Optimistic Parallelisation of SystemC

Samuel Jones

M2R Placement Report

2011

e-mail:
jones@imag.fr

Tutors:
Matthieu Moy - Claire Maiza

Unité Mixte de Recherche 5104 CNRS - Grenoble-INP - UJF
Centre Equation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
http://www-verimag.imag.fr

Optimistic Parallelisation of SystemC

Samuel Jones

Université Joseph Fourier: MoSiG DEMIPS

2011

Abstract

Systems-on-chip (SoCs) are becoming more complex and more widespread. Virtual prototyping tools are critical to the rapid development of embedded software. SystemC is the industry standard for simulation of SoCs today; however, its performance is becoming an issue. We describe an experiment in parallelising SystemC for SMP machines by running multiple schedulers each responsible for a subset of the available SystemC processes. Each scheduler has its own local time and does not synchronise automatically with the others. We provide an interface for specifying coarse-grain and fine-grain timing constraints to enable the programmer to write correct models in an intuitive way. Our work identifies the problems that occur when using fine-grain timing constraints and evaluates some solutions. We show that it is possible to obtain good speed-up even for relatively complex models.

Résumé

Les systèmes sur puce deviennent de plus en plus élaborés et répandus. Le développement de logiciels embarqués nécessite des outils de prototypage virtuel. SystemC est la norme industrielle dans le domaine de la simulation de systèmes sur puce modernes, mais sa performance devient insuffisante. Nous présentons une tentative de parallélisation de SystemC utilisant plusieurs ordonnanceurs indépendants, chacun responsable d'un sous-ensemble des processus SystemC disponibles. Chaque ordonnanceur dispose d'une horloge locale et ne se synchronise pas de manière automatique avec les autres ordonnanceurs. Nous fournissons une interface permettant la spécification de contraintes à gros grain et à grain fin, afin de permettre au programmeur d'écrire facilement des modèles corrects. Notre travail identifie les problèmes qui surviennent lors de l'utilisation de contraintes temporelles à grain fin et évalue quelques solutions. Nous démontrons qu'il est possible d'obtenir une accélération satisfaisante même pour des modèles assez complexes.

Keywords: SystemC, parallelisation, optimistic, quantum

Mots Clés: SystemC, parallélisation, optimiste, quantum

Tutors: Matthieu Moy - Claire Maiza

Acknowledgements

I'd like to take this opportunity to thank Matthieu Moy and Claire Maiza, my project supervisors, who dedicated their time and energy to guiding me through this project. They were always available to explain and advise, and very patiently went through things as many times as was necessary for me to understand. I'd also like to thank Hélène, my wife, for her patience and support throughout my Masters degree.

Contents

List of Figures

Chapter 1

Introduction

Systems-on-chip (SoCs) are omnipresent in the modern world. These are hardware components that contain one or several processors, memories, external interfaces, and everything necessary for a complete system on one integrated circuit. Found in mobile phones, cars, aeroplanes, DVD players, and all kinds of electrical appliances, these devices are becoming more and more powerful and complex.

When manufacturing SoCs, it used to be the case that, in order for developers to write software to run on a chip, there had to be a physical version of the chip available. This meant that the development of software for a SoC could not begin until a hardware prototype came off the production line. In order to accelerate the development of SoCs, manufacturers began to use software simulations of the chip's behaviour to enable software writers to develop code while the hardware design was still being specified. Such simulations are known as *virtual prototypes*.

The industry-standard simulation engine in use today is *SystemC*. SystemC is a set of `C++` classes and macros which provide a simulation kernel and a language for describing hardware elements. SystemC has an entirely sequential simulation kernel: simulation threads are executed one after another using co-routine semantics.

As the complexity of SoCs increases, so does the complexity of the simulation. Unfortunately, typical modern processors are no longer increasing in speed, but are increasing in parallelism instead. Problems with heat dispersion caused by increasing processor clock speed have led to the approach where several processor cores are placed on a single chip. SystemC, being sequential, is unable to harness this increase in computing power, and is therefore limited in the types of simulations it can execute in a reasonable period of time. This is particularly frustrating given that the simulated environment is typically highly parallel. As SoCs become more and more complicated, it is becoming less and less feasible to simulate them using SystemC.

To avoid this situation, it is necessary to develop a parallel simulation kernel for SystemC, enabling it to use the extra computing power provided by multi-core machines. A naïve approach to parallelisation involves sharing out the concurrently eligible processes over machine cores and executing them in parallel. While this method shows good results for certain types of models, it is ineffective for other types.

When parallelising SystemC, there are three major difficulties to overcome:

1. SystemC uses co-operative multi-tasking semantics. This means that each running process (representing some hardware component) can assume that it is the only running

process, and accesses to shared resources do not need to be protected. In a parallel kernel, this assumption will not hold.

2. For a certain class of SystemC models (see Section 2.4.1), the scheduler typically has only one item that it can execute at a time. In order to parallelise this class of models, it is necessary to run processes that normally would not yet be eligible. Avoiding the temporal inconsistencies this causes is difficult.

3. SystemC provides a language for modeling very detailed, cycle-accurate system descriptions, and for more abstract, functional models. It is difficult to write a general purpose kernel that will perform well for all types of simulation.

We present an *optimistic* parallelisation approach, meaning that we allow simulation events to occur out of order. We provide two types of tools designed to enable the user to express constraints on event ordering, and therefore maintain simulation correctness. Firstly, we use a *global quantum*, an integer value which fixes an upper limit on the temporal separation between any two schedulers in our simulation. Secondly, we introduce a set of timing specifiers local to each TLM transaction, which allow the programmer to assert that certain temporal properties should hold between the two schedulers before or after the transaction is executed.

