Parallel SystemC/TLM Simulation of Hardware Components described for High-Level Synthesis

Denis Becker

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 25 mai 2016

Présentée par

Denis Becker

Thèse dirigée par **Matthieu Moy**
et co-encadrée par **Jérôme Cornet**

préparée au sein **du laboratoire Verimag et de STMicroelectronics**
et de **l'école doctorale de Mathématiques, Sciences et Technologies de l'Information et Informatique (MSTII)**

Simulation Parallèle en SystemC/TLM de Composants Matériels décrits pour la Synthèse de Haut-Niveau

Parallel SystemC/TLM Simulation of Hardware Components described for High-Level Synthesis

Thèse soutenue publiquement le **11 décembre 2017**,
devant le jury composé de :

M. Frédéric Pétrot
Professeur, Université Grenoble Alpes, France, Président

M. Rainer Dömer
Professeur, University of California, Irvine, CA, États-Unis, Rapporteur

M. François Pêcheux
Professeur, Université Pierre et Marie Curie, France, Rapporteur

Mme Diana Göhringer
Prof. Dr.-Ing., Technische Universität Dresden, Allemagne, Examinatrice

M. Philipp Hartmann
Senior System Design Methodology Engineer, Intel Corp., Allemagne, Examinateur

M. Tanguy Sassolas
Ingénieur-chercheur, CEA, France, Examinateur

M. Matthieu Moy
Maître de Conférences, Université Lyon 1, France, Directeur de thèse

M. Jérôme Cornet
CAD Dev. Senior Staff Engineer, STMicroelectronics, France, Co-Encadrant de thèse

Contents

— Chapter 1 —

General Introduction

This manuscript explains the work of my CIFRE[1] thesis, a three-years joint work between STMicroelectronics in Grenoble and Verimag in Saint-Martin d'Hères. At STMicroelectronics, I joined the team "System Design and Services" working on the virtual prototyping of systems on chip, notably to enable debugging the embedded software at early development stages. At Verimag, I joined the team "synchronous" working on formal verification, testing, synchronous languages, and modeling. There has been a long collaboration between both teams, further presented in the next chapter, that presents the technical background. The following introduction presents the general context of my thesis.

Systems on Chip

STMicroelectronics produces millions of systems on chip yearly. Systems on chip are used in many areas including home automation, autonomous driving, set-top boxes, smartphones, spacecrafts; in short, in places where computing power is required, but where specific constraints prevent the use of a "classic" computer. Most of the time, those constraints are a lack of space, the need for a low power consumption, or the need for efficiency, which requires custom components. The "chip" part of the "system on chip" is an integrated circuit including various components: memories, peripherals such as sensors, Wi-Fi or Bluetooth interfaces, hardware acceleration blocks, and so on, but most importantly one or several generic-purpose Central Processing Units (CPUs). This last component, the CPU is what justifies the term "system" on chip. The CPU executes a program, the embedded software. The system results from the execution of the embedded software on the chip.

[1] Convention Industrielle de Formation par la REcherche — Industrial partnership of learning/training by research.

By definition, the software part can be updated during the lifetime of the product; for example, some set-top boxes were updated to run video games, even though that was not planned at first. Practically, updating the software means changing data in the chip memory. The case of the hardware is very different. The hardware is an integrated circuit, printed on a semiconductor material. To mass-produce this circuit, the process consists in producing a mask of the chip. The mask can be seen as a stamp, used to reproduce the same circuit. The production of a mask costs several million dollars: this step is critical in the production flow. With this in mind, one can easily see why the software can be updated — it costs some man/days of work to develop the new version of a program — and why the hardware cannot be updated.

The miniaturization of transistors (*cf.* Moore's law) enabled semiconductor companies to put more powerful components, and also more functionalities on a single chip. The ability to produce complex products mainly depends on two factors: abstraction and computer-aided design. The abstraction characterizes the model of a system. A model at a low abstraction level is close to the real system; it models the system in details. Such level requires a comprehensive knowledge of the internal behavior of the system. A high abstraction level does not model the details of the system, an approximate representation is used. High levels enable dealing with very complex systems as they are simpler to understand, but they are not sufficient to actually make the system. The notions of "high" and "low" levels are relative, they depend on the context.

In our context, the Register Transfer Level (RTL) description of a system on chip is considered to be at a low level of abstraction. It represents the system in a detailed way, which enables the generation of plans for an integrated circuit from its RTL description. This generation is done by an automatic tool. This is an example of computer-aided design. Another example of computer-aided design is a High Level Synthesis (HLS) tool. An HLS tool generates an RTL design from a function written in C or C++, written at a high level of abstraction. The functional description of the block is behavioral. It describes *what* the block does, but not *how*; the micro-architecture of the RTL design is decided during the generation process, by the hardware designer.

Simulation of Systems on Chip

The embedded systems market has been growing very fast during the last decades. This growth is a consequence of such systems being in many promising and already successful areas. That includes home automation, smart items (*e.g.* phones, watches, televisions) and autonomous cars, amongst others. Anticipation is one of the most important quality in a growing market: not only in terms of "what new feature will be the determining factor for our product", but in terms of solving problems *before* the product is available. Moreover, in systems on chip design, it is not possible to wait for the availability of the physical chip to start testing because the cost of an error is too expensive (several million dollars to produce a mask).

A model of the chip is used to start testing the system on chip (hardware and software) before the chip is available. This model is developed as soon as possible, even though the full specifications of the chip are not available. It must be possible to run the embedded software on the model of the chip, and to identify potential issues that would happen on the real chip. In summary, the essential properties of the required simulations are:

Reproducibility The same inputs must produce exactly the same outputs. With reproducibility ensured, when a bug is encountered, the user can reproduce it to identify and solve the problem.

Speed Simulations must be fast enough to be reasonably used. When software developers are fixing the code, they need to quickly get a result.

Debug A debug environment must be available to quickly find the root cause of bugs (when a piece of software runs on a chip and a bug occurs, it is hard to get more information than "it didn't work").

Accuracy The model must be accurate enough to enable the identification of software bugs or hardware synchronization issues.

The Transaction Level Modeling (TLM) abstraction level has been created specifically to answer those needs. A TLM model is at a high abstraction level, and it is implemented using the SystemC library in C++. In particular, such a simulation satisfies each point: simulations are reproducible thanks to SystemC coroutine semantics (**Reproducibility**); they are fast as they are written at a high level of abstraction, plus it also benefits from the C++ execution speed (**Speed**) and debug environment (**Debug**); and they model enough of the chip by definition (**Accuracy**). The embedded software is run in the TLM model of the chip without any modification.

Simulation Speed

The development of systems on chip involves many different teams working together. Some components of a chip are bought from external companies, some others are internally designed from scratch, and some others are reused or improved from previous chips. Even for components that are internally designed, the design method is specific to the type of component. For example, an Intellectual Property (IP) block for video decoding, has to be first mathematically designed, described and validated, before it is described in terms of hardware components. We mentioned the HLS process, that generates a hardware description of a component based on a functional description. Such process is typically used for this type of IP blocks. Since this process implies to write C/C++ code, and since a TLM description of a chip also consists in C++ code, developers reuse the first piece of code into the second one. Indeed, the development of a TLM model must be fast (a few months) and there is no time to re-write existing pieces of code. The fact that the same code is re-used also ensures consistency between the different models. This example illustrates that when we say "a model", we actually refer to different models for different components that are simulated together in order to check the whole system.

The growing complexity of systems on chip is reflected in their models. And more complex models lead to slower simulations, while the time-to-market constraints become shorter in a tight market. Reducing the simulation time of models became a major research challenge for more than two decades. One solution to speed up computer programs is to exploit the parallel computation resources available. Indeed, today's computers often have several CPUs, as well as powerful Graphics Processing Units (GPUs). However, exploiting parallel resources for TLM simulations is hard, and is a major research concern; this thesis falls within this context.

This manuscript is organized as follows. Chapter 2 presents technical background information that are part of the state of the art, and exposes the problem statement. Before the presentation of existing parallel simulation approaches for SystemC models, we examine the profile of an industrial case study from STMicroelectronics in Chapter 3. For this profiling, we developed a profiling and visualization tool for SystemC. Then, Chapter 4 presents the existing parallel simulation approaches. Chapter 5 introduces our proposed approach and details a proposed algorithm used for communication between concurrent simulators. Finally, Chapter 6 details the proposed approach based on the HLS design flow using the previous communication algorithm. Performance results are presented in this last chapter.

In summary, the contributions of this thesis are:

- Identification of profile metrics on SystemC simulations, and development of a visualization tool and instrumentation of a SystemC kernel in order to obtain the corresponding measurements.
- Profiling results on an industrial case-study from STMicroelectronics and analysis of the results, put in perspective with existing parallel SystemC simulation approaches.
- Identification and implementation of an efficient First In, First Out (FIFO) communication algorithm for intensive data exchanges (unidirectional), and adaptation of this algorithm to SystemC simulations, called Fast Ordered First In, First Out data exchaNge (FOFIFON).
- Development of DistemC, a non-intrusive multi-process infrastructure of SystemC simulators communicating through FOFIFON structures.
- Application of DistemC, to simulate in parallel the SystemC/TLM model of a hardware acceleration block (a JPEG decoder) whose behavior is described for an HLS design flow.

— Chapter 2 —

Background and Problem Statement

2.1 General Information

The simulation of systems on chip involves to use multitasking. Indeed, hardware systems are intrinsically parallel. In this section, we remind different multitasking concepts in computer science, that are necessary to understand SystemC simulations. We also emphasis the difference between *describing* parallel systems, and *executing* a program in parallel. This difference is a key to understand why parallel simulation of systems on chip is a major research concern. Finally, we introduce discrete event simulation.

2.1.1 Multitasking Concepts

2.1.1.1 Coroutines, threads and processes

A *task* is a generic term that refers to a unit of execution. This unit of execution is a sequence of instructions with context information (local variables, stack of function calls, *etc*). When a system consists of multiple tasks, a *scheduler* manages their execution. The scheduler can either be *preemptive*, which means it has the ability to pause and resume tasks, or *cooperative*, which means it cannot. In cooperative multitasking, the scheduler must wait until a task pauses itself before having the chance to run another one. The cooperative term comes from the fact that a task must cooperate, *i.e.* pause itself sometimes, to enable the execution of other tasks. The fact that a tasks suspends itself is called a *yield*. In cooperative multitasking, tasks are called *coroutines*. In preemptive multitasking, tasks generally consists in *threads* or *processes*.

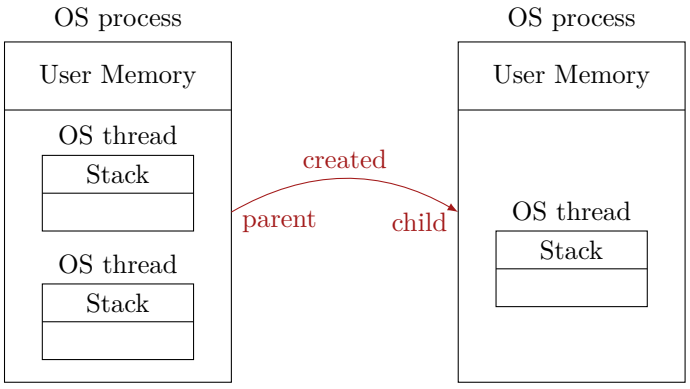


Figure 2.1: OS processes and OS threads.

Figure 2.1 reminds the major differences between OS processes and threads, that lies in the amount of context information. A process (the child) is created by another process (the parent). The child and the parent processes do not share the same memory space. Thus, if the child process modifies the value of a variable, the parent process will not see

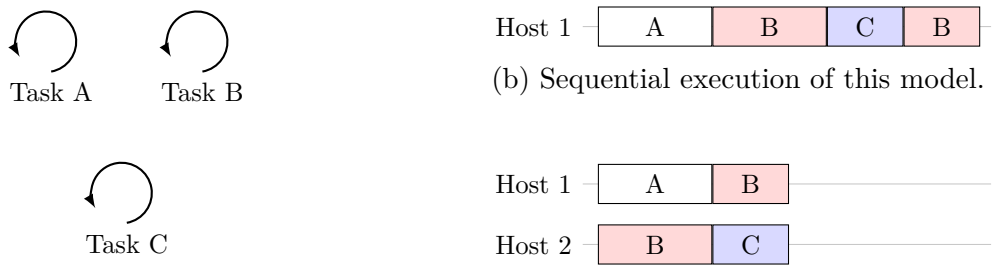
the modification. A thread is created by a process. Each thread belonging to the same process share the same memory space. Thus, if a thread modifies the value of a variable, other ones will eventually see the modification. Threads are often referred to as lighter concurrency features than processes, because their creation and context switching are faster, and communication between them is easier because they share the same memory space.

2.1.1.2 Atomicity, race conditions and critical sections

When multiple tasks are running, atomicity is required to build correct functions. Etymologically, something atomic is something that cannot be divided. In computer science, an *atomic* operation is a set of instructions that either occur completely or does not occur at all. This means that none of the intermediate states of an atomic operation must be observable from the *rest of the system* (*i.e.* the scheduler and the other tasks scheduled by the same scheduler).

A *race condition* is a situation where the observable state of an application changes depending on the execution order of tasks. A typical example of race condition is when two threads increment a shared variable without any "protection". The increments performed on a variable can be translated into load, add and store assembly instructions. Individually, those instructions happen atomically, but they can interleave such as the value is only incremented once at the end. To avoid this race condition, operations on the shared variable should be performed in a *critical section*. At any given time, there can be at most one task that runs code from a critical section. This is called *mutual exclusion*, and it is typically set up with a *mutex* variable or by using fences.

2.1.2 Parallel Description and Parallel Execution



(a) Description of a parallel model.

(b) Sequential execution of this model.

(c) Parallel execution of this model.

Figure 2.2: Example of parallel description, either executed sequentially or in parallel.

We already stated that hardware systems are parallel: indeed, each component is working at the same time as the others. Thus, in order to describe hardware systems, there is a need for parallel *description*. However, parallel description does not imply parallel *execution*: it is completely possible to execute sequentially a model that is described with

parallel components. Figure 2.2 shows an example on which three processes are described (a), and executed either sequentially (b) or in parallel (c).

2.1.3 Discrete Event Simulation

The principle of simulation is to model the *evolution* of a dynamic system through time. Discrete Event Simulation (DES) must be distinguished from continuous simulation. Continuous simulations are time-driven: the simulated time evolves with a specific step (possibly changing during simulation) and the components of the simulation evolve with it. In DES, which is event-driven, the state of components changes in response to event occurrences. Thus, the simulated time "jumps" from one instant to the other, also in response to event occurrences. In summary: time is an input parameter in continuous simulation, while in DES time is an output value. A consequence is that in DES, it is not possible *a priori* to know the different simulated time instants the simulation will go through.

Algorithm 1 Typical loop of a discrete event simulator.

```
1:  sim_time ← 0
2:  while runnable_processes not empty do
3:      while runnable_processes not empty do
4:          p ← runnable_processes.pop( )
5:          p.run( )
6:      end while
7:      if events not empty then
8:          e ← events.pop( )
9:          sim_time ← e.timestamp( )
10:         e.trigger( )          ▷ Processes sensitive to e are put in runnable_processes
11:     end if
12: end while
```

A DES simulator enables the execution of a model. The simulator must maintain a list of processes, a list of events (generally a priority queue), and a variable for the current simulated time value. An *event* is an object that has a timestamp value, corresponding to the simulated date it occurs. Processes can indicate that they have to run computation when a specific event occurs: in this case, they are called *sensitive* to such event. The queue of events is sorted in growing timestamps. The general principle of a discrete event simulator is presented on Algorithm 1. It consists in a loop program. First, each process from the model is run. During their execution, they register events to occur at future simulated time instants. When the execution of processes is over, the first event from the list of events is retrieved, the simulated time variable takes the event timestamp as value, and processes that are sensitive to this event are triggered. The simulation stops either when the simulated time has reached a specific value, or when there are no more processes to run.

2.2 The SystemC Library

2.2.1 Presentation of SystemC

SystemC is a C++ hardware modeling library. Since 2005 it is under IEEE standard, updated in 2011 as IEEE 1666–2011 [1]. The standard notably defines the requirements to implement a SystemC simulator. A SystemC simulator is a discrete event simulator. The SystemC library can be used to write models of hardware at different levels of abstraction. Each level of abstraction uses different constructs from SystemC. Transaction Level Modeling (TLM) is one of them, and is described as a part of the SystemC standard [1, pp. 413–561]. The goal of this section is not to exhaustively define the features of SystemC but some key ones, and to explain how the SystemC simulator works.

2.2.1.1 Elements for Discrete Event Simulation

SystemC offers different types of tasks to describe hardware parallelism: `SC_METHOD`, `SC_THREAD` and `SC_CTHREAD`. The SystemC clocked threads (`SC_CTHREAD`) are mentioned for exhaustiveness, but are only used in cycle-accurate models which are not in the scope of this manuscript. Whenever the expression *SystemC process* is used, it means indifferently SystemC threads or methods.

Both SystemC methods and threads enable the description of concurrent behaviors, their only difference lies in how this description is made. SystemC methods are implemented as stackless processes, *i.e.* simple C++ function calls. As a consequence, their state is not recorded from one run to the other. This execution model is relevant for simple behaviors, but for complex ones they are not comfortable to use, because manual recording of the current state would be necessary. For this reason, SystemC also offers threads, which are implemented as stackful processes.

Simulated time is modeled with the `sc_time` type. It models the fictional time taken by the simulated platform (*i.e.* estimated time that the real chip would take). It is completely disconnected from the wall-clock time, which is the human perception of time passed while a simulation is running. Figure 2.3 illustrates the duality between wall-clock time and simulated time. A SystemC method can schedule a future run for itself, after a certain simulated time elapse, using the function `next_trigger`. Similarly, a SystemC thread may suspend itself for a certain amount of simulated time, using the function `wait`.

Events are modeled in SystemC with the `sc_event` class. Events can be triggered explicitly in a SystemC process, either immediately or after a simulated time delay. They are used as conditions to wake up suspended SystemC processes. Each SystemC process can have a sensitivity list, which is a list of events. Each time an event present in

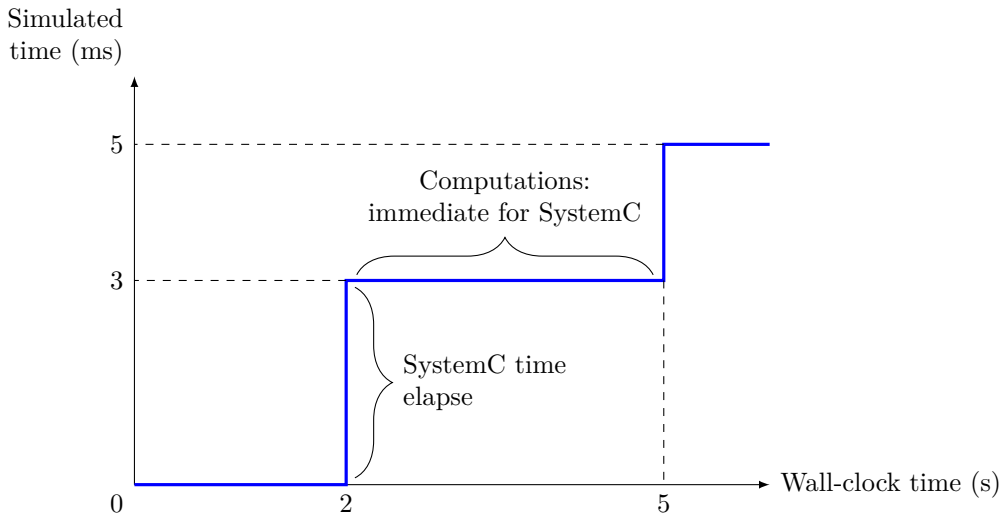


Figure 2.3: Evolution of wall-clock time versus simulated time. In SystemC, as in discrete event simulation, computations are "immediate" with respect to simulated time, and simulated time progresses "immediately" with respect to wall-clock time.

a suspended process' sensitivity list is triggered, it causes this process to run. This sensitivity list can be specified both statically when the process is created or dynamically during the process execution. For SystemC methods, the dynamic sensitivity is described with the family of functions `next_trigger`. For SystemC threads, it is specified with the family of functions `wait`.

Finally, the `sc_main` function is the equivalent of the `main` function in a standard C++ program. The role of this function is to instantiate the model components, bind them together and start the simulation by calling the function `sc_start`. The instantiation and binding of components is called the *elaboration phase* in SystemC. It stops when `sc_start` is called, then starts the *simulation phase*. The hierarchy and bindings of components can only be modified in the elaboration phase.

2.2.1.2 Elements for Hardware Modeling

Hardware systems are represented as a hierarchical set of blocks which are bound together. The C++ language, as an object-oriented language, already has the required constructs for this. Indeed, each kind of hardware component can be described with a C++ class, and each hardware component is an instance of the corresponding class. To encapsulate common behavior of all hardware components, SystemC offers the base class `sc_module`. Each part of the hardware system identifiable as a building block will be described in a class inheriting from SystemC's `sc_module`, as shown on Figure 2.4.

A module communicates with other modules through its ports (`sc_port` in SystemC) and exports (`sc_export`). A port enables access to a set of services declared in an

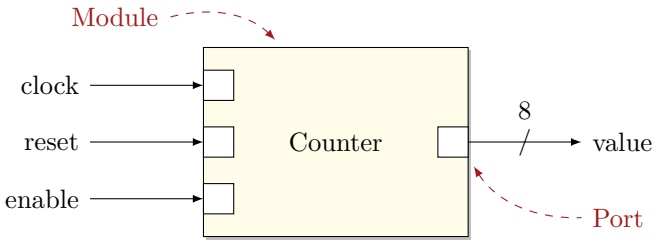


Figure 2.4: Example of an 8-bit counter represented in SystemC. This representation is at the RTL abstraction level. This manuscript is about TLM models, however this example is given to illustrate that similar SystemC constructs are used at RTL and TLM. However, we see in further examples (*e.g.* Figure 2.8) that, for example, a "module" is much more abstract at TLM than at RTL.

`sc_interface`. An export is used to enable access to an implementation of a SystemC interface. A SystemC interface is a set of functions.

Depending on the level of abstraction of the model, the elements defined for communication in SystemC are used differently. As an example, we compare the two main communication paradigms in SystemC. The first one uses *channels* inheriting from SystemC's `sc_prim_channel` class, as shown on Figure 2.5. In this example, there is a shared channel (called "Channel" on the figure) that stores a state. The channel has a current state (relatively to simulated time) and a next state. When "Module 1" writes on the channel, it modifies its next value. When "Module 2" reads the channel, it reads its current value. Thus, both modules do not access the same memory location. This introduces a specificity of SystemC simulation (with respect to discrete event simulation) called the *update phase*. It consists in affecting the "next" value to the "current" value. This does not correspond to a simulated time progress, but simply as a "delta" progress in the sense of a very small amount of time (in fact, the implementation uses a zero time value). This update phase creates a *delta cycle*. We present how the SystemC simulator handles delta cycles further.

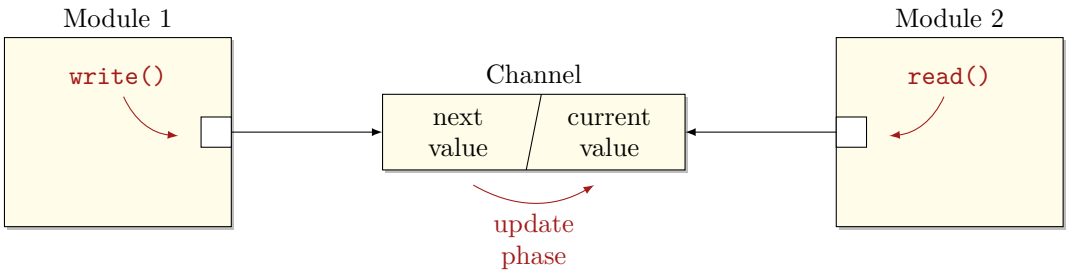


Figure 2.5: Communication between SystemC processes using primitive channels.

Using such channels creates delta cycles (thus slowing down the simulation), consequently, higher abstraction levels use an alternative communication paradigm. At the TLM abstraction level, communication between modules is done with Interface Method Calls (IMCs), as shown on Figure 2.6. This example presents one of the methods defined in the TLM-2.0 interface: the *blocking transport* interface (`b_transport`). When "Mod-

ule 1" writes to "Module 2", it *directly* calls a method from "Module 2". This enables faster communication, simply because less operations are done. The blocking transport interface is typical of Loosely-Timed (LT) TLM models. We further present TLM and LT in the next section, but introduced them here to illustrate that the SystemC library is used differently depending on the abstraction level.

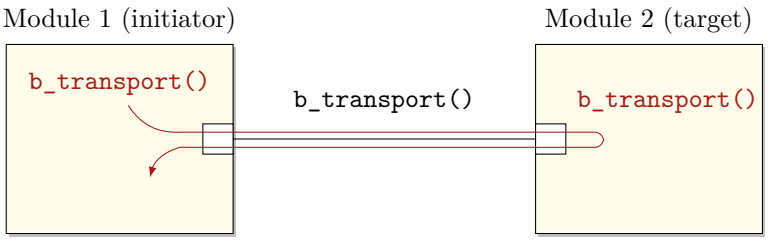


Figure 2.6: Communication using IMC, at the TLM abstraction level.

2.2.2 SystemC Simulation Kernel

A SystemC simulation kernel is an implementation that enables to run a SystemC model as described in the standard. The Accelera Systems Initiative (ASI), formerly Open SystemC Initiative (OSCI), is the consortium of companies that wrote the SystemC standard. They propose a reference implementation of the SystemC library, including a simulation kernel [1] (under the Apache 2.0 License since 2016). Commercial implementations also exist with different extra features.

A SystemC simulator contains a scheduler. It has to manage the execution of the different SystemC processes described in the model. According to the standard, a SystemC scheduler must fulfill coroutine semantics [1]. This means that each section of code between two yield (`wait`) statements must be atomic. To do so, a scheduler can either use cooperative multitasking, or use preemptive multitasking but behave *as if* it was using cooperative. In the latter case, the scheduler must notably ensure that new race conditions are not introduced because of the preemptive scheduling or physical parallelism [2]. This does not prevent multiple processes from running in parallel, but there must exist a sequential scheduling that reproduces the *same* simulation. In practice, one must ensure that shared variables are not modified concurrently by different processes. This semantic choice has been made to ease the writing of models by avoiding many race conditions between processes of a model. Indeed, it is frequent for modules to have several threads using the module's fields (in C++ terms the class attributes). With coroutine semantics, the different processes can use these shared variables without any protection (such as mutual exclusion). Another rule for a SystemC scheduler is that it must ensure the determinism of simulations: running multiple simulations with the same input must produce the same result. This makes SystemC simulations reproducible, which is a key property for debugging.

[1] `http://www.accellera.org/downloads/standards/systemc`

The ASI implementation consists in the sequential execution of the processes in the model. An advantage of a sequential implementation is that coroutine semantics are easy to fulfill and the determinism is easy to ensure, provided that multiple runnable processes are always run in the same order. Moreover, sequential implementation leads to a lightweight context-switch because it can be performed without underlying system calls. An obvious drawback is that it does not exploit the multiple cores of the host machine running the simulation. With the increasing size of models, the simulation time is a major issue of complex hardware simulation. Achieving parallel execution of SystemC simulations is not straightforward, notably because of the SystemC scheduler requirements, and is a major research concern.

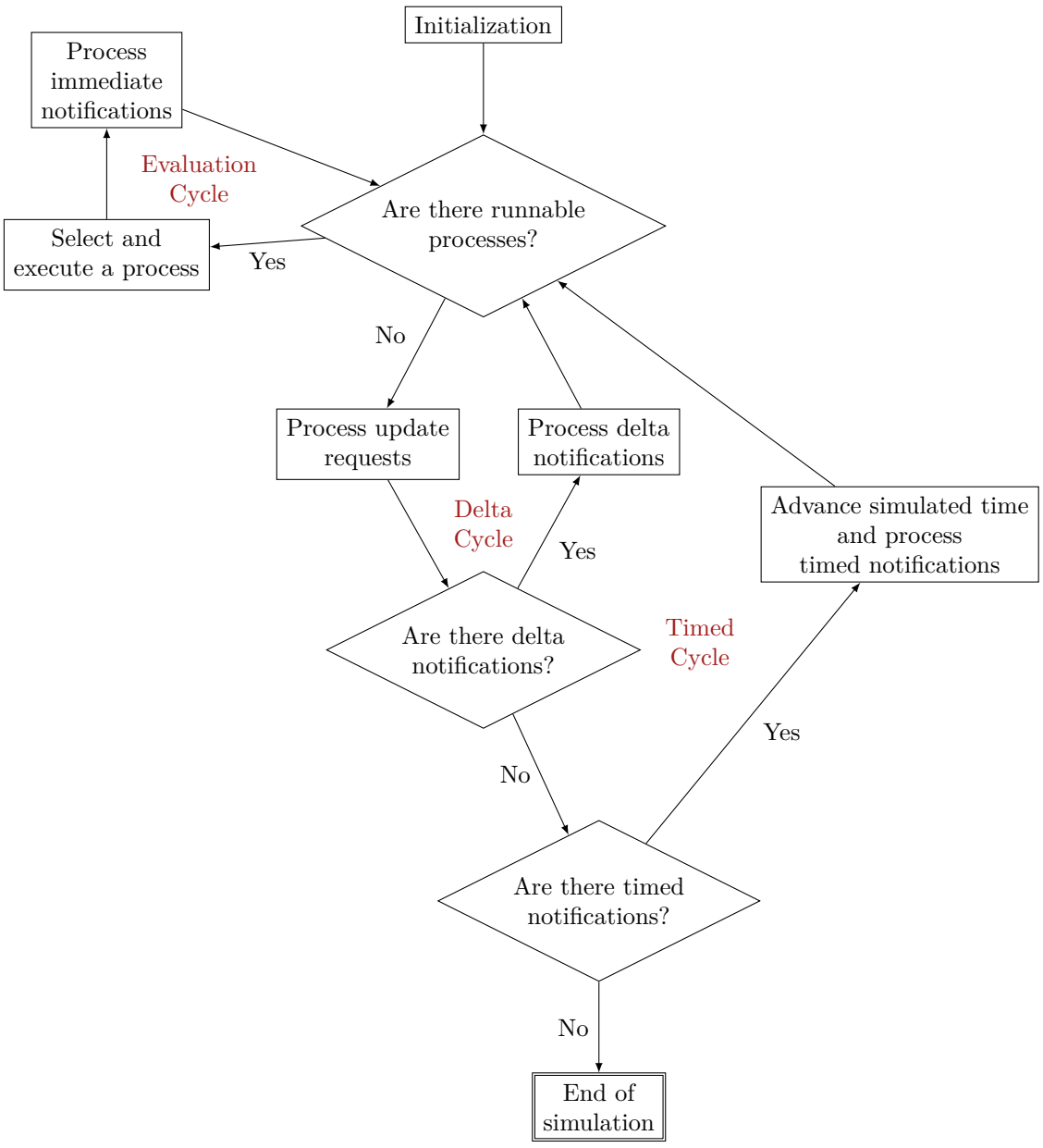


Figure 2.7: Behavior of a SystemC scheduler.

Figure 2.7 shows the expected behavior of a SystemC scheduler during the simulation phase (the elaboration phase has already been done at this point). The scheduler starts with an initialization phase that we do not detail here, but it consists in running the processes of the model once, to let them reach an "initialized" state. Then starts the simulation loop. As long as there are runnable processes, the scheduler remains in the same *evaluation phase*. During this phase, processes can trigger other ones by notifying immediate events. When there are no more runnable process, update requests are processed: this is the update phase. The value of primitive channels, previously presented, is updated in this phase. If the update phase created delta notifications (*i.e.* values of channels were actually changed) they are processed, and an evaluation phase starts again. If no such notification exists, the scheduler checks for timed notifications, *i.e.* notifications that can modify the value of simulated time. If there are timed events, the earliest one is picked, the simulated time is set to its time, and notifications are processed. The evaluation phase starts again. Otherwise, the simulation is over. The simulation also ends if a specific simulated time value has been reached (it can be defined when calling `sc_start`).

2.3 Transaction Level Modeling

We previously presented the SystemC library, that can be used to model hardware systems at various levels of abstraction. In this manuscript, we address the case of models at the Transaction Level Modeling (TLM) abstraction level. It has been created to enable modeling systems on chip already at early stages of the development. It answers the need for a fast simulation, quickly made available. But even when the real chip is available, on-chip software or hardware debug is a tedious task because there is little observability of what happens inside the chip. Thus, TLM models are useful during all the lifetime of the product.

Figure 2.8 presents a simple platform as represented at the TLM abstraction level. We present the different elements from this representation in the following section. This figure is "unzoomed" compared to Figure 2.4. Components like bit-adders, counters or clock signals are not represented at this level: they are part of the microarchitecture (which is often not known when the TLM model is written).

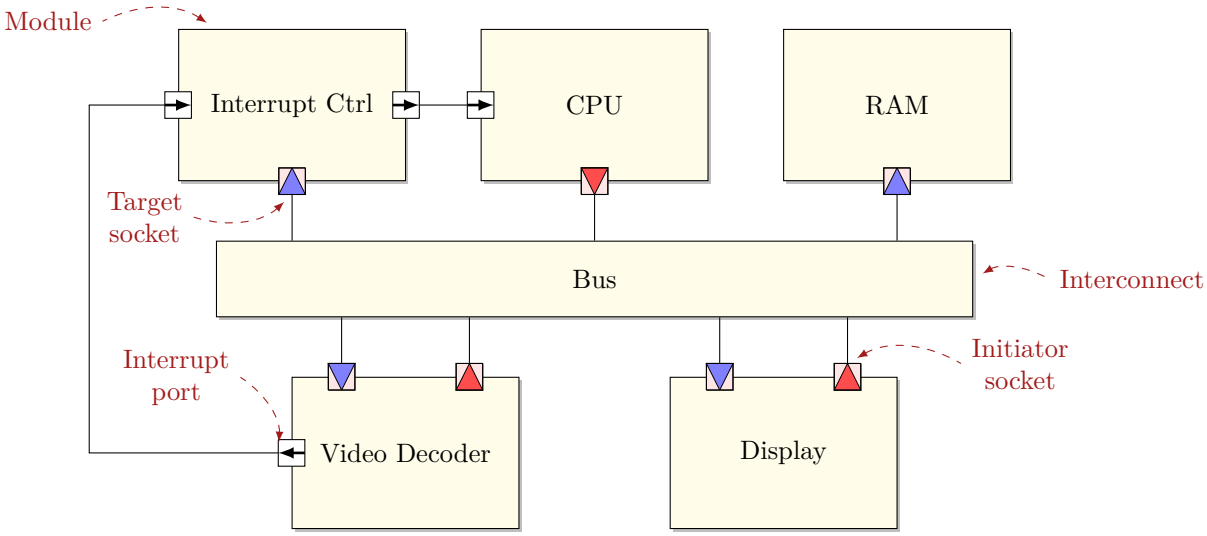


Figure 2.8: Example of a simple platform at the TLM abstraction level.

2.3.1 The TLM Layer

The SystemC standard has been extended to define the TLM layer, which adds core interfaces and utilities to the SystemC library, mainly specifying interprocess (in the sense of SystemC processes) communication. The main goal of the TLM standard is to ensure interoperability to enable the use of hardware models coming from different vendors. Two complementary versions of TLM are currently used: TLM-1.0 and TLM-2.0.

2.3.1.1 Key Concepts of TLM

A *transaction* is an atomic exchange of information between an initiator module and one or multiple targets. Most of the time, a transaction consists in a read or a write operation. Since TLM-2.0, a *generic payload* is defined to be the base information object. It contains a set of default fields to address most of the bus communication protocols (*e.g.* address field, data field). Generic payload also support protocol-specific extensions. A transaction is *transported* from one module to the other through `transport` methods. We previously presented an example on how transport methods work, using direct method calls in the target module. TLM also supports the modeling of *interrupts*, which are boolean wires asynchronously triggering actions on a state change.

To send or receive transactions, TLM offers initiator and target sockets, as shown on Figure 2.8. In terms of SystemC, an initiator socket is an `sc_port` that contains an `sc_export` and a target socket is an `sc_export` that contains an `sc_port`.

The TLM standard defines initiator, target and interconnect components. An initiator component has at least one initiator socket, a target component has at least one target socket and an interconnect component has at least one of each. For example, the Central Processing Unit (CPU) from Figure 2.8 is an initiator component, the memory is a target component and the bus is an interconnect. Figure 2.9 represents a possible processing of a transaction with an interconnect component. Even in this case, the transport is still a function call, performed *in* the final target *by* the thread from the initiator component.

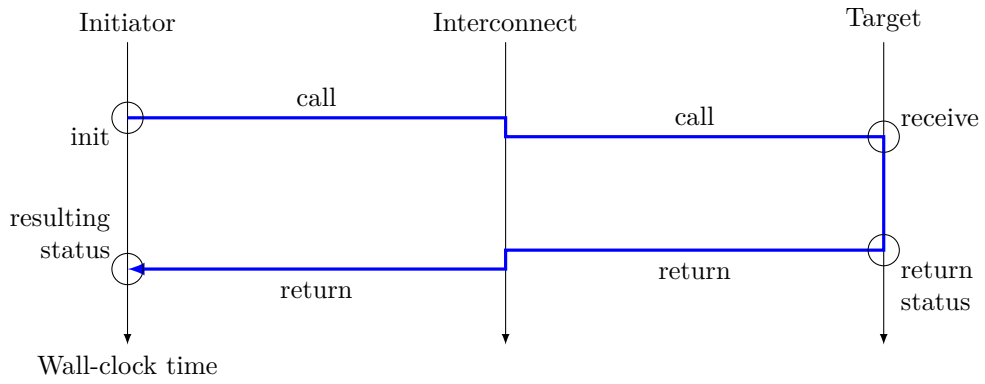


Figure 2.9: Diagram illustrating the transport of a transaction through an interconnect component.

