queue gets cleared in every scheduling step and all the threads move into the timed READY queue. Then, when the scheduler picks ready threads to run, it prefers earlier ones over threads with later time stamps. This order prevents threads from starving in the READY queue and also minimizes conflicts among the ready threads.

Potential conflicts are strictly averted by the NoConflicts(*th*) condition in Fig. 3 when runnable threads are picked. Here, detailed dependency analysis is used to avoid potential data, event, and time advance hazards among the set of threads in RUN that are executing in parallel. Only if a thread *th* has NoConflicts(*th*) it can be issued for parallel execution.

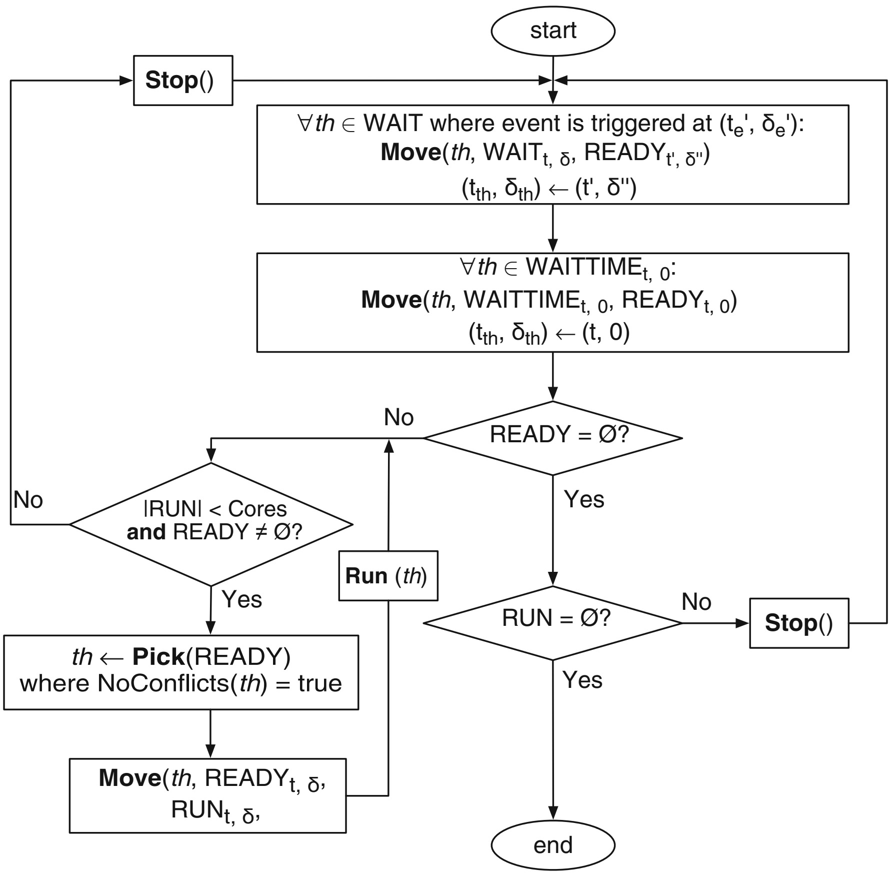

start
Stop()
∀th ∈ WAIT where event is triggered at (te', δe'):
Move(th, WAITt, δ, READYt', δ'')
(tth, δth) ← (t', δ'')
∀th ∈ WAITTIMEt, 0:
Move(th, WAITTIMEt, 0, READYt, 0)
(tth, δth) ← (t, 0)
No
READY = Ø?
Yes
No
|RUN| < Cores and READY ≠ Ø?
Yes
Run (th)
RUN = Ø?
No
Stop()
Yes
th ← Pick(READY) where NoConflicts(th) = true
Move(th, READYt, δ, RUNt, δ)
end

**Fig. 3** Out-of-order parallel discrete event simulation (OOO PDES) scheduler for SystemC

## 4.4 OOO PDES Scheduling Algorithm

Algorithm 1 formally defines the scheduling algorithm of OOO PDES. At each scheduling step, the scheduler first evaluates notified events and wakes up corresponding threads from WAIT. If a thread receives its event $e$ with time stamp $(t'_e, \delta'_e)$, it becomes ready to run and its local time advances to either $(t'_e, \delta'_e)$ for an immediate notification or $(t'_e, \delta'_e + 1)$ for regular delta notifications.

After event notifications, the scheduler processes local time advances and moves any threads in WAITTIME to the $\mathsf{READY}_{t,0}$ queue corresponding to their wait time.

Then the scheduler issues threads for parallel execution as long as idle CPU cores and threads without conflicts are available. Finally, if no threads can run, that is when $\mathsf{RUN} = \mathsf{READY} = \emptyset$, the simulator terminates.

**Algorithm 1** OOO PDES scheduling algorithm

```
1: procedure OOOPDES_SCHEDULER
2:     for all $th \in$ WAIT do                                        ▷ Process event notifications
3:         if $\exists e = (id_e, t'_e, \delta'_e)$ where $th$ awaits $e$ and $(t'_e, \delta'_e) \geq (t, \delta)$ then
4:             if $e$ is an immediate notification then
5:                 Move($th$, $\text{WAIT}_{t,\delta}$, $\text{READY}_{t'_e,\delta'_e}$)
6:                 $t_{th} \leftarrow t'_e$; $\delta_{th} \leftarrow \delta'_e$
7:             else
8:                 Move($th$, $\text{WAIT}_{t,\delta}$, $\text{READY}_{t'_e,\delta'_e+1}$)
9:                 $t_{th} \leftarrow t'_e$; $\delta_{th} \leftarrow \delta'_e + 1$
10:            end if
11:        end if
12:    end for
13:    for all $th \in$ WAITTIME do                                    ▷ Process local time advances
14:        Move($th$, $\text{WAITTIME}_{t,\delta}$, $\text{READY}_{t,\delta}$)
15:        $t_{th} \leftarrow t$; $\delta_{th} \leftarrow 0$
16:    end for
17:    for all $th \in$ READY do                                       ▷ Out-of-order evaluation phase
18:        if $|\text{RUN}| <$ Cores and NoConflicts($th$) then
19:            Run($th$)
20:        end if
21:    end for
22:    return                                                          ▷ End of simulation
23: end procedure
```

Note that Algorithm 1 allows to enable/disable the parallel out-of-order execution at any time by setting the Cores parameter. For example, when in-order execution is needed for debugging purposes, we can set Cores to 1, and the algorithm will behave the same way as the traditional DES where only one thread is running in order at all times.

OOO PDES relies heavily on efficient conflict detection. At run time, the scheduler calls the function NoConflicts(*th*) listed in Algorithm 2. NoConflicts(*th*) checks for potential conflicts with all concurrent threads in the RUN and READY queues that run at an earlier time than the candidate thread *th*. For each concurrent thread, function Conflict($th_1$, $th_2$) checks for any data, time, and event hazards. We will explain these hazards and their analysis in detail in Sect. 5.3 below, because we can rely on the compiler to carry the heavy burden of this complex analysis and pass prepared conflict tables to the simulator. At run time, the scheduler can then perform these checks in constant time (O(1)) by use of table lookups.

## 5 Recoding Infrastructure for SystemC (RISC)

We have realized the OOO PDES approach for the SystemC language with our RISC. This proof-of-concept prototype environment consists of a compiler and simulator with examples and documentation. The RISC software package is available as open source on our website [20] and can be installed on any regular multi-core Linux host.