We apply our approach to typical communication idioms and evaluate the strengths and weaknesses of our interface. We also provide some performance evaluations, giving an idea of the potential gain of our work.

We give an overview of how SoCs are modeled in chapter 2, followed by a description of the SystemC simulation engine. In chapter 3 we discuss work that has been done up until now on SystemC parallelisation, before discussing our approach in some detail in chapter 4. In chapter 5 we present performance results for our kernel and we draw some overall conclusions in chapter 6.

Chapter 2

SystemC/TLM

In this chapter we will give a brief overview of approaches to modeling Systems-on-Chip before going into relevant details about SystemC.

2.1 Modeling Systems-on-Chip

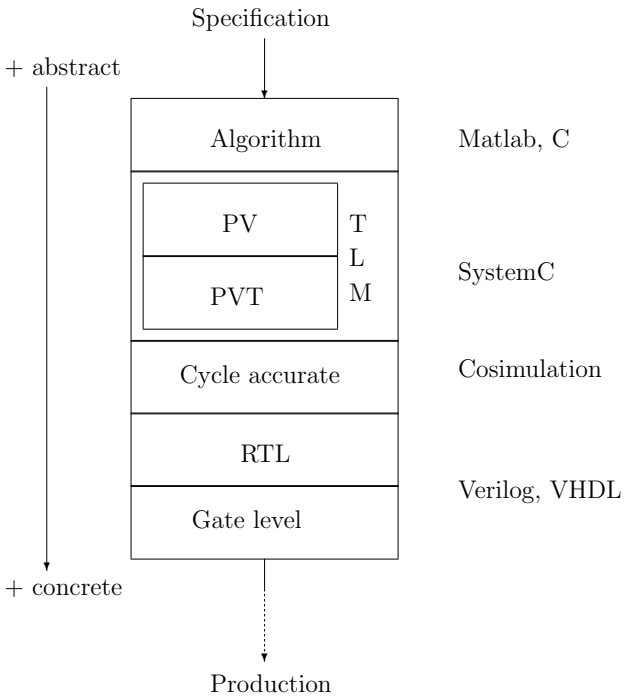


Figure 2.1: The different levels of abstraction, reproduced with permission from [4]

When modeling any system it is necessary to decide how much detail should be included about the way the underlying system works. The principal levels of abstraction used when designing SoCs are shown in Figure 2.1. When modeling at the algorithm level, no hardware implementation detail is maintained and only the properties of the algorithm used are studied. The next most concrete level is Transaction-Level Modeling, an abstract level, modeling communications between components as simple transactions, which we discuss in Section 2.1.2.

Following TLM are cycle-accurate models. These are models where the behaviour of components is modeled at the level of clock cycles. Register Transfer Level (RTL) models are the classical way of modeling systems; we discuss them in the following section. Finally there is the logic gate level of abstraction, which contains all the information necessary to synthesise the hardware.

2.1.1 RTL

The classical approach to designing and modeling SoCs is to describe the system using a Register Transfer Level (RTL) description. This relatively low level of abstraction describes the system in terms of signals flowing between hardware registers and logical operations performed on the signals. Hardware Description Languages (HDL) such as VHDL allow a designer to describe a system at the RTL. Furthermore, such descriptions can be automatically converted to gate-level descriptions using logic synthesis tools. Since RTL descriptions describe how every element behaves at every cycle, they provide a highly accurate model of the system.

As SoCs become ever more complex, using the RTL approach for simulation becomes less and less practical. The sheer number of elements that have to be simulated at every clock cycle means that simulations take a long time to execute, even on modern hardware. For example, for one description of an MPEG video codec, simulating the encode and decode operations of a single frame takes roughly one hour [4].

2.1.2 TLM

Transaction-Level Modeling (TLM) is a more abstract modeling approach. Systems are modeled in terms of transactions between concurrent components connected by channels. Components are written in some high-level programming language, and details of the hardware implementation of behaviour are forgotten. Communications are modeled as operations occuring over channels that connect components. The advantage of this approach is that by implementing only what is necessary to provide a view of the hardware sufficient for the software, considerable gains in execution time can be made. Depending on how much timing information is included in the model, models are termed as either *loosely-timed* or *approximately-timed*. Broadly speaking, approximately-timed models include communication protocol timing details, whereas loosely-timed models treat communications as a simple function call and use coarse grain timing annotations.

Furthermore, the TLM model can serve as a reference implementation with regard to the RTL description. This is useful because the TLM model may well be developed first, then subsequently refined and made more accurate, leading towards an RTL model.

SystemC supports TLM by means of the SystemC/TLM standard, released by the Open SystemC Initiative (OSCI) in 2005. It defines a series of templates, built on top of SystemC primitives, whose goal is to enable interoperability between transaction-level modelers.

2.2 SystemC/TLM

SystemC is a set of libraries and templates written in the `C++` programming language. It provides a means of describing systems both at RTL and transaction-level, and a simulation kernel. The first draft version of SystemC was published in 1999 and it underwent IEEE

standardisation in 2005 [2]. SystemC is defined by the Open SystemC Initiative (OSCI), a standardising group created in 2000.

A SystemC simulation is made up of two parts: a structural part and a behavioural part. The structural part describes the chip components and their interconnections; the behavioural part describes the actions executed by components. A component in SystemC is a class inheriting from the `sc_module` class. The class contains the component's internal state, its ports (for communication with other modules), and processes. Processes implement the behavioural part of the simulation.