We remind here that the "wires" represented on Figure 2.8 are modeled in TLM with transport methods, using Interface Method Calls (IMCs). Consequently, performing a TLM transaction is equivalent to calling a function in the final target component.

2.3.1.2 TLM Coding Styles

The TLM standard defines two coding styles: Loosely-Timed (LT) and Approximately-Timed (AT). The major difference between those coding styles are how transactions are performed. In the LT style, transactions only has two timing points: before the transaction (call) and after the transaction (return). For this coding style, the *blocking transport* interface is used, which means that the initiator cannot continue its execution before having received the response to a transaction. Figure 2.10 illustrates how transactions occurs with the LT coding style. In the AT style, transactions can be split in phases, each associated with two timing points (before and after).

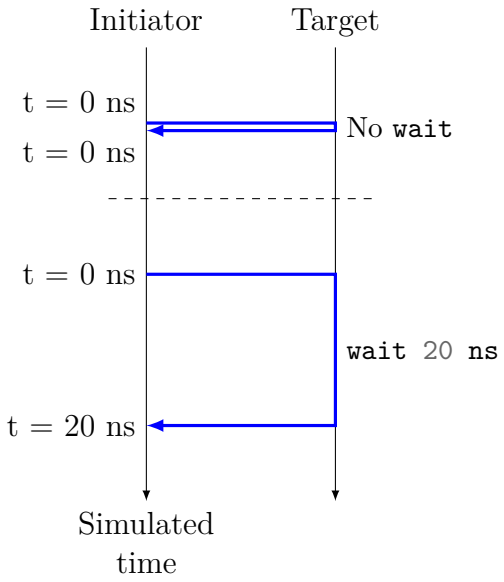


Figure 2.10: Two different transactions in loosely-timed style.

Each of those coding styles corresponds to different use cases. Both can be used for embedded software development or software performance analysis. Both can also be used for hardware functional verification and architectural analysis. Moreover, the AT style also enables hardware performance verification. The more transactions are split closely to the real hardware, the more accurate the performance evaluation will be. However, to answer the use cases enabled by both styles, the LT style is used for a simple reason: simulations run faster, because fewer details are specified in the model. Moreover, to develop AT models, more information is required, that is not available at early development stages.

2.3.1.3 Modeling CPUs

The simulation of CPUs is particular compared to other hardware components, as there are two very different approaches. A first approach is to use an Instruction Set Simulator (ISS). An ISS is, as the name indicates, a simulator of the instruction set of the target (*i.e.* the chip) processor architecture. An ISS is an accurate way to simulate a specific

architecture (*e.g.* ARM) on another one (*e.g.* x86). With an ISS, the embedded software is compiled for the target architecture (cross-compilation). This approach targets accuracy, in the sense that the software execution on the target processor can be simulated instruction per instruction. An ISS includes a translation mechanism from the simulated architecture to the host machine architecture; in the end, the ISS is a program that runs on the host machine. Another approach is to use native execution of the embedded software. This means that the embedded software is compiled for the host machine. The embedded software is run nearly in the same conditions as the rest of the model, *i.e.* as a regular program, except that the hardware functionalities (*e.g.* memory accesses) are captured and translated into transactions on the SystemC/TLM model. Instead of translating each instruction, only bus transactions need to be translated to communicate with the simulated platform. This results in simulations much faster than with an ISS.

The two approaches answer different needs. An ISS is used to simulate the execution of the software instruction by instruction: this can be used for an accurate performance evaluation of the software. Native code execution includes everything for fast simulation at a high abstraction level such as LT TLM.

2.3.2 Time Modeling

2.3.2.1 Temporal Decoupling

Temporal decoupling has a specific meaning in TLM-2.0, but before presenting this particular case, we introduce the idea. *Temporal decoupling* consists in enabling SystemC processes to run ahead of the current simulated time, thus without actually advancing simulated time. Each process has a local value modeling its own simulated time. Each process may increase its local simulated time without performing a SystemC `wait`. This is a low-cost timing annotation. To keep time consistency in the whole simulation, each process must at some points yield to the kernel to advance the SystemC simulated time. The purpose of temporal decoupling is to increase computation locality, which reduces the overhead of context switches between processes.

In TLM-2.0, temporal decoupling is implemented in transactions, with a simulated time annotation. This timing annotation is not necessarily transformed into a SystemC `wait` call. Different strategies for taking such annotation into account are possible. To limit the maximum advance a process is allowed to have (compared to the least advanced process), a *time quantum* can be defined. If any local time goes further than the quantum, a SystemC `wait` is performed.

In practice, temporal decoupling was already used in SystemC models, notably at STMicroelectronics, before TLM-2.0. Internally, timing annotations are performed with a family of functions called `annotate`. The synchronization points can either be implicit (*e.g.* using TLM-2.0 time quantum, performed in transactions) or explicit (*synchronization-on-demand* in TLM-2.0) by using a `synchronize` method.

2.3.2.2 Loose Timing: Time Ranges

One of the purposes of the TLM abstraction level is to enable the modeling and simulation of a chip that is not already fully specified. However, at such early stages of the development, not only the chip is not fully specified, but the micro-architecture is not fixed yet. Consequently, it is not possible to have precise timing information for the operations performed by the chip. Yet, the model can and must still be written, without such information. Different strategies are possible to overcome this issue: not using time, over-specifying or partially specifying.

The first one, not using time, results in an untimed TLM model. It often corresponds to under-specifying the model, because at least some timing information can be derived from the early specifications. The over-specification strategy consists in choosing an arbitrary value. The choice can be constrained in different ways. An example is to use an approximated value, plus or minus a certain percentage. Of course, the approximated value and the percentage are also arbitrary values. This leads to a model that *looks like* it is specified, but the values that are used do not actually mean a lot, thus it is an over-specification. A third strategy is to add in the model the fact that the information is partially known. Indeed, for most transactions, it is possible to find a minimum and a maximum time value that bounds the actual one (even if at start, this interval is wide). Such annotation indicates a partial information: we do not know the actual timing, but it will be within those bounds.

Among those three strategies, the one that adds *the most information* on the model is the third one, using time ranges. On one side, it adds information on the model, but on the other side it introduces the need for the simulator to support ranges. That has to do with how those time ranges are implemented. In practice, the SystemC kernel does not support time ranges, it needs time values. This is why, at STMicroelectronics, time ranges are part of the internal implementation of LT TLM, in an overlay on the SystemC kernel. A time range is specified using the function:

```
annotate_loose_timing(sc_time min, sc_time max)
```

In production code, this function picks a random time value within the bounds. We can notice that adding random values in the simulation does not remove the reproducibility: if the random seed, fixed at simulation start, stays the same then the same series of random numbers will be produced.

At first sight, using a random implementation for time ranges seems like the over-specification approach. However, the fact that we define time ranges as bounds for the real value enables the use of this information for other analysis tools. For example, Helmstetter [3] studied different scheduling orders of SystemC processes for different values chosen in time ranges. By studying the consequences of the different scheduling orders, it was possible to establish a list of orders that actually change the simulation results, thus are worth to simulate. We give another example of how those ranges can be exploited in the next chapter.

2.3.3 History of the STMicroelectronics/Verimag Collaboration

The Verimag laboratory is specialized in embedded systems, more precisely in improving their development, both in time and quality. In particular, Verimag is specialized in formal verification, testing, synchronous languages and modeling. Those fields are in line with the branch of STMicroelectronics in charge of systems on chip modeling, that has actively contributed to the standardization of SystemC and TLM. Over more than a decade, Verimag and STMicroelectronics have collaborated on SystemC/TLM modeling. The first thesis, by Matthieu Moy, studied the formal verification of SystemC/TLM models [4], *i.e.* proving properties on models. Claude Helmstetter worked on partial reduction of scheduling orders in the case of loosely timed models [3]. This work also includes study on simulation semantics, further discussed in Chapter 4. Jérôme Cornet worked on the refinement of models [5], by adding non-functional information to models without changing the functionality. Giovanni Funchal studied different aspects of TLM [6], in particular the consistency of models in comparison with the real chip, the semantic signification of TLM constructs and time modeling. Yussef Bouzouzou[2] studied static analysis of SystemC/TLM code, in order to implement a semantics-preserving parallel SystemC simulator for TLM models [7]. This work, as a parallel simulation approach, is further discussed in Chapter 4.

[2]Verimag and Silicomp (Orange Business Services), in collaboration with STMicroelectronics.

2.4 High Level Synthesis

The High Level Synthesis (HLS) design flow consists in transforming code from a high abstraction level, generally written in C++, into synthesizable code in a Hardware Description Language (HDL), generally at Register Transfer Level (RTL). At STMicroelectronics, it is mainly used for the design of hardware acceleration blocks containing Intellectual Property (IP).

2.4.1 Hardware Acceleration Blocks

Systems on chip are composed of hardware and embedded software, sometimes called firmware. Embedded software run on one or several Central Processing Units (CPUs), and use the different hardware blocks available in the system. Communications are made through bus components. We can distinguish application-specific hardware blocks from CPUs and bus components. Such blocks are finely tuned (*e.g.* in terms of area, performance or power consumption) to implement a specific functionality (*e.g.* video decoding, motion detection, signal handling). They contain hardware intellectual property, and are a key differentiation factor from one vendor's chip to another.

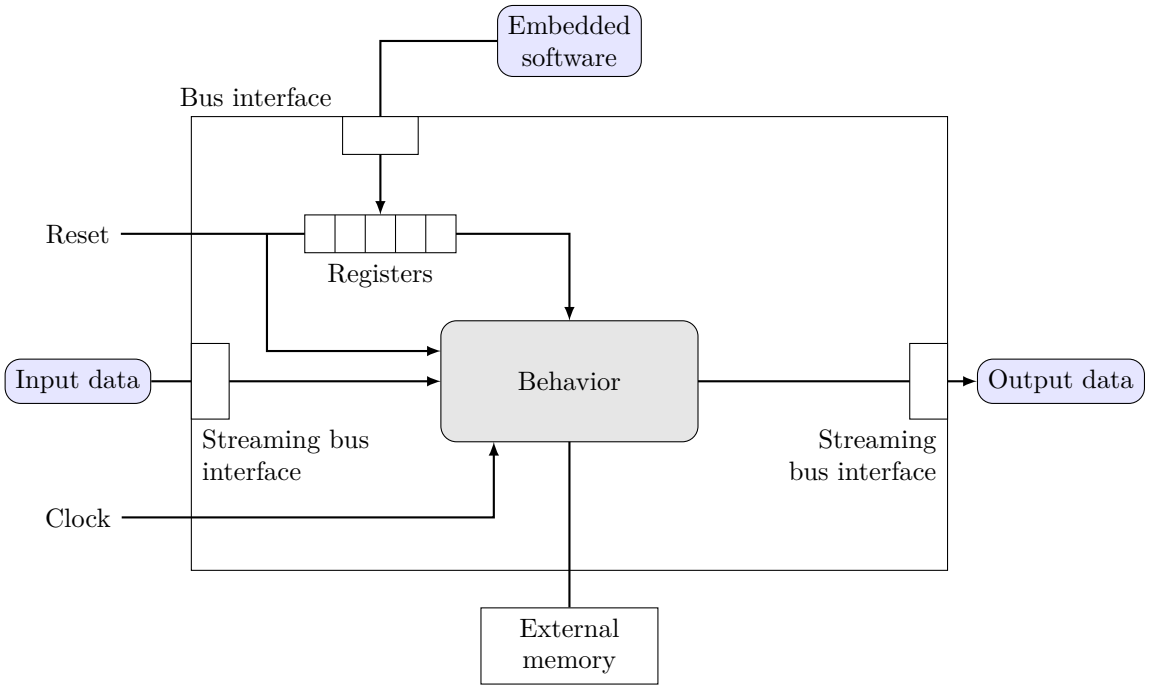


Figure 2.11: Example of an IP block.

An typical example of IP block is shown on Figure 2.11. An important point to highlight on this figure is how the embedded software interacts with the block. This

communication is enabled by hardware registers. Hardware registers have properties similar to memories in the sense that they contain data that can be read or written. But contrary to memories, registers can also trigger hardware functions. Registers can be private in the block, which means that the software cannot see them, or can be accessible to the software. To read or write a register, the software performs a bus transaction with the address of the register, just as it would do to read or write a memory. These registers are memory-mapped and in this context, they are often simply called *registers*.

2.4.2 Principle of High Level Synthesis

2.4.2.1 HLS Automatic Tool

The transformation of high level behavioral code into synthesizable RTL code is enabled by the use of an HLS tool. Several commercial tools exist in the market, including CatapultC (Mentor Graphics), SymphonyCC (Synopsys) or Stratus (Cadence Design Systems). Figure 2.12 illustrates the step performed by the HLS tool. Before HLS, this output design was handwritten by RTL designers directly from the specification. During the synthesis performed by the HLS tool, the user provides input parameters to the tool to influence the resulting design. These parameters include component libraries, details about the micro-architecture (*e.g.* whether to pipeline computations, to unroll loops) and the target frequency of the design.

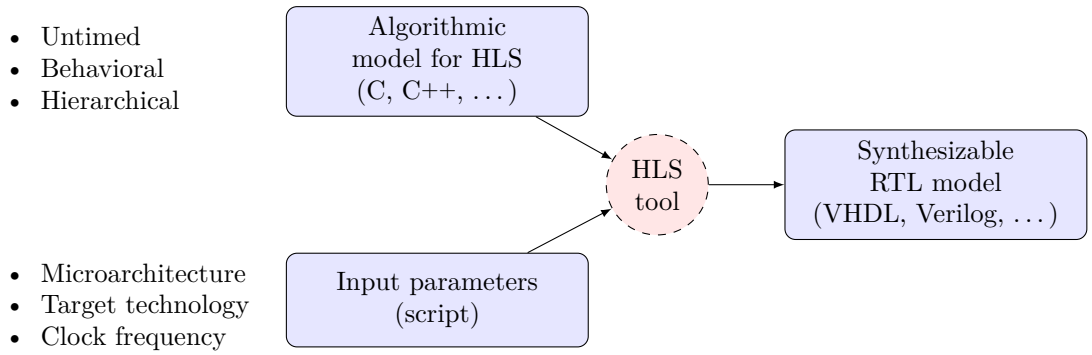


Figure 2.12: Principle of HLS.

2.4.2.2 Behavioral Code

The input of this flow is a high level code, generally written in C or C++. An example code is given in Figure 2.13. To disambiguate the meaning of *high level code* in this case, we assume the following list of properties for HLS code:

- The behavior is described without clock.

- Computations are described using classic C++ operators (*e.g.* line 7 on Figure 2.13). In particular, micro-architecture details (*e.g.* how many hardware multipliers are used, how these operations are scheduled) are not present in the code.

```
1 ac_int<8, false> coef[64]; // table of coefficients, defined in another file
2
3 void mult_coef_stream(ac_channel< ac_int<8, false> > & data_in,
4                        ac_channel< ac_int<16, false> > & data_out)
5 {
6     for (int i = 0; i < 64; ++i) {
7         data_out.write(data_in.read() * coef[i]);
8     }
9 }
```

Figure 2.13: Example of code for the HLS tool CatapultC. This describes a block that reads 64 data from an input stream, and outputs them multiplied by an array of coefficients.

With C++ datatypes, only "classic" bit length are available (8, 16, 32, 64). However, describing an algorithm for an HLS tool may require to deal with non-standard bit lengths. Each tool proposes its own solution to solve this issue. For example, CatapultC uses an algorithmic C library including the `ac_int<SIZE, SIGNED>` type (*e.g.* lines 1 and 3–4 on Figure 2.13). HLS tools that use SystemC code as input (*e.g.* Stratus) directly use SystemC bit-accurate datatypes like `sc_int<SIZE>` or `sc_uint<SIZE>` in this purpose. Similarly, the same answer can be made to deal with fixed point numbers.

2.4.2.3 Design Flow

The design flow at STMicroelectronics with HLS is presented on Figure 2.14. The SystemC/TLM model of the system includes parts written for HLS. The design contains parts that interact with the embedded software (*e.g.* registers) then there is a need to simulate the HLS code with a Transaction Level Modeling (TLM) platform, to enable the software to run. TLM users need the functionality described in the HLS code to have a comprehensive simulation.

From the previous explanations, one might think that a whole TLM platform could be given as input to a tool that outputs a synthesizable RTL code. In practice, there are several reasons explaining why this is not the case. The main reason is that a TLM model does not contain enough details to generate a synthesizable hardware design. Moreover, a TLM model contains simulation features that do not model hardware, but are used for convenience (*e.g.* using a graphical library to open a display, using third party libraries for better performances). There is also an issue with microprocessors of the platform: either they are modeled with an Instruction Set Simulator (ISS) or using native execution. In both cases, such models of processors are not intended to be synthesized by an HLS tool.

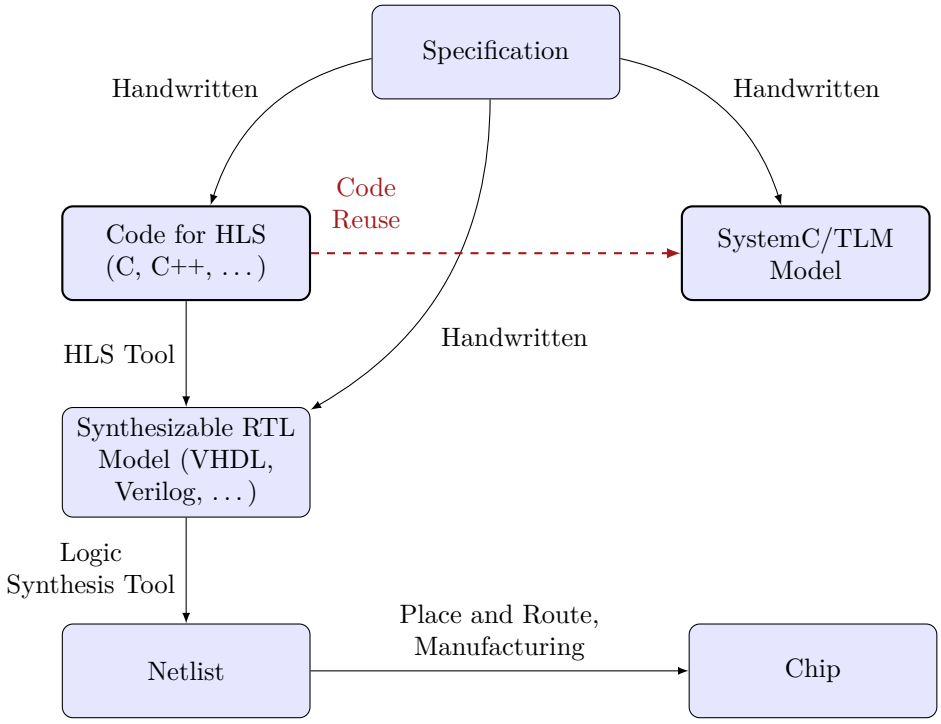


Figure 2.14: Design flow using HLS.

2.4.3 Studies and Experience on HLS

The current generation of HLS tools, starting in 2005 [8], is used by major semiconductor companies including STMicroelectronics. The maturity of HLS tools is growing and enables industrial use for the development of hardware IP blocks. Even if it is hard to accurately measure the benefits of HLS over RTL handwriting [9], we present research work that compares them for specific examples. Pagliari *et al.* presented a study on a medical imaging application [10]. The authors notably compared the development time and performance of RTL handwriting versus HLS for two different image processing methods. The development by an RTL designer took three months. Their result is that even if the development with HLS was slower, four months by a person with no prior knowledge about the implemented algorithms, the resulting design was three times faster. But the difference in the development time should be put in perspective with the lack of experience of the developer mentioned in the paper. Another result is that using the RTL flow, the authors only have one implementation for one of the two methods. With the HLS flow, they could explore 104 alternative RTL designs for one method and 80 for the second one, by changing input parameters of the HLS tool. Here is a potential benefit offered by HLS: with similar development times, the exploration possibilities on both the architecture and the algorithm have dramatically increased. Inggs *et al.* presented a case study on finance algorithms, where the conclusion is that the HLS tools under survey were ready for industrial needs considering the performance of produced designs [11].

Gajski *et al.* discussed about which language was the best for HLS [12]. Indeed,

HLS code does not have to be written in pure C++. In particular, they emphases the advantages of using SystemC code as HLS input. SystemC adds missing features to C++ that are required to model hardware, such as parallel description, hierarchical description and bit-accurate datatypes. The choice of using mostly C++-based inputs for HLS is motivated by the following reasons:

- C++ is a standard and well-known language. Since many developers already know at least its syntax, it avoids to learn a whole new specific language for HLS.
- Quality of existing tools around C++, especially when it comes to work with an abstract syntax tree, which is needed for HLS tools.
- High speed of execution when compiled with a C++ compiler (for simulation).
- Substantial existing code bases, which can ease the effort needed to develop an algorithm from legacy code.

At STMicroelectronics, HLS is used mostly to design hardware acceleration blocks for systems on chip. The overall feedback is that even though HLS tools have their defects, the benefits on development time are good enough to continue to use this design flow. Moreover, the reusability of HLS code from one project to the other is valued. Indeed, some teams have encapsulated frequent functions (*e.g.* fast Fourier transform) into template libraries to save development time. This library code is valuable in several ways: it is already functionally verified, and it is already known that it will produce an efficient design with the HLS tool for a specific set of input parameters.

To write good quality HLS code, the developer must be aware that the code will be translated into hardware description code. Using HLS does not mean that one shifts from hardware design practices to software design practices: designing hardware is still a job for hardware designers. However, the technology changes and so does the level of abstraction. With HLS, the micro-architecture aspects are settled by the tool, and depending on the tools, different choices are possible. Depending on how the designer writes HLS code, the tool tries to infer what the designer had in mind in terms of hardware implementation.

We make the assumption that the use of an HLS flow will in time replace manual writing at RTL level for the development of IP blocks. Similarly, for software development, there are very few use cases requiring writing assembly code. Instead, developers mostly program using high level programming languages like C++ or Python. But before the shift to fully happen, the main conditions are that HLS tools reach a sufficient state of maturity to replace the previous flow and that previous design flow users are ready to shift to a new technology. Those conditions are not completely related because some developers had bad experiences with the first generations of HLS tools [8], which introduced distrust in the technology. However, recent work cited in this part, and the current level of industrial use shows that the level of maturity is high enough at least in specific cases.

To sum up what was explained in this part, the rationale behind the use of HLS is the reduced amortized time-to-market compared to RTL handwriting. The *amortized*

qualifier comes from the fact that the first design may be obtained in more time with HLS than with handwritten RTL, because of the learning curve for using HLS tools and HLS code writing. However, the higher reusability of algorithms developed for HLS and the ability to perform exploration on the resulting architecture make the shift to an HLS design flow interesting for the design of hardware IP blocks.

2.4.4 Interface of a Block Designed with HLS

The HLS/RTL partitioning is the name we give in this manuscript to the choice hardware designers make when they use an HLS design flow. It consists in answering the question: "which part of an hardware block will be generated by the HLS tool, and which part will be handwritten in RTL?". This is a different question from: "which IP block will be generated with HLS and which will be handwritten in RTL?". In this case, we are speaking of an IP block for which the choice has been made to use an HLS design flow.

Figure 2.15 presents an example IP block whose behavior is described with algorithmic code for HLS. This example is inspired from a real-life design from STMicroelectronics, simplified and generalized for confidentiality. This block performs processing on video data coming from an acquisition device and the output is directly connected to a display device. The HLS tool used for this project is CatapultC.

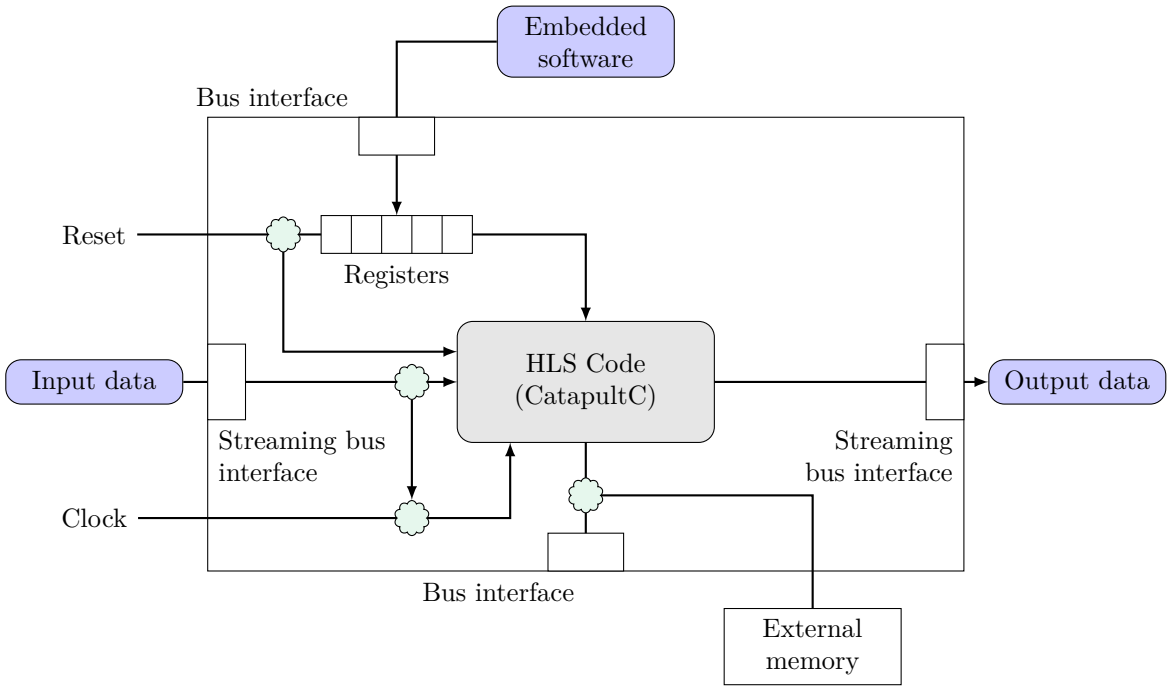


Figure 2.15: An IP designed with HLS.

In this paragraph, we describe the elements on Figure 2.15 only added in the RTL model of this IP block. The input and output data exchanges are performed through a

high frequency streaming bus (AXI[3] stream). To enable this, RTL code is added to send and receive data accordingly with the bus protocol. The IP block contains registers that can be read or written by the embedded software through a low speed system bus. The registers are used by the HLS code, where they are given as arguments in the form of an array or encapsulated in a custom structure. Another point is that even if the HLS code uses the registers, their hardware design is often not synthesized by the HLS tool, so they are added in the RTL model. In the industry, registers are often described using the IEEE standard format IP-XACT, and IP-XACT to RTL tools already exist, so there is no rationale to generate the registers differently. One of the registers is notably used by the software to perform a reset. A hardware reset signal is also provided as input to the IP block, with some logic to interact well with the software reset. In the HLS code there is no reset signal, so all this logic is added in the RTL code. The block needs access to a memory area, which can be accessed directly from within the block. It is also possible for the software to access that memory, through another bus access. To enable this, RTL code for memory arbitration is added. This code is not generated by the HLS tool because it was already available in RTL and is quite stable from one design to the other. The clock signal is completely absent in the HLS code for CatapultC. In this example, combinational logic is added in the RTL to disable the clock signal when no input data is available.

2.4.5 Necessity to Split Designs in Sub-Blocks

In order to design complex hardware IP blocks, the development must be split into sub-blocks. We can detail at least three reasons that justifies that statement. Firstly, it enables the design to be done in parallel within a team, or even by different teams. This decreases the overall development time, provided a clear specification of the interfaces. Secondly, it is easier and less error-prone to develop several simple functions than one complex function. Thus, in addition to enable multiple designers to work in parallel, their individual work is also simplified. The two reasons previously presented justifies that splitting designs in sub-blocks speeds up the development, but not that it is mandatory for complex designs.

The main reason is that the resulting hardware designs have to be validated. We remind that IP blocks are key differentiating factors from one product to the other, thus formal validation is mandatory for such parts of the design. This verification is made by covering the different inputs possible. Here, by “inputs”, we mean all the different wires produced by the synthesis tool. Not only the ones at the interface of the block, but also all the internal ones. Yet, it is not possible to formally validate, for example, a complete JPEG decoding block (as a whole), because the combination of values to test is too wide; there is a combinatory explosion. Thus, the development of complex blocks requires to split the design into sub-blocks. Moreover, this enables to build pipelined architectures when assembling the sub-blocks, which in many applications speeds up the throughput.

[3]Advanced eXtensible Interface, from ARM.

To describe hierarchical blocks, CatapultC distinguishes the top-level block and sub-blocks. Each block corresponds to a C++ function (but each C++ function does not need to result in a block: they can also be inlined). For Symphony, a design is completely described in one function and sub-blocks are extracted from code sections, *e.g.* from a loop nest. For Stratus, a block is a SystemC module.

Communication between blocks can be done with memories (or registers), with a simple wire, or through a First In, First Out (FIFO) streaming interface. As an example, CatapultC algorithmic library proposes the `ac_channel<TYPE>` class. Used in a block interface (*e.g.* lines 3–4 on Figure 2.13) it will be replaced by a FIFO interface. On the other hand, Stratus requires the user to describe in a cycle-accurate manner (using SystemC clocked threads) the input and output protocol used to read and write data (even if the computational part is described without using the clock).

2.4.6 Wrapping HLS code for TLM

In Section 2.4.4, we presented an example of HLS/RTL partitioning. From the HLS code, the set of elements missing for TLM is a subset of the elements missing for RTL. But it is not the same set. For example, the clock signal does not exist in a Loosely-Timed (LT) TLM model nor in the HLS code, but it is does exist in RTL.

To wrap HLS code for TLM means to enable the use of HLS code in a TLM simulation. In the case of CatapultC for example, this means to adapt block interfaces from `ac_channel` to use TLM protocols and to write an `sc_module` for the block. The latter part consists in writing a module having an `SC_THREAD` that executes the HLS function in a loop. About the block interfaces, since a data streaming protocol is already available internally at STMicroelectronics, a mapping has been done between the read and write functions of `ac_channel` to the read and write functions of this streaming protocol.

2.5 Problem Statement

The main objective of this thesis is to speed up the execution of SystemC/TLM simulations. More accurately, an interesting improvement is to reduce the time to get a first answer after a code change. Indeed, during the development of a model and of the embedded software, each change in the source code induce another run of the platform in order to test the modification. A significant reduction in the duration needed to get a result directly impacts the development speed, which reduces the time-to-market of the final product.

Using parallel computation resources is a natural idea to speed up the execution of a program. We stated that the reference implementation of the SystemC kernel is sequential. Thus, there is potential to speed up simulations with parallel execution. However, this is hard to achieve, as a SystemC parallelization approach must not introduce race conditions. As stated in Section 2.3.1.1, communication is done by function calls, which creates shared resources. For example, two initiators (*e.g.* processors) that concurrently access the same target (*e.g.* a memory) will concurrently call the same function of this target. That makes the target component itself a shared resource, introducing a race condition if not protected. Consequently, a great part of the work to achieve parallel execution of SystemC model is to extract potential for parallelism (namely safe parallelism, without race conditions) from a model.

Another huge challenge for SystemC parallelization is the adaptability on existing platforms. Indeed, as for every technology change, the migration has a cost. This cost must be put in perspective with the time saved if the parallelization solution was in production: a solution that requires an important effort may not be profitable even if it shows substantial performance benefits.

Also, the different flows and technologies presented in this chapter, namely TLM modeling and High Level Synthesis (HLS), are part of the problem. Indeed, industrial design flows result of the interweaving of different specific design flows. The development of a model does not come from a single team, but is partly done by several different teams. Each team has its own requirements and constraints. For example, the team that develops the code of a hardware component for HLS must follow the syntax of the HLS tool and complete a validation step. This impacts the way models are written. Since we cannot remove those constraints, we must include them as assumptions.

To conclude with this section, we emphasis an important point regarding the conception of a parallelization technique: the knowledge of the profile of a simulation. Analogically, parallelizing huge independent computations on matrices is not performed the same way as parallelizing a shortest path algorithm. In our case, a simulation is mostly characterized by the model and not by the simulation kernel in itself. In other words, it is mandatory to characterize a simulation in order to find potential parallelization.

— Chapter 3 —

Simulation Profiling

3.1 Introduction

Before making optimizations on a program, a natural step to go through is to identify its performance bottlenecks. The size and complexity of industrial SystemC models, as at STMicroelectronics, makes it hard if not impossible to have comprehensive knowledge of the whole simulation. To get a big picture of such simulations, we therefore needed to measure and visualize different parameters. For example, how the different SystemC processes of the model are run, how they consume the wall-clock time and how many processes are runnable at each simulation cycle. We have not found an existing tool for SystemC models, that helps getting those measurements, so we developed SycView, presented in Section 3.2. Then, we used SycView to profile a case study from STMicroelectronics. Those results are presented and analyzed in Section 3.3. In summary, the contributions presented in this chapter are:

- Identification of profile metrics on SystemC simulations, interesting with parallel simulation in mind. Development of a visualization tool and instrumentation of the SystemC kernel in order to obtain the corresponding measurements. This tool has been presented at the Design Automation for Understanding Hardware Designs (DUHDe) workshop in 2016 [13].
- Profiling results on an industrial case-study from STMicroelectronics and analysis of the results. The results presented in this section have been published at the Rapid Systems Prototyping (RSP) international symposium in 2015 [14] and further discussed in an article of the MDPI Electronics journal in 2016 [15].

3.1.1 Motivation for Developing SycView

Setting up parallel computations in a computer program can be done using several different approaches. The choice of an efficient approach depends on the *shape* or *profile* of the program. In the case of SystemC, a program consists in different parts: the SystemC kernel, the hardware model and the embedded software. Very little information on how the whole program behaves is available from the kernel. It is mostly the model of the hardware and the embedded software that defines the *profile* of a simulation. For example, a Register Transfer Level (RTL) model is essentially composed of clocked threads executing small computations at each clock cycle, while a Loosely-Timed (LT) Transaction Level Modeling (TLM) model uses as little `wait` as possible, targeting execution speed. Thus, finding a suitable parallelization approach for SystemC is through knowledge of the *profile* of the simulated models.

The word *profile*, previously used hastily, is referring in our case to a set of measurements performed during the execution of a SystemC simulation. Those measurements are

relative to wall-clock time, because our target is to speed up simulations, thus to reduce the wall-clock duration of a simulation. The following list gives interesting measurements for this purpose:

1. How many SystemC processes are runnable at each simulation cycle? What is the partitioning between `SC_THREAD` and `SC_METHOD`?
2. How many wall-clock time does each transition consume?
3. From which process, module or part of the model comes the most wall-clock time-consuming processes?

SycView can also be used in a purpose different than profiling a simulation. In an industrial context, complex models are partially developed by several teams, including external ones from different companies. This results in a large quantity of source code on which it is hard to have comprehensive knowledge. Moreover, some parts of the model or the embedded software may only be available in binary form. In this type of project, it is precious to be able to take a step back and see the big picture. Contrary to the *profile* measurements, the views mentioned here are relative to simulated time because they are used for understanding the simulated platform. With this in mind, we identify a set of questions that are interesting to answer:

1. Which SystemC processes are run, and in which order?
2. How does the simulated time advance in the model?
3. Can we identify patterns in the execution of processes? If so, are they meaningful with respect to the platform under test?

3.1.2 Existing Tools

Existing profiling tools such as *valgrind*[1] or *gprof*[2] are lacking the separation between the SystemC kernel and the model. For example, without instrumentation, they cannot find which transitions are the most time-consuming and which SystemC process and module they belong to. Consequently, to answer the needs formulated in the previous part, we need tools that understand SystemC constructs and can show results with respect to them.

Most of the literature about SystemC visualization focus on structural visualization. Große *et al.* [16] present a Graphical User Interface (GUI) for a system view showing the different modules of a design with their ports and bindings. Berner *et al.* [17] extract structural information from a SystemC model using the documentation system Doxygen.

[1] `http://valgrind.org/`
[2] `https://sourceware.org/binutils/docs/gprof/`

Albrecht *et al.* [18] extend SystemC with a GUI called *gSycC* that is not only used for structural visualization, but also for simulation control. Such a tool could be extended to collect traces during execution. However, annotations must be added to the model. *ViSyC* is a tool developed by Genz *et al.* [19, 20] enabling interactive system exploration of SystemC models. It performs static analysis on a SystemC program to extract structural information to display on a GUI and bind them with the source code describing their behavior.

As the complexity of models increases, so does the difficulty to understand them. This difficulty is illustrated by the diversity of visualization tools previously outlined around SystemC. How to visualize SystemC simulations is still a very open question, as much as how to retrieve data from SystemC simulations. This depends of course on the purpose of the observer. A recent overview on these questions is given by Drechsler and Stoppe [21]. SycView falls within this approach to improve how users and developers can understand the models and improve them. Our goal is to find the bottlenecks of a simulation as well as to help finding rationale to explain these bottlenecks.

3.2 SycView: A Visualization Tool

SycView results of two development axes. The first one is an instrumented version of a SystemC kernel, that generates traces online during a simulation. The second one is a visualization tool that enables the offline view of tables or charts based on the traces. We developed both parts during this thesis.

3.2.1 Trace Recording

We added instrumentation code for trace recording in the reference implementation of the SystemC kernel, as shown on Figure 3.1. We remind that the version we call the *reference* is the one developed by the Accelera Systems Initiative (ASI). The trace recording is done in the SystemC kernel, because we needed SystemC information in the traces. We do not instrument the user model code, to make the tool easy to use on different models.