**Algorithm 2** Conflict detection in OOO PDES scheduler

```
 1: function NoConflicts($th$)
 2:     for all $th_2 \in \mathsf{RUN}_{t,\delta} \cup \mathsf{READY}_{t,\delta}$ where $(t,\delta) < (t_{th},\delta_{th})$ do
 3:         if Conflict($th$, $th_2$) then
 4:             return false
 5:         end if
 6:     end for
 7:     return true
 8: end function

 9: function Conflict($th_1$, $th_2$)
10:     if $th_2$ has data conflicts with $th_1$ then                    ▷ check data hazards
11:         return true
12:     end if
13:     if $th_2$ may enter another segment before $th_1$ then           ▷ check time hazards
14:         return true
15:     end if
16:     if $th_2$ may wake up another thread $th_3$ to run before $th_1$ then   ▷ check event hazards
17:         return true
18:     end if
19:     return false
20: end function
```

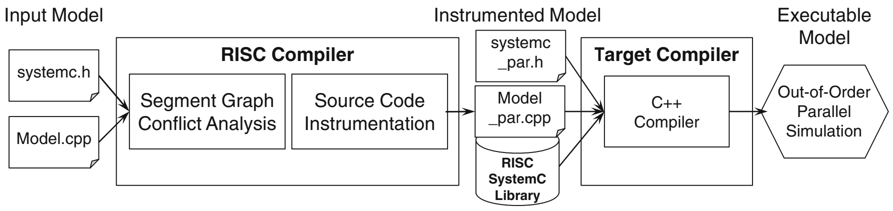


**Fig. 4** RISC compiler and simulator for out-of-order PDES of SystemC

To perform semantics-compliant parallel SystemC simulation with out-of-order scheduling, we introduce a dedicated SystemC compiler that works hand in hand with a new simulator. This is in contrast to the traditional SystemC simulation flow where a SystemC-agnostic C++ compiler includes the SystemC headers and links the input model directly against the reference SystemC library.

As shown in Fig. 4, our RISC compiler acts as a frontend that processes the input SystemC model and generates an intermediate model with special instrumentation for OOO PDES. The instrumented parallel model is then linked against the extended RISC SystemC library by the target compiler (a regular C++ compiler) to produce the final executable output model. OOO PDES is then performed simply by running the generated executable model.

From the user perspective, we simply replace the regular C++ compiler with the SystemC-aware RISC compiler (which in turn calls the underlying C++ compiler). Otherwise, the overall SystemC validation flow remains the same as before. It will be just faster due to the parallel simulation.

Internally, the RISC compiler performs three major tasks, namely, segment graph construction, conflict analysis, and source code instrumentation.

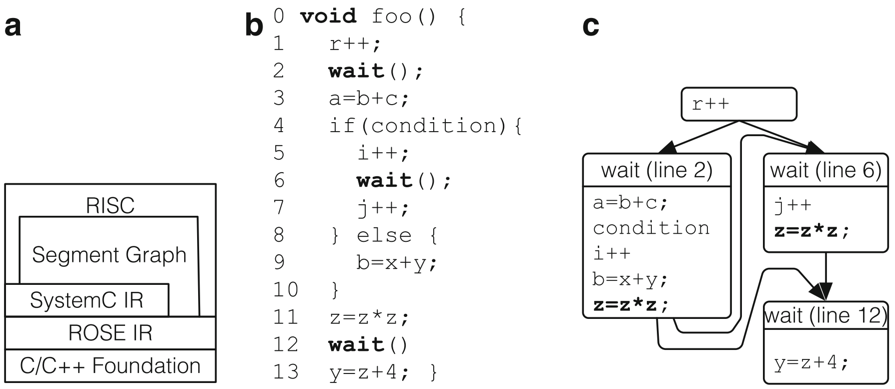
## 5.1 Segment Graph

The first task of the RISC compiler is to parse the SystemC input model into an Abstract Syntax Tree (AST). Since SystemC is syntactically regular C++ code, RISC relies here on the ROSE compiler infrastructure [25]. The ROSE internal representation (IR) provides RISC with a powerful C/C++ compiler foundation that supports AST generation, traversal, analysis, and transformation.

As illustrated with the RISC software stack shown in Fig. 5a, the RISC compiler then builds on top of the ROSE IR a SystemC internal representation which accurately reflects the SystemC structures, including the module and channel hierarchy, port connectivity, and other SystemC-specific constructs. Up until this layer, the RISC software stack is very similar to the SystemC-clang framework [19].

On top of this, the RISC compiler then builds a segment graph data structure. A *Segment Graph (SG)* [8] is a directed graph that represents the code segments executed by the threads in the model. With respect to SystemC simulation, these segments are separated by the scheduler entry points, i.e., the `wait` statements in the SystemC code. In other words, the discrete events in the SystemC execution semantics are explicitly reflected in the SG as segment boundaries.

Note that a general segment graph may be defined with different segment boundaries. In fact, the RISC infrastructure takes the segment boundary as a flexible parameter that may be set to any construct found in the code, including function calls or control flow statements, such as `if`, `while`, or `return`. Here, we use `wait` statements as boundary (specified as "segment graph [`wait`]" in Fig. 5c) since for



```
0  void foo() {
1    r++;
2    wait();
3    a=b+c;
4    if(condition){
5      i++;
6      wait();
7      j++;
8    } else {
9      b=x+y;
10   }
11   z=z*z;
12   wait()
13   y=z+4; }
```

**Fig. 5** Recoding infrastructure for SystemC (RISC) and a segment graph (SG) example [26]. (**a**) RISC software stack. (**b**) Example source code. (**c**) Segment graph [wait]

the purpose of parallel simulation we are interested in the segments of code executed between two scheduling points.

Formally, a segment graph consists of nodes and edges [6]. While the nodes are defined by the specified boundaries, the edges in the SG are defined by the possible control flow transitions. A transition exists between two segment nodes $S_1$ and $S_2$ if the flow of control starting from segment $S_1$ can reach segment $S_2$.

For example, the SystemC source code in Fig. 5b results in the SG shown in Fig. 5c where the segment boundary is chosen as SystemC `wait`. Here, the control flow from the start can reach the two `wait` statements in lines 2 and 6, resulting in the two edges to segment "wait(line2)" and "wait(line6)." Note also that source code lines may become part of multiple segments. Here, the assignment `z=z*z` is part of both segments "wait(line2)" and "wait(line6)" because it can be reached from both nodes.

## 5.2 Segment Graph Construction

The automatic construction of a segment graph is a complex process for the compiler. In this section, we first outline the main aspects and then provide a formal algorithm for the SG generation.

In contrast to the initial SpecC-based implementation [6, 8] (which is part of ► SCE: System-on-Chip Environment) which has had several limitations, the RISC SG generator can build a graph from any given scope in a C/C++-based code and the user can freely choose the segment boundaries (as stated above for a general SG). There are also no control flow limitations. The RISC compiler fully supports recursive functions, jump statements `break` and `continue`, as well as multiple `return` statements from functions. Finally, expressions with an undefined evaluation order will be properly ordered to avoid ambiguity.

Algorithm 3 formally defines the central function BUILDSG used in the RISC SG generator. Function BUILDSG is recursive and builds the graph of segments by traversing the AST of the design model. Here, the first parameter *CurrStmt* is the current statement which is processed next. The set *CurrSegs* contains the current set of segments that lead to *CurrStmt* and thus will incorporate the current statement. For instance, while processing the assignment `z=z*z` in Fig. 5, *CurrSegs* is the set {wait(line2), wait(line6)}, so the expression will be added to both segments.