The SystemC kernel builds and executes the model described by the user. It places each process in a separate thread, and it contains a scheduler that decides when each thread should run.

Abstractions designed to support TLM were introduced into SystemC by the TLM standard, currently at version 2.0 [9]. These abstractions are intended to allow users to perform TLM modeling in a standardised way and thereby maintain interoperability between models produced by different users. SystemC/TLM enables the modeling of generic transactions over a memory-mapped bus. The TLM standard defines sockets, which connect distinct modules to the bus, as well as laying down rules about the contents of messages sent over connections established by sockets.

2.3 Processes in SystemC

SystemC provides two key ways of modeling processes: threads and methods.

```
SC_MODULE(example){   //declare SystemC module called 'example'

  SC_CTOR(example){   //declare its constructor
    SC_THREAD(go);    //declare the function 'go' to be a thread
  }

void go(){
  while(!not_done){
    do_something();
    wait(20, SC_MS); //yield to SysC kernel, return 20ms later
  }
}
```

Figure 2.2: A SystemC thread

2.3.1 SC_THREAD

Threads are `C++` functions. They are invoked only once, and run from beginning to end, pausing at specific points in order to yield control of the processor to the simulation kernel. Threads can call the function `wait` to suspend their own execution for a period of simulation time (`wait(time)`), or until a specific event occurs (`wait(event)`). A common paradigm is to place code in an infinite loop which calls `wait(time)` once per iteration. This represents a hardware component performing some action at regular intervals, as shown in Figure 2.2.

```
SC_MODULE(example){                  //declare SystemC module named 'example'
  sc_in<bool> input_channel;         //declare a simple boolean input channel

  SC_CTOR(example){                  //declare its constructor
    SC_METHOD(interrupt);            //declare function 'interrupt' as method
    sensitive << input_channel;      //declare the method to be sensitive
  }                                  //  to changes on the input channel

  void interrupt(){
    std::cout << ``interrupted!'' << std::endl;
  }
}
```

Figure 2.3: A SystemC method

2.3.2 SC_METHOD

Methods are similar to function invocations: they define services that a component can perform when solicited by another component. Methods run when events to which they are sensitive occur. These events can be the edge of a clock, a change of value of some arbitrary signal, or user-defined events. When an event occurs, the SystemC scheduler runs any methods that are sensitive to this event.

Methods cannot call `wait()` and do not yield the processor. An example method is shown in Figure 2.3.

2.3.3 Co-operative Semantics

The SystemC scheduler elects threads in some reproducible order. The thread that is elected runs until either it calls `wait` or it terminates. During its execution, a thread cannot be preempted (interrupted by another SystemC thread). This means that each thread can safely assume that it is the only thread being executed, and therefore accesses to shared resources do not need to be protected.

Moreover, if a thread enters an infinite loop that does not call `wait()`, the thread will never give up the processor and the entire simulation will be frozen.

2.4 Time in SystemC

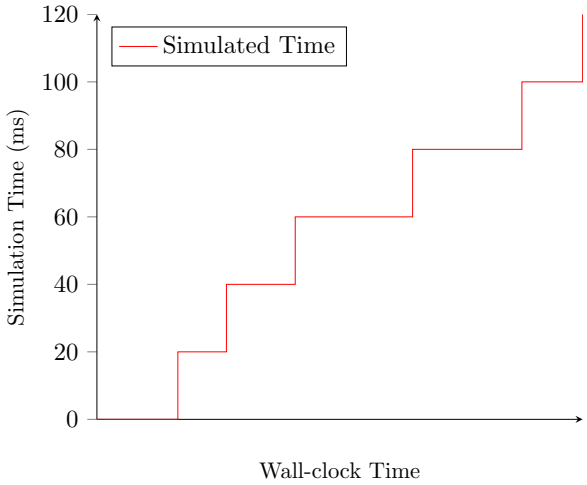
SystemC allows the modeling of systems both at the cycle-accurate level and at the transaction level. Cycle-accurate, or strictly-timed, simulations model the behaviour of each component at each clock tick. This is generally done using clocked threads and the SystemC communication primitives such as `sc_signal`. This involves a high level of detail but provides a very accurate simulation model, which can be used for performance analysis.

Transaction-level models, however, abstract out precise timing details and represent a system as independent components communicating via transactions on a central bus. They use time in a more complicated way. This means that SystemC has to contain the notion of *simulation time*. Simulation time represents the progress of time within the simulation, which has no relation to wall-clock (real-world) time. That is to say, the simulation of 1 second could

```
void MPEG::encode(){
  wait(20, SC_MS);
  //do the encoding
  ...
  return;
}

void Controller::control(){
  while(1){
    encode();
  }
}
```

(a) An action that consumes time



(b) Time moves forward in leaps and bounds

Figure 2.4: The passing of SystemC time

take 30 wall-clock seconds if the model is complex, or only 50 wall-clock milliseconds if the model is simple. Therefore, a notion of the passing of time within the simulated environment is maintained. For this purpose, a simple scalar value can be used that counts simulated nanoseconds (or smaller) since the beginning of the simulation.