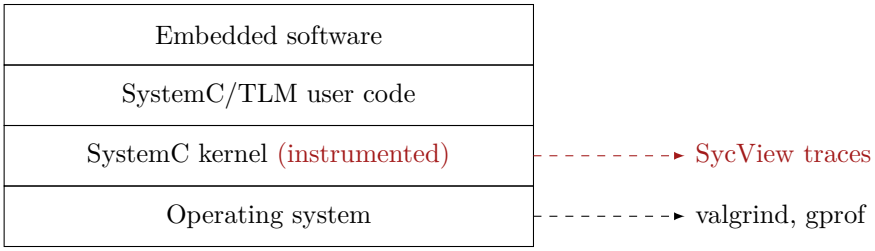


Figure 3.1: Positioning of the trace recording mechanism.

The trace recording consists in writing data into files at interesting points. Each time a process yields to the kernel, a trace is recorded, as illustrated on Figure 3.2. The trace contains the name of the yielding SystemC process, the type and arguments of the `wait` performed (if it is a SystemC thread) and the wall-clock duration of the transition that just ended.

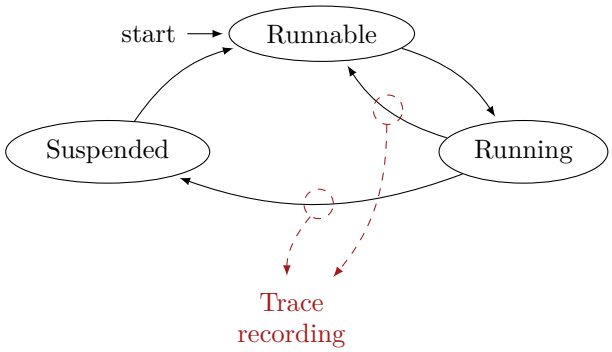


Figure 3.2: Trace recording for a SystemC process.

Moreover, at the end of each delta cycle, we record the maximum number of runnable processes among all the subsequent evaluation cycles. The purpose of this measurement requires further explanations, which itself needs a quick presentation on parallel simulation approaches (further presented in Chapter 4). A common parallel simulation approach consist in running multiple SystemC threads in physical parallelism when they are runnable at the same cycle. Such approaches use a synchronization barrier at each evaluation cycle. Thus, an upper bound of the potential for such parallel simulation is given by our measurement. Figure 3.3 sums up the states of a SystemC simulation where traces are recorded.

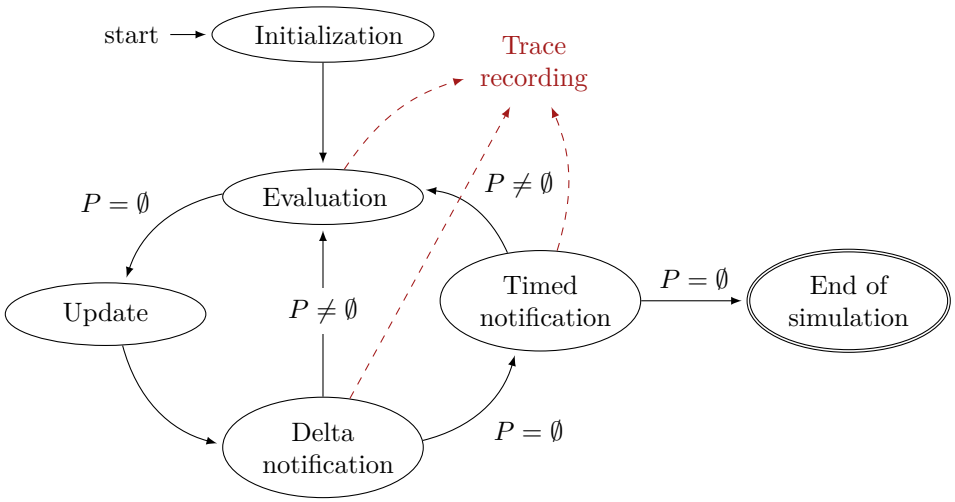


Figure 3.3: Trace recording in the SystemC kernel (P is the set of runnable processes).

3.2.2 Evaluation of the Trace Recording Overhead

We measured the influence of trace recording on simulation performance, in order to quantify the bias it introduces on the measurements. This enables to qualify our measurements as "reliable" or "not reliable". The relative overhead is obtained by comparing the wall-clock duration of a simulation with the instrumented kernel versus the reference version. The time measurements were done using `clock_gettime` from Linux's `time.h`, using the `CLOCK_MONOTONIC` parameter. It has a resolution of 1 ns according to `clock_getres`. The experiments were done on a host with an Intel® Xeon® CPU at 2.4 GHz.

On our host machine, the average time needed for trace recording is barely less than 50 µs per process transition. This does considerably increase the time normally used to switch from one SystemC method to the other (given that methods are just function calls) and also increases the context-switch duration for SystemC threads (normally executing only a few assembly instructions). In case a transition consumes a large amount of time compared to the duration of a recording, the measurement is accurate. In case a transition is very short compared to the duration of a recording, the measurement is not reliable and the measured duration should not be taken as accurate. Figure 3.4 shows the overhead

of trace recording on the SystemC thread context switches, compared to the reference version.

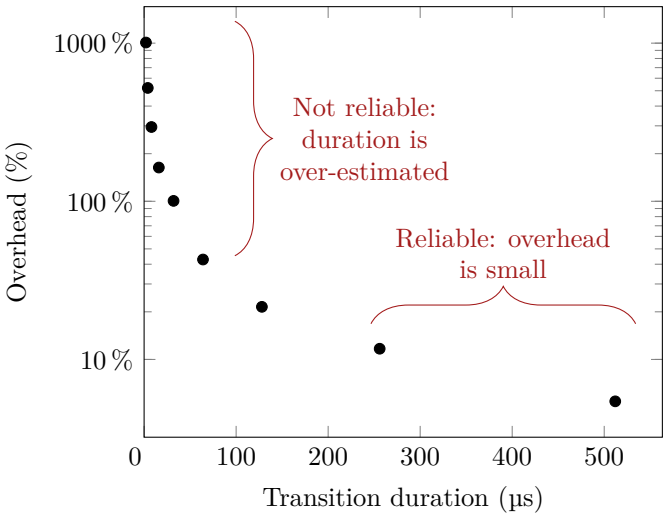


Figure 3.4: Trace recording overhead for the SystemC thread context switches, as a function of the duration of the transition (wall-clock time) preceding the context switch.

3.2.3 Visualization

We have implemented a Graphical User Interface (GUI) called SycView providing different views to exploit the data collected by our instrumented kernel during a simulation. To improve the readability of this manuscript, the measurements made by our tool are represented in tables or charts rather than screenshots when it is possible to do so. In this part, we show data from example platforms: the focus is not on the results as they are, but on *how we present and interpret* them. The focus will be on the results in Section 3.3, where data from an industrial case study are presented and discussed.

3.2.3.1 Quantitative Views

The views presented in this part display quantitative measurements. The wall-clock time consumption per SystemC process can be displayed. In particular, a table shows information for each process as a whole, and also about each transition. The per-process information are the name of the process, its type (thread or method), the total wall-clock duration of the process, as well as the proportion relatively to the overall simulation duration. The per-transition information are statistical metrics (minimum, quartiles, maximum) and the number of transitions that occurred.

Table 3.1 shows an example of a wall-clock time consumption table for the processes of a simulation. For each process, the table indicates its SystemC name, its type, its total wall-clock duration, the percentage it represents against the total wall-clock duration of

the simulation, and the two last columns are the number of transitions executed and the mean duration of a transition. The three columns we skipped indicate statistical metrics. The first quartile (1st Q.) is a duration such that 25 % of the process' transitions are shorter than this value. For the median it is 50 % of the transitions and for the third quartile (3rd Q.) it is 75 %. In this example, the first process (named "decoder.compute"), which is a SystemC thread, consumed 13 % of the total simulation duration (wall-clock time) in 5411 transitions. The third process (named "vga.compute") is also a SystemC thread, and consumed 8.4 % of the simulation duration, but only in 10 transitions. For the SystemC kernel itself, the measured times are mostly very short, *i.e.* the duration of a trace recording is not negligible compared to most of the measured times. This artificially exaggerated the percentage spent in the SystemC kernel compared to the overall simulation duration.

Name	**Type**	**Total (ms)**	%	**1st Q.**	**Median**	**3rd Q.**	**Transitions**	**Mean**
decoder.compute	Thread	7328.9	13.0	1.2	1.3	1.8	5411	1.4
cpu.compute	Thread	5658.4	10.1	76.4	78.6	80.2	72	78.6
vga.compute	Thread	4723.4	8.4	400.6	499.7	511.4	10	472.3
dma.compute	Thread	3972.6	7.1	133.8	135.0	232.1	25	158.9
SystemC kernel	-	3159.2	5.6	< 1	< 1	< 1	400	7.9

Table 3.1: Wall-clock time usage of some SystemC processes of a simulation. Time values are expressed in ms.

Another view available with SycView is the partitioning of simulation cycles depending on the number of runnable processes there were at the beginning of the cycle. Table 3.2 shows one example. In this example, there were 23,694 cycles with only one runnable process (at the beginning of the cycle). There were 127,070 cycles with only three runnable processes. Moreover, 96.6 % of the cycles had less than four runnable processes. This type of information gives an upper bound of the potential for parallel simulation there is in the execution of a model, in the case of an approach that runs simultaneously runnable processes in parallel. Such approaches are probably the most natural ones when it comes to parallel SystemC simulation, and were studied in several research work (see Chapter 4). This illustrates why we made such measurements: it gives in one table a good reason to study parallel simulation approaches other than the "classical" ones.

3.2.3.2 Simulation Charts

Additionally to the views shown on Tables 3.1 and 3.2, two other views are available with SycView. Those views are graphical and consist in plotting the scenario of an execution relatively to either wall-clock time or simulated time. Contrary to the previously presented views, those ones do not give quantitative information about a simulation. However, let us remind that models such as the ones developed at STMicroelectronics are developed

Nb. Runnable	Nb. Cycles	% (rounded)	Cumulated %
1	23,694	14.6	14.6
2	6078	3.7	18.3
3	127,070	78.3	96.6
4	621	0.4	97.0
5	94	0.1	97.0
6	17	0.0	97.0
7	1	0.0	97.0
17	3205	2.0	99.0
21	14	0.0	99.0
25	1585	1.0	100.0

Table 3.2: Number of cycles per number of runnable processes.

by many different teams, and even by external companies. Having the comprehensive knowledge on a model is hardly possible, because of both the size and complexity of the source code. The simulation charts can help to understand the global behavior of a simulation, from an external point of view, and to raise up questions on how models are written.

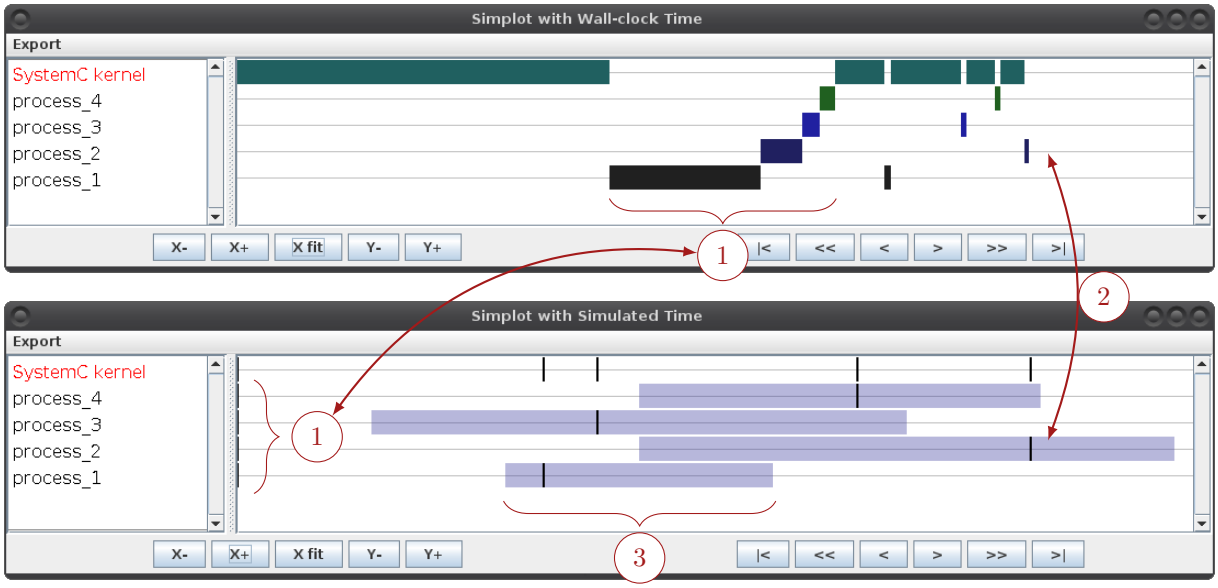


Figure 3.5: Example of simulation charts, relatively to wall-clock time (above) and simulated time (below).

Figure 3.5 shows two simulation charts for an example simulation. The chart above represents wall-clock time on the x-axis, and the one below represents simulated time on

the x-axis. The y-axis shows the different processes of the simulation. Let us start with the above graph for wall-clock time. Each time a process executes a transition, a rectangle is drawn, with a width proportional to the wall-clock time duration of the transition. Since the reference kernel we instrumented executes SystemC processes sequentially, there is always at most one rectangle for each value of time. In the graph below, for simulated time, each time a process executes a transition, a vertical stoke is drawn. This illustrates the instantness of transitions with respect to simulated time.

Mark (1) represents the initialization of the simulation. During this phase, each process executes its first transition, at simulated time zero. The strokes in the simulated time graph are stuck to the left, thus barely visible. On the wall-clock time graph, however, we can see that each initialization transition takes some time to run. On this simple example, we can easily map each transition execution in the wall-clock time chart with one in the simulated time chart, as we did for the initialization phase. As another example, Mark (2) indicates that the stroke below represents the same transition as the rectangle above. Mark (3) refers to the rectangles around some strokes. They represent a time range in which the execution of the transition *could have occurred*. This information comes from the time ranges annotations, notably used at ST and previously presented in Section 2.3.2.2. Of course, if only one of the transition occurred at a different time, this may change all the rest of the simulation. Those potential changes are not represented on this chart: it would be too complex not only to plot, but also to get such information. Thus, the rectangles represents time ranges relatively to the execution that *actually happened*.

3.3 Case Study: Model of a Chip for a Set-Top Box

3.3.1 Overview

This section presents the results obtained with SycView on an industrial case study from STMicroelectronics. The case study consists in the model of a chip for a set-top box. This chip includes video encoding and decoding capabilities. It has one Central Processing Unit (CPU) containing four general purpose cores, as well as hardware acceleration blocks for video processing, as illustrated on Figure 3.6. A modified Linux kernel runs on the CPU. Other dedicated cores that we do not detail here are also present in the chip.

The SystemC model is composed of $\sim 900,000$ lines of code including $\sim 750,000$ lines of C++ code as counted by `cloc`. It contains 850 modules hierarchically organized. Counting only the leaf modules leads to the number of 750 modules. There are 1068 registered SystemC threads and 163 SystemC methods in the whole model.

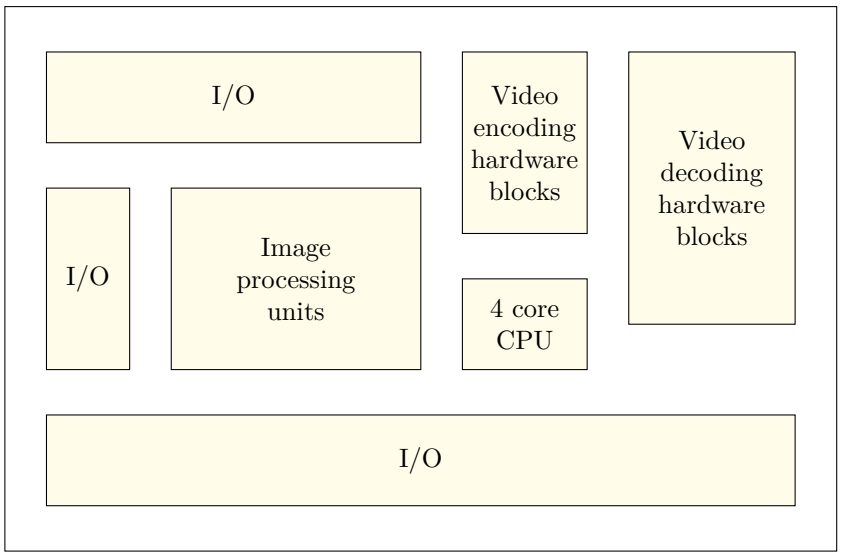


Figure 3.6: Overview of the chip from the case study.

Different test scenarios have been experimented. The first one is the boot and initialization of the platform, which starts in the beginning of the simulation and ends when a command shell is available. All the other scenarios exclude the boot and initialization phase. The two following cases are display video broadcast tests for h264 and mpeg2-encoded streams. They consist in decoding a previously encoded video stream and then send it for display. The last case is a transcoding, which redirects the decoded video stream to encoding hardware blocks, to produce a h264-encoded stream from a mpeg2-encoded stream.

3.3.2 Simulation Charts

As an overview, we first present simulation charts obtained from the execution of the mpeg2 decoding and display scenario. Figure 3.7 shows two simulation charts of the whole simulation where six frames have been decoded and displayed. It is not possible, with this level of zoom, to see the details of the simulation. However, we can identify different patterns that correspond to the test case. Mark (1) shows the hardware block threads that are computing when a frame is decoded. On the top chart, we can see that the execution take a non-negligible amount of wall-clock time. However, on the chart below, the same transitions were executed at the same simulated time instant. With this level of zoom, we only see that to each frame decoding and display corresponds one chunk of transitions. Between each frame, the CPU executes some transitions; the SystemC thread for the main CPU is pointed by Mark (2).

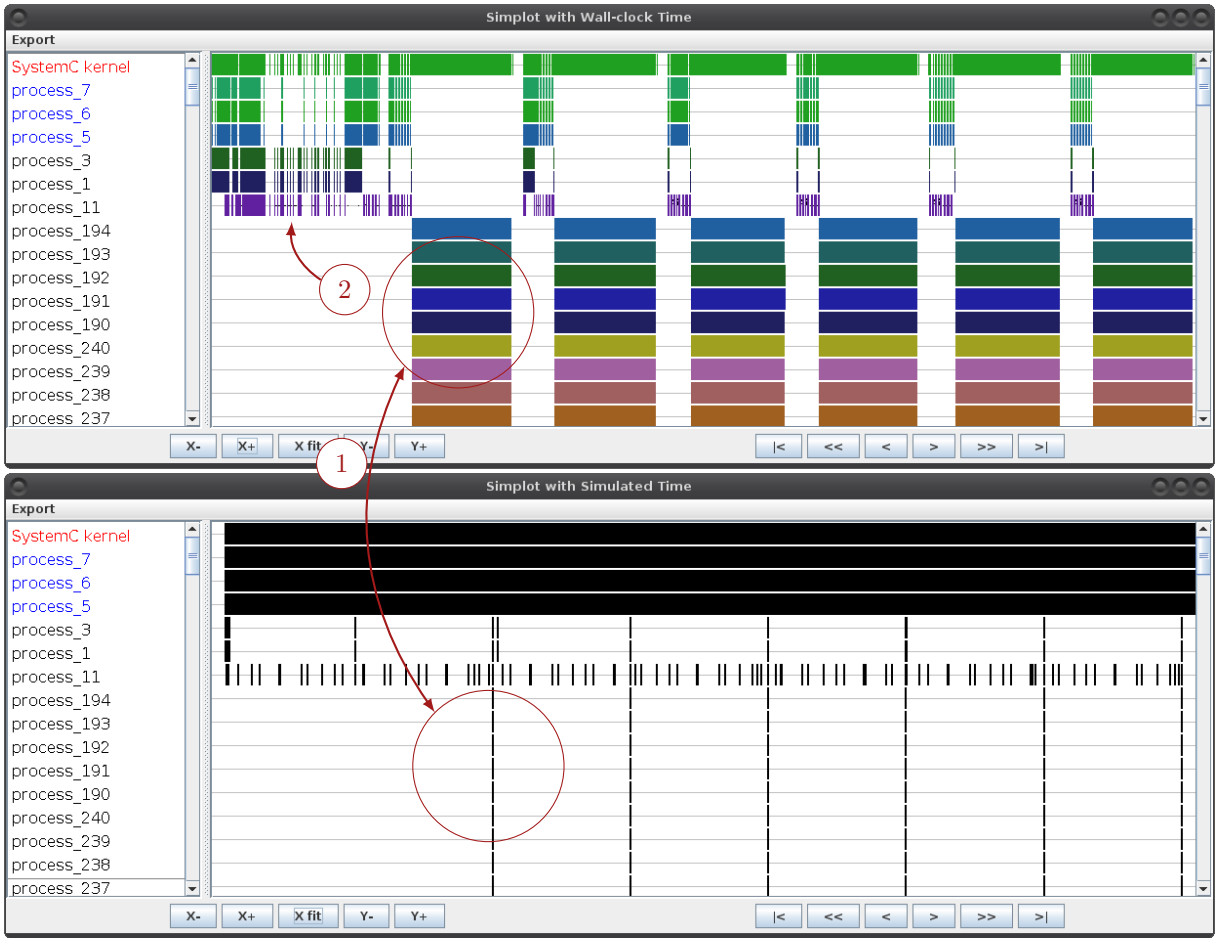


Figure 3.7: Simulation charts for the simulation of our case study when running the mpeg2 decoding and display scenario.

Figure 3.8 proposes a zoom on one of the chunks of transitions. Understanding the details of the simulation on such a chart is not possible because of its complexity. However, in some places, we observed what seems to be a "chain triggering" of short transitions

executing in one order (1) and later in the reverse order (2). Those transitions are triggered by the notification of SystemC events and happen in the same simulated time instant. Seeing this, one could question the coding style of this part of the model. Indeed, in the case of a Transaction Level Modeling (TLM) model, communications are mostly made using Interface Method Calls (IMCs). This tends in average to result in long transitions: one process performing an IMC executes code from another module, but still in the same SystemC process. In this part of the model, processes are notifying each other and the transitions are very short. If such patterns are frequent in the simulation, it may be slowed down because such parts of the model are not using at best the principles of high level of abstraction of loosely-timed TLM.

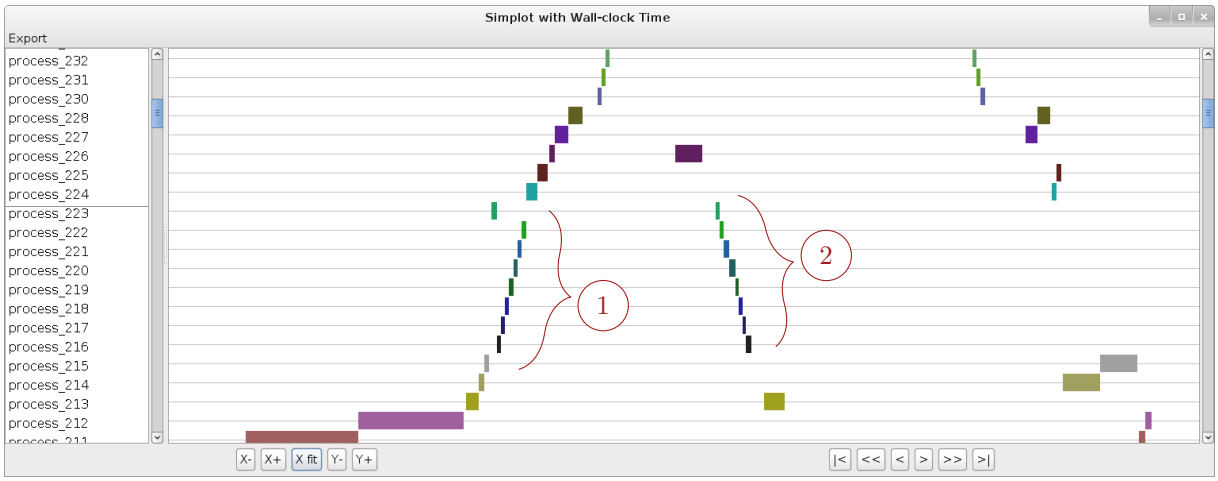


Figure 3.8: A zoom on processes from hardware blocks (see Figure 3.7) involved in the decoding of images.

3.3.3 Wall-Clock Duration

From the simulation charts, we could only have a feel on the complexity and on the overall behavior of the simulation. With the quantitative measurements, we get information that are both easier to understand and to exploit. To start, Figure 3.9 presents the wall-clock time consumption per type of SystemC process (thread, method) or spent in the SystemC kernel. Rather expectedly, a huge proportion of the time, namely 91.3 %, was spent in SystemC threads (*i.e.* not SystemC methods). For this reason, in case a distinction is made, we focus on SystemC threads rather than methods for the following results.

Figure 3.10 presents the distribution of wall-clock time between three different categories: SystemC processes from CPU core models, from hardware Intellectual Property (IP) blocks and the SystemC kernel itself.

For the boot and initialization case, 79 % of the wall-clock time was spent in core models, 10 % in hardware blocks and the remainder in the SystemC kernel. The explanation is rather straightforward: the system boot is performed by the CPU, and hardware

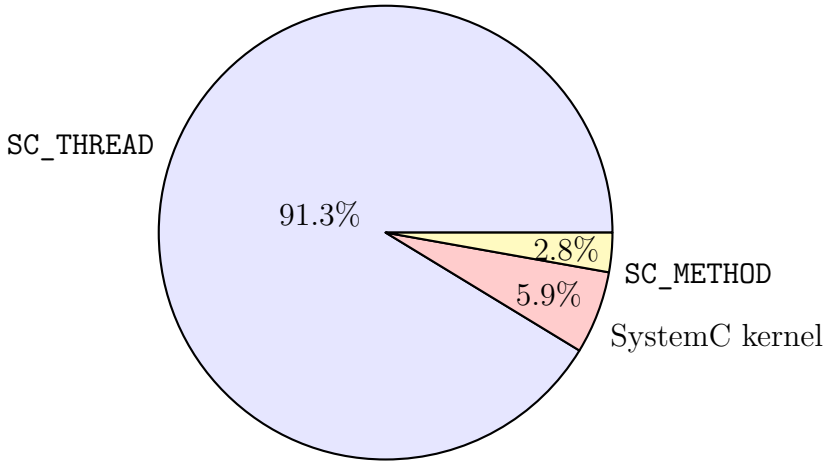


Figure 3.9: Distribution of the wall-clock time between SystemC threads, methods and kernel.

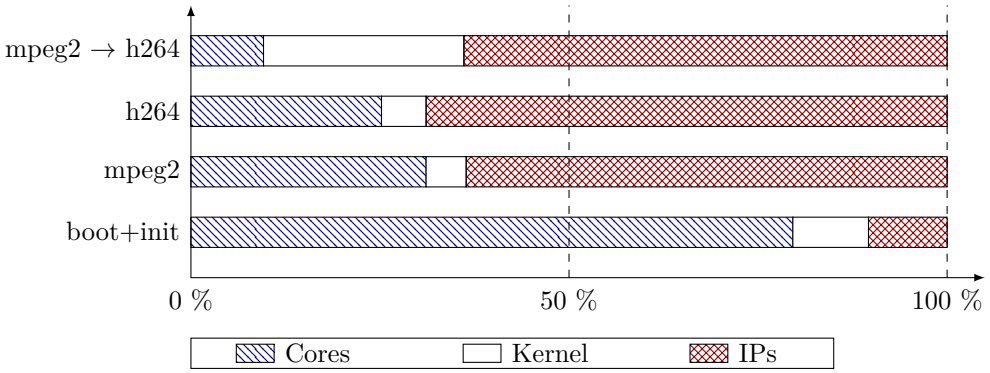


Figure 3.10: Partitioning of the wall-clock time elapsed, by category of processes, for four different test cases.

blocks are only initialized. In the video stream decoding cases (both h264 and mpeg2), one third of the time (respectively 25 and 31 %) was spent in simulated cores and the rest mostly in hardware blocks. The time spent in the SystemC kernel is below 6 %. This illustrates that the decoding computations are mostly done by IP blocks. Finally, for the transcoding case, 10 % of the time was spent on the core, 25 % on the SystemC kernel and 65 % on IP blocks. Again, we notice that hardware blocks are performing most of the computations, even if in this case there is more time spent in the SystemC kernel than in the decoding cases.

To get into more details, we present in Table 3.3 the results for the four most time-consuming SystemC processes in the test case h264. More than 35 % of the total time was spent on those processes. However, if we look at the number of transitions, we see that the first process (part of the IP category) consumed 12.9 % of the total time in 10,111 transitions while on the other hand, the second and third processes (also IP blocks) performed considerably fewer transitions (less than 100) but still consumed around 8 % of the time each. This means that transitions of the second and third processes are performing

Category	Type	Part	Exec.	Min (ms)	Median (ms)	Max (ms)
IP block	Thread	12.9%	10,111	< 0.1	1.1	18.4
IP block	Thread	8.6%	93	73.7	77.5	84.1
IP block	Thread	8.0%	14	395.2	486.6	495.7
SystemC kernel	—	5.9%	635,129	< 0.1	< 0.1	1.0

Table 3.3: Measurements of wall-clock time consumption for the four most consuming processes. Min, median and max corresponds to the execution time of transitions. For confidentiality reasons, we did not show the SystemC names of those processes, but only their category (IP block, core or kernel).

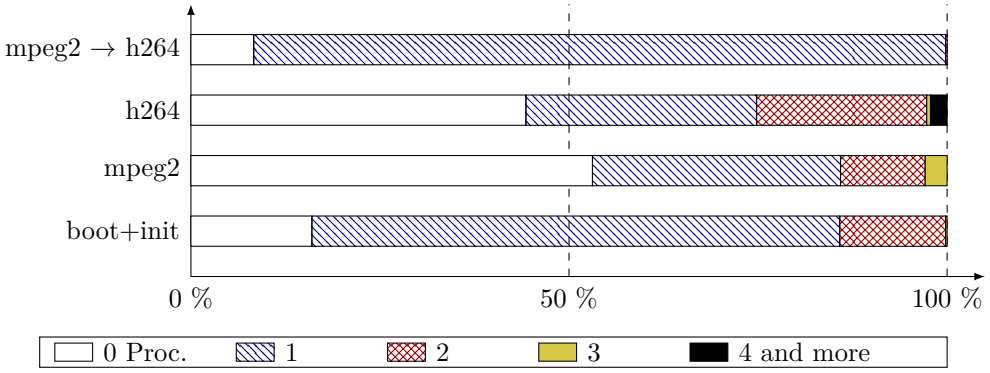
long computations, while transitions of the first process are short in comparison. This is confirmed by the minimum, maximum and median execution times for those processes' transitions. The fourth row represents the SystemC kernel.

3.3.4 Number of Runnable Processes per Cycle

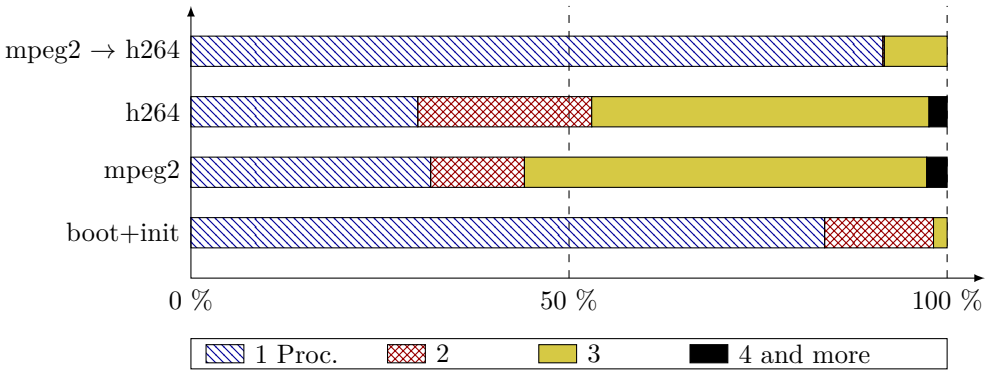
In this part we present the measurements of the number of runnable processes at each cycle, performed by SycView (as in Table 3.2, however, for clarity we show these results on charts).

Figure 3.11 shows the proportion of simulation cycles having a specific number of runnable processes. Figure 3.11a only considers SystemC threads. As an example, we analyse the case of mpeg2 video display. For 53 % of the cycles, there were no threads to run (*i.e.* only methods). For 32 % of the cycles, there were one runnable thread. For 12 % there were two runnable threads and finally for the remaining 3 % there were three threads to run. The h264 case showed the same trend. For the transcoding case, most of the cycles only had one thread to execute. Figure 3.11b shows the measurements considering both SystemC threads and methods. The big picture is that most simulation cycles only contain one to three transitions to run before moving to the next cycle.

This measurement gives an upper bound of the degree of parallelization achievable with approaches that use parallelization inside cycles. The number of runnable processes at the beginning of each cycle is too low to expect interesting speed-ups with such parallelization approaches. Due to shared resources, the real degree of parallelization might be lower. Indeed, a runnable process may share a dependency with another one, making the concurrent run of those two processes not consistent with coroutine semantics.



(a) SystemC threads only



(b) SystemC threads and methods

Figure 3.11: Partitioning of simulation cycles, per number of runnable processes, in four different test cases.

3.4 Influence of Time Ranges on the Number of Processes per Cycle

3.4.1 Discussion on Previous Results

The fact that the number of processes ran during each cycle is low can be explained by at least two factors:

1. Since the implementation of time ranges in production at STMicroelectronics consists in choosing a random time within the range (see Section 2.3.2), there is little chance of having two processes waking up at the same instant. For example, even two processes executing the exact same code will not be temporally synchronized if they use random timing within an interval.

2. Platforms are described at a high abstraction level, where both the time and space granularity are coarse. Instead of modeling the behavior of small pieces of circuit at each clock cycle, as at Register Transfer Level (RTL), the overall behavior of components is modeled, minimizing the amount of `wait` statements. The coarse space granularity leads to fewer processes than at RTL. Since hardware clock cycles are not modeled at Loosely-Timed (LT) Transaction Level Modeling (TLM), the chances of simultaneous execution of multiple processes is reduced.

Item 2 is intrinsic to the way we describe the platforms, but Item 1 is a side effect of the random time choosing policy for time ranges. The idea behind the use of time ranges (instead of time values) was to enable the exploration of different time values. In production code, random time choosing is used, as it may reveal synchronization bugs that are specific to one scheduling. Simulations are still reproducible because the random seed can be set at simulation start to control the generated series of random numbers.

We can try taking advantages of time ranges in another purpose. Since the time values may be chosen arbitrarily within an interval, we can pick a value that increases parallelism. This section describes such implementation and the results on our case study, which shows that Item 1 is actually not significant here.

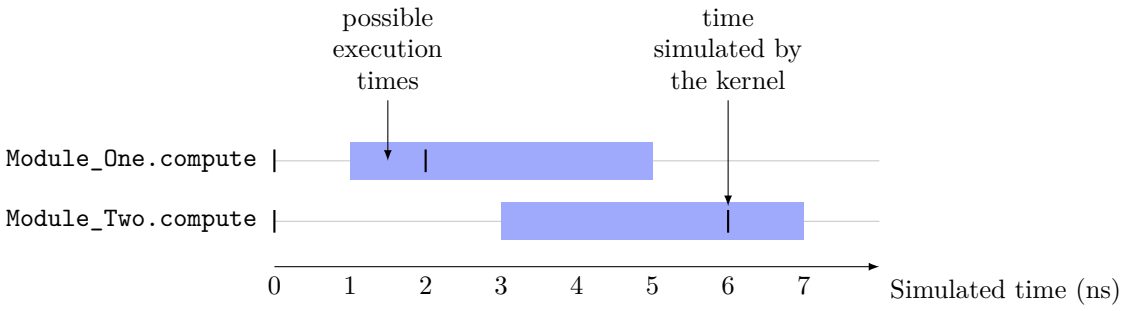
3.4.2 Example

The fact for a process to wait for a time range rather than a time value has the following semantics: *yield to the kernel, and wake up the current thread at any time within the given range.* In any case, the SystemC kernel eventually chooses *one* simulated time instant for

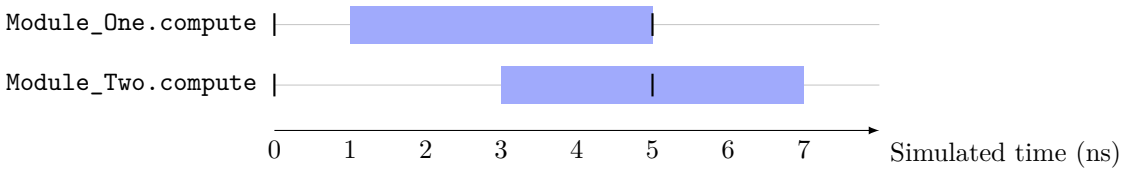
the simulation. But with time ranges, multiple choices are possible, and a careful choice of which time values are simulated can lead to increase the number of runnable processes at specific cycles. As an example to illustrate this, let us consider the following code, in two distinct `SC_THREAD` that we consider independent from each other:

```
void Module_One::compute() {
    annotate_loose_timing(
        sc_time(1, SC_NS),
        sc_time(5, SC_NS));
    synchronize();
}
```

```
void Module_Two::compute() {
    annotate_loose_timing(
        sc_time(3, SC_NS),
        sc_time(7, SC_NS));
    synchronize();
}
```



(a) Random time picking.



(b) “Better” time picking.