If *CurrStmt* is a boundary statement (e.g., `wait`), a new segment is added to *CurrSegs* with corresponding transition edges (lines 2–7 in Algorithm 3). Compound statements are processed by recursively iterating over the enclosed statements (lines 8–12), and conditional statements are processed recursively for each possible flow of control (from line 13). For example, the `break` and `continue` statements represent an unconditional jump in the program. For handling these keywords, the segments in *CurrSegs* move into the associated set *BreakSegs* or *ContSegs*, respectively. After completing the corresponding loop or `switch` statement, the segments in *BreakSegs* or *ContSegs* will be moved back to the *CurrSegs* set.

**Algorithm 3** Segment graph generation

```
 1: function BUILDSG(CurrStmt, CurrSegs, BreakSegs, ContSegs)
 2:     if isBoundary(CurrStmt) then
 3:         NewSeg ← new segment
 4:         for Seg ∈ CurrSegs do
 5:             AddEdge(Seg, NewSeg)
 6:         end for
 7:         return CurrSegs ∪ { NewSeg }
 8:     else if isCompoundStmt(CurrStmt) then
 9:         for Stmt ∈ CurrStmt do
10:             CurrSegs ← BUILDSG(Stmt, CurrSegs, BreakSegs, ContSegs)
11:         end for
12:         return CurrSegs
13:     else if isIfStmt(CurrStmt) then
14:         AddExpression(IfCondition, CurrSegs);
15:         NewSegSet1 ← BUILDSG(IfBody, CurrSegs, BreakSegs, ContSegs)
16:         NewSegSet2 ← BUILDSG(ElseBody, CurrSegs, BreakSegs, ContSegs)
17:         return NewSegSet1 ∪ NewSegSet2
18:     else if isBreakStmt(CurrStmt) then
19:         BreakSegs ← BreakSegs ∪ CurrSegs
20:         CurrSegs ← ∅
21:         return CurrSegs
22:     else if isContinueStmt(CurrStmt) then
23:         ContSegs ← ContSegs ∪ CurrSegs
24:         CurrSegs ← ∅
25:         return CurrSegs
26:     else if isExpression(CurrStmt) then
27:         if isFunctionCall(CurrStmt) then
28:             return AddFunctionCall(CurrStmt, CurrSegs)          ▷ See Fig. 6a
29:         else
30:             AddExpression(CurrStmt, CurrSegs)
31:             return CurrSegs
32:         end if
33:     else if isLoop(CurrStmt) then
34:         return AddLoop(CurrStmt, CurrSegs)                      ▷ See Fig. 6b
35:     end if
36: end function
```

For brevity, we illustrate the processing of function calls and loops in Fig. 6. The analysis of function calls is shown in Fig. 6a. In step 1 the dashed circle represents the segment set *CurrSegs*. The RISC algorithm detects the function call expression and checks if the function is already analyzed. If not and it is encountered for the first time, the function is analyzed separately, as shown in step 2. Otherwise, the algorithm reuses the cached SG for the particular function. Then in step 3, each expression in segment 1 of the function is joined with each individual segment in *CurrSegs* (set 0). Finally, segments 4 and 5 represent the new set *CurrSegs*.

Correspondingly, Fig. 6b illustrates the SG analysis for a while loop. Again the dashed circle in step 1 represents the incoming set *CurrSegs*. The algorithm detects the `while` statement and analyzes the loop body separately. The graph for the body

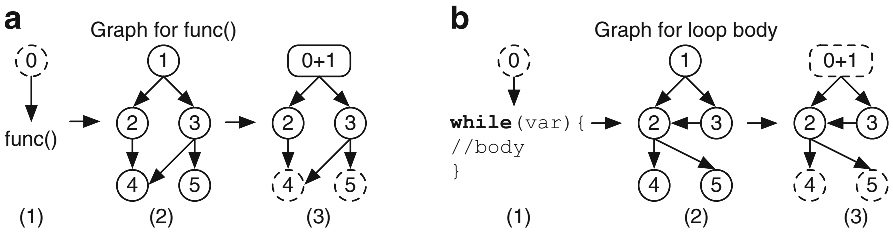


**Fig. 6** Segment graph generation for functions and loops. (**a**) Function call processing. (**b**) Loop processing

of the loop is shown in step 2. Then each expression in segment 1 is joined into the segment set 0, and the new set *CurrSegs* becomes the joined set of 0+1, 4, and 5. Note that we have to consider set 0+1 for the case that the loop is not taken.

## 5.3 Static Conflict Analysis

The segment graph data structure serves as the foundation for static (compile time) conflict analysis. As outlined earlier, the OOO PDES scheduler must ensure that every new running thread is conflict-free with respect to any other threads in the READY and RUN queues. For this, we utilize the RISC compiler to detect any possible conflicts already at compile time.

Potential conflicts in SystemC include data hazards, event hazards, and timing hazards, all of which may exist among the current segments executed by the threads considered for parallel execution [6].

### 5.3.1 Data Hazards

Data hazards are caused by parallel or out-of-order accesses to shared variables. Three cases exist, namely, read after write (RAW), write after read (WAR), and write after write (WAW).

In the example in Fig. 7, if the simulator would issue the threads $th_1$ and $th_2$ in parallel, this would create a race condition, making the final value of $s$ nondeterministic. Thus, the scheduler must not run $th_1$ and $th_2$ out of order. Note, however, that $th_1$ and $th_2$ can run in parallel in the segment after their second `wait` statement if the functions `f()` and `g()` are independent.

Since, data hazards stem from the code in specific segments, RISC analyzes data conflicts statically at compile time and creates a table where the scheduler can then at run time quickly look up any potential conflicts between active segments.

Formally, we define a data conflict table $CT[N, N]$ where $N$ is the total number of segments in the application model: $CT[i, j] = true$, iff there is a potential data conflict between the segments $seg_i$ and $seg_j$; otherwise, $CT[i, j] = false$.

To build the conflict table, the compiler generates for each segment a variable access list which contains all variables accessed in the segment. Each entry is a

**Fig. 7** Example of WAW conflict: Two parallel threads $th_1$ and $th_2$ start at the same time but write to the shared variable $s$ at different times. Simulation semantics require that $th_1$ executes first and sets $s$ to 0 at time $(5, 0)$, followed by $th_2$ setting $s$ to its final value 1 at time $(10, 0)$

```
int s;

thread1()
{ wait( 5, SC_MS);
  s = 0;
  wait(10, SC_MS);
  f();
}

thread2()
{ wait(10, SC_MS);
  s = 1;
  wait(10, SC_MS);
  g();
}
```

tuple (*Symbol*, *AccessType*) where *Symbol* is the variable and *AccessType* specifies read only (R), write only (W), read write (RW), or pointer access (Ptr).

Finally, the compiler produces the conflict table $CT[N, N]$ by comparing the access lists for each segment pair. If two segments $seg_i$ and $seg_j$ share any variable with access type (W) or (RW), or there is any pointer access by $seg_i$ or $seg_j$, then this is marked as a potential conflict.