One consequence of the semantics of SystemC is that it is impossible to state that an action occurs during a certain period of time. It is impossible to say that an action takes x nanoseconds, because the advancing of time is an atomic action (it is a simple integer add operation). In order to give the appearance of an action taking time, one first advances time by use of the `wait()` SystemC call (see figure 2.4a), and then performs the action. This feature is key: *operations do not occur during a certain period of simulation time, they happen instantaneously with regard to the simulation time.*

One might naïvely expect that SystemC time progress linearly with regard to wall-clock time; however, SystemC time tends to advance in "leaps and bounds". This is illustrated in figure 2.4b, with the example of a controller that continually triggers an encoding every time the previous one finishes. The example code is not representative of the way SystemC components communicate, it is merely a simplified example to demonstrate the way time passes in SystemC. One sees that, for SystemC, there is no time between successive instants. If the simulation starts at time 0, and the next event occurs at time 20ms, then there is no time instant at 10ms - the intermediary time simply does not exist.

2.4.1 Transaction-Level Modeling

In a transaction-level model, a component such as an MPEG encoder performing an encode operation might be represented as shown in Figure 2.4a. The MPEG encoder is treated as a black box: provided with data, it performs the encoding in precisely 20 milliseconds. Rather than littering the encoding operation with calls to `wait(1, SC_MS)` every time a millisecond's worth of computation is performed, one waits all 20 milliseconds first and then performs the operation.

A value such as 20 milliseconds is used here because it is not known precisely how long

```
void do_trans(){
  payload trans; //the object that expresses the transaction
  trans.set_command(TLM_WRITE_COMMAND);
  trans.set_addr(MPEG_ENCODE);
  trans.set_data(1);

  sc_time(40, SC_MS) time;

  out_socket.b_transport(trans, time);
}
```

Figure 2.5: The loosely-timed coding style

the operation will take. It is also possible to add a degree of uncertainty by using a bounded random value, such as between 5 and 35 milliseconds. These values can be refined as the precise hardware performance details become available.

The SystemC/TLM LRM describes the three principal *coding styles* used when modeling systems at the transaction level: *untimed*, *loosely-timed*, and *approximately-timed*. Coding styles are not levels of abstraction but rather different approaches within the transaction level of abstraction. They concern the way that the TLM interface is used and are not precisely defined by the standard. Furthermore, although in theory only one coding style is used in a given model, in practice a mix of styles can be observed within the same model.

Untimed

Untimed simulations have no notion of time or a clock. Communication is performed entirely using event notifications to communicate between processes. Untimed simulations may also be referred to as *Programmer's View* (PV) simulations, since a software programmer is typically unconcerned with the timing details of the hardware.

Loosely-timed

The TLM standard provides two types of transaction (communication) - blocking and non-blocking. Blocking transactions execute the entire transaction and return when it is complete. Non-blocking transactions break the transaction into stages and several function calls are necessary to complete a single transaction. This allows for more detailed modeling.

Loosely-timed simulation transactions are modeled using blocking interfaces, and coarse-grain timings are used with transactions. For example, in a loosely-timed simulation, a memory write may be modeled as a simple function call which performs the write and returns, and takes x microseconds in total. An example showing the loosely-timed coding style can be found in Figure 2.5.

Approximately-timed

In an approximately-timed simulation transactions are modeled using non-blocking interfaces and are separated into several phases, each of which has timings. In an approximately timed simulation, a memory write could be modeled as several phases: an opening handshake (x μs), followed by the data transfer over the bus (y μs), followed by a closing handshake (z μs).

```
void start_trans(){
  payload trans; //the object that expresses the transaction
  trans.set_command(TLM_WRITE_COMMAND);
  trans.set_addr(MPEG_ENCODE);
  trans.set_data(1);

  sc_time(10, SC_MS) time;

  //begin a handshake. The MPEG Encoder will reply by
  //making a call back to us, which will be handled by
  //continue_trans()
  out_socket.nb_transport_fw(trans, time);
}
void continue_trans(payload trans){
  command = trans.get_command(); //inspect the incoming payload
  ...
  sc_time(20, SC_MS) time;

  //continue the transaction.
  out_socket.nb_transport_fw(trans, time);
}
```

Figure 2.6: The approximately-timed coding style

The name *Programmer's View with Time* (PVT) includes both loosely and approximately-timed coding styles. Figure 2.6 shows an example of the approximately-timed coding style.

2.4.2 TLM Temporal Decoupling

TLM Temporal Decoupling, or *quantum keeping*, was introduced into SystemC by the TLM 2.0 standard. It is advocated as a way of reducing the number context switches incurred. It is a technique which allows each thread to run in its own "time warp", that is, to run ahead of SystemC global time up to a certain amount. Instead of synchronising every time they wish to communicate, threads can choose to advance their local time ahead of global simulation time. The maximum gap allowed between the global time and a thread's local time is known as the global quantum. Because the approximately-timed coding style requires timing accuracy, it is less suitable for use with temporal decoupling than the loosely-timed style [9]. The loosely-timed coding style, by definition, does not use time for synchronisation between components but events, and therefore can make use of temporal decoupling.

Temporal decoupling reduces the number of context switches made between threads and therefore decreases the execution time. Clearly, this approach can have its toll on correctness. The responsibility is left to the system designer to know whether it is dangerous to advance ahead of simulation time. This approach gives the user a degree of control. By setting a large *quantum*, or distance that a thread is allowed to advance ahead of global time, the user gains speed at the cost of accuracy. The TLM designers provide certain guidelines about how to choose the quantum to minimise inaccuracies. However, in this approach, any thread that does not use the technique is likely to become a bottleneck.

We draw on the temporal decoupling approach for our work.

2.5 Communication in SystemC

Communications in SystemC are provided by user-defined *events*, as well as a combination of *ports*, *exports*, and *channels*. These mechanisms exist to provide encapsulation of components and to enable generic components. A port defines the services that are required by a component from the outside. An export defines the services that a component provides. Ports and exports are connected, or *bound*, to channels, which connect the two. Ports and exports can be bound indirectly, that is, via other ports and exports.