Figure 3.13: Execution diagrams of two processes using loose timing, for two different time choice policies: a random value (a) and a value which maximizes the number of runnable processes (b) in the next cycle.

A graphical representation of an execution for this model is shown on Figure 3.13. We used the same formalism as shown in SycView simulation charts: a filled rectangle represent the valid time range for a transition, and the time instants actually simulated are represented by strokes. With this representation, it clearly appears that the number of runnable processes varies depending on which time instants are simulated by the SystemC kernel:

- In the case of Figure 3.13a (random time picking), there are two simulated time instants, each with one runnable process.
- In the case of Figure 3.13b (“better” time picking), there is only one simulated time instant, with two runnable processes. Thus, this case induces a higher degree of parallelism for the simulation.

3.4.3 Implementation of a Time Picking Policy

In its current form, the modification can be implemented neither with the loose timing Application Programming Interface (API) (which does not have visibility on the number of runnable processes at a specific simulated time) nor in the SystemC kernel in its current form (which is called through `wait` statements *after* the random time choice). Thus, for this experiment, we added the following function to the SystemC kernel:

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

We have bound the synchronize/annotate API from ST to this function. Having delegated the choice of the time value to the kernel, there are more information that can be used to choose the time instant. When there are no runnable processes and no pending delta notifications, the SystemC kernel picks the first event from the event queue and set the value of simulated time to its timestamp. We changed the way time elapses in the kernel. The first change modifies the events, so they don't have one but two time values, representing the range in which they are considered valid. Thus, when the kernel needs to elapse simulated time, it will go through the list of recorded future events, and compute a time value based on the ranges of those events. We want this time value to maximize the number of runnable processes in the next cycle. In the following explanations, we call this time value $t_{\max}$. Moreover, we assume that:

- the kernel always triggers an event if its time range includes the chosen time value,
- the old `wait` using one time value is replaced by the ranged `wait` with the same value for both bounds.

Let I_E be the time interval given for an event E, and T be the set of timed intervals registered in the kernel. We define $t_{\max}$ as follows:

$$t_{\max} = \min\left(\{\, \max(\mathrm{I}_E) \mid \mathrm{I}_E \in \mathrm{T} \,\}\right)$$

In other words, it is the minimum value of all the maxima of the registered timed ranges, as illustrated by the example on Figure 3.14. In particular, we can prove two properties about $t_{\max}$.

Property 1: Given that the simulation is currently at a given time instant, $t_{\max}$ is a correct choice as next instant.

Property 2: $t_{\max}$ maximizes the number of runnable processes of the next cycle.

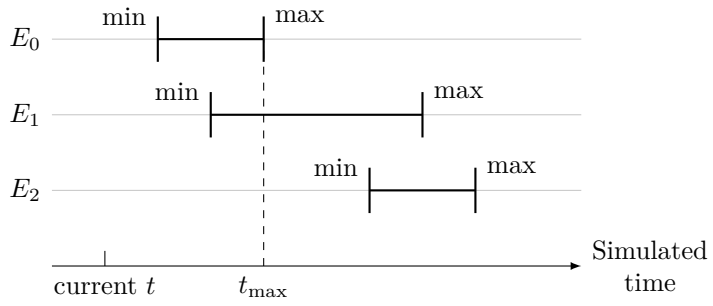


Figure 3.14: Example of a $t_{\max}$ value for three events.

Property 1 is correct if choosing $t_{\max}$ does not skip any event. An event E is skipped if $t_{\max} > \max(\mathrm{I}_E)$. However, by definition of $t_{\max}$, such a process does not exist, because $t_{\max}$ is the minimum of $\max(\mathrm{I}_E)$ for each event E. Thus, Property 1 holds.

To prove Property 2, let $\mathrm{E}_{\max}$ be the set of events that will be triggered if $t_{\max}$ is chosen, and F a future event such that $F \notin \mathrm{E}_{\max}$ (on Figure 3.14 for example, $\mathrm{E}_{\max} = \{E_0, E_1\}$ and $F = E_2$). "F not triggered" means that F's minimum triggering date is in the future, thus $t_{\max} < \min(\mathrm{I}_F)$. Let G be one of the events whose upper bound equals $t_{\max}$ (such an event exists by definition of $t_{\max}$, on our example $G = E_0$). Then $t_{\max} = \max(\mathrm{I}_G)$ which means that $\max(\mathrm{I}_G) < \min(\mathrm{I}_F)$.

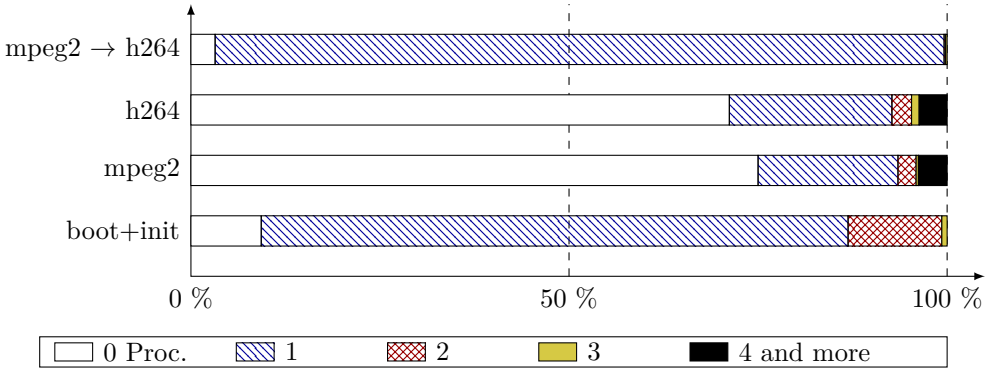
Now if we try to add F to $\mathrm{E}_{\max}$, the minimum value we could take for the time is $\min(\mathrm{I}_F)$. Yet, since $\max(\mathrm{I}_G) < \min(\mathrm{I}_F)$, choosing this value would exclude G from $\mathrm{E}_{\max}$. In other words, if we include F in $\mathrm{E}_{\max}$, we immediately exclude G, which violates Property 1 shown before. Therefore, a process F (not initially in $\mathrm{E}_{\max}$) cannot be added in $\mathrm{E}_{\max}$, which means that $\mathrm{E}_{\max}$ already contains the maximum number of processes for the next cycle. Thus Property 2 holds.

Two points are to be noticed. First, there may be multiple values that fulfills Properties 1 and 2. In our example, all the values between the minimum of E_1 and the maximum of E_0 are valid. The defined value $t_{\max}$ is only one of those values that it is easy to compute. Second, this only maximizes the number of processes *for the next cycle*. An optimal time choosing policy would rather try to globally maximize the number of runnable processes, instead of just maximizing it for the next cycle. However, this would require a very complex code analysis to anticipate the influence of the execution of each transition on the rest of the simulation, in terms of runnable processes. On the contrary, the chosen policy is simple to implement and we will see that it did not show promising results, thus this track has not been explored further.

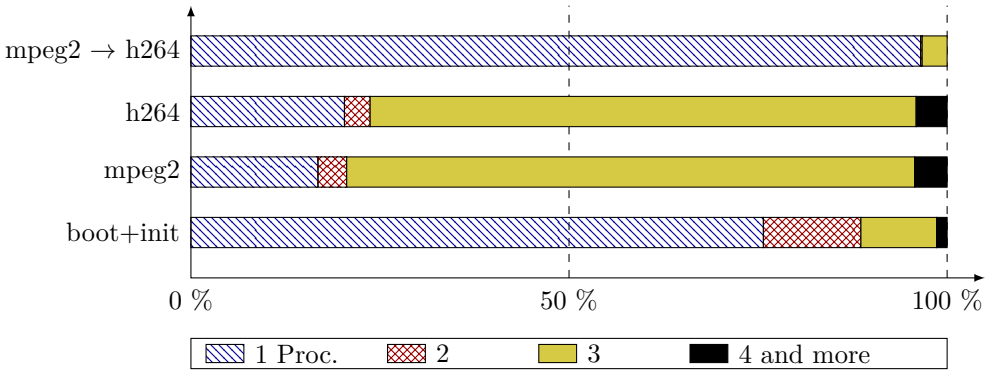
3.4.4 Results

We have run again the simulations of our industrial platform for the same test cases to measure the same metrics with this modification. Figure 3.15 shows that the number of

processes is still very low, even though there are more occurrences of cycles with 3, 4 or more runnable processes.



(a) `SC_THREAD` only



(b) `SC_THREAD` and `SC_METHOD`

Figure 3.15: Partitioning of delta cycles, per number of runnable processes, for four different test cases, with optimized time picking within time ranges.

From those results, we see that trying to run multiple processes in the same simulation cycle in parallel will not be an efficient parallelization approach. We have seen in Figure 3.9 that most of the simulation duration is spent in SystemC threads. Thus, let us consider Figure 3.15a: even if, say 5 % of the simulation is accelerated by running the four runnable threads in parallel, Amdahl's law tells us that the expected speed up is 1.04 at best. Moreover, we completely neglected dependencies between processes, which can forbid the concurrent execution of specific transitions. This question is further discussed in the survey on existing parallelization approaches, presented in the next chapter.

3.5 Conclusion

According to the previously presented results, we can establish a list of characteristics for our case study as follows:

- Most of the wall-clock time is spent on models of Intellectual Property (IP) blocks (Figure 3.10).
- Most of the wall-clock time is spent on SystemC threads (Figure 3.9).
- Most of the simulation cycles contain less than four runnable SystemC processes (Figure 3.11) and even if we exploit time ranges to maximize this value (Figure 3.13), the results remain similar (Figure 3.15).
- In cycles containing several runnable processes, most of the time there is zero or one SystemC thread, the remaining being methods (Figure 3.11 and Figure 3.15). The wall-clock time consumed by these methods is very low compared to the time consumed by the threads (Figure 3.9).

Obviously, those characteristics may vary depending on the test scenario. However, they have been observed for common uses of a set-top box, *e.g.* video display tests, that are run several times during the system on chip development. In this part, we gave the reader an idea of the complexity of such types of platforms. We also highlighted the fact that in our case, the complexity is located in hardware IP blocks, and not in the embedded software. Consequently, the performance problem we try to solve mainly comes from the models of IP blocks, and not from the Central Processing Unit (CPU).

The parallelism exposed by our case study is, at first, not promising. Indeed, the number of runnable processes at each simulation cycle is very low, 4 at best for 5 % of the cycles at best. Thus, an immediate implication of the results from this chapter is that if we want to find a parallelization approach to speed up such models, we must find a potential for parallelization elsewhere than in the number of runnable SystemC processes.

— Chapter 4 —

Survey: Existing Parallelization Approaches

4.1 Introduction

Parallel Discrete Event Simulation (PDES) consists in running a Discrete Event Simulation (DES) on a parallel computer. In this manuscript, we use the term PDES with this general meaning [22]. Research work on PDES includes theoretical techniques, and some of them have been applied to SystemC simulations. For each technique, we remain general in the theoretical description and explain in further details the implementations for SystemC. Then, this opens the discussion on the applicability of the existing techniques for Loosely-Timed (LT) Transaction Level Modeling (TLM) models, such as our case study, for which we enumerated a list of properties in the previous chapter.

4.1.1 Different Types of Simulated Architectures

Systems on chip include different types of designs and architectures. This section reminds the most common ones.

4.1.1.1 Shared Resources and Hardware Acceleration Blocks

Multi-Processor System on Chips (MPSoCs) are systems on chip with more than one generic purpose processor and heterogeneous hardware components. A Symmetric Multi-Processing (SMP) architecture is generally used in this case. The processors share most of the hardware resources, including memories, as shown on Figure 4.1. There may be multiple processors, however, the performance of such systems mostly relies on the efficiency of hardware acceleration components. The processors generally run a classic Operating System (OS).

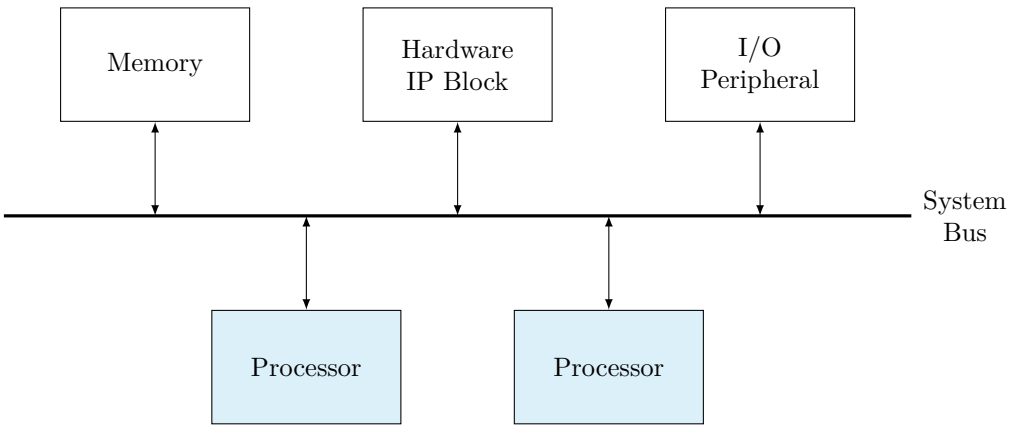


Figure 4.1: Example block diagram of an MPSoC model.

4.1.1.2 Massively Parallel Architectures: Many-Cores

Contrary to the previous architecture, some systems do not or barely use hardware acceleration blocks. Instead, the system consists in a many-core processor, counting a large number of processing cores (generally more than 16). Individually, cores are slower than in the previous architecture, but good performances are obtained thanks to the parallel execution of the embedded software. In such systems, the embedded software is partitioned in several processes: it is finely tuned for efficiency on massively parallel architectures. Thus, many-core, or Massively Parallel Processing (MPP) systems on chip are used when a high degree of parallel processing is needed. Many-core systems on chip often use the Network on Chip (NoC) architecture, which is easily scalable. Figure 4.2 illustrates an example of NoC for a many-core system. Each memory contained in the cores is private and cannot be accessed from other cores.

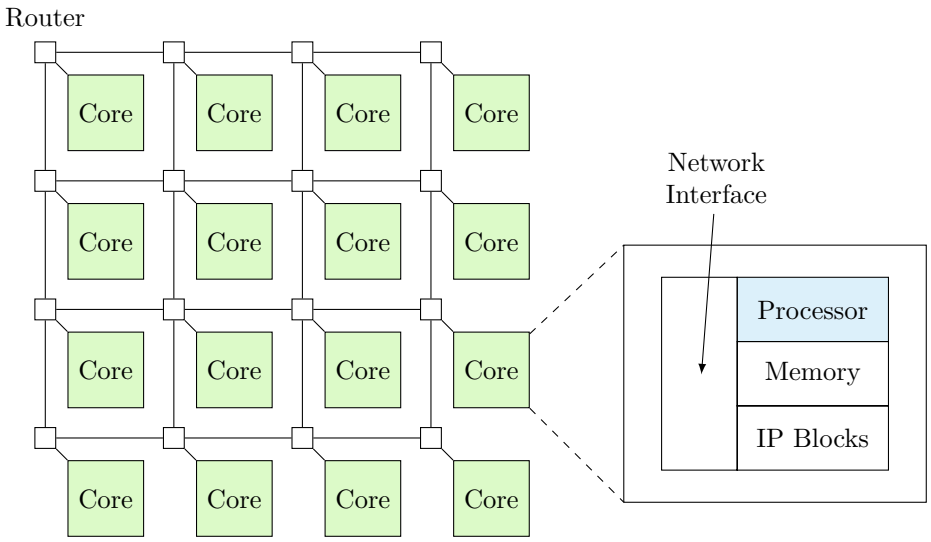


Figure 4.2: Example block diagram of a NoC architecture for a many-core processor, like Kalray's MPPA [23].

4.1.1.3 Consequences on Models

The two architectures presented here are not incompatible with each other: a processor from Figure 4.1 can be a many-core processor using a NoC like on Figure 4.2. This variety in systems on chip is reflected on their models, which leads to very different types of simulations. The NoC architecture is used precisely to avoid synchronizations between clusters as much as possible. This specificity can be exploited to speed up the simulations: because of the natural partitioning and isolation of the clusters, there are few obstacles to running them in physical parallelism. Moreover, the isolation between clusters induces that communication costs are not performance critical. In the case of an architecture with hardware acceleration blocks, of which our case study is an example, there are hardware synchronizations between components. Most of them are performed through system buses.

We saw from profiling results that there is, at first sight, little potential for parallelism in this type of simulations. Consequently, the same parallelization approach will not work for both types of simulations.

4.1.2 A Reminder on SystemC Semantics

Before detailing the approaches for parallel simulation, we clarify one point. The following inference is common in research work:

> *SystemC simulation semantics state that within a delta cycle, the execution order of processes it not predefined. Thus, all the runnable processes of a delta cycle can be run concurrently.*

The first sentence is correct with respect to the SystemC standard. However, the "thus" implying the second sentence is technically incorrect, because of possible race conditions. Indeed, if two SystemC processes modify the same variable, a physically concurrent run of those processes produces a race condition on this variable. In fact, the SystemC language reference manual already covers this [1, p. 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 to constrain their execution to match the coroutine semantics.*

The last sentence indicates a technical solution to fulfill coroutine semantics. A parallel SystemC kernel, that enables the parallel run of any kind of SystemC program, must indeed analyze dependencies. However, a formal dependency analysis can be avoided if assumptions on inter-process dependencies are made. For example, at the Register Transfer Level (RTL) abstraction level, one assumption could be that processes do not share any variable, and the only form of inter-process communication is achieved through SystemC signals. With this assumption, a parallel SystemC kernel, that correctly handles concurrent accesses to signals, can be considered valid. At the TLM abstraction level, there are many shared resources due to the use of Interface Method Calls (IMCs).

4.1.3 Introductory Example

In this chapter, we use the following example to explain the main categories of existing PDES techniques. The example consists in three SystemC threads, whose execution planning is represented on Figure 4.3. This representation uses the same visual conventions

as the SycView simulated time charts from the previous chapter. Figure 4.4 represents the work of the host processor running this SystemC simulation. At each time, the figure indicates which SystemC process the simulator is currently running (the time spent in the kernel itself is neglected in this figure). With a sequential simulator, only one process is running at each wall-clock time instant. The figure represents a possible scheduling, but it is not unique. For example, at simulation start, processes could have been run in a different order.

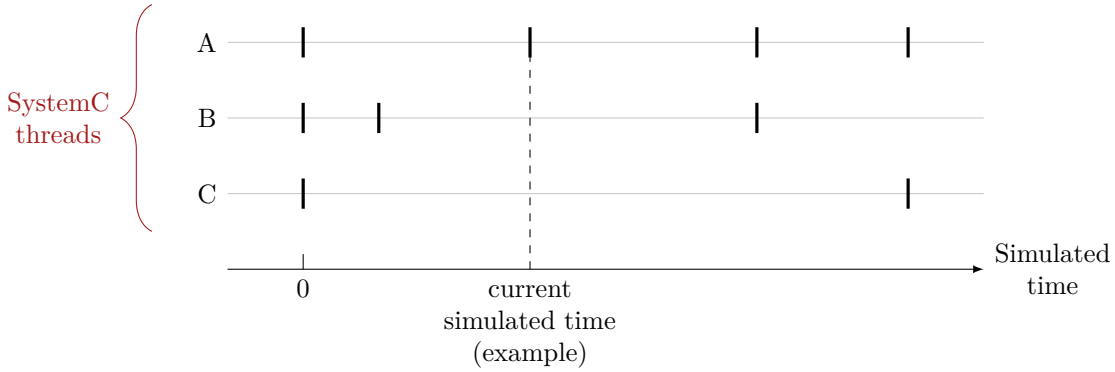


Figure 4.3: Planning of three SystemC threads. A vertical stroke represent a simulated time instant where the process has a transition to run. An example current simulated time instant is indicated.

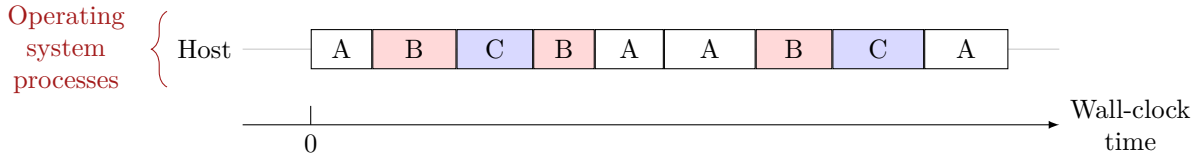


Figure 4.4: Possible execution of this simulation in a sequential simulator such as the reference implementation of SystemC.

4.2 Overview

To achieve the parallel simulation of a model, one must find potential parallelism in the model, and use parallel computing resources to exploit the parallelism from the model. The second point is not the main problem in our case: recent computers have multiple processors or cores, have powerful Graphics Processing Units (GPUs), and multiple computers can also be used to get more parallel computing power. The problem is to find parallelism in the model, exploitable by parallel computing resources, while keeping the results correct and of course achieving better performances.

This chapter presents the different approaches for parallel SystemC simulation, summed up on Figure 4.5. We start with space partitioning techniques in Section 4.3, that keep simulations conservative with respect to simulated time. Then, Section 4.4 presents techniques that exploit the relaxation of constraints to increase parallelism. This section includes most parallel simulation approaches. The following sections are discussions, on simulation semantics in Section 4.5, and on simulation replication and time partitioning in Section 4.6. The two latter approaches are not applied to SystemC, or not to speed up simulations in a development context.

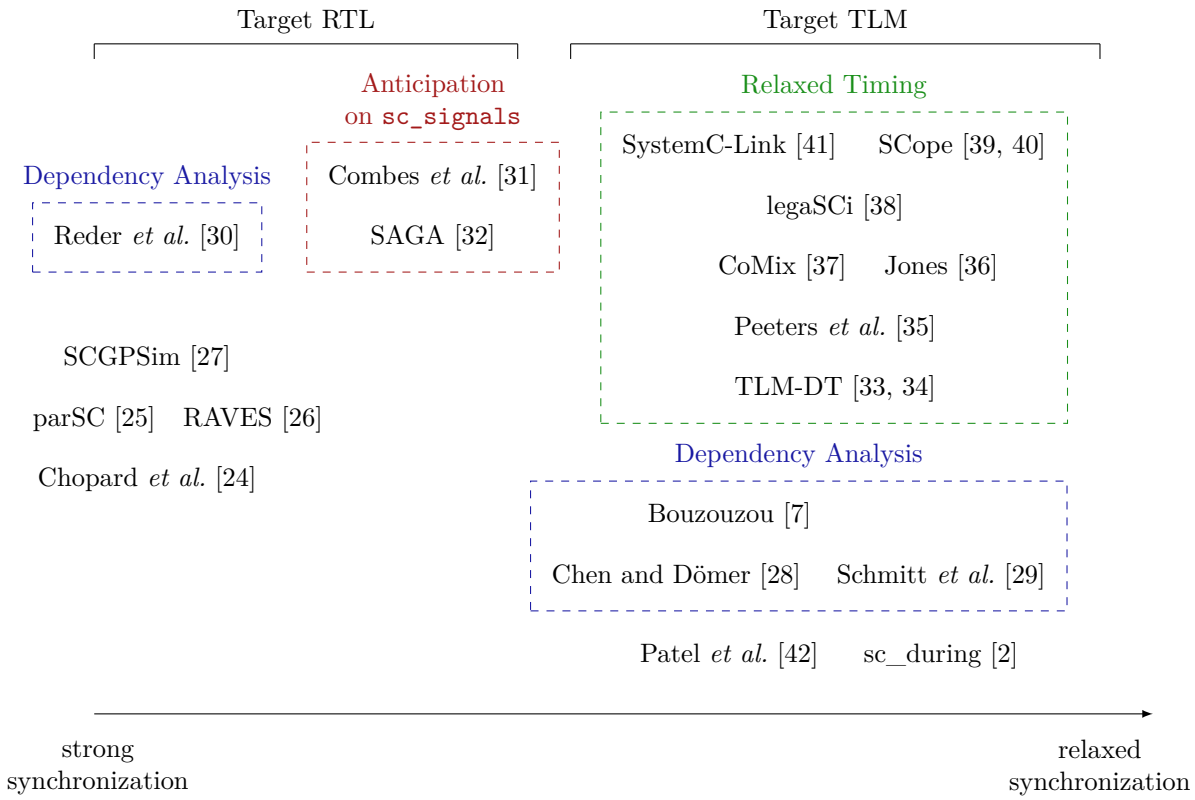


Figure 4.5: Existing work on parallel SystemC simulation.

4.3 Space Partitioning

4.3.1 Presentation

Space separation consists in partitioning data and computations, and run them in physical concurrency. Implicitly, we assume that in the case of space separation, all the different workers are synchronous with each other in terms of simulated time. These approaches are known as *conservative* parallel simulation approaches. Formally, a *conservative* synchronization mechanism keeps the same order of simulation cycles, and the same set of transitions per cycle as in sequential simulation [43].

In SystemC, the strongest synchronization mechanism consists in allowing only processes that are in the exact same simulation cycle (*i.e.* same time and same delta cycle) to be candidate for parallel execution. An obvious implication is that the number of runnable processes at each cycle is a limiting factor to the achievable speed up. Figure 4.6 shows an example of a conservative parallel simulation for our example.

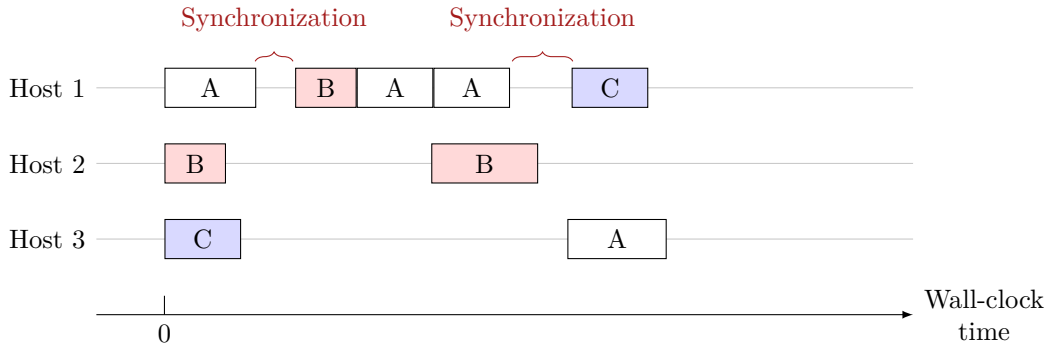


Figure 4.6: Example of a parallel execution with conservative synchronization. The performance costs due to synchronization are illustrated as “idle” moments.

4.3.2 Applications to SystemC

Chopard *et al.* proposed a parallel implementation of a SystemC kernel using conservative PDES [24]. Multiple SystemC scheduler instances are created, and run on different computing units. The SystemC processes of the model are partitioned into groups, attributed to one of the schedulers. There is a synchronization barrier between simulators at each delta cycle. This is an example of a strongly conservative synchronization mechanism. However, this approach typically uses the assertion in Section 4.1.2, assuming that processes from a delta cycle can be run in physical parallelism. In this case, the approach targets models at Register Transfer Level (RTL) level, that communicates with SystemC channels. Thus, the authors assume that shared variables only consist in channels, which

is a reasonable assumption at RTL level. In this approach, the time elapse mechanism is centralized in a master simulator. Before every time cycle, each simulator sends its next simulated time value to the master, which answers back with the minimum when it has answers from each simulator. This approach is efficient in specific cases, namely where several processes from different modules are runnable at each simulation cycle. When this is not the case, synchronization overheads are very high, and the master simulator acts as a performance bottleneck.

To address these limitations, Combes *et al.* have studied possible solutions to relax the synchronizations, while remaining conservative [31]. For example, they propose to use the information on SystemC signal bindings to anticipate the update of signals when it is known that the value will not change (or not be read) in the current delta cycle. Therefore, the relaxation is still limited to delta cycles. Another idea is to decentralize the simulation time between simulators, contrary to their first approach [24]. The simulators are still in the same simulated time instant, but it saves some communications: it removes the communication bottleneck of having a master simulator. The number of processes runnable at each cycle stays a strong limiting factor.

A parallel SystemC kernel called *parSC* has been proposed by Schumacher *et al.* [25]. The simulation kernel used in parSC is a modified version of the reference implementation. There is one master thread, that manages the states of the simulation, and several worker threads. Each worker thread runs transitions during an evaluation step. A synchronization barrier is done at the end of each evaluation step. Then, the master thread performs the update requests and further notifications. This approach is similar to the previous ones, except the fact that it is a parallel implementation of a SystemC kernel, whereas the first ones use different sequential SystemC kernels to achieve parallel simulation.

4.3.3 Discussion

The approaches presented in this section are all located on the left part of our overview graph (see Figure 4.5). This left part corresponds to strongly conservative synchronization mechanisms. We also indicated on the figure that such approaches target RTL models. The first reason for this is that the implementation uses features of SystemC for RTL modeling, such as `sc_signal` or clocked threads. The second reason is that such type of parallel execution is naturally efficient when a lot of independent work has to be performed at each simulation cycle. This corresponds to simulations at low levels of abstraction, which models the internal details of each component, as at the RTL abstraction level.

With conservative approaches, the number of runnable processes per simulation cycle is a dramatic limiting factor of the efficiency of parallel simulation. Yet, we focus on models where most of the wall-clock time is elapsed by SystemC threads (minimum 90 % in our study) and most of the simulation cycles (minimum 70 % in our study) only have one runnable thread, or even only methods to run.

4.4 Relaxed Parallel Simulation

Most of parallel simulation approaches for SystemC use both space and time separation (the slider being much closer to space than time). To increase the potential for parallel execution, compared to conservative approaches, relaxed synchronization mechanisms are used. Processes from different cycles can be allowed to run in parallel under some conditions. Those conditions can be checked by an analysis tool, or assumed to be true.

4.4.1 Common Techniques

4.4.1.1 Lookahead Time

When it comes to relaxing synchronizations in a conservative approach, the use of a lookahead time is a commonly used technique. It consists in allowing different parts of a simulation (or different simulators working together) to be at a different simulated time instants. The maximum time difference allowed within a simulation is defined by a fixed value, the *lookahead time*. Figure 4.7 shows an example of parallel simulation with a lookahead time. A possible corresponding execution is shown in Figure 4.8. We can see that this execution took less wall-clock time than the one in Figure 4.6, thanks to the relaxation of the conservative synchronization with the lookahead time.

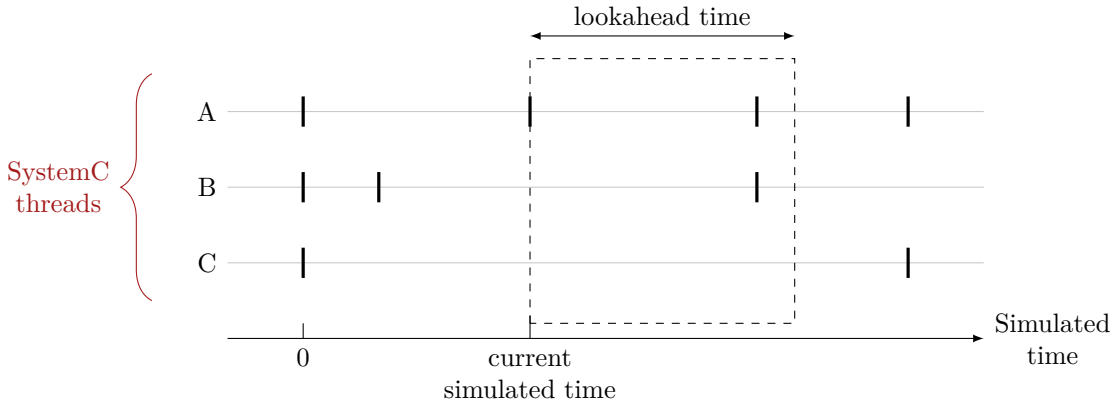


Figure 4.7: Possible simulated time state with a lookahead time. Graphically, the simulated time can be seen no longer as a line but as a window.

The benefits of a lookahead value comes at a price. A lookahead value of zero does not relax anything compared to standard simulation (thus without any benefit), and a too high lookahead value will cause problems during the simulation, by breaking causality. Consequently, the choice of a lookahead value is let to the model developer, as it depends on the simulated architecture [39]. Different choices for lookahead values are discussed in the following approach presentation.

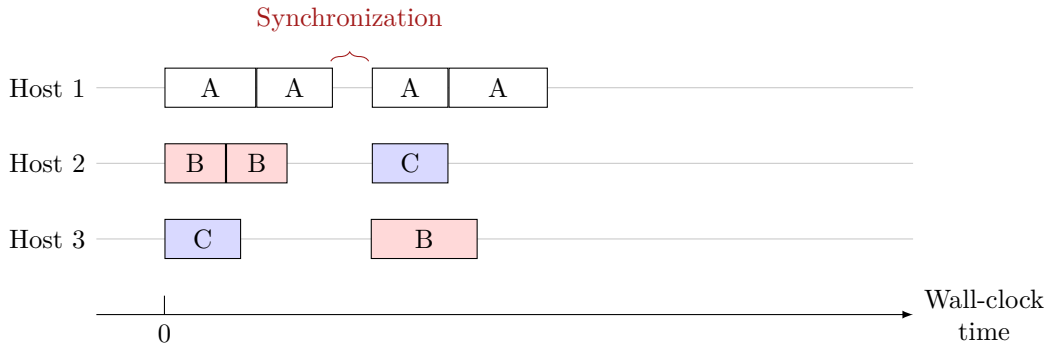


Figure 4.8: Possible execution for our example, using three host simulators and a lookahead time as shown on Figure 4.7.

4.4.1.2 Optimistic Synchronization

Optimistic synchronization mechanisms are an answer to the inefficiency of conservative mechanisms when there are not enough processes to run at each simulation cycle, even with some relaxed constraints [43]. Optimistic synchronization relies on two mechanisms. The first one is a forecast mechanism, that enables some processes to take advance on others. By definition, this forecast mechanism is sometimes wrong. In this case, a rollback to a previous valid state is done. This is the second mechanism involved. Figure 4.9 shows an optimistic simulation for our example. To rollback a simulation, state saves are needed, which implies to record the value of each variable and the state of each process. Further difficulties of using an optimistic mechanism for SystemC simulations have been discussed by Trams [44]. To the best of our knowledge, there have been no parallel SystemC simulation approach using optimistic synchronization, *i.e.* with a forecast and rollback mechanism. This is mostly due to the fact that a rollback mechanism is costly both to implement and at runtime.

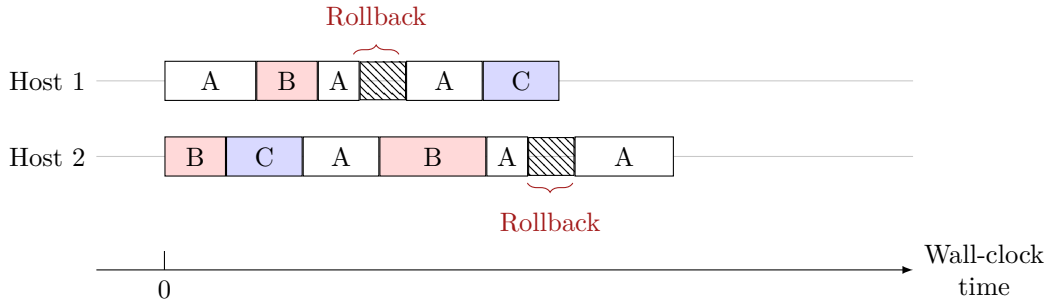


Figure 4.9: Possible execution for our example, with two host simulators using optimistic synchronization. When wrong predictions have been made (on timing for example), the process rollbacks to a previous state.

4.4.1.3 Platform Partitioning

Partitioning the platform is a common technique to specify that parts of a model can be run quite independently with each other. This does not mean that they do not communicate *at all*, but such partitioning can be used to make assumptions. For example, one can ask the user to define partitions such as there is no shared variables between them (except with TLM transactions). Formally, this is a "space separation", however it presented here because it is often used to desynchronize different partitions in terms of simulated time.

In SystemC terms, partitioning the model consists in partitioning the different SystemC objects of a simulation. Each partition is simulated with a different instance of a SystemC scheduler. Bounds between objects that go in different partitions are replaced by inter-simulator connectors (the terminology varies with the approaches). Partitioning is most of the time done manually, *i.e.* by the simulation user. Figure 4.10 illustrates an example of partitioning for different model architectures, previously presented. In models with a central system bus, a natural cutting point is across this bus. For many-core architectures, a natural partitioning is groups of clusters.

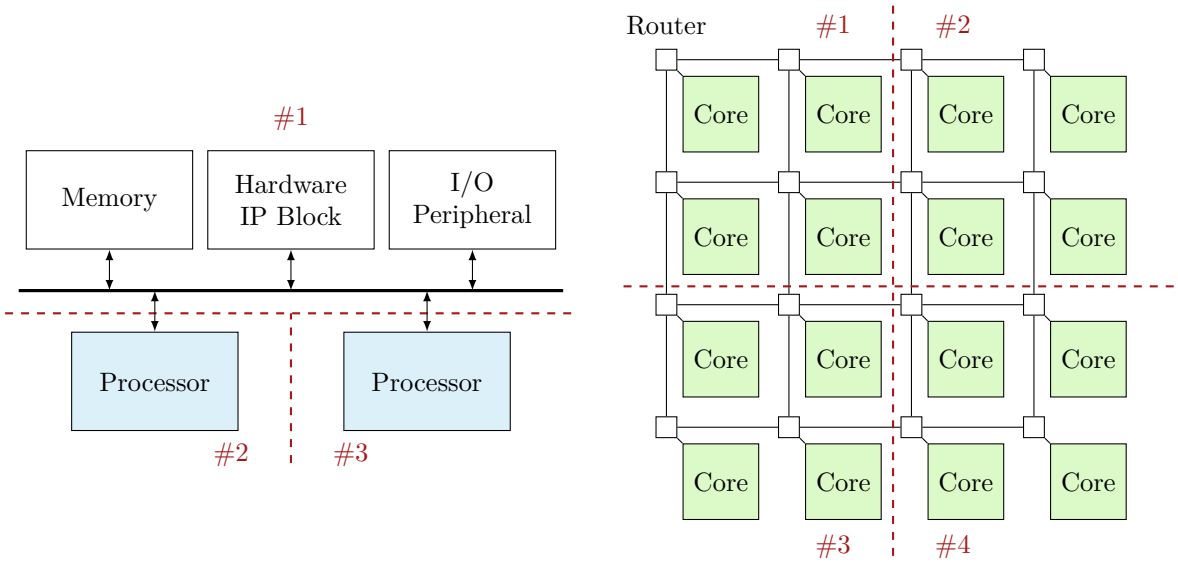


Figure 4.10: Example of partitioning for two different platform architectures.