Figure 8 shows an example SystemC model where two threads $th_1$ and $th_2$ which run in parallel in modules *M1* and *M2*, respectively. Both threads write to the global variable $x$, $th_1$ in lines 14 and 27, and $th_2$ in line 35 since reference $p$ is mapped to $x$. Before we can mark these WAW conflicts in the data conflict table, we need to generate the segment graph for this example. The SG with corresponding source code lines is shown in Fig. 9a, whereas Fig. 9b shows the variable accesses by the segments. Note that segments 3 and 4 of thread $th_1$ write to $x$, as well as segment 8 of $th_2$ which writes to $x$ via the reference $p$. Thus, segments 3, 4, and 8 have a WAW conflict. This is marked properly in the corresponding data conflict table shown in Fig. 10a.

In general, not all variables are straightforward to analyze statically. SystemC models can contain variables at different scopes, as well as ports which are connected by port maps. The RISC compiler distinguishes and supports the following cases for the static variable access analysis.

1. Global variables, e.g., $x$, $y$ in lines 2 and 3 of Fig. 8: This case is discussed above and is handled directly as tuple (*Symbol*, *AccessType*).
2. Local variables, e.g., *temp* in line 10 for Module $M1$: Local variables are stored on the stack and cannot be shared between different threads. Thus, they can be ignored in the variable access analysis.
3. Instance member variables, e.g., $i$ in line 2 for Module $M2$: Since classes can be instantiated multiple times and then their variables are different, we need to distinguish them by their complete instance path added to the variable name. For example, the actual symbol used for the instance variable $i$ in module $M2$ is $m.m2.i$.

```
#include "systemc.h"
int x = 0;
int y;
SC_MODULE(M1) {                                // Module M1
  SC_HAS_PROCESS(M1);
  sc_event &event;
  M1(sc_module_name name, sc_event &e): event(e)
  { SC_THREAD(main); }
  void main() {
    int temp = 0;
    while(temp++<2) {
      wait(1, SC_MS);
      wait(event);
      x = temp;
    }
    wait(3, SC_MS);
    x = 27;
  }
};
SC_MODULE(M2) {                                // Module M2
  SC_HAS_PROCESS(M2);
  int i;
  int &p;
  sc_event &event;
  M2(sc_module_name name, int &pp, sc_event &e):
    sc_module(name), p(pp), i(0), event(e)
  { SC_THREAD(main); }
  void main() {
    do {
      wait(2, SC_MS);
      y = i;
      event.notify(SC_ZERO_TIME);
    } while(i++<2);
    wait(4, SC_MS);
    p = 42;
  }
};
SC_MODULE(Main) {                              // Module Main
  sc_event event;
  M1 m1;
  M2 m2;
  Main(sc_module_name name):
    sc_module(name), m1("m1", event), m2("m2", x, event)
  { }
};
int sc_main(int argc, char **argv) {
  Main m("main");
  sc_start();
  return 0;
}
```

**Fig. 8** SystemC example with two parallel threads in modules *M1* and *M2*

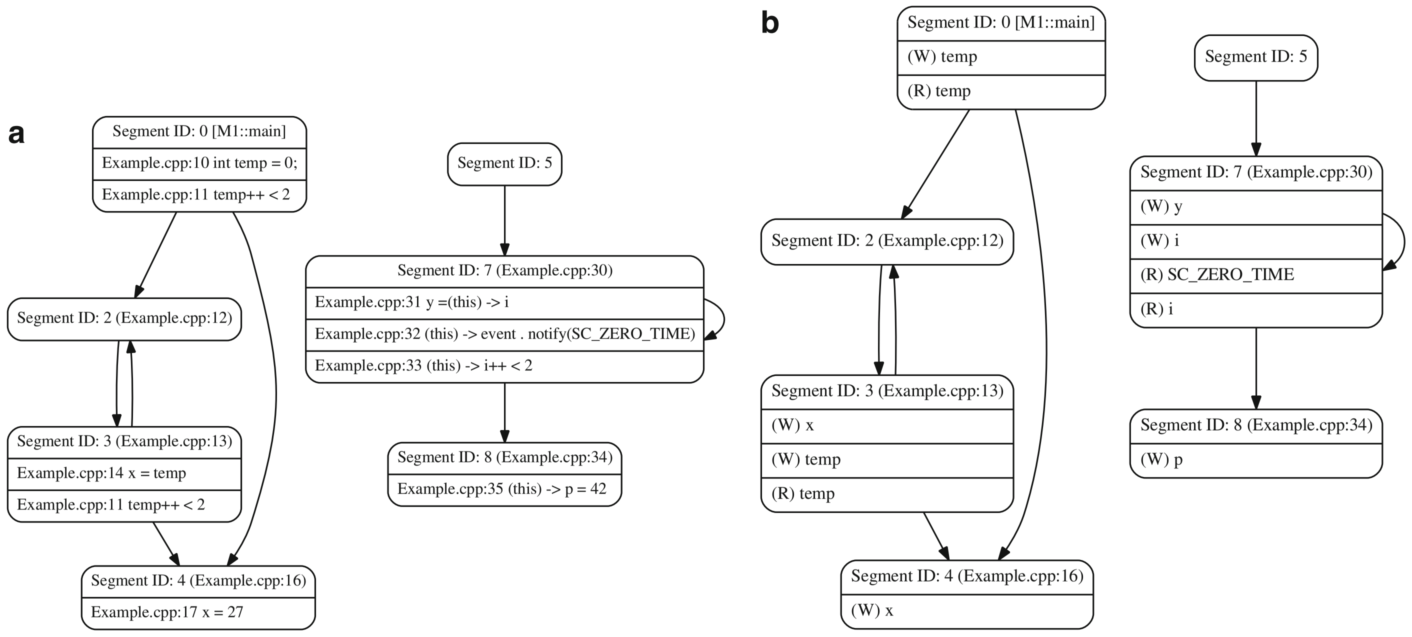

a
Segment ID: 0 [M1::main]
Example.cpp:10 int temp = 0;
Example.cpp:11 temp++ < 2
Segment ID: 2 (Example.cpp:12)
Segment ID: 3 (Example.cpp:13)
Example.cpp:14 x = temp
Example.cpp:11 temp++ < 2
Segment ID: 4 (Example.cpp:16)
Example.cpp:17 x = 27
Segment ID: 5
Segment ID: 7 (Example.cpp:30)
Example.cpp:31 y =(this) -> i
Example.cpp:32 (this) -> event . notify(SC_ZERO_TIME)
Example.cpp:33 (this) -> i++ < 2
Segment ID: 8 (Example.cpp:34)
Example.cpp:35 (this) -> p = 42
b
Segment ID: 0 [M1::main]
(W) temp
(R) temp
Segment ID: 2 (Example.cpp:12)
Segment ID: 3 (Example.cpp:13)
(W) x
(W) temp
(R) temp
Segment ID: 4 (Example.cpp:16)
(W) x
Segment ID: 5
Segment ID: 7 (Example.cpp:30)
(W) y
(W) i
(R) SC_ZERO_TIME
(R) i
Segment ID: 8 (Example.cpp:34)
(W) p

**Fig. 9** Segment graphs generated by RISC for the example in Fig. 8. (**a**) Source code SG for Fig. 8. (**b**) Variable access SG for Fig. 8

**Fig. 10** Data conflict and event notification tables for the example in Fig. 8. (**a**) Data conflict table. (**b**) Event notification table

**a**

| | 0 | 2 | 3 | 4 | 5 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 0 | | | | | | | |
| 2 | | | | | | | |
| 3 | | | T | T | | | T |
| 4 | | | T | T | | | T |
| 5 | | | | | | | |
| 7 | | | | | | T | |
| 8 | | | T | T | | | T |

**b**

| | 0 | 2 | 3 | 4 | 5 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 0 | | | | | | | |
| 2 | | | | | | | |
| 3 | | | | | | | |
| 4 | | | | | | | |
| 5 | | | | | | | |
| 7 | | | T | | | | |
| 8 | | | | | | | |

4. References, e.g., $p$ in line 23 in module $M2$: RISC follows references through the module hierarchy and determines the actual mapped target variable. For instance, $p$ is mapped to the global variable $x$ via the mapping in line 43.
5. Pointers: RISC currently does not perform pointer analysis. This is planned as future work. For now, RISC conservatively marks all segments with pointer accesses as potential conflict with all other segments.