In the TLM standard, a higher level approach is used. The system is modeled as components communicating over a common memory-mapped bus. The TLM standard provides interfaces representing *initiator* and *target sockets*. Initiator sockets are analagous to ports, and target sockets to exports, while the common bus plays the rôle of the channel. Components call the transport functions of the initiator socket, which must be bound to a target socket on the bus. The target socket on the bus should look up the memory address of the operation, and route the request onward to the bus's initiator socket bound to the appropriate target.

When SystemC processes write values on channels, any process sensitive to a change of the channel value will be marked as runnable. However, the sensitive processes are awoken an infinitesimal amount of time afterwards, known as a delta-cycle.

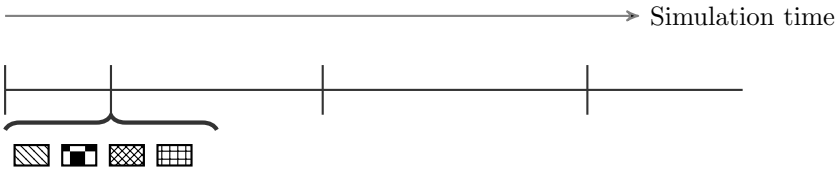


Figure 2.7: A δ-cycle inside a time instant

2.5.1 Delta-cycles

When several processes are eligible at the same time instant, the execution order is defined by the implementation. However, it may be the case that some processes depend on the outputs of others, and that therefore some kind of order is implied. In Figure 2.7, the vertical lines represent instants in simulation time. The brace represents a zoom on a given time instant, showing the eligible processes as shaded boxes. These processes are executed in some order. So as to ensure that the order of execution does not determine the output, SystemC uses the concept of a delta-cycle (δ-cycle). Conceptually, the available threads within a given time instant are run repeatedly until the outputs stabilise, in a manner analogous to the computation of a fixed point. A δ-cycle is one round of execution inside a time instant. In actual fact, processes are not executed repeatedly, but the mechanism is used whereby processes are *sensitive* to their input channels, and when the value of their input channels changes, they are made runnable. Using this approach, one can maintain determinism and coherency when modeling parallel events sequentially.

```
SC_MODULE(encode){                          //declare a SystemC module
  sc_in<bool> m_in;                         //declare an input signal
  sc_event worker_event;

  SC_CTOR(encode){                          //declare the module's constructor
    SC_METHOD(int);                         //the function ''int'' is a method
    sensitive << m_in.pos_edge();           //and it is sensitive to m_in
    SC_THREAD(worker);                      //the function ''worker'' is a thread
  }

  void int(){
    worker_event.notify();
  }

  void worker(){
    while(true){
      wait(worker_event);
      work();
    }
  }


}
```

Figure 2.8: An event used to wake up a worker thread

2.5.2 Events

SystemC contains user-defined events. Threads can halt their execution until events are signalled by another process, by *wait*ing on events. Threads or methods which awake sleeping threads are said to *notify* an event. Events in SystemC are non-persistent, which is to say that if an event is notified and no thread is waiting on it, then the notification is lost. Events are often used inside a module to awaken a worker thread, as shown in Figure 2.8.

2.5.3 TLM Transport

In SystemC/TLM, all components are connected to a memory-mapped bus. A memory-mapped bus maps virtual memory address ranges to the connected components, and then routes transactions by the virtual memory address used by the components when communicating. Components are connected *via* sockets, which are directional (we refer to *initiator* or *target* sockets). Figure 2.9 shows of how sockets connect components to a bus, including a memory with an input/output connection to the bus, an input port and an output port.

The transport interface defines two types of communicating, blocking and non-blocking, as we have seen earlier. These functions accept as an argument an object of the generic payload introduced by the TLM standard. The generic payload is a class that is intended to represent the transaction being carried out on the bus and its use is recommended for interoperability. The argument t can be used for temporal decoupling.

The implementation of the bus is left to individual vendors. The TLM standard simply specifies what is necessary to achieve interoperability between vendors.

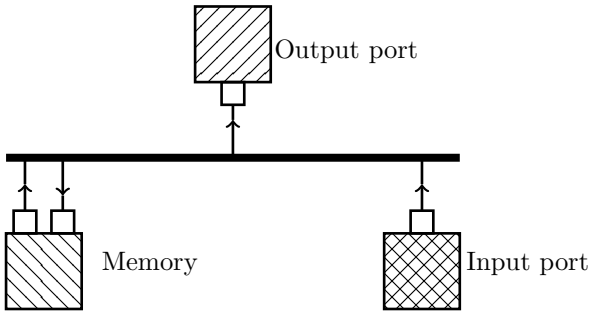


Figure 2.9: A TLM model

The TLM transport interfaces are the glue that holds distinct components together. When component A calls `b_transport` on a socket, the call is routed over the bus to a function on the appropriate component B. This function is executed by the thread that was running inside component A. This means that an entire transaction can be performed without a context switch. This also means that the TLM transport mediates between two pieces of arbitrary, user-written code. Threads from component A executing code in component B is not a problem in classical SystemC/TLM as the co-operative semantics mean that the threads are executed in turn and there can be no race conditions.

Where they are not relevant to the example, we leave out the TLM syntax in our code listings, as it is rather verbose. So, instead of declaring a function as returning type `tlm::tlm_response_status` and setting properties on the generic payload, we simply declare the function as returning type `status`.