4.4.2 Application to SystemC

In 2004, Mario Trams proposed a parallel simulation approach for SystemC using lookahead time [44]. This work targets RTL simulations and is not applicable as-is, because it leaves many future work directions. Most of them are addressed by more recent papers.

Viaud *et al.* proposed a set of modeling rules to describe multi-processor systems on chip at the Transaction Level Modeling (TLM) abstraction level for parallel simulation [33]. The key idea is to distribute the simulated time to each processor of the simula-

tion. Thus, the SystemC simulated time is no longer used. Each component advances its local simulated time depending on messages received from other components. A message can be a request packet, a response packet or an interrupt packet. To prevent deadlock situations, the simulators need to send *null messages*, as shown on Figure 4.11. Null messages are messages that only contain a simulated timestamp. The handling of interrupts is done by defining timestamped interrupt messages. Each Central Processing Unit (CPU) loop starts with a check on interrupt messages. This removes the asynchronous characteristic of interrupts, but guarantees that they will be handled in a meaningful time. Acceleration is obtained by trading "accuracy" for "speed up" by using a lookahead time, that must be specified by the model developer. Higher lookahead values cause a lost of accuracy in the simulation of processors (which are simulated cycle by cycle).

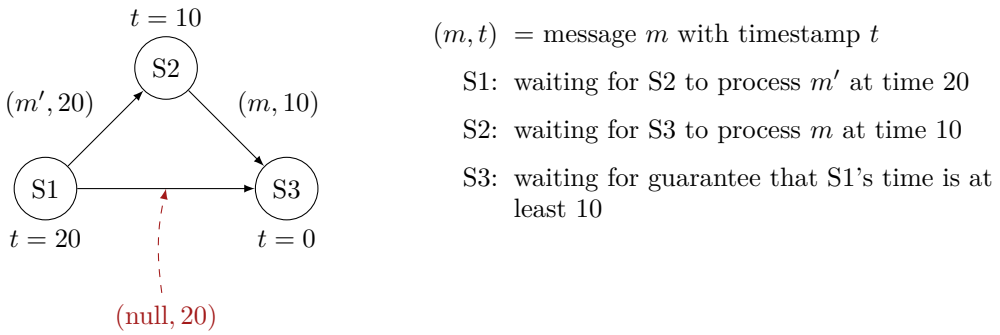


Figure 4.11: Example of potential deadlock situation, if no *null messages* are sent in a conservative parallel simulation. The null message sent by S1 to S3 tells S3 that S1 will not send messages anterior to that null message, thus it unlocks the situation.

The previous set of modeling rules was later extended by Mello *et al.* as *TLM-DT* (TLM with Distributed Time) [34]. A parallel SystemC engine called *SystemC-SMP* has been developed to simulate models developed with TLM-DT. In TLM-DT, the behavior of components is modeled using SystemC threads only. The only yield operations allowed are delta notifications and waits on events. To the best of our knowledge, there have been no study on the migration cost of a complete TLM model to TLM-DT. Also, there is no compatibility between non-TLM-DT and TLM-DT modeling styles (time management, handling of interrupts, SystemC primitives allowed). The conducted experiments were done on the model of a multi-cluster Network on Chip (NoC) [34]. In this case, computations are CPU-intensive, the embedded software is developed to exploit the parallelism of the architecture, and running the clusters in parallel indeed leads to a good speedup (*e.g.* a speedup of 1.9 using 2 cores for simulation). In our case study (see Section 3.3), the major slowness comes from the complexity of hardware acceleration blocks that, for example, intensively exchanges video streams. The embedded software is run mostly on one CPU of the platform. With such a distributed simulation approach, the platform first needs to be partitioned. Considering our profiling results, the partitioning would occur near hardware acceleration blocks, exchanging high definition video streams. Thus, using a timestamped-message passing technique in this situation will lead to a communication bottleneck.

Peeters *et al.* proposed a simulation framework, reusing some ideas from TLM-DT, but without changing the modeling rules [35]. The model developer has to partition the model into clusters, each cluster will be run on a different simulator, concurrently with the other ones. Each inter-cluster TLM transaction is turned into an asynchronous call. This means that the initiator continues to run while the target processes the transaction. To avoid temporal inconsistencies, two synchronization mechanisms can be used: "implicit" and "explicit". The "implicit" one is an *exclusive write policy*: a writer must wait for all the readers to finish reading before writing. The "explicit" synchronization is performed by simulation-specific modules, one for each cluster, that bounds the time intervals between the different clusters by exchanging messages. The models experimented with this approach have a high degree of parallelism (high number of tasks) and consists in many-core platforms (64 CPU with 64 memories). Thus, even with a high degree of interconnection, relaxing synchronizations leads to very good speed-ups with such models.

Jones developed an optimistic parallel simulation approach. The optimistic term in this approach does not refer the usual definition of "optimistic" in parallel simulation (*i.e.* with a forecast/rollback mechanism) but to a weak synchronization mechanism, with a lookahead time [36]. The platform is divided into groups of modules called partitions. Each group is simulated by a different instance of a SystemC simulator. This approach makes strong assumptions about variables shared between partitions: it considers that shared variables (except the ones from the SystemC kernel) has been either well protected or purposely not protected. The value of lookahead time is advised to be set to a value below the frequency at which different partitions have to communicate. This value depends on models, and no general rule can be used. One possible empirical solution proposed is to run different simulations with growing lookahead time values, until the simulation fails. However, this solution does not guarantee that this lookahead time is still valid after only a slight change in the model.

A methodology called *legaSCi* is proposed to integrate existing SystemC models into parallel SystemC simulators, with a focus on achieving thread-safety [38]. Similarly to the previous proposition, the key step involves the model developer, who must describe partitions, here called *containment zones*, in the model. The definition of zones consists in partitioning the SystemC objects (*e.g.* modules, channels or sockets). A process group is created for each zone, with all the processes defined in the zone. Within a zone, the processes are scheduled sequentially. Containment zones are independent with each other, *i.e.* they do not share variable dependencies. Thus, parallel execution is not achieved when a model contains many data inter-dependencies. In particular, with TLM, each Interface Method Call (IMC) leads to a possible race condition. To answer this specific problem, the authors propose to intercept IMC, and change the group of the initiating process to the target group. This solution is technically possible, however it has drawbacks: either the other processes from the initial group must wait for the completion of the IMC, then there is no parallelism, or they can go on and this inserts a yield point which breaks the atomicity of TLM transactions.

CoMix is a concurrent model interface library for distributed SystemC simulation, developed at Cadence Design Systems [37]. The model developer must partition the

model, each part is connected to the other through a bridge component. Each partition is running in a separate process, with its own SystemC kernel instance. The bridges relay sockets between peers, thus a natural cutting point is around a system bus. Lookahead time is used in the form of *credit*, each simulator can get a different credit value. The credit is mostly used to group processor instructions, which indeed enables to get an acceleration for loosely-timed models. Nonetheless, this work mostly answers to performance problems coming from the simulation of models of many-core systems on chip. Such models offer natural and most possibly efficient cutting points, since the workload is balanced between the different cores.

SCope is a parallel simulation kernel focusing on support for TLM communications [39, 40]. Multiple SystemC schedulers are instantiated on different worker threads. It uses a similar idea as the one proposed for legaSCi [38], which is to partition the simulation into safe zones. The only allowed form of communication between process from different zones is using a *remote event* and a *remote transactions* mechanism. A lookahead time is used to relax synchronizations. To avoid causality errors when using remote events, this lookahead time must be set to a value that depends on the simulated system. In fact, the notification of remote events must be delayed by at least the lookahead time value. This needed delay is the same when using remote transactions (*i.e.* through different simulators). The remote transaction mechanism is implemented in target sockets. It checks if the initial thread belongs to a different group than the target object, and if so, puts the transaction to a queue. A relay process, in the same group as the target, then dequeues transactions and process them sequentially. SCope has been tested on the TLM model of a NoC with 64 tiles, each one having private memory and a processor (for which the model is cycle-accurate). As mentioned on Section 4.1.1, the natural partitioning and regularity of the NoC architecture fits well with this parallel simulation approach, that requires manual partitioning of the system.

One common point of Parallel Discrete Event Simulation (PDES) approaches applied to SystemC is that the reproducibility is hard to ensure, since multiple processes from the same simulation cycle are run in parallel. A SystemC kernel called *SCale* address this issue, by adding a replay mechanism based on traces recorded during simulation [45]. SCale also uses an annotation mechanism on shared resources to detect race conditions during runtime. This does not prevent race conditions to happen, but it informs the model developer where to insert protection mechanisms in the model to ensure thread-safety during parallel simulation.

Weinstock *et al.* proposed another parallel simulation framework called *SystemC-Link* [41]. The formerly called *zones* are here referred to as *simulation segments*. It is still required that the model developer identifies the segments. The major difference with SCope is the use of a simulation controller that manages the different segments. Again, the efficiency of parallel simulation depends on temporal decoupling. This makes the approach best fit for cycle-accurate models, with precise timing, where relaxing temporal constraints removes more constraints than in Loosely-Timed (LT) models.

4.4.2.1 Applicability in our Case

The approaches presented in this section mostly target simulation of many-core platforms. Indeed, such platforms offer a natural partitioning and a strong independence between the cores. Moreover, the efficiency is best when multiple processors are on different partitions. In this case, with a cycle-accurate Instruction Set Simulator (ISS), a substantial benefit can be taken out of relaxed timing with a lookahead time.

In our case, we do not model many-core platforms where the software is the major part of a simulation, but platforms with Intellectual Property (IP) hardware acceleration blocks, which mostly are the performance bottleneck. The simulation of processors have already been made fast enough for our software needs, *e.g.* using native host code execution. Moreover, the exact effect of a lookahead time on a platform with hardware acceleration blocks has not been studied yet: most approaches using a lookahead time focused on simulations of many-core platforms with cycle-accurate CPUs.

4.4.3 Approaches Based on Dependency Analysis

A generic and semantics-preserving parallel SystemC simulator needs a dependency analysis mechanism to identify independent processes. This section sums up the approaches based on dependency analysis to achieve parallel simulation. Figure 4.12 illustrates the general principle of dependency analysis. The different approaches presented in this section use various mechanisms to find the dependencies (*e.g.* static analysis, dynamic analysis) and have different definitions of a dependency.

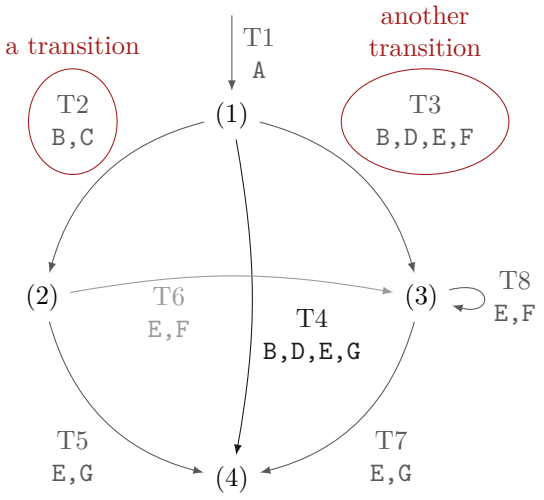
4.4.3.1 Presentation of the Approaches

Bouzouzou presented a parallel SystemC simulator and proposed to use static code analysis to identify dependencies between SystemC transitions [7]. A static analysis tool first scans the model in order to produce a dependency scheme. Then this dependency scheme is provided as input to an altered SystemC simulator, which uses it to schedule different transitions in parallel. A runnable process can be run if it is independent with each currently running process. However, the proposed approach has a limiting factor due to IMCs. When a transaction is sent to a system bus, the address is decoded to determine the transaction target. During the static analysis, this address decoding is not done, which implies to suppose that any target component bound to the same bus is a potential target. Thus, in the case of a system where a central system bus exists, this supposition is too pessimistic and leads to no physical parallelism.

Segment conflict analysis is introduced by Chen et al. [46]. In the context of the authors, a *segment* is a portion of code executed by a SystemC thread between two scheduling steps, what we call a thread *transition* in this manuscript. A *segment boundary*

```
A();
wait();              // (1)
if (B()) {
    C();
    wait();          // (2)
} else {
    D();
}
while (E()) {
    F();
    wait();          // (3)
}
G();
wait();              // (4)
```

(a) Pseudo-code of a thread.



(b) Transition graph for this thread.

	T1	T2	T3	T4	T5	T6	T7	T8
T1		×			×			
T2	×						×	×
T3				×			×	
T4			×					×
T5	×					×	×	×
T6					×			
T7		×	×		×			
T8		×		×	×			

(c) Dependency relations between transitions. A cross in the table means that the two tasks share a dependency and cannot be run in physical parallelism.

Figure 4.12: The different steps of dependency analysis [46]: from user-code (a), identify the different transitions (b) and then find dependencies between the transitions (c).

is a statement that calls the scheduler (*e.g.* `wait`). A *segment graph* is a representation of a SystemC model. It is an oriented graph, where boundaries are nodes and segments are transitions between nodes. Thus, a boundary is connected to another if and only if they can be executed in direct sequence (considering only boundaries) in a simulation. A conflict table is built with the graph, to associate to each pair of segment a boolean value telling if the segments share dependencies or not.

Following this, Chen and Dömer use the conflict tables to implement an out-of-order simulator [28]. Three types of conflict tables are made: one for data hazards, one for timing hazards and one for event hazards. Each pair of segments is then associated to

three boolean values, one for each table, and the conjunction of those values tells if the segments are in conflict or not. To increase the number of processes candidate for parallel execution, the authors use techniques similar to branch predictions in hardware.

In the latest version, as proposed by Schmitt *et al.*, the segment graph and conflict analysis tables are built using a hybrid approach, that combines static and dynamic analysis [29]. A dynamic design analysis is first made to determine the components hierarchy and the location of SystemC processes in those components. This is possible thanks to the introspection API offered by the SystemC kernel. The results of the design analysis leads to a first instrumentation of the model source code. Then a static conflict analysis is done on the instrumented model. During this analysis, a *may-happen-in-parallel* table of segments can be built [47], helping to produce a suitable design for thread-safe parallel simulation. The added value of this last approach is the support of third-party libraries. Indeed, it is hardly possible to statically analyze a piece of code containing calls to functions from a third-party library. Within binary code, segment boundaries or used variables cannot be identified. The authors propose an annotation mechanism to indicate if a function contains segment boundaries, or can induce conflicts. By default, to deal with standard libraries functions, no segment boundaries nor conflicts are assumed to be present.

Reder *et al.* propose to use both static and dynamic analysis [30]. They target to achieve the parallel simulation of SystemC/RTL models on many-core architectures (as simulation host). The authors use conservative PDES with a lookahead time and null messages. The current solution is based on SystemC/RTL communications, using `sc_signal`. For the time being, they do not support IMCs, however supporting TLM models is planned as a future work by the authors.

4.4.3.2 Applicability in our Case

To develop a generic simulation method that enables the parallel run of multiple tasks, SystemC offers theoretically no other choice than performing conflict analysis (either statically or dynamically). Even within a delta cycle, coroutine semantics implies that the parallel run of transitions is safe only if there is no race condition between them. In practice, it is reasonable to make assumptions that specific conflict situations will not occur. For example, a reasonable assumption can be that processes from different SystemC modules do not directly share a variable (except through TLM transport methods). Another assumption taken in most work for SystemC/RTL is that processes do only share signals, and not directly variables. In order to develop a generic kernel (*i.e.* no such assumptions), the approaches presented in this section have chosen to use conflict analysis to achieve parallel simulation.

In an industrial context like at STMicroelectronics, virtual platforms contain parts only available in binary form, as a black box. The combination of static and dynamic analysis with an annotation mechanism can reduce this problem [29]. However, an annotation mechanism still requires that someone has the sufficient knowledge on the internal

behavior of the binary code to add annotations. An applicable approach must also be scalable for large code bases. We remind our case study presented on Section 3.3, which consists in 750 000 lines of C++ code. Moreover, previously presented analysis approaches do not support TLM transport calls [30] or use over-approximations that do not extract potential parallelism [7]. Thus, dependency analysis applied on TLM models remains an open problem.

4.4.4 Exploit Massively Parallel Computing Architectures

In the case study we presented, the problem is not to find enough workers on the simulation host, but to identify independent tasks in the model to run in parallel. This situation is common at high abstraction levels, as LT TLM. Indeed, at such a level, the number of transitions tends to be reduced, as the coding style asks to group computations, and use mechanisms as temporal decoupling, notably to limit the need for context switches. In models where many independent tasks have been identified, the problem is to find enough computing units in the simulation host. A possible solution is to exploit the computing abilities of Graphics Processing Units (GPUs) [27, 48, 32]. This can be done using frameworks like NVIDIA's Compute Unified Device Architecture (CUDA), or OpenCL [49].

SCGPSim is a simulation infrastructure that enables parallel simulation of synthesizable SystemC code at Register Transfer Level (RTL) level [27]. It uses a source-to-source translator to produce code that uses the CUDA framework, therefore exploiting the computation capabilities of NVIDIA's GPUs. SCGPSim cannot be used for non-synthesizable SystemC code, all the more so for models at the TLM abstraction level. Sinha *et al.* later proposed to reuse concepts from SCGPSim into a simulator compatible with TLM [48]. It exploits multiple CPUs along with GPUs. In this approach, the user must first identify "GPU-suitable" SystemC processes. The code of those processes is then translated into CUDA code (as with SCGPSim) and wrapped to enable communication with the SystemC kernel. It is then possible to use SystemC immediate, delta or timed events, as well as signal handling across the GPU/CPU boundary. Other SystemC processes are run on CPUs. They are compiled with an altered SystemC kernel to enable the use of multiple CPUs. This approach also uses the principle of a parallel evaluation phase.

SAGA is another parallel simulation infrastructure for SystemC programs, at RTL level [32]. As SCGPSim, it exploits GPUs by using CUDA. SAGA determines a process dependency graph, based on the analysis of reads and writes done on SystemC signals. This dependency graph is used to build a static scheduling of the processes. Then, the tool partitions the graph by identifying independent dataflow graphs. A concurrent scheduling scheme is established from this, which leads to the generation of CUDA code.

An original implementation of a parallel SystemC kernel have been proposed by Ventroux *et al.* [26]. Indeed, it is a hardware implementation of a SystemC simulator, called *RAVES*. It consists in a parallel implementation of a SystemC kernel, similar to previ-

ously seen approaches, with parallel execution of runnable threads in the beginning of each evaluation phase followed by a synchronization barrier. This SystemC kernel is run on RAVES, a special-purpose many-core platform (64 cores) offering a highly parallel execution architecture. On RAVES there is only a custom micro-kernel (not a complete Linux kernel) made to support and optimize the execution of SystemC processes. This platform provides hardware acceleration features for the evaluation of processes. Even though the hardware acceleration along with the parallel kernel in RAVES showed good results, the same problem still exists with a parallel evaluation phase: there are not enough processes in our type of models for this to be efficient.

4.4.5 Tasks with Duration

An experimentation framework in Java, called *jTLM*, has been developed to try the concept of tasks with duration in TLM [50]. This framework had also another goal, namely study preemptive scheduling for SystemC, that we further discuss in Section 4.5. Tasks with duration are an alternative to instantaneous computation followed by a time elapse. Figure 4.13 illustrates this change with a SystemC code. Tasks with duration have later been implemented with SystemC in the *sc_during* library [2].

```
void thread() { // SC_THREAD
    compute();
    wait(time);
}
```

(a) Common pattern with SystemC.

```
void thread() { // SC_THREAD
    during(time, compute);
}
```

(b) Tasks with duration.

Figure 4.13: Comparison between SystemC classic time elapse and sc_during.

Semantically, this means that the task can be executed in parallel with the rest of the simulation. This solution needs model refactoring to exploit host parallelism, but it has the advantage to let legacy SystemC code running as-is, sequentially. This solution is relevant for platforms where wall-clock time consuming parts can be clearly identified, and are present in sufficient number to justify the parallelization. This solution explicitly targets LT models; the notion of tasks with duration is not meaningful for clock sensitive processes. It is also possible to synchronize a during task with the SystemC kernel. For example, the duration of the task can be extended, or a SystemC primitive can be called. However, in the latter case, there is a major overhead because the sc_during thread must synchronize with the SystemC kernel thread. Consequently, too frequent synchronization induces a dramatic slowness in the simulation.

4.5 Discussion on Simulation Semantics

In the context of system on chip simulation, there exist different languages to write models. They are referred to as System-Level Description Languages (SLDLs) in the literature [51]. Each SLDL enables the definition of objects and comes with specific simulation semantics, that are mandatory to enable the implementation of a simulator, making models executable. For example, SystemC defines modules, ports, threads, methods, and its simulation semantics specifies that the processes are run with respect to coroutine semantics. We have seen many approaches developed specifically for SystemC, but there are also approaches for other languages than SystemC. The answer to the question "is it possible to adapt for SystemC an efficient parallelization approach applied to another SLDL?" has no simple answer in the general case, as it mainly depends on the simulation semantics of the language.

Dömer *et al.* proposed a comparison of multi-threading semantics of two SLDLs, namely SpecC and SystemC [51]. With SystemC, a simulator must use cooperative multitasking semantics. The consequence is that model developers do not have to avoid race conditions on variables shared by multiple SystemC processes, including Interface Method Calls (IMCs). We have already explained this in details in Section 2.2.1. On the contrary, SpecC simulators can use preemptive multitasking and/or physical parallelism. Thus, model developers must protect shared variables from race conditions. This difference in the semantics alone leads to the fact that a parallel SystemC simulator requires conflict (or dependency) analysis, if no specific assumptions are made, whereas it is not the case for SpecC. In the latter case, only shared variables in channels need a protection mechanism [52].

The language is not the problem in itself, if we put aside the syntactic aspects. The fact that a parallelization approach works for one language or another depends on its simulation semantics. This opens discussion on the comparison of the two types of scheduling, cooperative or preemptive, in the context of a simulation. In cooperative scheduling, yield points are explicit: the model developer must place them. Research work have studied how to place them in an efficient manner, depending on the modeling needs. In preemptive scheduling, yield points are implicit: they are not part of the model description. Thus, a correct model needs to be written by taking into consideration all the possible preemption points. In other words, it makes model writing harder by adding potential race conditions that were not possible in cooperative scheduling [50]. A possible solution to this difficulty is to offer a dedicated structure for communication between processes (which is the case in SpecC). However, this restricts the flexibility in the sense that it forces model developers to use this structure, where it may be more efficient to use another one. A practical example is SystemC signals: they are used for Register Transfer Level (RTL) models and could play this role of communication restriction. But in Transaction Level Modeling (TLM), SystemC signals are not (or barely) used because the higher abstraction level enables more efficient communication means.

The question of whether using preemptive or cooperative scheduling is hard to judge. But what is probably the most important point in favor of cooperative scheduling is that it ensures the reproducibility. This is a critical property for simulation. The problem with preemptive scheduling, is that the exact same scheduling order can hardly be reproduced (it requires a mechanism to record each yield point at assembly level). This has a major drawback, that most developers using parallel programming have experienced: a bug can occur only once in a while, with exactly the same input parameters. Of course, proper programming should avoid that, but in the context of model developing the effort is focused on the platform that is modeled, not on the model itself (as a piece of software). Thus, asking model developers to deal not only with modeling problems, but also with software engineering problems, makes modeling harder. This can eventually increase the overall development time, which is opposite to the intended goal.

4.6 Discussion on Simulation Replication and Time Partitioning

For exhaustiveness, this section briefly discusses theoretical parallel simulation approaches that either are not applicable to SystemC (we explain why) or do not address the same performance issue as ours.

4.6.1 Simulation Replication

Simulation replication consists in running concurrent simulations of the same model with different inputs. The different simulations do not communicate with each other. A limitation is the quantity of memory available on the host machine. Hybinette *et al.* [53] proposes a clone/prune system. Simulations are cloned at specific decision points, and pruned simulations when an answer is obtained. Figure 4.14 shows a possible example.

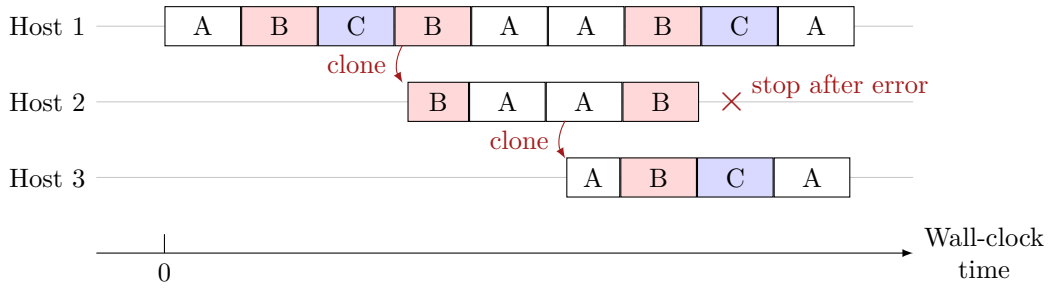


Figure 4.14: Possible execution of the example simulation with three host processors. Simulation are cloned when decision points are reached, and pruned when a result has been obtained.

Simulation replication is useful for non-regression tests, that ensures that new versions of the model do not break previously working features. In non-regression tests, a large quantity of independent tests needs to be run, thus it is possible to speed up such tests by running them in parallel. The problem we address is the slowness of simulation in the case of development or debugging tests. The purpose is to identify as fast as possible the root cause of an error, or to get an early validation of the current development. Simulation cloning/pruning is be useful for the exploration of scheduling orders, as done by Helmstetter [3], but does also not address the case of development and debugging tests.

4.6.2 Time Partitioning

Formally, time separation in parallel simulation consists in splitting the simulation with respect to simulated time [22]. Each worker gets a simulated time interval to compute

(the intervals are non-overlapping with each other).

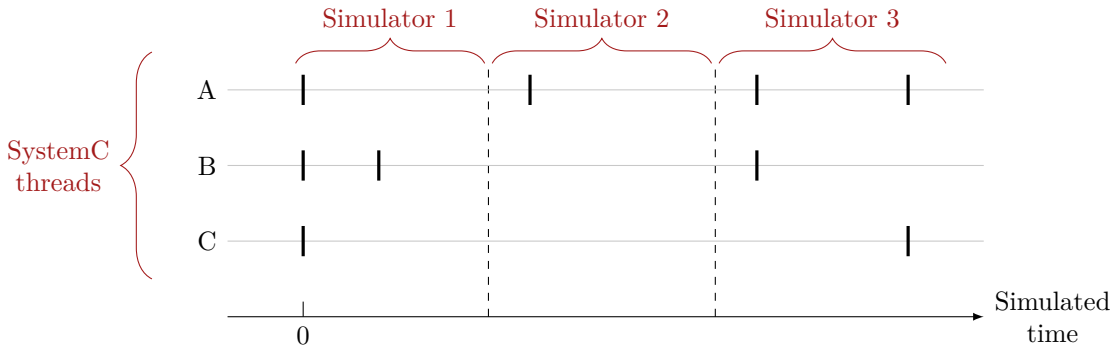


Figure 4.15: A possible simulated time-partitioning for our example.

A naive example of such approach applied to SystemC is given as illustration on Figure 4.15. The obvious requirement to apply this approach is to be able to compute the initial state of each simulator. Generally, this is possible only when each state of the model is a function of time, and not of the previous states. Another possible example is when traces for a simulation are available (*e.g.* from previous runs). In this case, the traces can be used to quickly reach the initial state of each simulator.

The time separation approach cannot realistically be applied to the parallel simulation of SystemC models. There are strong dependencies between the different transitions of a SystemC process. From one transition to the other, the context (in the sense of OS context: program counter, stack, registers) is restored exactly as it was just before the previous yield. Moreover, SystemC simulations are discrete event simulations (see Section 2.1.3). Thus, time is not an *input* value, but an *output* value that depends on how events occur in a simulation. It is therefore not possible to know *a priori* which times will be simulated. Consequently, a separation purely based on simulated time is not applicable to SystemC simulations.

4.7 Conclusion

The literature review presented in this chapter showed that many approaches have been proposed through the last decades to address the question of parallel simulation of SystemC models. The first approaches were made for SystemC/RTL models [24, 31, 25]. Those approaches propose to run multiple processes concurrently within simulation cycles. Low-level communication features from SystemC are exploited in this purpose. It is assumed that running multiple processes in parallel causes no race conditions. This assumption is reasonable at the RTL abstraction level, but less at the Transaction Level Modeling (TLM) abstraction level, where Interface Method Calls (IMCs) make processes dependent with each other. In our case, there is another issue with conservative approach: there is little potential for parallelism. Indeed, we have seen that the results of our case study showed low potential for parallelism within simulation cycles.

More recent approaches were made with SystemC/TLM models in mind [34, 35, 37, 41]. Some of them impose constraining modeling rules in exchange for parallel simulation, while others focus on the integration of legacy models. A common point between these approaches is that they are based on a manual partitioning of the platform. Each partition is simulated in parallel with the others, but still the different partitions are tightly coupled. For this type of approaches, the number of runnable processes per cycle remains a limiting factor. The acceleration is obtained mostly be using lookahead time between simulators, taking advantage of the fact that processors are simulated with cycle-accurate models, where relaxing timing synchronizations is efficient. This is particularly the case where processors are simulated with Instruction Set Simulators (ISSs). In our context, the simulation of processors (and thus the execution of the embedded software) is not a performance bottleneck. Moreover, the complexity is located on hardware Intellectual Property (IP) blocks.

— Chapter 5 —

Proposed Parallel Simulation Infrastructure

5.1 Introduction

At this point, one question came on the table. How can we address the performance problem of simulations such as our case study (see Chapter 3), considering that the analysis of existing parallel simulation approaches showed that there is a discrepancy between the type of models addressed, and the types of models we simulate (see Chapter 4). Beyond our situation, there is a deeper problem that consists in identifying potential for parallel simulation in SystemC/TLM models, taking into account the industrial context.

In an industrial context, complex models are not developed by one developer, or even one team. They are made of several different pieces coming from different teams, with different expertise. Let us take one example, not chosen at random, that is the case of the High Level Synthesis (HLS) design flow. The HLS design flow is used by one or several teams to design an Intellectual Property (IP) block. This team uses one HLS tool at disposal, carrying specific constraints. For example, the validation process requires that the component is split in sub-blocks, and the HLS tool requires a specific coding style to describe the behavior in an understandable manner (for the tool). When this HLS code has been written, and the team in charge of developing a Transaction Level Modeling (TLM) virtual platform meets short development time, there is an absolute need to reuse the work from HLS teams, *i.e.* their code. Rewriting the SystemC/TLM model of IP blocks is not possible: platforms are complex, there are many such components and others to integrate, and not enough time for all of that.

When code written for an HLS tool is used in a TLM model, both sets of constraints meet each other and may conflict: while in TLM modeling, computations are grouped, for HLS the developers may have to split them up. Moreover, the HLS design flow is used for IP blocks, thus they represent finely-tuned components that are crucial in the final product performance. Such components use pipelining of sub-blocks, which produce inefficient sequential simulations: it is also one result of our case study. Again, beyond our case, the issue is bigger: our assumption is that HLS will be more and more used to develop IP components, and with the tight embedded systems market, the code for HLS must be reused in virtual platforms.

In consequence, we decided to address the integration and efficient simulation of hardware acceleration blocks, whose behavior is written for an HLS design flow. This implies to enable both TLM and HLS users to develop their code only with their own set of constraints in mind, and at the same time to increase the performance of code written for HLS when simulated within a TLM model. Our proposition is to run in physical parallelism the models of hardware IP blocks. By our experience, the HLS design flow is mostly used to design hardware IP blocks that performs computations on large amounts of data, such as image processing blocks, in a pipeline of blocks communicating through First In, First Out (FIFO) accesses. Thus, finding an efficient FIFO communication mechanism is also part of the problem.

Our proposition is compatible with both the HLS requirements, and the TLM requirements. It consists in wrapping the code for HLS in SystemC modules, and then simulate them with a parallel infrastructure. The parallel infrastructure is presented in this chapter. The principle of *wrapping* HLS code is presented in Chapter 6, but an overview is given in this chapter to provide the reader with context information. In summary, the contributions presented in this chapter are:

- Identification and implementation of an efficient FIFO communication algorithm for intensive data exchanges (unidirectional), and adaptation of this algorithm to SystemC simulations. The FIFO structure that implements this algorithm is called Fast Ordered First In, First Out data exchaNge (FOFIFON).
- Development of DistemC, a non-intrusive multi-process infrastructure of SystemC simulators, which communicates through FOFIFON structures.

5.2 Wrapping HLS Code for TLM Simulation

5.2.1 Principle

This section introduces what we call the *wrapper* on HLS code. The purpose of this wrapper is to help the integration of High Level Synthesis (HLS) code into Transaction Level Modeling (TLM) models. Section 2.4 already explained that each HLS tool has its own input syntax, or design rules. In our case, we considered the HLS tool CatapultC, because it is currently used at STMicroelectronics. From here, we often refer to C++ code written to be used in CatapultC as *HLS code*.

Figure 5.1 shows a typical example of wrapping. The TLM/HLS interface consists in using TLM interfaces to communicate with the rest of the platform ("SystemC/TLM Model" on the figure). The HLS/HLS interface is purely internal to the component and does not expose information to the rest of the platform. The fast First In, First Out (FIFO) data exchanges take place in the HLS/HLS interfaces.

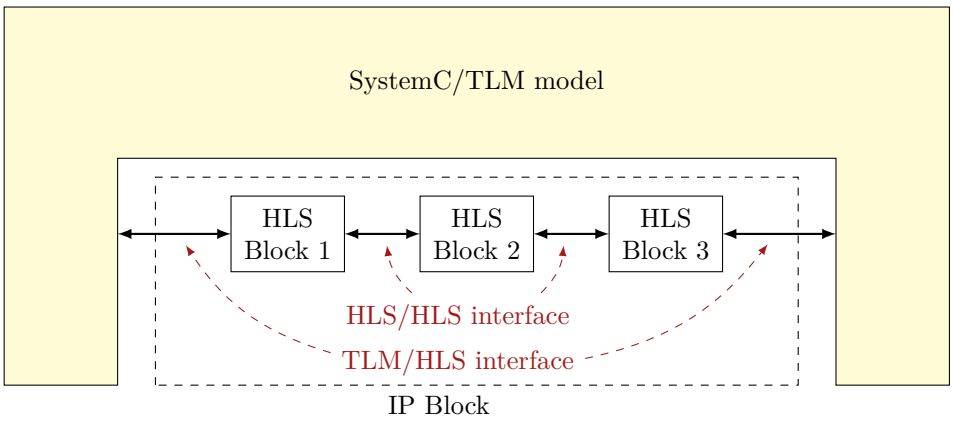


Figure 5.1: Typical example of HLS code with interfaces for integration in a TLM model.

The TLM/HLS is the interface that exists between an Intellectual Property (IP) block and the rest of the TLM model. An IP block is typically connected to a system bus and has some others input or output wires. The term "TLM/HLS" is a convenience term used because we interface code written for HLS with a TLM model. However, there is no HLS abstraction level and thus HLS is not comparable in itself to the TLM abstraction level.

5.2.2 Link with the Kahn Process Network Model

We put this model of computation (pipeline of blocks) in perspective with the Kahn Process Network (KPN) model [54]. A KPN is a model of computation where a group

of processes communicate with each other through FIFO channels. On this model, the following assumptions hold:

1. The FIFO channels are the only way of communication.
2. The FIFO channels are unbounded.
3. The FIFO channels transmit data in a finite amount of time.
4. Each process executes sequential code.
5. At any given time, a process is either computing or waiting on one FIFO channel.

Item 2 means that an operation on a FIFO channel is blocking if and only if it is a read operation and the buffer is empty. A write operation cannot be blocking since buffers are unbounded. In particular, a process cannot test for the emptiness or query the numbers of token of a buffer. Item 5 implies that a process cannot wait on several channels.

The determinism of executions is a very important property of SystemC simulations. The KPN model of execution ensures the determinism of executions, thus it is a good property in our case. Moreover, the ability to run computations in parallel is what we are looking for to speed up the simulation.

When this model of computation is applied, the theoretical unbounded size of buffers cannot be satisfied. Indeed, the amount of memory available in a computer is not infinite. This has the consequence that FIFO channels are bounded, and that the formerly non-blocking write operation becomes a blocking one, in case the buffer is full. This may lead to artificial deadlock situations, in the sense that they cannot occur in the KPN model. A very simple example of such deadlock is illustrated on Figure 5.2.

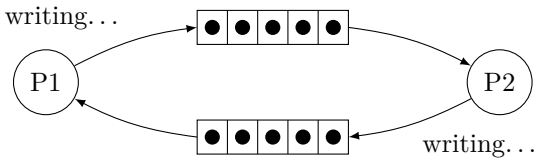


Figure 5.2: Simple example of an artificial deadlock, induced by bounded FIFO channels. Two processes P1 and P2 are blocking on a write operation, because both buffers are full.