### 5.3.2 Event Hazards

Thread synchronization through event notification also poses hazards to out-of-order execution. Specifically, thread segments are dependent when one is waiting for an event notified by another.

We define an event notification table $NT[N, N]$ where $N$ is the total number of segments: $NT[i, j] = true$, iff segment $seg_i$ notifies an event that $seg_j$ is waiting for; otherwise, $NT[i, j] = false$. Note that in contrast to the data conflict table, the event notification table is not symmetric.

Figure 10b shows the event notification table for the SystemC example in Fig. 8. For instance, $NT[7, 3] = true$ since segment 7 notifies the event $e$ in line 32, which segment 3 is waiting for in line 13.

Note that in order to identify event instances properly, RISC uses the same scope and port map handling for events as described above for data variables.

### 5.3.3 Timing Hazards

The local time for an individual thread in OOO PDES can pose a timing hazard when the thread runs too far ahead of others. To prevent this, we analyze the minimum time advances of threads at segment boundaries. For SystemC, there are three cases with different time increments, as listed in Table 1.

In order for the scheduler to avoid timing hazards, we let the compiler prepare two time advance tables, one for the segment a thread is currently in and one for the next segment(s) that a thread can reach in the following scheduling step.

The current time advance table $CTime[N]$ lists the time increment that a thread will experience when it enters the given segment. For the SystemC example in Figs. 8 and 11a shows the corresponding current time advance table. Here for instance, the `wait(2,SC_MS)` in line 30 at the beginning of segment 7 defines $CTime[7] = (2, 0)$.

**Table 1** Time advances at `wait` segment boundaries

| Segment boundary | Time increment | | Add to $(t', \delta')$ |
|---|---|---|---|
| `wait`($t$) | Increment by time $t$ | $(t : 0)$ | $(t' + t, 0)$ |
| `wait`(event) | Increment by one delta count | $(0 : 1)$ | $(t', \delta' + 1)$ |
| `wait`(immediate event) | No increment | $(0 : 0)$ | $(t', \delta')$ |

**a**

| 0 | 2 | 3 | 4 | 5 | 7 | 8 |
|---|---|---|---|---|---|---|
| (0:0) | (1:0) | (0:0) | (3:0) | (0:0) | (2:0) | (4:0) |

**b**

| 0 | 2 | 3 | 4 | 5 | 7 | 8 |
|---|---|---|---|---|---|---|
| (1:0) | (0:0) | (1:0) | ∞ | (2:0) | (2:0) | ∞ |

**Fig. 11** Current and next time advance tables for the example in Fig. 8. (**a**) Current time advance table. (**b**) Next time advance table

**Table 2** Examples for direct and indirect timing hazards

| Situation | $th_1$ | $th_2$ | Hazard? |
|---|---|---|---|
| Direct timing hazard | $(10 : 2)$ | $(10 : 0)$, next segment at $(10 : 1)$ | Yes |
| | $(10 : 2)$ | $(10 : 0)$, next segment at $(12 : 0)$ | No |
| Indirect timing hazard | $(10 : 2)$ | $(10 : 0)$, wakes $th_3$ at $(10 : 1)$ | Yes |
| | $(10 : 2)$ | $(10 : 1)$, wakes $th_3$ at $(10 : 2)$ | No |

On the other hand, the next time advance table $NTime[N]$ lists the time increment that a thread will incur when it leaves the given and enters the next segment. Since there may be more than one next segment, we list in the table the minimum of the time advances, which is the earliest time the thread can become active again. Formally: $NTime[i] = min\{CTime[j], \forall seg_j$ which follow $seg_i\}$.

For example, Fig. 11b lists $NTime[0] = (1, 0)$ since segment 0 is followed by segment 2 with increment $(1, 0)$ and segment 4 with increment $\infty$.

There are two types of timing hazards, namely, direct and indirect ones. For a candidate thread $th_1$ to be issued, a direct timing hazard exists when another thread $th_2$, that is safe to run, resumes its execution at a time earlier than the local time of $th_1$. In this case, the future of $th_2$ is unknown and could potentially affect $th_1$. Thus, it is not safe to issue $th_1$.

Table 2 shows an example where a thread $th_1$ is considered for execution at time $(10 : 2)$. If there is a thread $th_2$ with local time $(10 : 0)$ whose next segment runs at time $(10 : 1)$, i.e., before $th_1$, then the execution of $th_1$ is not safe. However, if we know from the time advance tables that $th_2$ will resume its execution later at $(12 : 0)$, no timing hazard exists with respect to $th_2$.

An indirect timing hazard exists, if a third thread $th_3$ can wake up earlier than $th_1$ due to an event notified by $th_2$. Again, Table 2 shows an example. If $th_2$ at $time(10 : 0)$ potentially wakes a thread $th_3$ so that $th_3$ runs in the next delta cycle $(10 : 1)$, i.e., earlier than $th_1$, then it is not safe to issue $th_1$.

## 5.4 Source Code Instrumentation

As shown above in Fig. 4 on page 15, the RISC compiler and parallel simulator work closely together. The compiler performs the complex conservative static analysis and passes the analysis results to the simulator which then can make safe scheduling decisions quickly.

More specifically, the RISC compiler passes all the generated conflict tables to the simulator, namely, the data conflict table, the event notification table, as well as the current and next time advance tables. In addition, the compiler instruments the model source code so that the simulator can properly identify each thread and each segment by unique numeric IDs.

To pass information from the compiler to the simulator, RISC uses automatic model instrumentation. That is, the intermediate model generated by the compiler contains instrumented (automatically generated) source code which the simulator then can rely on. At the same time, the RISC compiler also instruments user-defined SystemC channels with automatic protection against race conditions among communicating threads. The source code instrumentation with segment IDs, conflict tables, and automatic channel protection is a part of model "recoding" (i.e., the "R" in RISC). For the future, we envision additional recoding tasks performed by RISC, such as model transformation, optimization, and refinement.

In total, the RISC source code instrumentation includes four major components:

1. Segment and instance IDs: Individual threads are uniquely identified by a creator instance ID and their current code location (segment ID). Both IDs are passed into the simulator kernel as additional arguments to all scheduler entry functions, including `wait` calls and thread creation.
2. Data and event conflict tables: Segment concurrency hazards due to potential data conflicts, event conflicts, or timing conflicts are provided to the simulator as two-dimensional tables indexed by a segment ID and instance ID pair. For efficiency, these table entries are filtered for scope, instance path, and reference and port mappings.
3. Current and next time advance tables: The simulator can make better scheduling decisions by looking ahead in time if it can predict the possible future thread states. This possible optimization is discussed in detail in [5] but remains as a future work item for the current RISC prototype.
4. User-defined channel protection: SystemC allows the user to design channels for custom inter-thread communication. To ensure that such user-defined communication remains safe also in the OOO PDES situation where threads execute truly in parallel and out of order, the RISC compiler automatically inserts locks (binary semaphores) into these user-defined channel instances (which are acquired at entry and released upon leaving) so that mutually exclusive execution of the channel methods is guaranteed. Otherwise, race conditions could exist when communicating threads exchange data.