2.5.4 Synchronisation

There are two types of inter-process synchronisation in SystemC/TLM: implicit or *time-based synchronisation*, and explicit or *event-based synchronisation*. By time-based synchronisation it is meant that it is known how long a remote component will take to perform a given operation, and so one simply requests that the component perform the operation, waits an appropriate amount of time, and then collects the result. Figure 2.10 shows an example of time-based synchronisation: the worker updates the value x every 10 ms, starting from time 0, and the reader samples the value every 10 ms starting at time 5.

Time-based synchronisation can be used in some very detailed models where precise timing information and the component network topology and characteristics are known. It is not generally used in more abstract models where all timing values are approximate and describe the general progression of the simulation rather than the precise state of components. In these models, event-based synchronisation is typically used.

By event-based synchronisation it is meant that components communicate using SystemC events and the `wait` and `notify` calls. Event-based synchronisation is illustrated in Figure 2.11. It is worth noting that the event in the worker module is made persistent by the addition of a Boolean variable, but not the event in the reader module. As will be explained in the following section, the order of execution of threads in a delta-cycle is implementation-dependent. Since both the reader and the worker will start with time 0, one will be elected by the scheduler to run first. If the reader runs first then the `notify` will be lost, as the worker

```
SC_MODULE(reader){
  //bound to worker
  init_socket bus_socket;

  SC_CTOR(reader){
    SC_THREAD(read_data);
  }

  void read_data(){
    wait(5, SC_MS);
    while(1){
      int y = bus_socket.read();
      wait(10, SC_MS);
    }
  }
}
```

```
SC_MODULE(worker){
  target_socket bus_socket;
  int x = 0;

  SC_CTOR(worker){
    SC_THREAD(work);
  }

  void work(){
    while(1){
      x++;
      wait(10, SC_MS);
    }
  }

  int read(){ return x; }
}
```

Figure 2.10: Time-based synchronisation

```
SC_MODULE(reader){
  //bound to worker::in_socket
  init_socket out_socket;
  target_socket in_socket;

  SC_CTOR(reader){
    SC_THREAD(read_worker);
  }

  int read_worker(){
    out_socket.write(START, true);
    wait(my_event);
    return bus_socket.read();
  }

  void write(addr, val){
    if(addr == DONE && val)
        my_event.notify();
  }
}
```

```
SC_MODULE(worker){
  //bound to reader::in_socket
  init_socket out_socket;
  target_socket in_socket;

  SC_CTOR(worker) : x(0) {
    SC_THREAD(work);
  }

  void work(){
    while(1){
      while(!start_work)
        wait(my_event);
      start_work = false;
      x++;
      out_socket.write(DONE, true);
    }
  }

  int read(){ return x; }

  void write(addr, val){
    if(addr == START && val){
        start_work = true;
        my_event.notify();
    }
  }
}
```

Figure 2.11: Event-based synchronisation

is not yet waiting on the event. This is why a persistent event is necessary. However, the reader's event can remain transient, as we can be sure that the reader will always be waiting on the event when the writer notifies it.

Approximately-timed models may use some time-based synchronisation. Loosely-timed models, by their very nature, use little time-based synchronisation. We say little rather than none because some assumptions are retained. For example, it is reasonable to expect that an event A occurring a *long time* before event B will have completed before B occurs. What is a *long time*? This depends on the timing detail of the model.

2.6 The SystemC Scheduler

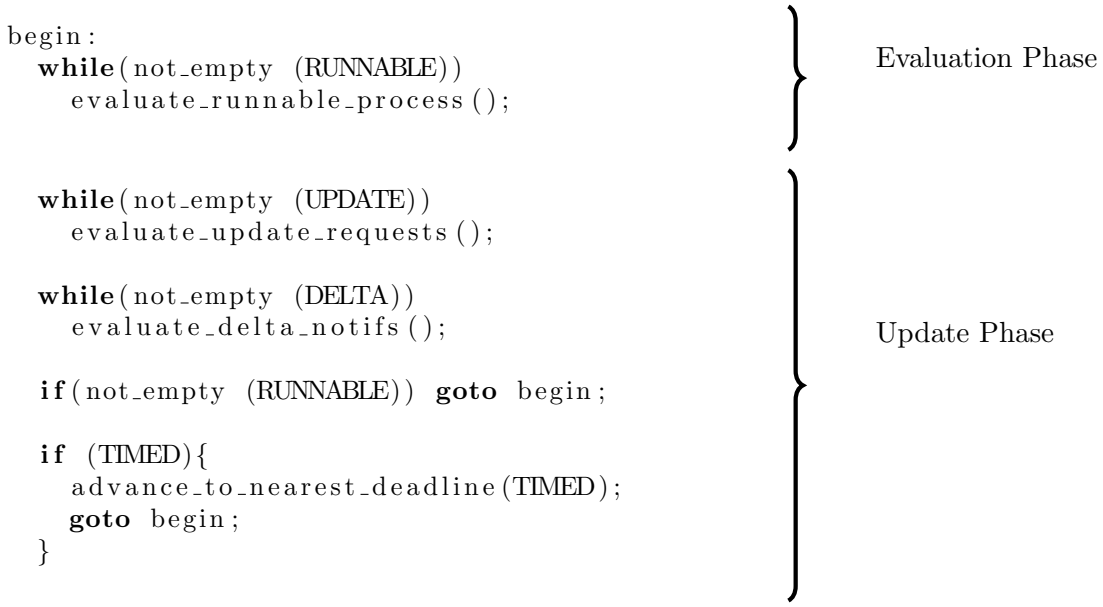


Figure 2.12: The SystemC scheduling algorithm

The SystemC scheduler elects processes to be executed, one at a time, from those which are in its runnable processes set. Its mode of operation can be expressed most easily in terms of four sets:

1. Runnable processes - processes that are ready to run at this time instant.
2. Update requests - channels that have been written to by processes and must have their values updated.
3. Delta notifications - methods that are sensitive to channel values and will be awoken after updates are performed. It is also possible for processes to issue delta notifications directly.
4. Timed notifications - processes that are waiting for time to pass.