Since this deadlock situation cannot happen in the KPN model, it means that a sufficiently high size for buffers prevents such deadlocks to occur. In general, the determination of the needed capacity for each buffer is an undecidable problem [55]. However, for specific algorithms, it is possible to find at least an overestimation of the size needed to avoid such artificial deadlocks.

5.3 DistemC, a Parallel Simulation Infrastructure

5.3.1 Multiple Simulators

In the example outlined on Figure 5.1, what is represented as the "rest of the SystemC model" includes buses, models of Central Processing Units (CPUs), other peripherals, *etc.* This is simulated by one single SystemC simulator as usual. We propose to execute in parallel the model of a hardware Intellectual Property (IP) block, by running in parallel the sub-blocks that constitute the hardware pipeline of the complete block.

In order to run multiple tasks in physical parallelism, two main choices are possible: using OS threads or OS processes. Threads are generally preferred, as they are lighter than processes (user memory is shared instead of copied). However, in our case, it was better to use multiple processes, because we want each worker to run a SystemC simulator. Running multiple instances of a SystemC simulator on different OS threads is not immediately possible: the reference SystemC simulator was not designed for that. For example, it contains a global variable for the simulation context. Thus, it is required to modify the SystemC kernel to run multiple instances on the same process. Having a SystemC simulator on each worker enables to develop the wrapper for each sub-block as a SystemC module, thus as a black box. When individual sub-blocks do not represent a sufficient computing effort (compared to the other sub-blocks), they can be grouped into the same simulator (thus scheduled sequentially) provided they follow each other in the pipeline.

The isolation provided between OS processes also have positive side-effects: it enables the possibility to run different versions of SystemC, and more generally different executables with different versions of external libraries (it is useful if two sub-blocks are developed using incompatible versions of libraries). As stated, in our case, this isolation is more of a side-effect and it was not the main reason to use multiple simulators on different OS processes.

5.3.2 Presentation of the Infrastructure

DistemC is a non-intrusive overlay on the SystemC kernel (*i.e.* the kernel remains unchanged), developed during this thesis. An example of a main SystemC function using DistemC is shown on Figure 5.3. DistemC enables to run in parallel a given number of SystemC simulators. By assumption, those simulators communicate exclusively through Fast Ordered First In, First Out data exchaNge (FOFIFON) structures, presented further in Section 5.4. During the elaboration of the model, each simulator instantiates the modules it is in charge of. The partitioning of modules can either be hard-coded, or specified in a separate file given as input parameter. It is possible to run a single executable (multiple Operating System (OS) processes are created in the `distemc::init` function),

or to run different executables, one for each part of the model.

```
int sc_main(int argc, char *argv[])
{
    distemc::init(argc, argv);
    Top top("top");          // elaboration of the model
    distemc::sync_start();  // start simulators synchronously
    distemc::finalize();
    return 0;
}
```

Figure 5.3: Example code of an `sc_main` using DistemC.

An example of a top module is shown on Figure 5.4. The `create_module` function instantiates the module only if it is attributed to the current simulator. In the example, the partitioning is not hard-coded (otherwise `create_module` takes as argument the simulator number in charge of the module), it is specified in a separate file, where each module name is associated with a simulator number. By default, modules are assumed to belong to the main simulator only. The `create_fofifon` function instantiates the shared buffers through which modules communicate with each other. The fact that a FOFIFON structure is used for "remote" communication (*i.e.* between modules on different simulators) or "local" communication (*i.e.* between modules on the same simulator) is deduced at runtime depending on the simulators of the two modules that uses the buffer. Remote communication buffers are instantiated on POSIX shared memory.

```
SC_MODULE(Top) {

    SC_CTOR(Top) {
        producer = distemc::create_module<Producer>("producer");
        sub_block_1 = distemc::create_module<Sub_Block_1>("sub_block_1");
        // ... (other modules)

        first_buffer = distemc::create_fofifon<uint32_t>(
                producer, sub_block_1, "prod_to_1", 200);
        // ... (other buffers)

        if (sub_block_1) { // if sub_block_1 is on current simulator
            sub_block_1->input_port(*first_buffer);
            sub_block_1->output_port(...);
        }
        // ... (other bindings)
    }

    Producer * producer;
    Sub_Block_1 * sub_block_1;
    // ... (other modules)

    distemc::fofifon<uint32_t> * first_buffer;
    // ... (other buffers)

};
```

Figure 5.4: Example of a top module. This code is an ellipsis of the code given in Appendix B.1, that corresponds to an example like Figure 5.1, with three sub-blocks modules.

5.4 Fast Communication using FOFIFON

The proposed infrastructure is intended to be used in the context of hardware components that processes large amounts of data, *e.g.* high definition video frames. Thus, the communication cost must be reduced to a minimum, as it is performance critical. When a shared resource is intensively used by different computation units, an lock-free implementation is required to achieve good performances [56]. The Fast Ordered First In, First Out data exchaNge (FOFIFON) structure presented in this section is based on lock-free programming.

5.4.1 Lock-Free Programming

5.4.1.1 Terminology

The terminology of lock-free programming mainly includes three levels of "lock-freeness": *obstruction-free*, *lock-free* and *wait-free*. The three terms are defined relatively to how the different thread behaves when they access or modify a shared data structure.

An *obstruction-free* program guarantees that each thread progresses even when all the other threads are suspended while not accessing the shared structure. Thus, a thread can access or modify the shared data structure in a bounded number of steps. This is the weaker property of a concurrent program.

A *lock-free* program guarantees the system-wide progression. Individual threads may starve, but without hindering at least one other thread's progression. All lock-free programs are obstruction-free programs. A typical example of non-lock free program is when two threads lock, at some point, the same mutex. If the first thread locks the mutex and is suspended by the operating system, then when the second thread reaches the lock of the mutex, it will starve until the execution of the first thread resumes and unlocks the mutex. As a consequence, mutexes cannot be used in a lock-free program.

A *wait-free* program guarantees the individual progression of each thread, even if all the other threads are suspended (possibly while accessing the shared structure). Thus, each thread completes any operation on the data structure in a bounded number of steps. All wait-free programs are lock-free programs. A counter-example is a program containing a First In, First Out (FIFO) structure with blocking operations. It is not wait-free because, for example, if a reader thread reads an empty buffer, it will be blocked until a writer thread pushes data to the structure.

Lock-free programming is enabled by using low-level instructions called memory fences. Memory fences are required because the order of memory operations can be changed,

compared to the description in the source code, by either the compiler or the hardware, as illustrated on Figure 5.5. This reordering is done to get better performances, but when a program uses concurrent tasks, reordering operations on memory can introduce race conditions on shared variables.

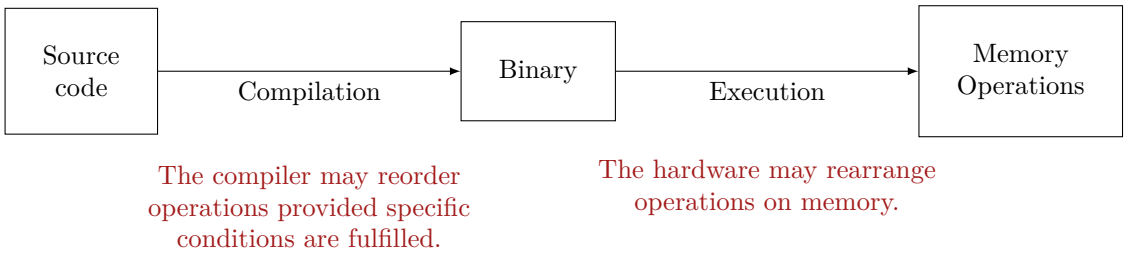


Figure 5.5: Places where memory operations can be reordered.

5.4.1.2 In C++11

C++11 atomic data types enables access and modification of memory in a thread-safe and portable manner. Atomic datatypes are defined using the `atomic` structure. This structure has a template argument which is the type of data stored by the atomic object. Using this structure guarantees that any access to the content of this object is thread-safe. In practice, using `atomic` variables adds constraints for the compiler, which reduces its reordering possibilities, and it may need to generate specific assembly code to ensure the expected order and atomicity is fulfilled.

Atomic data types in C++11 allows fine-grain control on memory ordering. Each operation on an atomic variable, *i.e.* a load or a store, accepts an extra argument: a memory order annotation. In order to define the C++11 memory orders, we define the *sequenced before* relation and the *acquire* and *release semantics*, that are used to define the *happens before* relation [57].

The *sequenced before* relation is defined between two operations on a single thread. If an operation A is sequenced before an operation B, then the execution of A shall precede the execution of B. For example, if the operation B uses the result of the operation A, then the execution of A shall precede the execution of B, thus A is sequenced before B.

The *acquire* and *release semantics* are defined using acquire and release operations. A *release operation* is a store operation on memory such that the preceding memory operations cannot be rescheduled after this release operation. An *acquire operation* is a load operation on memory such that the following memory operations cannot be rescheduled before this acquire operation.

The *happens before* relation is defined between two operations. Let A and B be two memory operations. If A and B are executed by a single thread, then A happens before B if A is sequenced before B. If A and B are executed by two different threads, then A

happens before B if A is sequenced before a release operation on a shared object M (let us assume the value stored is x), and B happens after an acquire operation on the shared object M that loads the value x.

In C++11, the acquire and release semantics can be used for atomic operations using annotations. The annotation `memory_order_relaxed` indicates that the operation happens only atomically at some point. Such order does not induce any memory synchronization operation. The `memory_order_acquire` and `memory_order_release` annotations specify acquire or release operations. The default order for atomic operation corresponds to `memory_order_seq_cst`. In this case, the operation happens in a sequentially consistent order with respect to each other atomic operation. This is more than a combination of acquire and release, as it defines a total order on the operations on the variable. Consequently, each thread sees such atomic operations in the very same order.

5.4.2 Presentation of FOFIFON

A FOFIFON is a lock-free, single-producer, single-consumer FIFO structure used as communication between modules on different simulators, using DistemC. This structure is not wait-free because the read and write operations are blocking, as in the Kahn Process Network (KPN) model discussed in Section 5.2.2. A C++ structure is allocated to store the data buffer. This structure is stored on shared memory; it must be directly accessible from both the reader and the writer. The algorithms for the read and write methods are implemented in a C++ class, which enables safe access to the structure. Figure 5.6 shows an overview of the different elements involved.

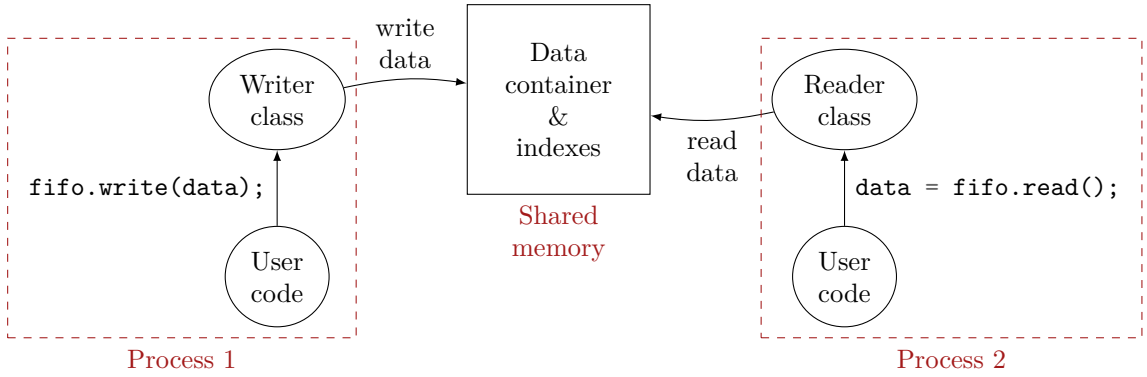


Figure 5.6: Overview of the architecture used for FIFO communication.

The data container has a fixed size and is allocated during the elaboration phase. Consequently, the FIFO buffer is bounded by a maximum capacity, and is used as a circular buffer. The following sections present the layout of the data container, and then the read and write algorithms.

5.4.2.1 Layout of the Data Container

Figure 5.7 introduces the graphical notation we use to represent the state of a FOFIFON data container. The central part represents the buffer. Indexes are represented pointing to the cell their value refer to. Cells colored with diagonal hatches or straight hatches represent the safe zone for the reader or writer. The notion of *full cell* or *empty cell* refers to the fact that the piece of data in the cell is meaningful or not. Initially, the buffer contains undetermined data, *i.e.* all the cells are empty.

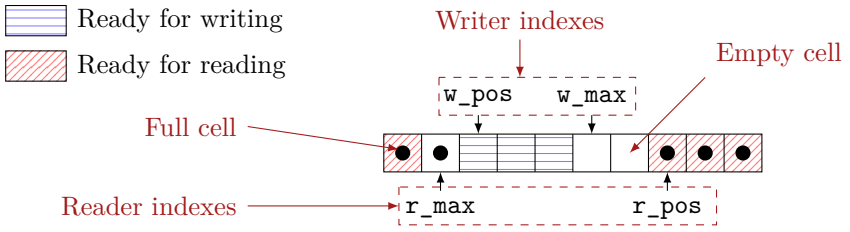


Figure 5.7: Graphical notation to represent the state of a FOFIFON data container.

Technically, the data container is a template structure with one parameter, which is the type of objects that must be stored in the buffer. This shared data container contains the following elements:

- `buffer`: contiguous array, used as a circular buffer.
- `w_pos`: index of the next location to write. By convention, the initial value of this index is zero. This index is an atomic variable.
- `r_pos`: index of the next location to read. By convention, the initial value of this index is also zero. It is also an atomic variable.

Along with `w_pos` and `r_pos`, which represent positions, there are also maximum indexes for each side. They are stored locally (they are not atomic variables) by the writer or reader. The two "max" indexes are defined as follows:

- `w_max`: index of the first cell where writing is forbidden (starting from `w_pos`). Initially, this is the last cell of the buffer (all the buffer is writable).
- `r_max`: index of the first cell where reading is forbidden (starting from `r_pos`). The initial value is zero (no cells are readable).

The value of the "max" indexes always points to existing cells of the buffer. For the initial case, this implies that there is an extra cell, compared to the visible size of the buffer. This extra cell acts like a sentinel cell, whose position varies during the execution. At any time, there is at least one cell of the buffer that cannot be written to and that

cannot be read from. The use of this sentinel cell avoids to handle a special case where the counters have made a complete turn of the buffer, thus the same mechanism can be used independently from the counter's positions. Figure 5.8 represents the initial state of a buffer of size 8.

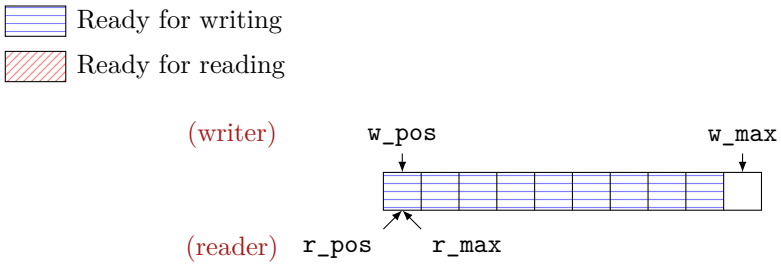


Figure 5.8: Initial state of a buffer of size 8 (internal size is 9). All cells are empty and ready for writing.

5.4.3 WeakRB Algorithm

The algorithm we propose for FIFO communication is based on WeakRB, developed by Lê *et al.* [58]. WeakRB is a single-producer, single-consumer FIFO queue with a portable implementation, using the C++11 memory model previously presented. This implementation has been proven correct by the authors, using a formalization of the C++11 memory model. We give the original algorithm of WeakRB in the form of C++11 code in Figure 5.9. This algorithm supports writing and reading batches of data at once, *i.e.* the memory synchronizations are only performed once for the whole batch of data.

5.4.4 Proposed Changes

The WeakRB's efficiency relies on three mechanisms: relaxed memory ordering (with C++11 acquire/release semantics), software caching (with local variables for foreign counters, namely `pfront` and `cback`), and batching (*i.e.* the "push" and "pop" operations work with arrays of tokens instead of tokens) [58]. This algorithm is non-blocking: "push" and "pop" return a boolean value that indicates if the operation was done or not.

The first change we made is to turn these non-blocking functions into blocking functions, to fulfill the KPN model. To make this modification, a simple solution is to call the former "push" and "pop" methods in a loop. The question is what to do when the operation failed, *i.e.* returned false. Doing "nothing" would result in a busy wait, and it would be counter-productive to use Operating System (OS) calls to suspend and wake-up the processes in a lock-free algorithm.

Consequently to this first change, we propose a better way to exploit the time spent waiting, so that a waiting writer or reader is not doing "nothing". Since operations are

```
atomic<size_t> front = 0;
size_t pfront = 0;
atomic<size_t> back = 0;
size_t cback = 0;

T data[SIZE];

bool push(const T * elems, size_t n) {
    size_t b = back.load(memory_order_relaxed);
    if (pfront + SIZE - b < n) {
        pfront = front.load(memory_order_acquire);
        if (pfront + SIZE - b < n) {
            return false;
        }
    }
    for (size_t i = 0; i < n; ++i) {
        data[(b + i) % SIZE] = elems[i];
    }
    back.store(b + n, memory_order_release);
    return true;
}

bool pop(T * elems, size_t n) {
    size_t f = front.load(memory_order_relaxed);
    if (cback - f < n) {
        cback = back.load(memory_order_acquire);
        if (cback - f < n) {
            return false;
        }
    }
    for (size_t i = 0; i < n; ++i) {
        elems[i] = data[(f + i) % SIZE];
    }
    front.store(f + n, memory_order_release);
    return true;
}
```

Figure 5.9: Code for WeakRB queue.

blocking, when the function returns, the whole batch of data have been read or written. Thus, it does not matter whether the whole batch is processed at once, or piece by piece. We propose to start writing or reading data as soon as possible, by using a concept of "safe zone". This safe zone is a portion of the buffer where a writer or a reader is allowed

to operate without risking race conditions. To illustrate this, we propose to see the case of the read mechanism in details (the write mechanism being symmetrical).

5.4.5 Read Mechanism

Figure 5.10 lists the code of the read function. It is split in three main parts, run in a loop (the loop that basically turns the non-blocking WeakRB algorithm into a blocking one). The three main parts are "get safe size", "actual read" and "update indexes". The first part, "get safe size", gets the immediately available size of the buffer for reading. Thus, if the size available for reading (given by the position of the writer) is smaller than the batch size (given as parameter), but greater than zero, then the process is not busy waiting. When the safe zone size is greater than zero, data are read from the buffer. This is the second step, the "actual read", which is straightforward. The third step, the "update indexes", updates the position of the reader, *i.e.* taking into account the data that have just been read. In terms of memory order specification, the acquire operation is performed during the "get safe size" step, and the release operation is performed during the "update indexes" step; thus the intermediate step, the "actual read", is safe.

5.4.6 Illustrative Examples

Figure 5.11 presents a minimal example with a buffer of size one, thus with an internal size of two (one extra cell). The hints in red indicates the elements that changed compared with the previous state. This example illustrates how the extra cell is used. States (1) and (5) are symmetrical. If the buffer's actual size was one (not two), the two position indexes (`r_pos` and `w_pos`) would point to an inexisting cell. This, it would be necessary to handle this case by reseting the values to the first cell of the buffer. Here, both indexes points to an existing cell, handling this case requires no special handling.

In (1), (3) and (5), the two sides are *up-to-date* with each other. It is the case when `w_pos` equals `r_max`, or when `r_pos` equals the cell that precedes `r_max`. In this state, there is exactly one cell that is neither in the safe zone of the reader or the writer (the sentinel) and the other cells are either in the safe zone of the reader or the writer. On the contrary, in (2) and (4), the two sides are not up-to-date with each other. An operation has been performed in the buffer (either write or read) but not yet observed by the other side. In this example, there cannot be a write that happens concurrently with a read, because the buffer size is only one. Thus the buffer can only be either empty or full.

Figure 5.12 shows an example with a buffer of size two (thus an internal size of three). In this example, a read and a write happen concurrently. In (1), both sides are up-to-date with each other. One cell contains data and one cell is available for writing. In (2) the reader and the writer observe a different state. In particular, after it has performed a write operation, the writer sees two full cells in the buffer, even though the reader has already emptied one of them. On the other side, after its read operation, the reader

```
void read(const T * elems, size_t n) {
    size_t nb_data_processed = 0;
    while (nb_data_processed < n) {

        // get safe size

        size_t pos;
        size_t safe_zone_size;
        do {
            pos = r_pos.load(memory_order_relaxed);
            safe_zone_size = min(r_max + SIZE - pos % SIZE,
                                 n - nb_data_processed);
            if (safe_zone_size == 0) {
                r_max = w_pos.load(memory_order_acquire);
            }
        } while (safe_zone_size == 0);

        // actual read

        for (size_t i = 0; i < safe_zone_size; ++i) {
            elems[nb_data_processed + i] = data[(pos + i) % SIZE];
        }

        // update indexes

        r_pos.store((pos + safe_zone_size) % SIZE,
                    memory_order_release);
        nb_data_processed += safe_zone_size;
    }
}
```

Figure 5.10: Proposed code for the blocking read function.

observes an empty buffer, even though the writer has already done its write operation. In (3) both sides updates their indexes. In the drawing they update their indexes at the same time, but in reality the updates most of the time happen at different times.

The fact that the writer, in (2), *sees* more data than what is actually in the buffer is not a problem. In fact, the concept of *seeing* data for the writer is not even defined. Indeed, our structure is a single-producer, single-consumer FIFO. The writer cannot read any data from the buffer; it only sees its safe zone, which consists in empty cells. The only consequence of overestimating the amount of data in the buffer is that the buffer could be seen as full when it is not full. This does not corrupt data in the buffer, or prevent the reader from reading correct data. When the writer updates its indexes, it will

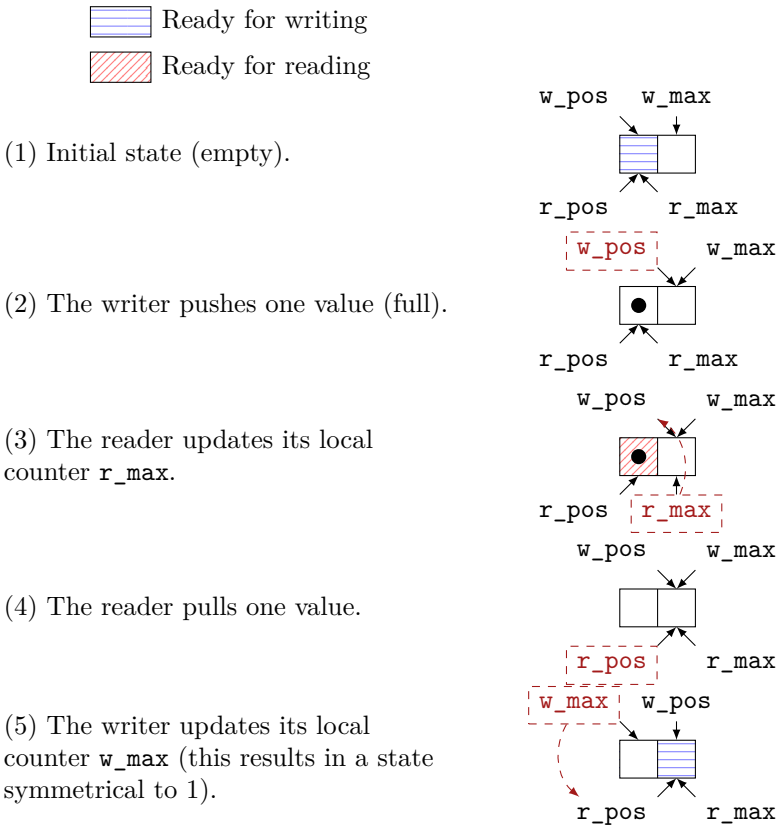


Figure 5.11: Illustration of the use of an extra cell for a buffer of size one. The changes from one state to the next are indicated with red hints.

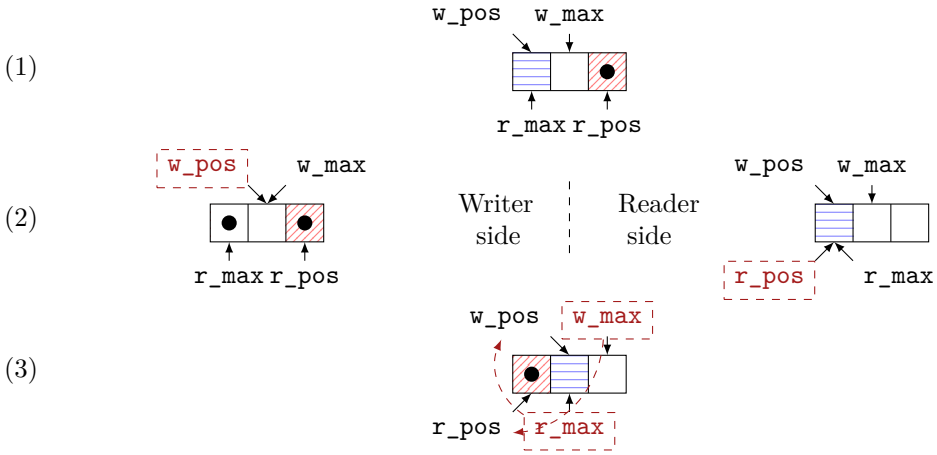


Figure 5.12: Example state with a buffer of size two, where a read and a write happen concurrently. The two states in the middle represent a point of the program where the reader and the writer observe a different version of the shared buffer.

see a more recent state of the buffer, as in (3), which may allow it to write more data. Symmetrically, the reader can underestimate the amount of data in the buffer, as in (2). An example with a greater buffer size is shown on Appendix B.2.

5.4.7 Validation

The correctness of the read and write algorithms have been tested using corner case tests and stress tests (with large amounts of data). However, such tests are not enough to guarantee the thread-safety of the algorithm. No matter how many times they are run, there is no guarantee that a scheduling that hides the race condition does not occur at each run. Moreover, our tests have been performed on Intel processors which are strongly-ordered. Thus, for short, they do not enable many reorderings of memory operations. In comparison, ARM processors are weakly-ordered. This implies that the same program, run on an ARM architecture, may exhibit race conditions that could not occur on Intel architectures.

For this reason, the thread-safety of the read and write algorithms have been checked with a tool called Relacy Race Detector[1]. In academia, this tool has been discussed [59, 60] and used to test, *e.g.* Dekker's mutual exclusion algorithms [61]. It represents each user thread as a coroutine, and adds yield points around operations on shared variables. Different interleaving orders are tested depending on the chosen scheduling strategy. For example, a comprehensive scheduling explores all the possible interleaving orders. Because of combinatorial explosion, it is often not possible to use this strategy. An alternative is to run a fixed number of random schedules. The user code must be instrumented; in particular each shared variable must be wrapped in a template class. We have run a large number of tests with a random scheduling strategy, which have shown no race conditions. We also checked that the memory orderings used are *at least* required for the algorithm to work (*e.g.* replacing release operations by relaxed operations leads to race conditions).

[1]The tool has been developed by Dmitry Vyukov. The presentation and download are available at: `http://www.1024cores.net/home/relacy-race-detector`.

5.5 Preventing Deadlock Situations

5.5.1 Dealing with Multiple SystemC Kernel Instances

Figure 5.13 shows a potentially problematic situation if not well handled. This situation is specific to the fact that we use this algorithm in a SystemC context, where SystemC processes from the same simulator are run under coroutine semantics. The two SystemC simulators are run on different Operating System (OS) processes. The first simulator contains two SystemC threads: one writes data to the first buffer, and one reads data from the second buffer. A trivial deadlock situation can happen if the `reader` thread is run before the `writer` thread. In this situation, the `reader` thread is blocked in a busy wait on the second buffer, while the `compute` thread is blocked in a busy wait on the first buffer. This deadlock situation does not happen when the `writer` thread is run before the `reader` thread busy-waits on its buffer. This situation is a specific case of a more general problem that we expose in the following paragraphs, with our solution.

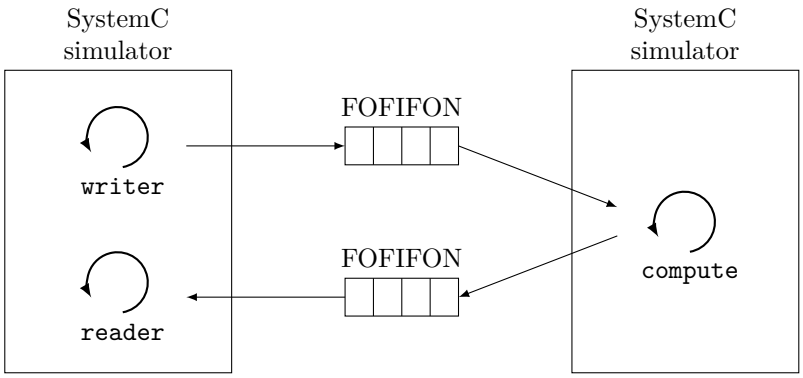


Figure 5.13: Simple example of a deadlock.

A deadlock situation could occur depending on the minimal number of tokens the writer or the reader require [62]. Let W be the number of tokens that the `writer` thread writes in the first buffer between each SystemC yield. Let R be the number of tokens required by the `compute` thread before producing any output to the second buffer. We assume that the `writer` thread is executed first. Then, after W write operations on the first buffer, the `reader` thread is blocked in a busy wait on the second buffer. Meanwhile, the `compute` thread reads W tokens from the first buffer. Two cases are possible:

- If $W < R$, the `compute` thread does not produce any output, and remains in busy waiting tokens from the buffer. This is a deadlock situation.
- If $W \geqslant R$, the `compute` thread produces output and the `reader` thread is unblocked and may yield to the `writer` thread. Then, the same situation happens after the next yield.

The deadlock situations presented in this section are solved by a function we named `systemc_wait_keep_alive`, defined in Algorithm 2. This function is called in the first part of the read or write method, in case the operation cannot be done immediately (*e.g.* after line 13 of Figure 5.10). It enables to yield to other SystemC processes of the current simulator when blocking operations could not be done immediately, without ending the simulation "by error" (*i.e.* because the local simulator is in process starvation). This problem of not ending simulations because of local process starvation is one of the first problems to solve when multiple simulators are used for parallel simulation, or even when multi-threading is used in a simulation.

The proposals for the next version of the SystemC kernel (2.3.2) includes a related feature. This feature applies in the case of simulations that use `async_request_update` (to asynchronously, *i.e.* from another thread of execution, require the execution of an "update" method in the next delta cycle). The problem is that a simulation that waits *only* on an event that is asynchronously triggered will end (because of starvation). The solution consists in adding a semaphore in the SystemC kernel, to which primitive channels can dynamically attach or detach during simulation. The semaphore is checked when no further events are available (*i.e.* the case in which the current kernel would end). Our solution avoided the use of asynchronous update requests and remote event notification: a simulator in the "starvation" situation (*i.e.* buffer full or empty) busy waits until the buffer state changed.

Algorithm 2 Definition of `systemc_wait_keep_alive`.

```
1: function SYSTEMC_WAIT_KEEP_ALIVE( )
2:     if SC_PENDING_ACTIVITY( ) then
3:         wait_time ← SC_TIME_TO_PENDING_ACTIVITY()
4:         if SC_TIME_STAMP() + wait_time > current_max_time then
5:             wait_time ← current_max_time − SC_TIME_STAMP()
6:         end if
7:         WAIT(wait_time)
8:     end if
9: end function
```

The function `sc_pending_activity` is part of the SystemC library and it returns false if and only if the set of runnable processes, the set of update requests and the set of delta and timed notifications or time-outs are all empty. The function `sc_time_stamp` returns the current SystemC time stamp (simulated time). The function `sc_time_to_pending_activity` returns the time to wait if one wants to reach the next registered notification or process execution (which is only meaningful if there is pending activity).

This introduces the `current_max_time` variable. The value of this variable is set each time the `read` or `write` functions are called. The value is equal to the current SystemC timestamp plus a constant offset. The value of the offset can be set in the model. The use of an absolute time value for `current_max_time` avoids that the simulator moves on too far with its simulated time in case another process constantly advances time: each operation on the buffer is bound by a maximum simulated time duration.

We illustrate the use of this value to solve the deadlock problem presented in the example of Figure 5.13. Indeed, let us consider the case where the `reader` thread is blocked in reading the second buffer (empty). The `compute` thread is waiting for input from the first buffer (also empty). If the `writer` thread already advanced its simulated time (after it has written data) then it will not get the hand back if the `reader` thread does not also advance its simulated time. If the time offset for this buffer is set to the simulated time the `writer` waits, then this deadlock situation does not occur.

5.5.2 SystemC Time Support

The global synchronization of the subsystem (HLS and wrapper code) is data-driven; in the Kahn Process Network (KPN) model of computation, the state of processes only changes when tokens are consumed or produced. Thus, the behavior and output of the subsystem is not sensitive to timing. Moreover, the loosely-timed coding style is not used for performance evaluation, and timing should not be used as a synchronization mean in such models. Moreover, quantitative timing is not relevant within the subsystem, because microarchitecture details are not modeled.

Even considering the previous points, one cannot simply put aside the SystemC time, as the subsystem is used in a SystemC simulation environment. To be consistent with SystemC simulation semantics, the FOFIFON buffers embbed a SystemC timestamp with the data. In other words, when a write operation is done on a FOFIFON buffer, the data is recorded with the current SystemC timestamp. Then, this timestamp can be used, for example, to keep the SystemC time evolution in remote DistemC partitions consistent with the SystemC time of the main simulator. This also adds the possibility to include components on remote DistemC partitions that need to know at least approximately the current SystemC timestamp.

5.6 Conclusion

This chapter presented DistemC, a parallel simulation infrastructure where simulators communicate only through First In, First Out (FIFO) structures. We developed a specific FIFO structure for this purpose, called Fast Ordered First In, First Out data exchaNge (FOFIFON). The proposed algorithm offers lock-free blocking accesses to the shared structure, enabling high performance in data exchanges. We introduced, in the beginning of this chapter, a location where this infrastructure is interesting to apply, namely for the parallel run of hardware acceleration blocks, designed as a pipeline of sub-blocks in High Level Synthesis (HLS).

The current proposal does not support asking the number of data available (for reading or writing) in a buffer. It is possible to add this feature, but not without adding costly communication. However, it is possible to get an overestimation of the number of available data by using the local values of counters (from the previous access). We discuss in the next chapter the consequences that such modification implies in the context of HLS code.

The purpose of this development is to exploit potential parallelism identified in hardware acceleration blocks, written for a HLS design flow. The remaining question is how to apply this infrastructure to integrate and efficiently run in a SystemC/TLM simulation such hardware Intellectual Property (IP) blocks. This question is addressed by the following chapter.

— Chapter 6 —

Parallel Simulation of a Hardware Component: Example of a JPEG Decoder Platform

6.1 Introduction

This chapter presents an application of the DistemC infrastructure for the parallel simulation of a hardware component of a SystemC model. To be clear, our proposition is not a parallel SystemC kernel for generic purpose, but an infrastructure that can be used under our assumptions.

Figure 6.1 summarizes how our approach integrates itself in the design flow. It addresses the integration of hardware Intellectual Property (IP) blocks, developed using an HLS design flow, with the tool CatapultC. Let us remind that the development of complex hardware IP blocks is split into sub-blocks. Splitting the design in sub-blocks eases the development, because within a sub-block, the function is simpler than the overall functionality, enables parallel development, but more importantly is mandatory because of the validation requirements (see Section 2.4.5). Using a pipeline architecture follows on from that: since the overall functionality is split in sub-blocks, it is more efficient to design the block as a pipeline of smaller functions, with a data stream flowing through the pipeline.

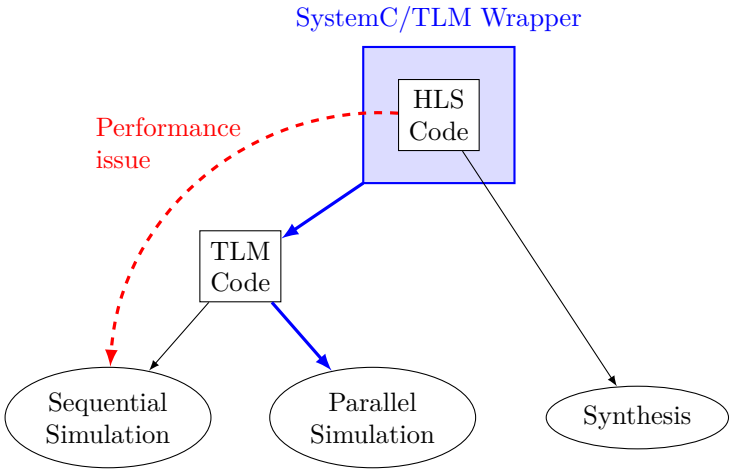


Figure 6.1: Integration of our approach in the design flow. We propose to wrap the HLS code into SystemC/TLM code for parallel simulation.

In this chapter, we use as example the model of a platform that does hardware-accelerated JPEG decoding. We remind that the JPEG encoding reduces the size of an image by removing some information and compressing the result using Huffman encoding. The encoded stream is in the frequency domain, pixels are ordered differently than in the visible image, the different color components of the Luma and Blue/Red-differences Chroma ($Y'C_BC_R$) color space are separated and the image is divided into macroblocks of 8×8 pixels. The decoding process transforms an encoded stream into a displayable image. Figure 6.2 shows a block view of the algorithm, composed of four steps: the Inverse Quantization and Inverse Zig-Zag (IQZZ), the Inverse Discrete Cosine Transform (IDCT),

the upsampling and the color model change, in our case where the output is expected in the Red, Green, Blue (RGB) format.