After this automatic source code instrumentation, the RISC compiler passes the generated intermediate model to the underlying regular C++ compiler which produces the final simulator executable by linking the instrumented code against the RISC extended SystemC library.

## 6 Experimental Evaluation

We now present two SystemC application models as examples and evaluate the performance of DES, PDES, and OOO PDES algorithms on modern multi-core hosts. As DES representative and baseline reference, we will use the open-source proof-of-concept library [16] provided by the SystemC Language Working Group [31] of the Accellera Systems Initiative [1]. As OOO PDES representative, we will use the RISC [21] which is also available as open source [20]. For synchronous PDES, we will use an in-house version of RISC where the out-of-order scheduling features are disabled. To ensure a fair comparison, all simulator packages are based on Posix threads and compiled with the same optimization settings, and of course run on the same host environment.

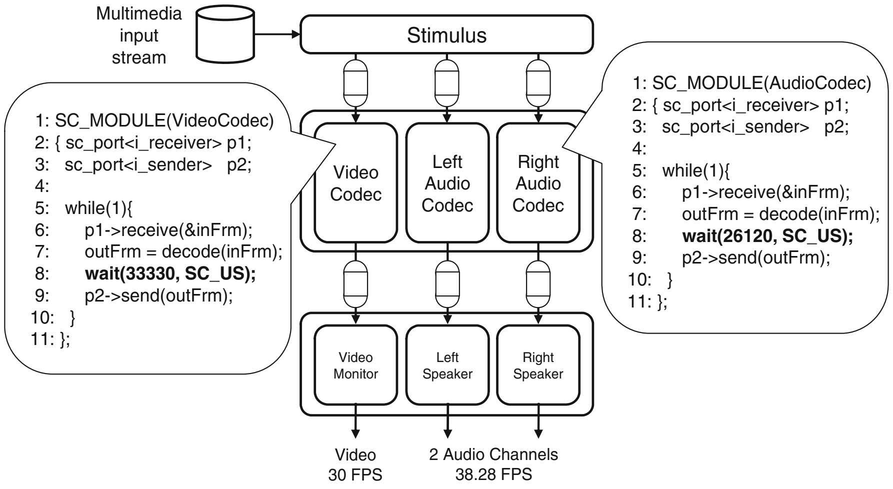
### 6.1 Conceptual DVD Player Example

Our first example is an abstract model of a DVD player, as shown in Fig. 12. While this SystemC model is conceptual only, it is well-motivated and very educational as it clearly demonstrates the differences between the DES, PDES, and OOO PDES algorithms.

As listed in Fig. 12, the SystemC modules representing the video and audio decoders operate in an infinite loop, reading a frame from the input stream, decoding it, and sending it out to the corresponding monitor modules. Since the video and audio frames are data independent, the decoders run in parallel and output the decoded frames according to their channel rate, 30 frames per second video (delay 33.33 ms) and 38.28 frames per second audio (delay 26.12 ms), respectively.

Figure 13 depicts the time lines of simulating the DVD player according to DES, PDES, and OOO PDES semantics. As expected, the DES schedule in Fig. 13a executes only a single task at all times. Synchronous PDES in Fig. 13b is able to parallelize the decoders for the left and right audio channels but cannot exploit parallelism for decoding the video channel due to its different frame rate. Only the OOO PDES schedule in Fig. 13c shows the fully parallel execution that we also expect in reality. Note that the artificially discretized timing in the model prevents PDES from full parallelization. In contrast, OOO PDES with thread-local timing achieves the goal.

Our experimental measurements listed in Table 3 confirm the analysis of Fig. 13. For both experiments, the synchronous PDES gains about 50% simulation speed



```
SC_MODULE(VideoCodec)
{ sc_port<i_receiver> p1;
  sc_port<i_sender>   p2;

  while(1){
     p1->receive(&inFrm);
     outFrm = decode(inFrm);
     wait(33330, SC_US);
     p2->send(outFrm);
  }
};
```

```
SC_MODULE(AudioCodec)
{ sc_port<i_receiver> p1;
  sc_port<i_sender>   p2;

  while(1){
     p1->receive(&inFrm);
     outFrm = decode(inFrm);
     wait(26120, SC_US);
     p2->send(outFrm);
  }
};
```

**Fig. 12** Conceptual DVD player example with a video and two audio stream decoders

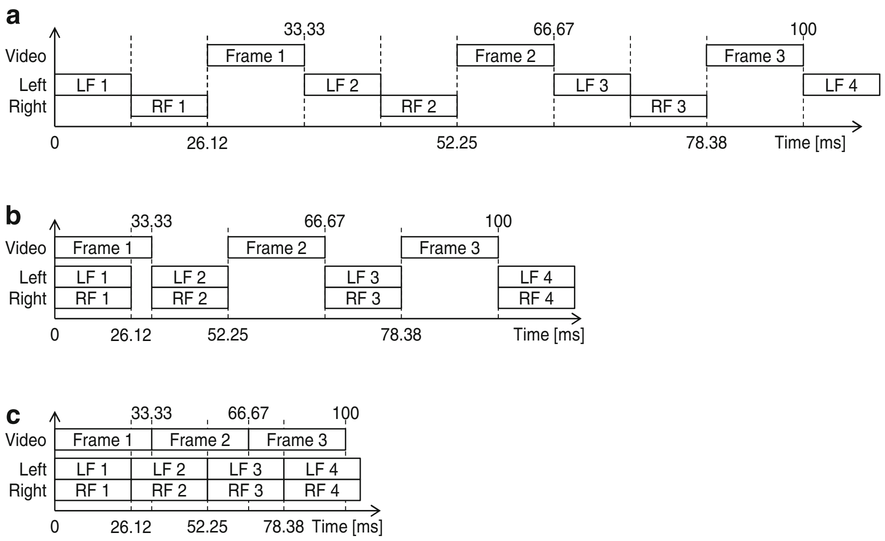


**Fig. 13** Time lines for the DVD player example during simulation. (**a**) DES schedule: sequential execution, only one task at any time. (**b**) Synchronous PDES schedule: parallel execution only at the same simulation times. (**c**) Out-of-order PDES schedule: fully parallel execution (the same as in reality!)