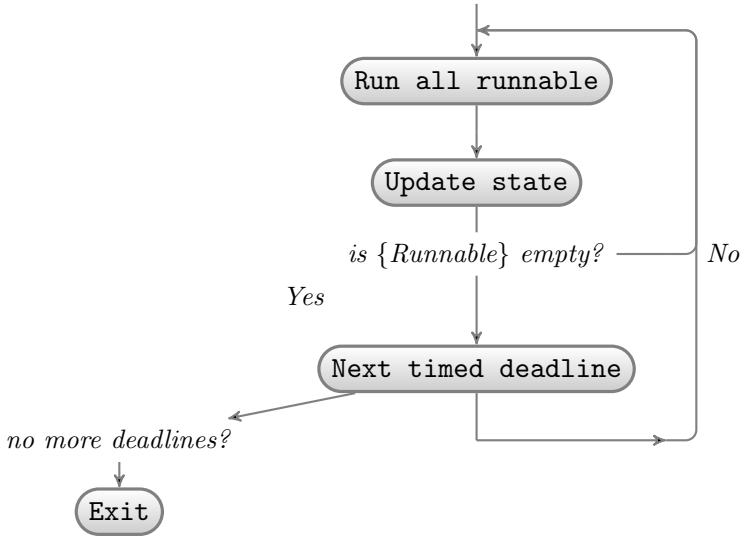


Figure 2.13: The SystemC scheduler

Figure 2.12 shows the SystemC scheduling algorithm in pseudo-code. Figure 2.13 shows the algorithm diagrammatically. First, the scheduler evaluates (executes) the processes that are in the runnable processes set. Each of these processes may generate update requests, immediate notifications, delta notifications and timed notifications. Immediate notifications allow a process to place another process in the runnable set immediately. Timed notifications will fire at a given deadline in the future, activating one or several processes. When there are no more runnable processes, the scheduler evaluates the update requests and the delta notifications. If the runnable set is no longer empty, it returns to the beginning of the algorithm. Otherwise, it advances time to the next deadline, as set by a timed notification and evaluates the runnable processes. When there are no more deadlines, the simulation is over.

The algorithm can also be thought of in terms of two phases: the evaluation phase and the update phase. The evaluation phase consists in evaluating runnable processes. This can be thought of as the useful work of the simulator. The update phase is the remaining operations: updating channel values, notifying events and awakening processes. The order in which runnable processes are evaluated within a δ-cycle is not defined by the standard, leaving the choice to implementations [2]. Since the system designer cannot rely on a specific process execution order when designing the system, he must make any dependencies explicit in the model. This is essential to parallel SystemC kernels, since an easy starting point for parallelisation is to execute all currently runnable processes in parallel. If a specific execution order were implied in the standard, such an approach would instantly violate the standard semantics.

We have given an outline of the key properties of SystemC and its Transaction-level Modeling layer that are relevant to our work. We have also introduced the naïve approach to SystemC parallelisation: at each time instant, take all runnable processes and execute them in parallel. We will now introduce the related work in the field and see the difficulties inherent to such an approach.

Chapter 3

Related Work

Parallelisation techniques aim to increase the execution speed of the SystemC/TLM simulation by running some of the simulation processes in parallel. If two threads are run in parallel then there are two risks in play. Firstly, the risk that running the two threads in parallel violates the order in which threads should be run, as specified by the SystemC standard. Secondly, the risk that running the two threads in parallel introduces race conditions, which can cause erratic behaviour.

There are two categories of approaches to parallelising languages based on the Discrete Event Simulation algorithm: conservative, and optimistic. Conservative approaches enforce the same global time over all copies of the simulator at all times. For SystemC, this means that the simulation is guaranteed not to violate the order in which threads should be run with respect to the standard. However, this is a source of difficulty as it imposes a large amount of communication and synchronisation.

Optimistic approaches have weaker synchronisation behaviour than conservative ones. They have to either implement some kind of rollback mechanism when violations are detected, provide some means of foreseeing and avoiding violations, or provide weaker guarantees about the correctness of the simulation with respect to the standard. Optimistic approaches have been used to parallelise some Discrete Event Simulators such as VHDL and Verilog [3], but we are not aware of the approach being applied to SystemC.

Furthermore, the work that has been done on SystemC has been focused on cycle-accurate models. This is to be expected, as these more detailed simulations were the first to require speeding up. However, now even transaction-level models are being found to be too slow.

We first review the work that has been done on parallelising SystemC and comparable languages and then discuss the problems that are left unadressed by existing work.

3.1 Conservative Approaches

Most work on conservative approaches has modified the SystemC kernel, and has been performed with distributed execution in mind. However, some work has also been done on parallelising SystemC for SMP workstations [15] and on introducing a thin layer above SystemC [8].

One of the first attempts to do parallelise SystemC was the work of Chopard *et al.* [3]. They place a copy of the SystemC scheduler on distributed machines and synchronise the schedulers at each update phase. A master node collects information from each scheduler

to determine when to advance the global simulation time. They require the developer to manually partition SystemC processes over the participating nodes. They achieve good results with a realistic system, their limiting factor being load balancing. However, their approach assumes a cycle-accurate simulation.