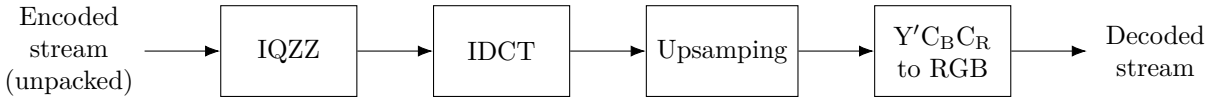


Figure 6.2: Block view of the JPEG decoding algorithm.

Section 6.2 presents the example platform that we use for our proof-of-concept. Before presenting the application of the DistemC infrastructure to this example, we need to further explain the specificities of developing code for the CatapultC tool. This is done in Section 6.3. The application itself is described in Section 6.4. Finally, a performance evaluation of the parallel simulation of our example platform is presented in Section 6.5. Appendix A gives more details on the steps of the JPEG decoding algorithm, and presents some implementation steps for HLS. We remind that the goal of this chapter is not to present an efficient implementation of JPEG for HLS; this is used as a representative example and for the performance evaluation.

In summary, the contribution presented in this chapter is the application of our proposed infrastructure to integrate code describing hardware acceleration blocks for the High Level Synthesis (HLS) tool CatapultC in a SystemC/TLM model. This enables the parallel execution of the hardware block. We did a performance evaluation of the speed-up enabled with the proposed infrastructure on a proof-of-concept model that consists in a JPEG decoder platform.

6.2 Description of the Example Platform

This section presents different block views of the JPEG decoding platform. The first one presents a block view of the different Transaction Level Modeling (TLM) components of the platform. Then, we zoom into the decoder block and present its internal structure.

6.2.1 Platform

Figure 6.3 shows the TLM block view of our example platform. The "Decoder" block is the JPEG decoder. The encoded picture is stored in the memory. The CPU executes a piece of software that reads the encoded stream from the memory, performs the Huffman decoding and then sends data to the decoder block. The Huffman decoding is done in software because this step requires more time to be adapted for an HLS implementation, which was not done within the time of this thesis.

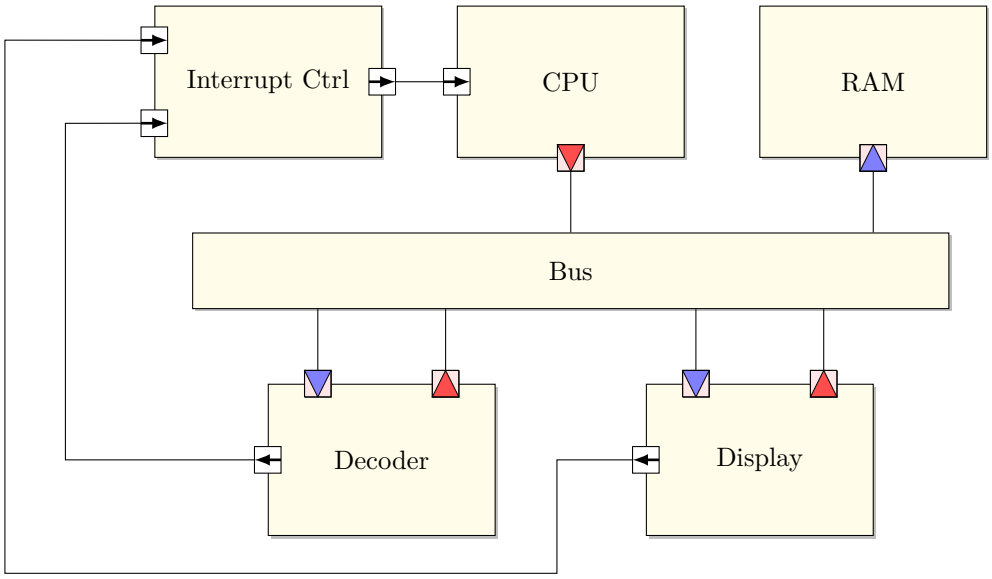


Figure 6.3: TLM view of the JPEG decoding platform.

6.2.2 Decoder Block

The decoder block has memory-mapped registers, programmed by the embedded software executed by the Central Processing Unit (CPU). The decoder block includes a custom Direct Memory Access (DMA) to fetch the encoded data from memory, and feed the pipeline constituted by the sub-blocks of the JPEG decoding algorithm. When the decoding of a frame is over, the decoder raises an interrupt. The behavior of the sub-blocks in the pipeline consists in code written for the High Level Synthesis (HLS) tool CatapultC.

6.3 Block Design with CatapultC

The explanations in this part are compiled from both our experience at STMicroelectronics, and from the High Level Synthesis (HLS) Blue Book written by Michael Fingeroff [63], that explains HLS with the CatapultC tool.

6.3.1 Hierarchical Designs

With CatapultC, the C++ functions are translated to Register Transfer Level (RTL) designs that are functionally equivalent. Each C++ function does not necessarily result in a hardware block, in the sense described on Figure 6.4. It is indeed possible to inline functions, that will be integrated into the "behavior" part of the hardware block. The inputs and outputs of the resulting design are determined by the parameters of the C++ function, and the direction (input, output or both) by how they are used in the code. During the synthesis, CatapultC adds the different missing signals from C++, like the clock signal, enables or resets.

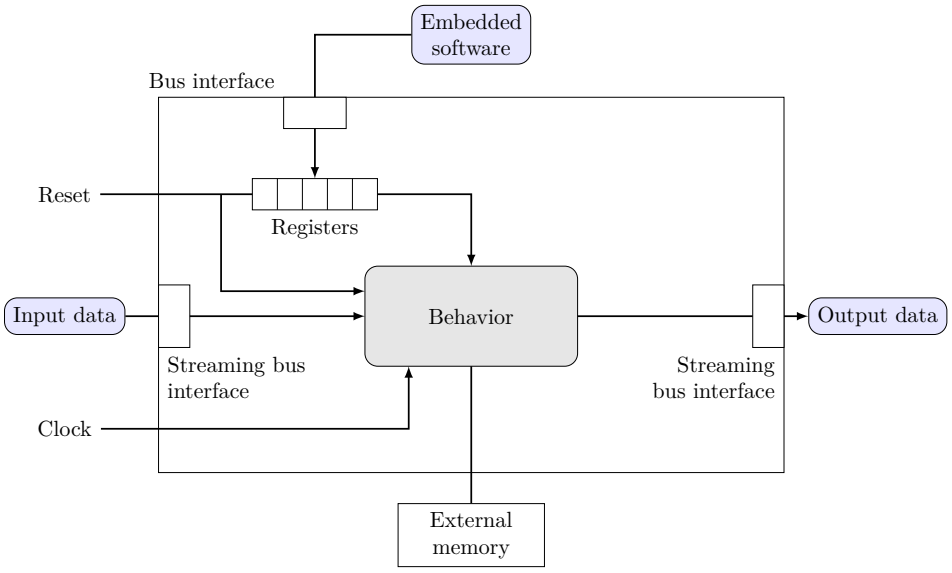


Figure 6.4: Reminder of a typical example of a hardware block.

We previously explained that in real-life designs, hardware designers build systems by assembling sub-blocks. In the complete design, in addition to the code describing each sub-block, there is also a top-level function that binds all the sub-blocks together, *i.e.* that relays the inputs and outputs of sub-blocks in the right order.

For the JPEG decoder example, Figure 6.5 shows a possible top-level function. In this example, we simplified the input and outputs, but the idea is that the input and outputs

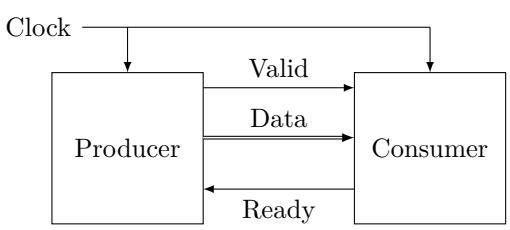
transits from one block to the other until the whole operation is done. In our approach to use HLS code in TLM simulations, we do not use such a top level function. There is no parallelism possible if the top level function is kept as in this example. Thus, our infrastructure replaces this top level function, by directly binding the different sub-block modules in the SystemC model.

```
void JPEG_top_level(A input, B output) {
    IQZZ(input, tmp_1);
    IDCT(tmp_1, tmp_2);
    Upsampling(tmp_2, tmp_3);
    ConvToRGB(tmp_3, output);
}
```

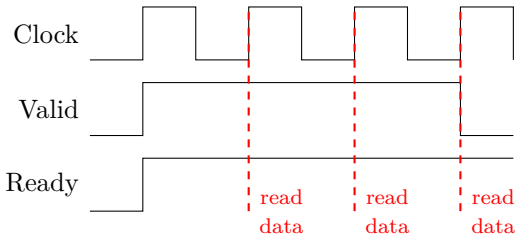
Figure 6.5: Example of a top-level function for the JPEG decoder.

6.3.2 Streaming Behavior

There is another aspect we must consider in hierarchical designs, it is how data goes from one sub-block to the other. Indeed, in the case where each sub-function becomes a sub-block in the hardware, there are multiple ways to exchange data between them. For example, one sub-block can write its results into a memory, then send a signal to the next sub-block, which will then write the memory and so on. This results in area and time inefficient designs, especially for blocks that handle large amounts of data, as it is the case for image processing blocks. Another way for data to transit is to use a hardware First In, First Out (FIFO) interface, also known as "valid, data, ready" interface (see Figure 6.6). In this case, data is transmitted at each clock cycle where the "valid" and "ready" signals are high. This kind of handshake protocol is used in common bus protocols, for example ARM's AXI4-Stream protocol, designed to enable fast communication in data-intensive applications.



(a) Hardware FIFO interface.



(b) Example corresponding chronogram.

Figure 6.6: Description of the hardware FIFO interface.

The following question is "how to tell CatapultC to use this interface between two blocks?". There are two situations where a streaming interface is used. The first situation

is when CatapultC can *prove* that the behavior described in the C++ code is a streaming behavior. In this case, the tool implicitly uses the streaming interface between the blocks. Otherwise, it uses memories and copies of data between blocks. However, in the latter case, it is possible that a streaming behavior was *possible*, but not used because the tool could not *prove* that it is the case (the proof is not straightforward). Figure 6.7 presents two example cases from the HLS blue book [63] that illustrates the two situations.

```
void block0(int in[4], int out[4]) {
   for (int i = 0; i < 4; ++i) {
      out[i] = in[i];
   }
}

void block1(int in[4], int out[4]) {
   for (int i = 0; i < 4; ++i) {
      out[i] = in[i];
   }
}

void top(int in[4], int out[4]) {
   int tmp[4];
   block0(in, tmp);
   block1(tmp, out);
}
```

(a) Streaming behavior can be proven by CatapultC, because accesses to the arrays are in order.

```
void block0(int in[4], int out[2]) {
   static int idx = 0;
   for (int i = 0; i < 4; ++i) {
      if (i & 1 == 0) {
         out[idx] = in[i] + in[i + 1];
         ++idx;
         if (idx == 2) {
            idx = 0;
         }
      }
   }
}

void block1(int in[2], int out[2]) {
   for (int i = 0; i < 2; ++i) {
      out[i] = in[i];
   }
}

void top(int in[4], int out[2]) {
   int tmp[2];
   block0(in, tmp);
   block1(tmp, out);
}
```

(b) Streaming behavior cannot be proven by CatapultC, because of the conditional operations on the index.

Figure 6.7: Two examples where the streaming behavior can or cannot be proved by CatapultC.

To force CatapultC to use streaming interfaces between blocks, the designer must write the C++ code with a specific C++ class called `ac_channel`. It offers functions for FIFO accesses, and the designer directly code the behavior using these functions. Figure 6.8 presents an example code that explicit the streaming behavior of the design described in Figure 6.7b.

On this figure, we see the usage of the two main methods of the `ac_channel` class, which are `read` and `write`. Amongst all the methods of this class, some of them are synthesizable, *i.e.* they are meant to be translated in a hardware design, and some are not synthesizable, *i.e.* they are used for simulation only, as in test benches.

```
void block0(int in[4], ac_channel<int> & out) {
   static int idx = 0;
   for (int i = 0; i < 4; ++i) {
      if (i & 1 == 0) {
         out.write(in[i] + in[i + 1]);
      }
   }
}

void block1(ac_channel<int> & in, int out[2]) {
   for (int i = 0; i < 2; ++i) {
      out[i] = in.read();
   }
}

void top(int in[4], int out[2]) {
   static ac_channel<int> tmp;
   block0(in, tmp);
   block1(tmp, out);
}
```

Figure 6.8: Explicit streaming behavior.

Non-synthesizable functions include the test for emptiness, comparison between channels or out-of-order peeks of values in the channel. Using those functions in a code for synthesis will simply not produce any output design. Amongst the synthesizable functions, there are the blocking read and write functions. There is also a non-blocking read function, that returns a Boolean value indicating if a value has been read from the channel or not. It is also possible to test if at least N tokens are available from a channel with the function `available`. This latter function is meant to be used for the C++ execution of code, where CatapultC forbids to read empty channels. When synthesized, it is replaced with a handshake. In other terms, it can be seen, for synthesis, as always returning True.

In summary, we consider HLS blocks that describe a streaming behavior. In the code, this behavior can be obtained using arrays as interfaces (see Figure 6.7a), or using `ac_channel` to explicitly describe it (see Figure 6.8).

6.4 Application

Figure 6.9 shows the infrastructure obtained as a result by the application of the proposed infrastructure. This section details the different steps to follow in order to get this infrastructure, starting from the set of individual sub-blocks code written for High Level Synthesis (HLS). The following explanations are split in two major steps: writing the top-level module, and writing individual wrappers for each sub-block.

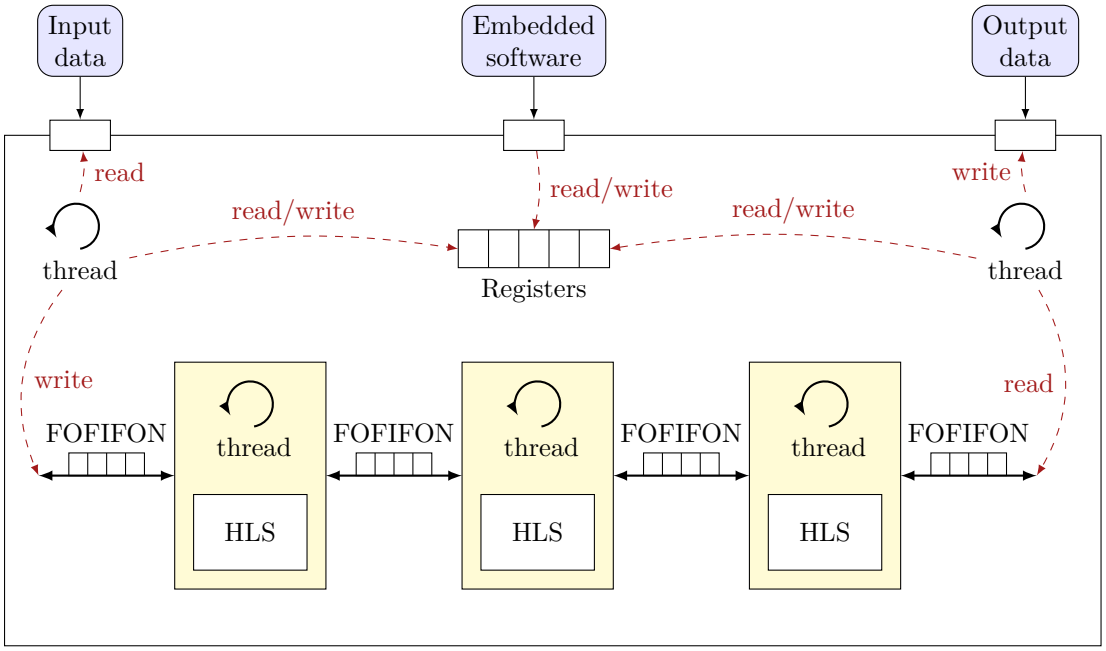


Figure 6.9: Our method consists in writing code at two places: for the top-level block (with registers, and proper interfaces) and for each sub-block (with our Fast Ordered First In, First Out data exchaNge (FOFIFON) structures).

6.4.1 Top-Level Module

This section explains how to write the top block of the hardware Intellectual Property (IP) block. This top block has two roles:

1. Interface the block with the rest of the platform, *i.e.* with code to handle TLM transactions and the programmable registers.

2. Instantiate and bind the different sub-blocks, and the different FOFIFON buffers.

The different interfaces of the module (Role 1) use common communication protocols (a system bus for the registers, and streaming bus for data). Each company that develops SystemC/TLM models has its own internal protocol models, as it is the case at STMicroelectronics. It is also the case for programmable registers, that are common features of hardware blocks, thus embedded in a development feature at STMicroelectronics. Moreover, registers for hardware blocks are often described in the IP-XACT format (see Section 2.4.4), and then translated with an automatic tool to SystemC/TLM registers. The specific behavior of control registers must be added manually as it is application dependent. The SystemC processes of this top-level module are scheduled by the same SystemC simulator than the rest of the platform, thus legacy interfacing code (that may vary from one company to the other) mentioned in the previous paragraph remains compatible.

We recommend to instantiate the top-level module and the different sub-blocks in different main files. This considerably reduce the refactoring effort in the case of large pre-existing main file, because the user does not have to explicitly put each component to the first simulator. On the main file, the user instantiates the top-level module as an "empty shell": it contains all the proper interfaces to communicate with the rest of the platform, the registers, but the sub-blocks are not instantiated in the main simulator. On Figure 6.9, this corresponds to all the surrounding module without the inner sub-blocks, and the buffers that connects them (except the ones at the two ends of the pipeline). This top-level module instantiates the required number of FOFIFON buffers to communicate with the first and last sub-block of the pipeline. We remind that the names of the buffer will be used to enable the different executables to actually use the same shared memory area. This main file produces a first executable, that will be executed sequentially by one simulator. This first version of the top-level module, that we called the "empty shell", fulfills Role 1.

Then, in a separate SystemC main file, the user instantiates another version of the top-level module, that contains the sub-block modules, and the FOFIFON buffers that connects them together. This version fulfills Role 2. It is important to use the same names, for FOFIFON buffers at the two ends of the pipeline, that were used in the "empty shell" of the main file. In this file, the user can attribute a simulator number to each sub-block (the processes for this executable will be created before simulation start), or use the specific constructors (see the example in Appendix B.1) to use a separate map file. This enables to change the mapping without recompiling the source file. Different strategies to assign sub-blocks to simulators are compared and discussed further when we present performance results. Intuitively, the best strategy consists in profiling sequentially the different sub-blocks (*e.g.* using SycView with a sequential execution of the model) and balance the work between the different workers depending on the parallel resources available on the host computer.

The instantiation of the different FOFIFON buffers needs further explanations. To construct a buffer, one need to provide a name and a capacity (we remind that we use bounded circular buffers). The choice of a capacity is application dependent, but the principle is to choose a sufficient capacity to avoid artificial deadlocks, high enough to

increase the throughput, and reasonable compared to the memory available on the host simulator [64]. In the example of our JPEG decoder block, let us take the case of the Inverse Discrete Cosine Transform (IDCT) block (see Appendix A). This block outputs 64 values each time 64 input values are read. Thus, the capacity of buffers around this block must be at least 64. Then, from 64 to a certain value, the overall throughput globally increases, because statistically less time is spent waiting because the input buffer is empty, or the output buffer is full. The user must specify a reasonable buffer size for each one.

6.4.2 Sub-Block Modules

The application of our infrastructure also requires that the code for each sub-block, written for the HLS tool CatapultC, is embedded in a SystemC module (represented as the inner rectangles on Figure 6.9). The SystemC module has one `SC_THREAD` that runs the HLS function of the sub-block in a loop. However, before or after calling the HLS function, it is necessary to add code to manage inputs and outputs. In the following explanations, we refer to this code as *extra sub-module code*. In order to enable the parallel execution of the different sub-blocks, they must communicate exclusively through FOFIFON buffers.

A situation that is likely to happen, is when sub-block modules need access to the values of the main block registers. In the JPEG example, the sub-blocks needs to have the value of the horizontal and vertical sampling factors. Those values are extracted by the embedded software (we remind that in our example, the header readout and the huffmann decode is done in software), and the software programs registers of the decoder block with those values. But the registers are part of the "empty shell" version of the top-level module, that is not on the same simulator as the sub-block modules. In order to enable the sub-blocks to access these values, the only choice we give is to send them through FOFIFON buffers. In order to do this, different strategies are possible. We propose an option to address this situation. The first step is to create local registers in SystemC for each sub-block module. The second step consists in writing an extra token to the data buffer each time the register values *may* change (this is application-dependent) and that indicates if the following data are special (*e.g.* register values) or are part of the data stream (in our example the compressed JPEG data). The third step is to add extra sub-module code to regularly read this extra token and consider the following data as register values (the number of token to read is application-dependent but it is known).

Then, another piece of extra sub-module code must be added depending on how the HLS block is written (in terms of interface). Each channel (CatapultC's `ac_channel`) is substituted with a FOFIFON buffer. Thus, in practice, each read or write on the buffer is done on shared memory. In this case, there is no need to do any particular operation before or after calling the HLS function, since the algorithm will perform the read and write operations directly on the shared buffer. In the current version of FOFIFON, the supported methods are `read` and `write`, and the `available` method that always returns True (as explained in Section 6.3.2).

When the interface of the HLS block consists in arrays, the SystemC module simulates the streaming using local arrays. Before calling the HLS function, the user must add code that reads the input FOFIFON until the required number of data is available. In this case, the "batch" read or write on the buffer is effectively used. Similarly, at the end of the function, code must be added to copy the output array to the output buffer.

6.5 Performance Evaluation

6.5.1 SycView Measurements

This section presents SycView measurements for our decoder platform, done on the sequential version. The first measurement is the wall-clock time consumption of processes, shown in Table 6.1.

Name	Type	Total (ms)	%	1st Q.	Median	3rd Q.	Transitions	Mean
decoder.idct.compute	Thread	11,806	55.7	0.4	1.1	1.1	14,689	0.8
cpu.compute	Thread	3474	16.4	0.1	0.3	0.3	14,743	0.2
decoder.upsampler.compute	Thread	2374	11.2	< 0.1	0.2	0.2	14,716	0.2
decoder.conv2rgb.compute	Thread	2036	9.6	< 0.1	0.2	0.2	14,716	0.1
decoder.iqzz.compute	Thread	1179	5.6	< 0.1	0.1	0.1	14,689	<0.1
SystemC_Kernel	Thread	185	0.9	< 0.1	< 0.1	< 0.1	36,988	<0.1

Table 6.1: Wall-clock time usage of the most consuming parts of the simulation. Time values are expressed in ms.

Another result obtained with SycView is the number of runnable SystemC threads per simulation cycle, shown in Figure 6.10. There are 65.0 % of the delta cycles that start with one runnable SystemC thread, and 33.8 % with two runnable threads. During the simulation, the processes from the decoder block are all waiting for data, most of the time. In the sequential simulation, while they are waiting for data, the sub-blocks are in practice waiting for the notification of a SystemC event, that notifies that data are available in the buffer.

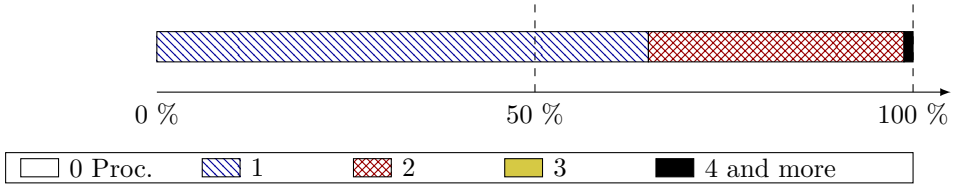


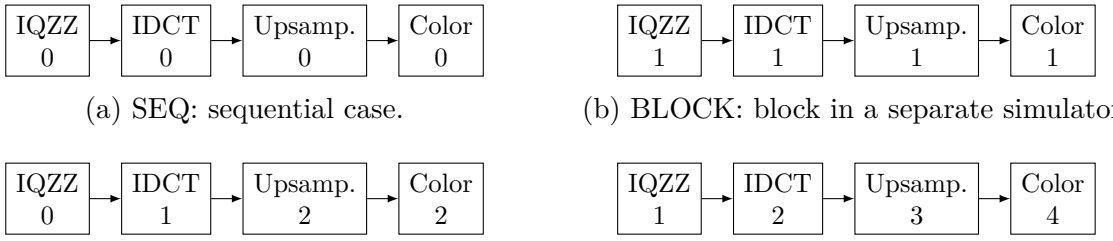
Figure 6.10: Partitioning of delta cycles, per number of runnable processes, for an execution of the decoder platform.

Those measurements are similar to the profiling results on the case study, presented in Chapter 3. Thus, even though this platform is much less complex than our case study, it represents a simpler instance of the same performance issue, due to the simulation of hardware acceleration blocks.

6.5.2 Results for Parallel Execution

From these measurements, one could, at first, see little potential for parallel execution. However, in this case there is such a potential as our results will show, using the technique previously presented. To evaluate the performance of the decoder platform previously presented, and thus measure the efficiency of our parallel simulation system, we measured the median wall-clock time needed to decode and display one frame of an MJPEG stream. This median value is computed after the decoding duration of frames stabilizes.

We measured this value for three different MJPEG streams, each with a different image size: 256×144, 1024×768 and 1920×800. For each different size, we have run four different cases, that correspond to four different ways of mapping SystemC processes to Operating System (OS) processes. Figure 6.11 describes the four cases.



(a) SEQ: sequential case.

(b) BLOCK: block in a separate simulator.

(c) BALANCE: group blocks to balance load.

(d) ONE-TO-ONE: each block in its simulator.

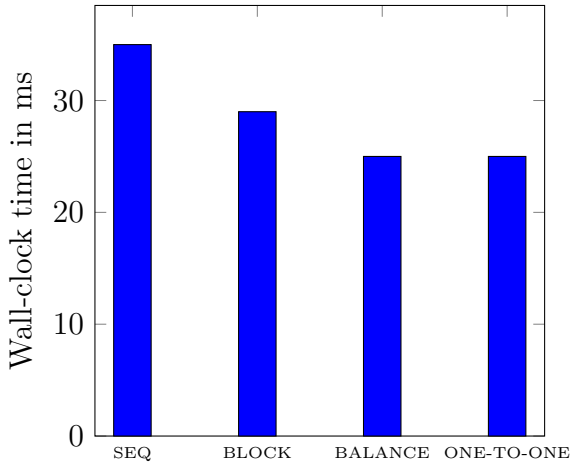
Figure 6.11: The four mapping cases we used for the decoder block. The number indicates the simulator in which the sub-block is run. The rest of the SystemC platform runs on simulator number 0.

Case (SEQ) is used as reference to compute the speed up value. Case (BLOCK) runs each sub-block on the same simulator, but a different one than the rest of the simulation. Case (ONE-TO-ONE) runs each sub-block on a different simulator. Case (BALANCE) balances the load amongst the simulators. Indeed, we have previously seen that in the sequential version, the blocks are, from the most wall-clock time consuming to the least: the IDCT (far ahead), the upsampler, the color converter and finally the IQZZ. This case represents a balance between the different parallel parts. Indeed, the simulator 1 groups processes that consumed 55.7 % of the wall-clock time (in fact only one block, thus indivisible with our approach), the simulator 2 groups 20.8 % (the upsampler and color conversion) and the simulator 0 the rest, which makes 23.5 %.

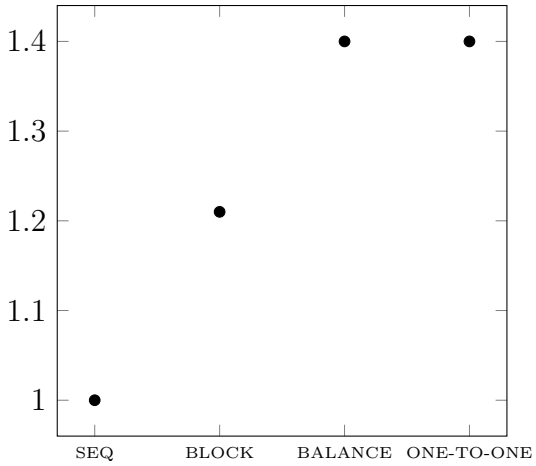
Figures 6.12 to 6.14 shows the results on the JPEG decoder platform for the four cases. The charts on the left hand side present the median wall-clock time to decode and display one frame, and the ones on the right hand side shows the speed up compared to the sequential case. Each figure shows results for a different encoded image size.

The best speed up in our experiments was achieved in Case (ONE-TO-ONE), the case where each block is in parallel with the other ones. On each experiment, there was a speed up of approximately 1.2 in Case (BLOCK). In this case, all the parts of the decoding algorithm are run on the same process, but in parallel with the rest of

the platform. This notably enables to run CPU computations and display operations in parallel with the decoding operations. An interesting point is that the same speed up (or very close) is reached in Case (BALANCE) and (ONE-TO-ONE). Indeed, Case (ONE-TO-ONE) uses the most parallel computing units, however Case (BALANCE) use them better by balancing (as possible) the load between them. For example, for the third experiment with the biggest image size, we reached a speed up of 1.57 using 4 processes (one for the model plus 3 for the decoder).

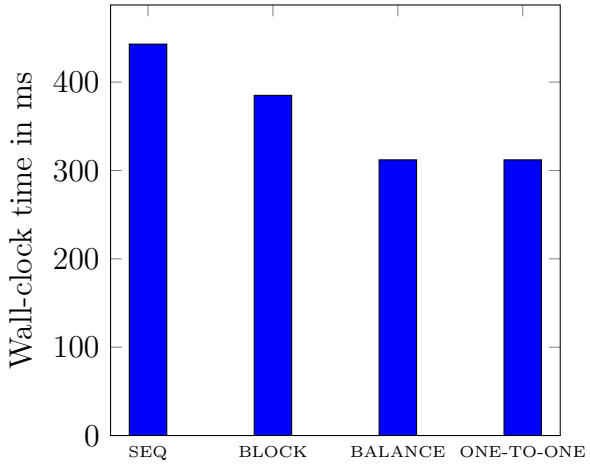


(a) Median wall-clock time to decode and display one frame.

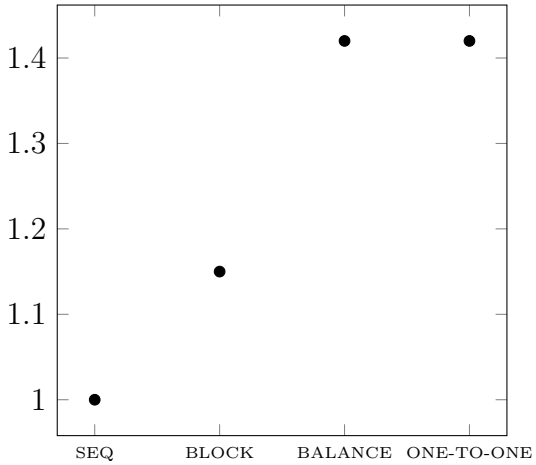


(b) Speed up compared to SEQ.

Figure 6.12: Wall-clock time to decode and display one frame of an MJPEG stream, in colors, of a resolution of 256×144.



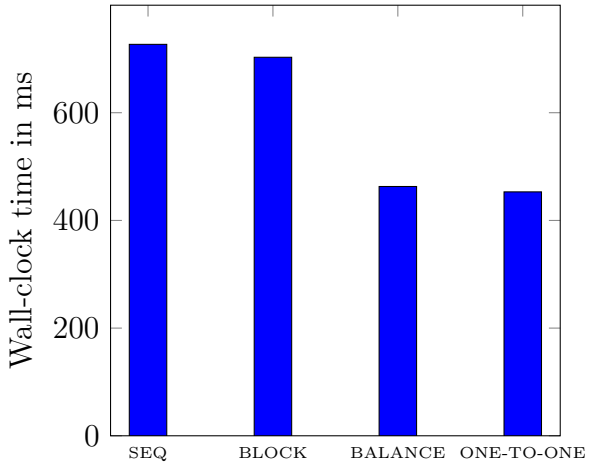
(a) Median wall-clock time to decode and display one frame.



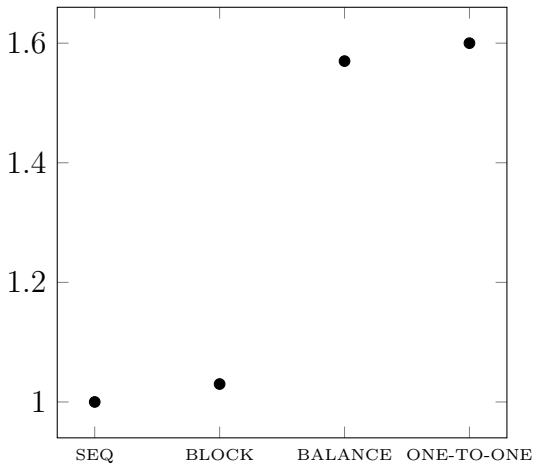
(b) Speed up compared to SEQ.

Figure 6.13: Wall-clock time to decode and display one frame of an MJPEG stream, in colors, of a resolution of 1024×768.

Currently, the mapping of each sub-block to a different concurrent unit is done statically, in a text file. However, it is possible in theory to make this mapping automatic,



(a) Median wall-clock time to decode and display one frame.



(b) Speed up compared to SEQ.

Figure 6.14: Wall-clock time to decode and display one frame of an MJPEG stream, in colors, of a resolution of 1920×800.

using the best load balancing, by running first a profiling on the platform and then use those results to generate the mapping. The results tend to indicate that at least in our case (with few sub-blocks) the best balancing was reached not by running each sub-block to a different computation unit, but by balancing each simulator. Thus, it has the advantage to save some of the parallel computing potential of the host machine for another parallelization technique, applied to another part of a SystemC model that one could identify as potentially executable in physical parallelism. Using extra-parallel computation potential will increase the speed up more than using the maximum possible cores for our approach only.

6.6 Conclusion

This chapter presented an application of the previously presented parallel simulation infrastructure to the case of a hardware design written for the HLS tool CatapultC. This results in integrating the HLS code into a SystemC/TLM simulation, that enables the parallel execution of the integrated component.

For this experiment, we developed an industrial platform, representative of the problem we address. The performance results notably shown a speed up of 1.6 for a simulation using 4 processes. The best performance results were obtained with the highest image resolutions. This is explained by the fact that there is more parallelism exploited by our approach, simply because there is more computations to do when images are bigger.

The speed up achieved is a good result, especially when it is put it in perspective with the rest of the work. In this thesis, we studied cases of SystemC models where finding potential for parallel computation is hard. Our case study, presented in Chapter 3, illustrates this problem well. In this work, the first and the main question to answer was not “how do we run this simulation in parallel?”, but “what can we run in parallel?”.

We did not find, at first sight, potential parallelism in the model, with existing approaches. Thus, we choose to take the problem differently. We first studied sequential profiling results of the simulation of a platform, in order to find which part of the platform effectively takes time. Then, from those results, we proposed an infrastructure that address one instance of this problem: the simulation of hardware components, described for the High Level Synthesis (HLS) tool CatapultC. In this situation, there is a potential for parallelism: the components are described as a hardware pipeline of blocks. We exploit this parallelism by turning it into physical parallelism during the simulation. For now, there is no better way to speed up such types of models with parallel computation. Further research work is necessary in order to study more complex designs, for example with data feedback loops, or designs made for other HLS tools. Moving from one HLS tool to the other is not simply a syntactic change, each tool has its own way to describe behaviors, although the underlying model of computation is resulting in a Kahn Process Network (KPN).

Beyond our specific parallel simulation infrastructure, further research work can identify other potentially parallel parts in real-life simulations. This can lead to the need of combining multiple parallel simulation approaches, applied to different parts of a model. This research direction, and other possible ones, are further discussed in the following conclusion chapter.

— Chapter 7 —

Conclusion

Summary

The main objective of this thesis is to study the performance issue that started to appear in complex models of systems on chip at the Transaction Level Modeling (TLM) abstraction level, and propose a solution for this issue. The proposed approach addresses both legacy and future designs, based on real-life design practices from the industry. The study of parallel execution approaches is natural in this context, because models describe parallel systems, and parallel computation resources are available on today's computers.

The first step of this thesis was to profile a complex case study that showed performance issues when running simulations. To have a synthetic view of this profile, we developed a profiling tool based on an instrumented SystemC kernel. This tool has been presented at the Design Automation for Understanding Hardware Designs (DUHDe) workshop [13]. We used this tool on the industrial model of a set-top box, that provided us with results that we studied to characterize the type of models in our scope, and what causes the slowness. The profiling results and analysis have been published at the Rapid Systems Prototyping (RSP) international symposium [14] and further discussed in an article of the MDPI Electronics journal [15]. From the profiling results, we notably saw that the computation complexity is located mostly in hardware Intellectual Property (IP) blocks.

The use of an High Level Synthesis (HLS) flow to design complex hardware acceleration blocks is common in the industry: video, audio, or signal processing blocks are embedded in a wide variety of systems on chip. Code for HLS describes complex hardware blocks, containing intellectual property. Once this description exists, it is critical to re-use it in the TLM model, because timing constraints on model development are tight. Moreover, re-using the same code ensures consistency between different representations. We assume that future designs will contain even more IP blocks described in HLS, and disqualify

with time the handwriting of Register Transfer Level (RTL) in this purpose. The re-use of HLS code in fast SystemC/TLM simulations is a situation that had to be addressed, as industrial actors have and will have to face it. Consequently, we decided to study this problem. We identified a potential for the parallel execution of such IP blocks. We proposed a parallel simulation infrastructure, and applied it to enable the integration and parallel simulation of such hardware blocks. In summary, our approach is based on the reasonable assumptions that future designs will still use hardware pipelining, with data streaming, and that there will be more IP blocks designed with an HLS design flow, that must be integrated quickly in fast SystemC/TLM simulations.