**Table 3** Experimental results for the DVD player example. RISC V0.2.1 simulator performance (Posix-thread based) on a 8-core Intel® Xeon® host PC (E3-1240 CPU, 4 cores, 2 hyper-threads) at 3.4 GHz

| Movie | Simulator | DES | PDES | OOO PDES |
|---|---|---|---|---|
| 10 second stream | Run time | 6.98 s | 4.67 s | 2.94 s |
| | CPU load | 97% | 145% | 238% |
| | Speedup | 1× | 1.49× | 2.37× |
| 100 second stream | Run time | 68.21 s | 45.91 s | 28.13 s |
| | CPU load | 100% | 149% | 251% |
| | Speedup | 1× | 1.49× | 2.42× |

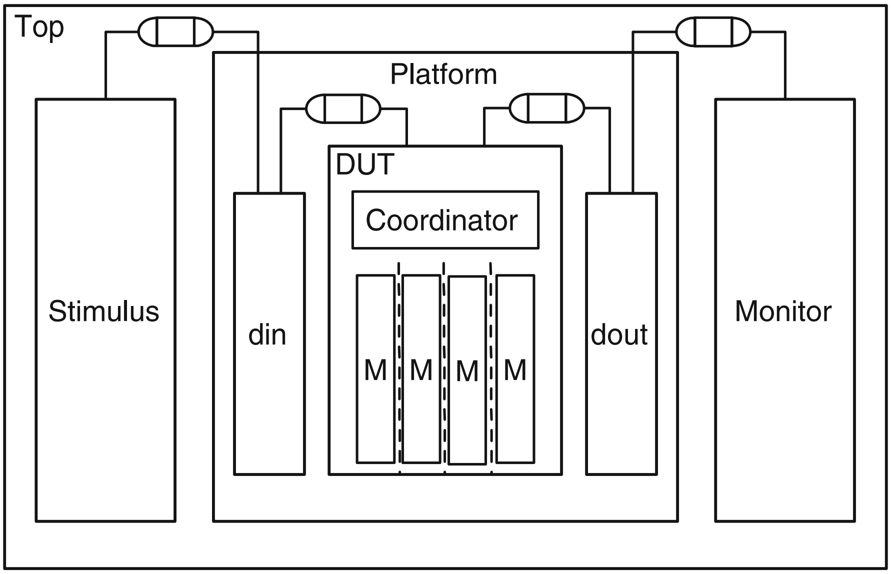


**Fig. 14** Mandelbrot renderer example: Stimulus generates target coordinates in the complex plane for which the DUT renders the corresponding Mandelbrot image and sends it to the Monitor module. In the DUT, a Coordinator distributes slices of the target coordinates to parallel Mandelbrot worker threads and synchronizes their progress

over the reference DES. However, the out-of-order PDES beats the synchronous approach by another 100% improvement.

## 6.2 Mandelbrot Renderer Example

As a representative example of very computation intensive and highly parallel applications, we have evaluated the three-simulation algorithms also on a graphics pipeline model that renders a sequence of images of the Mandelbrot set. Figure 14 shows the block diagram of our Mandelbrot renderer model in SystemC.

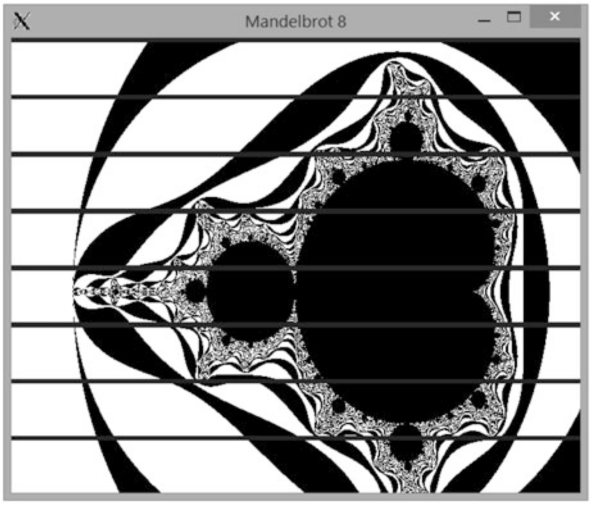


**Fig. 15** Screenshot of the Mandelbrot set renderer in action: The progress of the parallel threads, which collaboratively compute a Mandelbrot set image, can be viewed live in a window on screen. Here, eight SystemC threads compute the eight horizontal slices of the image in parallel. Note that this visualization clearly shows the difference between the sequential DES simulation (which computes only one slice at any time) and the parallel PDES algorithms (which are shown here)

**Table 4** Experimental results for the Mandelbrot renderer example. RISC V0.2.1 simulator performance (Posix-thread based) on a 32-core Intel® Xeon® host PC (2 E5-2680 CPUs, 8 cores, 2 hyper-threads) at 2.7 GHz

| | DES | | PDES | | | OOO PDES | | |
|---|---|---|---|---|---|---|---|---|
| Slices | Run time | Load | Run time | Load | Speedup | Run time | Load | Speedup |
| 1 | 162.13 s | 99% | 162.06 s | 100% | 1.0× | 161.90 s | 100% | 1.0× |
| 2 | 162.19 s | 99% | 96.50 s | 168% | 1.7× | 96.48 s | 168% | 1.7× |
| 4 | 162.56 s | 99% | 54.00 s | 305% | 3.0× | 53.85 s | 304% | 3.0× |
| 8 | 163.10 s | 99% | 29.89 s | 592% | 5.5× | 30.05 s | 589% | 5.4× |
| 16 | 164.01 s | 99% | 19.03 s | 1050% | 8.6× | 20.08 s | 997% | 8.2× |
| 32 | 165.89 s | 99% | 11.78 s | 2082% | 14.1× | 11.99 s | 2023% | 13.8× |
| 64 | 170.32 s | 99% | 9.79 s | 2607% | 17.4× | 9.85 s | 2608% | 17.3× |
| 128 | 174.55 s | 99% | 9.34 s | 2793% | 18.7× | 9.39 s | 2787% | 18.6× |
| 256 | 185.47 s | 100% | 8.91 s | 2958% | 20.8× | 8.90 s | 2964% | 20.8× |

The number of Mandelbrot worker threads is user configurable as an exponent of 2, for example, 4 as illustrated in Fig. 14. Each worker thread computes a different horizontal slice of the image and works independently and in parallel to the others. If enabled in the SystemC model, the progress of the workers' computation is displayed in a window, as shown in Fig. 15.

Table 4 shows the measured experimental results for the Mandelbrot renderer with different numbers of worker threads, one per image slice, as indicated in the first column. Since the example is "embarrassingly parallel" in the DUT and otherwise contains only comparatively little sequential computation, both parallel simulators respond with impressive performance speedups, more than 20× compared to the Accellera reference simulator. With growing parallelism, the simulation speed increases almost linearly with respect to the number of parallel workers, up until to the point where the number of software threads reaches the number of available hardware threads (2 CPUs, 8 cores with 2 hyper-threads each, so 32 in total).

We can also observe that, for this example, the synchronous PDES and the OOO PDES perform the same since the differences are within the noise range of the measurement accuracy. This matches the expectation, because the Mandelbrot workers are all synchronized (locked in) due to their communication with the coordinator thread and thus out-of-order execution cannot be exploited here.

As for scalability, PDES and OOO PDES approaches scale very well, if we base our expectation on the host hardware capabilities and the amount of parallelism exposed in the application model (which is the fundamental limitation of any PDES). Overall, we observe that parallel simulation has the potential to improve simulation speed by an order of magnitude or more. In this book chapter we do not evaluate the overhead of the static analysis incurred at compile time because our current compiler implementation does not produce meaningful results due to its unoptimized ROSE foundation. Generally, compile time for OOO PDES increases moderately, but is amortized by the typically longer and more frequent simulations [6].

## 7 Conclusion

In the era of stagnant processor clock frequencies and the growing availability of inexpensive multi- and many-core architectures, parallel simulation is a must-have for the efficient validation and exploration of embedded system-level design models. Consequently, the traditional purely sequential DES approach, as provided by the open-source SystemC reference simulator, is inadequate for the future when the system complexity keeps growing at its current exponential pace.

In this chapter, we have reviewed the classic discrete event-based simulation techniques with a focus on state-of-the-art parallel solutions (synchronous and out-of-order PDES) that are particularly suited for the de facto and official IEEE standard SystemC. While many approaches have been proposed in the research community, we have detailed the OOO PDES approach [6] being developed in the Recoding Infrastructure for SystemC (RISC) [21]. The open-source RISC project provides a dedicated SystemC compiler and corresponding out-of-order parallel simulator as proof-of-concept implementation to the research community and industry.

The OOO PDES technology stands out from other approaches as an aggressive yet conservative modern simulation approach beyond traditional PDES, because it can exploit parallel computing resources to the maximum extend and thus achieves fastest simulation speed. At the same time, it can preserve compliance with traditional SystemC semantics and support legacy models without modification or loss of accuracy.

**Acknowledgements** This work has been supported in part by substantial funding from Intel Corporation. The authors thank Intel Corporation for the valuable support and fruitful collaboration. The authors also thank the anonymous reviewers for valuable suggestions to improve this chapter.

## References

1. Accellera Systems Initiative. http://www.accellera.org
2. Amdahl GM (1967) Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the spring joint computer conference, AFIPS'67 (Spring), 18–20 Apr 1967. ACM, New York, pp 483–485. doi10.1145/1465482.1465560
3. Cai L, Gajski D (2003) Transaction level modeling: an overview. In: Proceedings of the international conference on hardware/software codesign and system synthesis, Newport Beach
4. Chandy K, Misra J (1979) Distributed simulation: a case study in design and verification of distributed programs. IEEE Trans Softw Eng SE-5(5):440–452
5. Chen W, Dömer R (2013) Optimized out-of-order parallel discrete event simulation using predictions. In: Proceedings of design, automation and test in Europe conference and exhibition (DATE)
6. Chen W, Han X, Chang CW, Liu G, Dömer R (2014) Out-of-order parallel discrete event simulation for transaction level models. IEEE Trans Comput Aided Des Integr Circuits Syst (TCAD) 33(12):1859–1872. doi10.1109/TCAD.2014.2356469
7. Chen W, Han X, Dömer R (2011) Multi-core simulation of transaction level models using the system-on-chip environment. IEEE Des Test Comput 28(3):20–31
8. Chen W, Han X, Dömer R (2012) Out-of-order parallel simulation for ESL design. In: Proceedings of design, automation and test in Europe conference and exhibition (DATE)
9. Dömer R, Chen W, Han X, Gerstlauer A (2011) Multi-core parallel simulation of system-level description languages. In: Proceedings of design automation conference. Asia and South Pacific (ASPDAC), pp 311–316
10. Dömer R, Gerstlauer A, Gajski D (2002) SpecC language reference manual, version 2.0. SpecC technology open consortium. http://www.specc.org
11. Ezudheen P, Chandran P, Chandra J, Simon BP, Ravi D (2009) Parallelizing SystemC kernel for fast hardware simulation on SMP machines. In: PADS'09: proceedings of the 2009 ACM/IEEE/SCS 23rd workshop on principles of advanced and distributed simulation, pp 80–87
12. Fujimoto R (1990) Parallel discrete event simulation. Commun ACM 33(10):30–53
13. Gajski DD, Zhu J, Dömer R, Gerstlauer A, Zhao S (2000) SpecC: specification language and design methodology. Kluwer Academic Publishers, Boston
14. Gerstlauer A (2010) Host-compiled simulation of multi-core platforms. In: Proceedings of the international symposium on rapid system prototyping (RSP), Washington, DC
15. Grötker T, Liao S, Martin G, Swan S (2002) System design with SystemC. Kluwer Academic Publishers, Dordrecht
16. Group SLW SystemC 2.3.1, core SystemC language and examples. http://accellera.org/downloads/standards/systemc
17. Huang K, Bacivarov I, Hugelshofer F, Thiele L (2008) Scalably distributed SystemC simulation for embedded applications. In: International symposium on industrial embedded systems, SIES 2008, pp 271–274
18. IEEE Computer Society (2011) IEEE standard 1666-2011 for standard SystemC language reference manual. IEEE, New York
19. Kaushik A, Patel HD (2013) SystemC-clang: an open-source framework for analyzing mixed-abstraction SystemC models. In: Proceedings of the forum on specification and design languages (FDL), Paris

20. Liu G, Schmidt T, Doemer R Recoding infrastructure for SystemC (RISC) compiler and simulator. http://www.cecs.uci.edu/~doemer/risc.html
21. Liu G, Schmidt T, Dömer R (2015) RISC compiler and simulator, alpha release V0.2.1: out-of-order parallel simulatable SystemC subset. Technical Report CECS-TR-15-02, Center for Embedded and Cyber-physical Systems, University of California, Irvine
22. Mukherjee S, Reinhardt S, Falsafi B, Litzkow M, Hill M, Wood D, Huss-Lederman S, Larus J (2000) Wisconsin wind tunnel II: a fast, portable parallel architecture simulator. IEEE Concurr 8(4):12–20
23. Nanjundappa M, Patel HD, Jose BA, Shukla SK (2010) SCGPSim: a fast SystemC simulator on GPUs. In: Proceedings of design automation conference. Asia and South Pacific (ASPDAC)
24. Nicol D, Heidelberger P (1996) Parallel execution for serial simulators. ACM Trans Model Comput Simul 6(3):210–242
25. Quinlan DJ (2000) ROSE: compiler support for object-oriented frameworks. Parallel Process Lett 10(2/3):215–226
26. Schmidt T, Liu G, Dömer R (2016) Automatic generation of thread communication graphs from SystemC source code. In: Proceedings of international workshop on software and compilers for embedded systems (SCOPES)
27. Schumacher C, Leupers R, Petras D, Hoffmann A (2010) parSC: synchronous parallel SystemC simulation on multi-core host architectures. In: Proceedings of the international conference on hardware/software codesign and system synthesis (CODES+ISSS), pp 241–246
28. Sinha R, Prakash A, Patel HD (2012) Parallel simulation of mixed-abstraction SystemC models on GPUs and multicore CPUs. In: Proceedings of design automation conference. Asia and South Pacific (ASPDAC)
29. Sirowy S, Huang C, Vahid F (2010) Online SystemC emulation acceleration. In: Proceedings of design automation conference (DAC)
30. Stattelmann S, Bringmann O, Rosenstiel W (2011) Fast and accurate source-level simulation of software timing considering complex code optimizations. In: Proceedings of design automation conference (DAC)
31. SystemC Language Working Group (LWG). http://accellera.org/activities/working-groups/systemc-language
32. SystemC TLM-2.0. http://www.accellera.org/downloads/standards/systemc/tlm
33. Weinstock J, Schumacher C, Leupers R, Ascheid G, Tosoratto L (2014) Time-decoupled parallel SystemC simulation. In: Proceedings of design, automation and test in Europe conference and exhibition (DATE), Dresden
34. Yun D, Kim S, Ha S (2012) A parallel simulation technique for multicore embedded systems and its performance analysis. IEEE Trans Comput Aided Des Integr Circuits Syst (TCAD) 31(1):121–131
35. Zhu J, Dömer R, Gajski DD (1997) Syntax and semantics of the SpecC language. In: International symposium on system synthesis (ISSS), Osaka