Results

Our parallel execution approach for hardware acceleration blocks developed for an HLS design flow showed promising results. We developed an example platform representative of industrial designs, on which a speed-up of 1.6 was achieved compared to a sequential execution, using 4 processes. This speed-up did not involve to modify the HLS code, and required to write relatively small wrappers for the SystemC/TLM simulation. Moreover, the rest of the platform and the interfaces with the IP block can use existing constructs, which induce no refactoring effort on the rest of the SystemC/TLM model.

This thesis is a step in this direction, which we believe address crucial overcoming challenges of system on chip simulation. The current version of our infrastructure has limitations, for example designs with feedback loops have not been studied. Moreover, the case of HLS tools that rely on a different description of blocks than CatapultC can cause further problems, since they may require to use code transformation. However, the underlying model of computation of such blocks still falls within Kahn Process Network (KPN) models, which is addressed by our approach.

Another important conclusion of this thesis is that there is no “miracle” solution for parallel SystemC simulation. This statement is not trivial: it comes from a study of both the state of the art and an industrial case study. An interesting question addressed by research work is how to turn a sequential SystemC simulator into a parallel simulator. Some approaches have chosen to aim for semantics-preserving simulation, and some others have made assumptions on models to enable more parallelism. The profiling results on our case study showed another problem: finding potential for parallel simulation when the model description does not expose it. In other words, simulation at TLM are not the same than at RTL, and even within the TLM abstraction level, different types of models shows different parallel simulation possibilities.

Prospects

Finally, we can discuss about longer-term considerations. One first consideration is the Internet of Things (IoT). The IoT is a paradigm where a wireless network of embedded

devices collect, process and exchange data with each other. One possible application is home automation: a temperature sensor can control the central heating, a light sensor can control the shutters, and so on. The biggest technical challenges in this area are low-power consumption, modularity, data processing, data management and security [65, 66, 67]. In particular, the modularity and communication aspects of such systems brings new challenges to the simulation. What makes the complexity of simulations here is not one complex chip, but the quantity of interconnected chips, and the handling of networking protocols between various devices. In other words, this is a system of systems (on chip), where the system is dynamic: components can be added or removed from the system while it is running. Parallel simulation comes naturally at stake in this area: in addition to speed up simulations, it is a solution for the dynamic aspect and the integration of various components in a model. Parallel simulation of dynamic systems is currently under development at STMicroelectronics, as it is an overcoming issue, and this thesis work will also contribute to advancing in this direction.

There are common problems to solve between our infrastructure and the case of parallel simulators for dynamic systems. For example, avoiding a simulation to terminate because it waits for something coming from a different thread or from another simulation. In our case, this situation could only happen while reading or writing an empty or a full buffer, thus we proposed a solution that addresses this case. More generally, each parallel approach that uses multiple simulators has to deal with such a problem.

One other challenge is the simulation of multi-domain systems; they include physical modules, for example analog signal processing (optical, acoustic, *etc*). Such simulations combine discrete event simulation for the digital components, and continuous simulation for analog components. This case is addressed by an extension of SystemC called SystemC Analog/Mixed-Signal (AMS), standardized by Accellera. SystemC/AMS has been further extended with SystemC Multi-Disciplinary Virtual Prototyping (MDVP) that precisely addresses the case of multi-physical domains simulations [68, 69]. SystemC MDVP focused on integrating multi-physical systems in a sequential SystemC simulation, which not so long ago was not possible. The next step will consist in studying the parallel simulation of those systems. They are *a priori* good candidates for parallel simulation, since they include independent physical parts. Existing parallel simulation approaches, including ours, build up a solid ground basis to come with a solution that addresses the simulation of such systems.

At one point, research on parallel simulation will have to mix different parallel simulation approaches. Our state of the art analysis have shown that an approach that works well in one context will not work in another context; not because the solution is bad, but because it is not the same problem. An interesting prospect is thus to study the integrability of different parallel simulation techniques on a single simulation. For example, a simulation where hardware acceleration blocks are run in parallel, where some analog parts of the system are also accelerated with parallel computation, and where the whole system communicates, through a network interface, with another similar system simulated in parallel. This would require first to study the interaction between different simulators, or between different parallel simulation techniques. In this situation, different SystemC

kernel implementations could be combined, and for example a parallel SystemC kernel implementation could be used for one part of the simulation, in parallel with sequential ones. This kind of research will mainly have to deal with communication and integration problems between different types of simulators. We believe this thesis is an important step in this direction.

Acronyms

API Application Programming Interface. 53, 73

ASI Accelera Systems Initiative. 16, 39

AT Approximately-Timed. 21

CPU Central Processing Unit. 5, 8, 20, 21, 25, 45, 47, 56, 68, 71, 74, 86, 106

CUDA Compute Unified Device Architecture. 74

DES Discrete Event Simulation. 12, 58

DMA Direct Memory Access. 106

FIFO First In, First Out. 8, 32, 82–85, 89, 91, 93, 96, 102, 108, 109

FOFIFON Fast Ordered First In, First Out data exchaNge. 8, 83, 86, 87, 89, 91, 92, 102, 111–114

GPU Graphics Processing Unit. 8, 62, 74

GUI Graphical User Interface. 37, 38, 41

HDL Hardware Description Language. 25

HLS High Level Synthesis. 6–8, 25–33, 82–84, 102, 105–107, 110, 111, 113, 119, 121, 122, 127

IDCT Inverse Discrete Cosine Transform. 104, 113, 126, 127

IMC Interface Method Call. 15, 16, 20, 47, 60, 69, 71, 73, 76, 80

IP Intellectual Property. 7, 25, 28–31, 47–49, 56, 71, 80, 82, 84, 86, 102, 104, 111, 121, 122

IQZZ Inverse Quantization and Inverse Zig-Zag. 104, 125–127

ISS Instruction Set Simulator. 21, 22, 27, 71, 80

KPN Kahn Process Network. 84, 85, 91, 93, 101, 119, 122

LT Loosely-Timed. 16, 21–23, 32, 36, 51, 58, 70, 74

MPP Massively Parallel Processing. 59

MPSoC Multi-Processor System on Chip. 58

NoC Network on Chip. 59, 68, 70

OS Operating System. 10, 58, 79, 86, 93, 99, 116

OSCI Open SystemC Initiative. 16

PDES Parallel Discrete Event Simulation. 58, 60, 63, 70, 73

RGB Red, Green, Blue. 104, 105, 127, 128

RTL Register Transfer Level. 6, 25–32, 36, 51, 60, 63, 64, 74, 76, 107, 121, 122

SLDL System-Level Description Language. 76

SMP Symmetric Multi-Processing. 58

TLM Transaction Level Modeling. 7, 13, 15, 16, 19–24, 27, 32, 36, 47, 51, 58, 60, 67–70, 73–76, 80, 82–84, 106, 121, 122

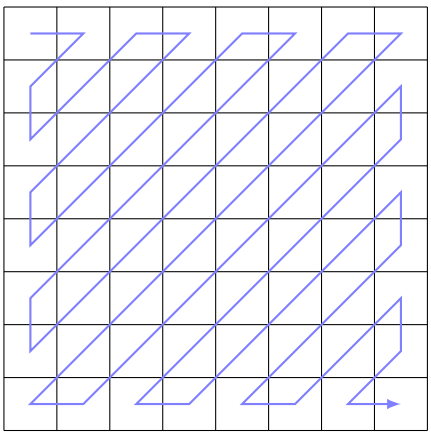
$\mathbf{Y'C_BC_R}$ Luma and Blue/Red-differences Chroma. 104, 105, 127, 128

— Appendix A —

JPEG Decoding Algorithm

A.1 Inverse Quantization and Inverse Zig-Zag

The first step is the Inverse Quantization and Inverse Zig-Zag (IQZZ). It consists in reordering values of a macroblock (the "zig-zag" part) and multiply them with a table of coefficients present in the encoded stream (the "inverse quantization" part). The table of coefficients is called the dequantization table. Figure A.1 illustrates this step of the decoding for a macroblock (8×8 pixels). There is generally one dequantization table per color component.



Macroblock

×

8	13	16	16	18	21	21	28
10	13	16	18	18	21	23	28
10	13	16	18	16	23	23	35
11	13	16	18	16	23	25	35
10	16	16	21	18	23	25	36
11	15	16	21	18	21	25	43
13	16	16	21	20	21	30	43
13	16	16	18	20	21	30	51

Dequantization table

Figure A.1: Illustration of the IQZZ.

A.2 Inverse Discrete Cosine Transform

A.2.1 Theory

The step after the Inverse Quantization and Inverse Zig-Zag (IQZZ) is the Inverse Discrete Cosine Transform (IDCT). This step transforms a macroblock from the frequency domain to the spatial domain. The formula for a macroblock of size $n \times n$ is:

$$S(x,y) = \frac{1}{\sqrt{2n}} \sum_{\lambda=0}^{n-1} \sum_{\mu=0}^{n-1} C(\lambda)C(\mu) \cos\left(\frac{2x+1}{2n}\lambda\pi\right) \cos\left(\frac{2y+1}{2n}\mu\pi\right) \times \mathrm{F}(\lambda,\mu),$$

where S is the spatial macroblock, F is the frequency macroblock, x and y are the coordinates in the spacial domain, λ and μ are the coordinates in the frequency domain and C is defined as:

$$C(k) = \begin{cases} \frac{1}{\sqrt{2}} & \text{if } k = 0, \\ 1 & \text{otherwise.} \end{cases}$$

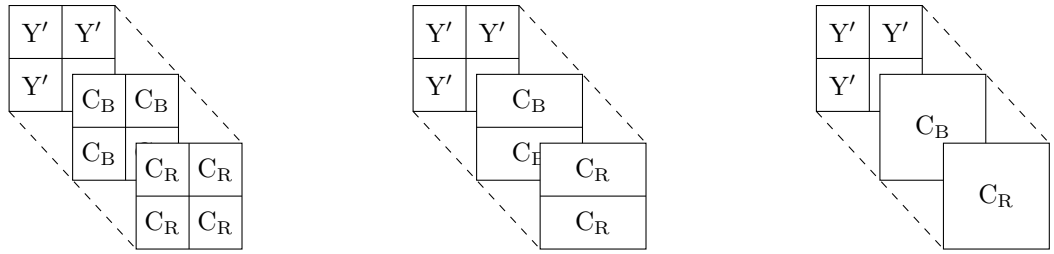
A.2.2 Implementation Notes for HLS

The IDCT is the step that needs the most work to adapt the mathematical formula for an efficient hardware implementation. Indeed, it contains costly operations for a hardware implementation, *e.g.* square root or cosine. There already exist various IDCT hardware implementations, and our goal here is not to discuss their differences or efficiency.

For our HLS implementation, the macroblock size is known and fixed to 8×8. This already removes the initial square root. Moreover, the computation of cosines can be avoided by using a precomputed cosine table. Indeed, the cosine parameters do not depend on pixel values, but only on indexes.

A.3 Upsampling

After the Inverse Discrete Cosine Transform (IDCT), the spatial information obtained is in the color space Luma and Blue/Red-differences Chroma ($Y'C_BC_R$). This color space is used in the JPEG encoding instead of the classic Red, Green, Blue (RGB) because it enables space saving by removing details on colors that are barely perceptible by the human eye. The luma (Y') component is present in any case, because the eye is sensitive to luminosity changes. The blue and red differences (C_B and C_R) are sometimes subsampled. Subsampling consists in using the same chrominance value for multiple pixels. Different strategies are possible for subsampling, the most common are represented on Figure A.2.



(a) 4:4:4 (no subsampling) (b) 4:2:2 (c) 4:2:0

Figure A.2: Most common subsampling strategies.

Similarly to the Inverse Quantization and Inverse Zig-Zag (IQZZ), the upsampling step does not need major refactoring for High Level Synthesis (HLS). The upsampling factors (whether to use 4:4:4, 4:2:0 or 4:2:2) are required inputs. The streaming behavior is not applicable at pixel-scale here, for the same reason as for IQZZ, then full macroblocks must be read before starting to output the resulting macroblocks.

A.4 Color Model Change

A.4.1 Theory

The final step is to convert colors from the Luma and Blue/Red-differences Chroma ($Y'C_BC_R$) color model to Red, Green, Blue (RGB). The conversion is straightforward, it consists in applying the following formula to each pixel:

$$
\begin{aligned}
R &= Y' + 1.402 \times (C_R - 128) \\
G &= Y' - 0.34414 \times (C_B - 128) - 0.71414 \times (C_R - 128) \\
B &= Y' + 1.772 \times (C_B - 128)
\end{aligned}
$$

A.4.2 Implementation Notes for HLS

Each color component either from the $Y'C_BC_R$ or RGB color models is an integer. However, the constants from the original formula are decimal numbers. For an hardware implementation, floating point computations are better avoided for performance reasons. Then, two options are possible: using fixed-point numbers or integer numbers.

In our context, we use integer constants. We use the implementation proposed by Manz [70], that consists in multiplying each decimal constant by 1024 and then we divide the result by the same value. The values of the constants multiplied by 1024 can be rounded to integer values without loosing too much accuracy. Moreover, the multiplication and division by 1024 is convenient in hardware because it simply consists in a 10-bit shift. After this modification, the formula are:

$$
\begin{aligned}
R &= \left(1024 \times Y' + 1436 \times (C_R - 128)\right) / 1024 \\
G &= \left(1024 \times Y' - 352 \times (C_B - 128) - 731 \times (C_R - 128)\right) / 1024 \\
B &= \left(1024 \times Y' + 1815 \times (C_B - 128)\right) / 1024
\end{aligned}
$$

— Appendix B —

Detailed Examples

B.1 Example Code of a Top Module using DistemC

```
#include "distemc.h"

SC_MODULE(Top) {

    SC_CTOR(Top) {
        // "create_module" checks the map file (given as input parameter)
        // to check if the module (by its name) has a given affinity,
        // otherwise it is 0
        producer = distemc::create_module<Producer>("Producer");
        sub_block1 = distemc::create_module<Sub_Block1>("Sub_Block1");
        sub_block2 = distemc::create_module<Sub_Block2>("Sub_Block2");
        sub_block3 = distemc::create_module<Sub_Block3>("Sub_Block3");
        consumer = distemc::create_module<Consumer>("Consumer");

        // "create_fofifon" takes the pointers to the two modules that will
        // use the FOFIFON (null => not on current simulator, not null => on
        // current simulator) the name of the buffer and its capacity
        producer_to_first = distemc::create_fofifon<uint32_t>(
                producer, sub_block1, "producer_to_first", 200);
        first_to_second = distemc::create_fofifon<uint32_t>(
                sub_block1, sub_block2, "first_to_second", 200);
        second_to_third = distemc::create_fofifon<uint32_t>(
                sub_block2, sub_block3, "second_to_third", 200);
        third_to_consumer = distemc::create_fofifon<uint32_t>(
                sub_block3, consumer, "third_to_consumer", 200);

        // The "producer" is always in simulator 0.
```

```
        // It can be e.g. a DMA that reads data from
        // memory and feeds the input stream with it.
        if (producer) {
            producer->output_port(*producer_to_first);
        }
        if (sub_block1) {
            sub_block1->input_port(*producer_to_first);
            sub_block1->output_port(*first_to_second);
        }
        if (sub_block2) {
            sub_block2->input_port(*first_to_second);
            sub_block2->output_port(*second_to_third);
        }
        if (sub_block3) {
            sub_block3->input_port(*second_to_third);
            sub_block3->output_port(*third_to_consumer);
        }
        // The "consumer" is also always in simulator 0.
        // It can be e.g. another DMA that pulls data
        // from the stream and writes it in memory.
        if (consumer) {
            consumer->output_port(*third_to_consumer);
        }
    }

    Producer * producer;
    Sub_Block1 * sub_block1;
    Sub_Block2 * sub_block2;
    Sub_Block3 * sub_block3;
    Consumer * consumer;

    distemc::fofifon<uint32_t> * producer_to_first;
    distemc::fofifon<uint32_t> * first_to_second;
    distemc::fofifon<uint32_t> * second_to_three;
    distemc::fofifon<uint32_t> * three_to_consumer;

};
```

B.2 Example Scenario on a FOFIFON Structure

(1) Initial state (empty).

(2) The writer pushes two values.

(3) The reader updates its local counters.

(4) The reader pulls one value.

(5) The writer updates its local counters.

(6) The writer pushes 8 values (full).

(7) The reader updates its local counters.

(8) The reader pulls 6 values.

(9) The writer updates its local counters.

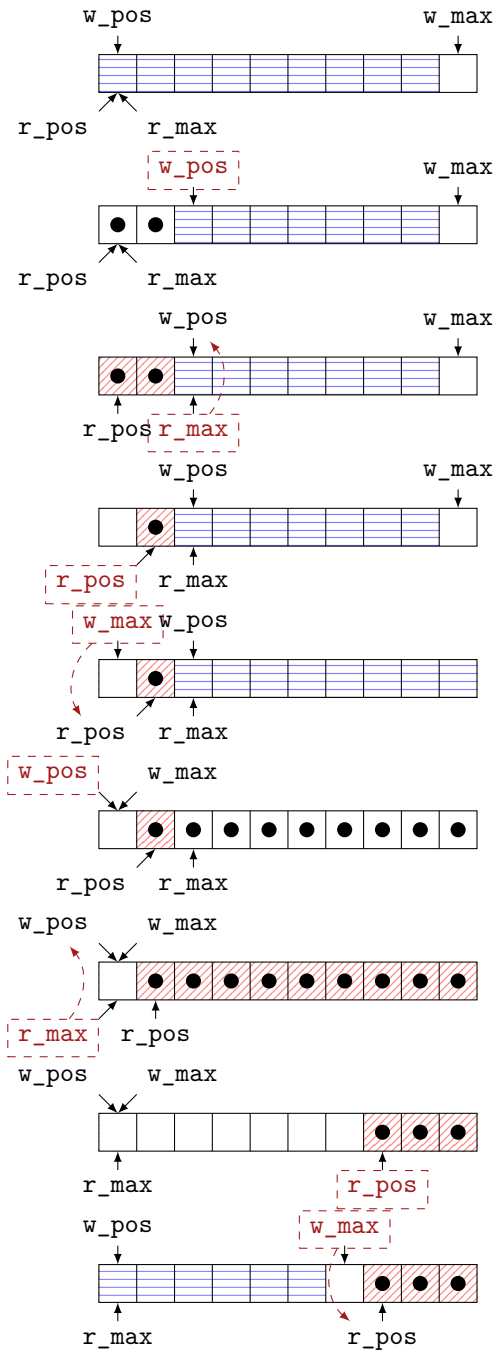


Figure B.1: Possible execution scenario for a sequence of reads and writes on the queue. A circular buffer is used. The visible size of the FIFO is 9, thus the implementation uses an array of size 10.

Bibliography

[1] IEEE Standard for Standard SystemC Language Reference Manual. *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pages 1–638, 2012.

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

[3] Claude Helmstetter. *Validation de Modèles de Systèmes sur Puce en Présence d'Ordonnancements Indéterministes et de Temps Imprécis*. PhD thesis, Institut National Polytechnique de Grenoble (INPG), 2007.

[4] Matthieu Moy. *Techniques et Outils pour la Vérification de Systèmes-sur-Puce au Niveau Transaction*. PhD thesis, Institut National Polytechnique de Grenoble (INPG), 2005.

[5] Jérôme Cornet. *Séparation des Aspects Fonctionnels et non-Fonctionnels dans les Modèles Transactionnels des Systèmes sur Puce*. PhD thesis, Institut Polytechnique de Grenoble (IPG), 2008.

[6] Giovanni Funchal. *Contributions to the Transaction-Level Modeling of Systems-on-a-Chip*. PhD thesis, Institut National Polytechnique de Grenoble (INPG), 2011.

[7] Yussef Bouzouzou. Accélération des Simulations de Systèmes sur Puce au Niveau Transactionnel. Diplôme de Recherche Technologique (DRT), Université Joseph Fourier, 2007.

[8] G. Martin and G. Smith. High-Level Synthesis: Past, Present, and Future. *IEEE Design Test of Computers*, 26(4):18–25, 2009.

[9] H. Ren. A Brief Introduction on Contemporary High-Level Synthesis. In *IEEE International Conference on IC Design Technology*, pages 1–4, 2014.

[10] D. J. Pagliari, M. R. Casu, and L. P. Cartoni. Acceleration of Microwave Imaging Algorithms for Breast Cancer Detection via High-Level Synthesis. In *IEEE International Conference on Computer Design (ICCD)*, pages 475–478, 2015.

[11] G. Inggs, S. Fleming, D. Thomas, and W. Luk. Is High Level Synthesis Ready for Business? A Computational Finance Case Study. In *International Conference on Field-Programmable Technology (FPT)*, pages 12–19, 2014.

[12] D. Gajski, T. Austin, and S. Svoboda. What Input Language is the Best Choice for High Level Synthesis (HLS)? In *Design Automation Conference (DAC)*, pages 857–858, 2010.

[13] Denis Becker, Matthieu Moy, and Jérôme Cornet. SycView: Visualize and Profile SystemC Simulations. In *Workshop on Design Automation for Understanding Hardware Designs*, 2016.

[14] Denis Becker, Matthieu Moy, and Jérôme Cornet. Challenges for the Parallelization of Loosely-Timed SystemC Programs. In *IEEE International Symposium on Rapid System Prototyping (RSP)*, 2015.

[15] Denis Becker, Matthieu Moy, and Jérôme Cornet. Parallel Simulation of Loosely Timed SystemC/TLM Programs: Challenges Raised by an Industrial Case Study. *Electronics*, 5(2):22, 2016.

[16] Daniel Große, Rolf Drechsler, Lothar Linhard, and Gerhard Angst. Efficient Automatic Visualization of SystemC Designs. In *FDL*, pages 646–658, 2003.

[17] David Berner, Jean-Pierre Talpin, Hiren D Patel, Deepak Mathaikutty, and Sandeep K Shukla. SystemCXML: An Extensible SystemC Front end Using XML. In *FDL*, pages 405–409, 2005.

[18] C. Albrecht, C. J. Eibl, and R. Hagenau. A Loosely-Coupled Graphical User Interface for Run-Time Control of SystemC Simulation Models. *IJSSST*, 2006.

[19] C. Genz and R. Drechsler. System Exploration of SystemC Designs. In *Emerging VLSI Technologies and Architectures, IEEE Computer Society Annual Symposium on*, pages 6 pp.–, 2006.

[20] C. Genz, R. Drechsler, G. Angst, and L. Linhard. Visualization of SystemC Designs. In *Circuits and Systems (ISCAS), IEEE International Symposium on*, pages 413–416, 2007.

[21] Rolf Drechsler and Jannis Ulrich Stoppe. Hardware/Software Co-Visualization on the Electronic System Level using SystemC. In *International Conference on VLSI Design*. IEEE, 2016.

[22] Jason Liu, James J. Cochran, Louis A. Cox, Pinar Keskinocak, Jeffrey P. Kharoufeh, and J. Cole Smith. *Parallel Discrete-Event Simulation*. John Wiley & Sons, Inc., 2010.

[23] B. D. de Dinechin, R. Ayrignac, P. E. Beaucamps, P. Couvert, B. Ganne, P. G. de Massas, F. Jacquet, S. Jones, N. M. Chaisemartin, F. Riss, and T. Strudel. A clustered manycore processor architecture for embedded and accelerated applications. In *IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2013.

[24] B. Chopard, P. Combes, and J. Zory. A Conservative Approach to SystemC Parallelization. In *Computational Science, ICCS*, volume 3994, pages 653–660. Springer Berlin Heidelberg, 2006.

[25] Christoph Schumacher, Rainer Leupers, Dietmar Petras, and Andreas Hoffmann. parSC: Synchronous Parallel SystemC Simulation on Multi-core Host Architectures. In *Hardware/Software Codesign and System Synthesis, IEEE/ACM/IFIP International Conference on*, pages 241–246. ACM, 2010.

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

[27] M. Nanjundappa, H.D. Patel, B.A. Jose, and S.K. Shukla. SCGPSim: A Fast SystemC Simulator on GPUs. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 149–154, 2010.

[28] Weiwei Chen and Rainer Dömer. Optimized Out-of-order Parallel Discrete Event Simulation Using Predictions. In *Design, Automation and Test in Europe (DATE)*, pages 3–8. EDA Consortium, 2013.

[29] T. Schmidt, G. Liu, and R. Dömer. Hybrid Analysis of SystemC Models for Fast and Accurate Parallel Simulation. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 226–231, 2017.

[30] Simon Reder, Christoph Roth, Harald Bucher, Oliver Sander, and Jürgen Becker. Adaptive Algorithm and Tool Flow for Accelerating SystemC on Many-Core Architectures. *Microprocessors and Microsystems*, pages 1063–1075, 2015.

[31] Philippe Combes, Eddy Caron, Frédéric Desprez, Bastien Chopard, and Julien Zory. Relaxing Synchronization in a Parallel SystemC Kernel. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, 2008.

[32] S. Vinco, V. Bertacco, D. Chatterjee, and F. Fummi. SAGA: SystemC Acceleration on GPU Architectures. In *Design Automation Conference (DAC)*, pages 115–120, 2012.

[33] Emmanuel Viaud, François Pêcheux, and Alain Greiner. An Efficient TLM/T Modeling and Simulation Environment Based on Conservative Parallel Discrete Event Principles. In *Design, Automation and Test in Europe (DATE)*, volume 1, pages 1–6, 2006.

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

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

[36] Samuel Jones. Optimistic Parallelisation of SystemC. Master's thesis, Université Joseph Fourier: MoSiG DEMIPS, 2011.

[37] C. Sauer, H.-M. Bluethgen, and H.-P. Loeb. Distributed Loosely-Synchronized SystemC/TLM Simulations of Many-Processor Platforms. In *Forum on Specification and Design Languages (FDL)*, volume 978-2-9530504-9-3, pages 1–8, 2014.

[38] Christoph Schumacher, Jan Henrik Weinstock, Rainer Leupers, Gerd Ascheid, Laura Tosoratto, Alessandro Lonardo, Dietmar Petras, and Hoffmann Andreas. legaSCi: Legacy SystemC Model Integration into Parallel SystemC Simulators. In *Proceedings of the Workshop on Virtual Prototyping of Parallel and Embedded Systems, in Proceedings of Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW)*, pages 2188–2193, 2013.

[39] Jan Henrik Weinstock, Christoph Schumacher, Rainer Leupers, Gerd Ascheid, and Laura Tosoratto. Time-Decoupled Parallel SystemC Simulation. In *Design, Automation and Test in Europe (DATE)*, pages 191:1–191:4. European Design and Automation Association, 2014.

[40] J.H. Weinstock, R. Leupers, and G. Ascheid. Parallel SystemC Simulation for ESL Design Using Flexible Time Decoupling. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), International Conference on*, pages 378–383, 2015.

[41] Jan Henrik Weinstock, Rainer Leupers, Gerd Ascheid, Dietmar Petras, and Andreas Hoffmann. SystemC-Link: Parallel SystemC Simulation using Time-Decoupled Segments. In *Design, Automation and Test in Europe (DATE)*, 2016.

[42] Hiren D. Patel and Eep K. Shukla. Towards a Heterogeneous Simulation Kernel for System Level Models: A SystemC Kernel for Synchronous Data Flow Models. In *Proceedings of the 14th ACM Great Lakes symposium on VLSI*, pages 248–253. ACM Press, 2004.

[43] R. Fujimoto. Parallel and Distributed Simulation. In *Winter Simulation Conference (WSC)*, pages 45–59, 2015.

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

[45] Nicolas Ventroux and Tanguy Sassolas. A New Parallel SystemC Kernel Leveraging Manycore Architectures. In *Design, Automation and Test in Europe (DATE)*, 2016.

[46] W. Chen, X. Han, and R. Dömer. Out-of-Order Parallel Simulation for ESL Design. In *Design, Automation and Test in Europe (DATE)*, pages 141–146, 2012.

[47] W. Chen, X. Han, and R. Dömer. May-Happen-in-Parallel Analysis based on Segment Graphs for Safe ESL Models. In *Design, Automation and Test in Europe (DATE)*, pages 1–6, 2014.

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

[49] Nicolas Bombieri, Sara Vinco, Valeria Bertacco, and Debapriya Chatterjee. SystemC Simulation on GP-GPUs: CUDA vs OpenCL. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2012.

[50] Giovanni Funchal and Matthieu 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.

[51] Rainer Dömer, Weiwei Chen, Xu Han, and Andreas Gerstlauer. Multi-Core Parallel Simulation of System-Level Description Languages. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2011.

[52] Rainer Dömer, Weiwei Chen, and Xu Han. Parallel Discrete Event Simulation of Transaction Level Models. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 227–231, 2012.

[53] Maria Hybinette and Richard M. Fujimoto. Cloning Parallel Simulations. *ACM Transactions on Modeling and Computer Simulation*, 11(4):378–407, 2001.

[54] Gilles Kahn. The Semantics of a Simple Language for Parallel Programming. *Information Processing*, 74:471–475, 1974.

[55] Thomas M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, University of California at Berkeley, 1995.

[56] Andrei Alexandrescu. Lock-Free Data Structures. *C/C++ User Journal*, 2004.

[57] Working Draft, Standard for Programming Language C++ (N4659). Retrieved from `https://isocpp.org/std/the-standard`. *International Organization for Standardization (ISO)*, 2017.

[58] N. M. Lê, A. Guatto, A. Cohen, and A. Pop. Correct and Efficient Bounded FIFO Queues. In *International Symposium on Computer Architecture and High Performance Computing*, pages 144–151, 2013.

[59] A. Nistor, D. Marinov, and J. Torrellas. Light64: Lightweight Hardware Support for Data Race Detection during Systematic Testing of Parallel Programs. In *42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 541–552, 2009.

[60] Milos Gligoric, Vilas Jagannath, Qingzhou Luo, and Darko Marinov. Efficient Mutation Testing of Multithreaded Code. *Software Testing, Verification and Reliability*, 23(5):375–403, 2013.

[61] Peter A. Buhr, David Dice, and Wim H. Hesselink. Dekker's Mutual Exclusion Algorithm Made RW-Safe. *Concurrency and Computation: Practice and Experience*, 28(1):144–165, 2016.

[62] Dmitry Nadezhkin, Sjoerd Meijer, Todor Stefanov, and Ed F. Deprettere. Realizing FIFO Communication When Mapping Kahn Process Networks onto the Cell. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), International Conference on*, pages 308–317, 2009.

[63] Michael Fingeroff. *High-Level Synthesis Blue Book*. Xlibris Corporation, 2010.

[64] Marc Geilen and Twan Basten. *Requirements on the Execution of Kahn Process Networks*, pages 319–334. Springer Berlin Heidelberg, 2003.

[65] L. Mainetti, L. Patrono, and A. Vilei. Evolution of Wireless Sensor Networks Towards the Internet of Things: A Survey. In *SoftCOM 2011, 19th International Conference on Software, Telecommunications and Computer Networks*, pages 1–6, 2011.

[66] D. Blaauw, D. Sylvester, P. Dutta, Y. Lee, I. Lee, S. Bang, Y. Kim, G. Kim, P. Pannuto, Y. S. Kuo, D. Yoon, W. Jung, Z. Foo, Y. P. Chen, S. Oh, S. Jeong, and M. Choi. IoT Design Space Challenges: Circuits and Systems. In *2014 Symposium on VLSI Technology (VLSI-Technology): Digest of Technical Papers*, pages 1–2, 2014.

[67] Eleonora Borgia. The Internet of Things Vision: Key Features, Applications and Open Issues. *Computer Communications*, 2014.

[68] Liliana Lilibeth Andrade Porras. *Principes et Réalisation d'une Interface de Synchronisation Interopérable entre Modèles de Calcul SystemC AMS pour le Prototypage Virtuel Optimisé de Systèmes Multi-Disciplines*. PhD thesis, École doctorale informatique, télécommunications et électronique (Paris), 2016.

[69] Cédric Ben Aoun. *Principes et Réalisation d'un Environnement de Prototypage Virtuel de Systèmes Hétérogènes Composables*. PhD thesis, École doctorale informatique, télécommunications et électronique (Paris), 2017.

[70] Sebastian Manz. *Development and Implementation of a MotionJPEG Capable JPEG Decoder in Hardware*. PhD thesis, Heidelberg, Univ., Dipl., 2008.

Résumé

Les systèmes sur puce sont constitués d'une partie matérielle (un circuit intégré) et d'une partie logicielle (un programme) qui utilise les ressources matérielles de la puce. La conséquence de cela est que le logiciel d'un système sur puce est intrinsèquement lié à sa partie matérielle. Les composants matériels d'accélération sont des facteurs clés de différenciation d'un produit à l'autre.

Il est nécessaire de pouvoir simuler ces systèmes très tôt lors de leur conception; bien avant que la puce ne soit physiquement disponible, et même avant que la puce ne soit complètement spécifiée. Pour cela, un modèle du système sur puce est réalisé à l'aide du langage SystemC au niveau d'abstraction TLM (Transaction Level Modeling). La partie matérielle d'un système sur puce est constituée de composants qui s'exécutent en parallèle. Pour autant, la simulation avec le simulateur SystemC de référence est séquentielle. Ceci permet de garantir les bonnes propriétés des simulations SystemC, en particulier la reproductibilité et le confort d'écriture des modèles.

Les travaux de cette thèse portent sur la simulation parallèle de modèles SystemC/TLM. L'objectif de l'exécution parallèle est d'accélérer les simulations dans un mode d'utilisation correspondant à la phase de développement, où il est primordial de disposer de simulations qui donnent rapidement un résultat. Afin de cerner le problème de performance remarqué sur des modèles complexes à STMicroelectronics, le premier travail de cette thèse a été d'analyser le profil d'exécution d'une étude de cas représentative de la complexité actuelle des platformes SystemC/TLM. Pour cette étude, nous avons développé un outil de collecte de traces et de visualisation. Les résultats de cette analyse ont indiqué que la lenteur d'exécution en simulation était due à la complexité des composants matériels d'accélération. L'étude de l'état de l'art en simulation parallèle de modèles SystemC nous a conduit à chercher d'autres pistes que celles actuellement existantes.

Pour réaliser les composants matériels plus rapidement et permettre d'augmenter la réutilisabilité de composants d'un projet à l'autre, le flot de conception HLS (High Level Synthesis) est utilisé, notamment à STMicroelectronics. Ce flot de conception permet, à partir de la description d'une fonction en C++, de générer un plan de composant matériel qui va réaliser la même fonction. La description des composants est découpée en sous-fonctions, individuellement plus simples. Afin d'obtenir de bonnes performances, les sous-fonctions sont assemblées en chaîne à travers laquelle circulent les données à traiter. Il est indispensable de pouvoir réutiliser le code écrit pour la HLS dans les simulations SystemC/TLM : cette situation deviendra de plus en plus fréquente, et il n'a pas assez de temps pour réécrire ces modèles dans ces projets courts.

Nous avons développé une infrastructure de simulation parallèle permettant d'intégrer et de simuler efficacement des composants de traitement de données écrits pour la HLS. L'application de cette infrastructure à un exemple a permis d'accélérer l'exécution de la simulation d'un facteur 1.6 avec 4 processeurs. Au-delà de ce résultat, les conclusions principales de cette thèse sont que la simulation parallèle de modèles à haut niveau d'abstraction en SystemC/TLM passe par la combinaison de plusieurs techniques de parallélisation. Il est également important d'identifier les parties parallélisables dans des simulations industrielles, notamment pour les nouveaux défis que sont les simulations multi-physiques et l'internet des objets.

Abstract

Systems on chip consist in a hardware part (an integrated circuit) and a software part (a program) that uses the hardware resources of the chip. Consequently, the embedded software is intrinsically connected to the chip hardware. Hardware acceleration components are key differentiation factors from one product to another.

It is necessary to simulate systems on chip very early in the design flow; before the chip is physically available and even before its full specification. For such simulations, developers write a model of the system on chip in SystemC, at the TLM (Transaction Level Modeling) abstraction level. The hardware part of a chip consists in components that behave in parallel with each other. However, the reference SystemC simulator execute simulations sequentially. The sequential execution enables to keep good properties of SystemC simulations, namely reproducibility and ease of model writing.

This thesis work address the parallel execution of SystemC/TLM simulations. The goal of parallel simulation is to speed up simulations in the context of the model development, where it is important to quickly get results. In order to identify the performance problem of complex models at STMicroelectronics, the first step of this thesis was to analyse the execution profile of a case study, representative of the complexity of current platforms. For this study, we developed a trace recording and visualization tool. The results of this study indicated that the performance critical parts of the simulation are hardware acceleration components. Studying existing parallel simulation approaches led us to look for other parallel simulation techniques.

To speed up the development of hardware acceleration components and increase the reusability from one project to another, the HLS (High Level Synthesis) design flow is used notably at STMicroelectronics. This design flow enables to generate a logically synthesizable model of a component from a high level behavioral description in C++. This design flow also constraints the development: it is split in sub-functions assembled in a pipeline. The code written for HLS must be re-used in SystemC/TLM models: this situation will become more and more frequent and there is no time to rewrite the models of such components within short delays.

We developed a parallel simulation infrastructure enabling the integration and efficient simulation of hardware components written for HLS. We applied this infrastructure to an example platform which resulted in speeding up the simulation. Beyond this result, one of the main conclusion of this thesis is that parallel simulation of abstract SystemC/TLM models will require to combine multiple parallelization techniques. Future research work can identify other types of potential parallelism in industrial models. This will become critical with the new challenges of simulation as multi-physical simulations and internet of things.