An extension to the same work [5], presents some optimisations that they applied to their approach, removing some unnecessary synchronisation between schedulers. They state that SystemC semantics *"require a high level of synchronization which can dramatically affect the performance"*. They partially decentralise their original algorithm and allow nodes to compute the next timed deadline locally. These optimisations enable better performance, but do not represent a major change of direction.

Another similar approach, this time aimed at SMP workstations rather than clusters, was presented by Schumacher *et al.* [15]. They also only parallelise the evaluation phase of the scheduler, using a barrier to synchronise threads. They achieve good speed-up (even super-linear, due to the extra cache memory available on extra cores), but they acknowledge that more needs to be done to deal with the TLM abstraction.

All of the approaches require some partitioning of the chip model into groups which will be placed on separate cores. Ezudheen *et al.* [13] look closely at the question of how to choose these groups. They implement a parallel evaluation phase and a sequential update phase, but they then investigate a work-sharing implementation, a work stealing implementation, and they implement an interface allowing the user to group related processes together, with the intention of improving cache usage. They analyse their implementation by varying the number of processes and the amount of computation that they perform. For large amounts of computation and a high number of modules they achieve good speedup. They conclude that the manual grouping of processes is the most efficient, though the user effort involved in grouping the modules together in order to get the best performance may be considerable.

Mello *et al.* introduce a new programming style [12], which they call TLM-DT (for "Distributed Time"), which they contrast with the approximately-timed and loosely-timed TLM programming styles. They present a parallel simulation kernel that uses the quantum-keeping technique of the SystemC TLM standard [9] to reduce context switches occurring in parallel simulation, while remaining conservative. Every thread has its own local time, which it advances when it executes work, and when it receives messages. These messages are used to synchronise with other processes. They disallow certain SystemC synchronisation functions in order to simplify their implementation and they change the semantics of SystemC events, making them persistent. They show very promising results, but they only provide one example. Since they use the TLM time quantum, their work provides the user with the ability to trade speed against accuracy. They achieve good results, but their approach cannot be applied to existing systems without rewriting them. They remove timed and immediate notifications. Furthermore, they require communicating threads to be placed on the same CPU, which limits the applicability of their system significantly.

Finally, Huang *et al.* suggest an approach to SystemC parallelisation that does not involve modifying the kernel [8]. They introduce a SystemC Distribution Library, a layer that sits above SystemC and manages remote and local simulators. They make use of the fact that the OSCI SystemC reference implementation includes a function that executes only one delta cycle of the simulation at a time. Their management layer handles consistency and shared time among simulators and causes them to execute this function only when the system is in a consistent state. They provide state machines for the control logic of the master and slave nodes which enables the preservation of correctness.

3.2 Optimistic Approaches

Optimistic approaches have been investigated in the domain of distributed algorithms. There are typically two types of approach to the problem of temporal ordering violations: avoidance and recovery. Algorithms such as Schneider's [14] use a system similar to Lamport's clocks [11] to avoid carrying out any messages until it is certain that no earlier message can arrive. On the other hand, Jefferson's *virtual time* [10] approach for ordering events correctly relies on using roll-back techniques. He advocates using *anti-messages* to propagate roll-back information when an ordering violation occurs, and delaying operations that are impossible to roll back (such as I/O) until he can be certain that no earlier messages exist in the system.

Optimistic approaches have been applied to hardware description languages such as VHDL. WARPED [16] was an optimistic simulation kernel for VHDL using a roll-back technique to recover from temporal violations. However, a roll-back technique would be difficult to adapt to SystemC/TLM given that inter-thread communications can take the form of arbitrary function calls.

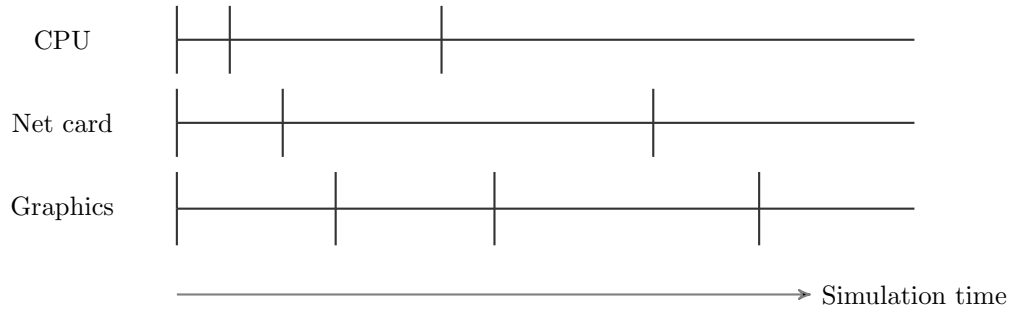
3.3 `jTLM` [7]

`jTLM` is not a SystemC kernel but a small, entirely separate parallel simulation kernel for TLM modeling. Presented by Funchal and Moy in [7], it includes a pre-emptive scheduling mode, where it hands off scheduling to the underlying operating system scheduler. It also includes a way of stating that an action occurs over a certain length of simulation time, in an attempt to parallelise loosely-timed models effectively (see section 3.4.1). Programmers writing models for `jTLM` have to be aware of locations in their code where mutual exclusion may be important and protect them appropriately. The authors wished to investigate what was possible in a simulation kernel enforcing different semantics to those of SystemC. `jTLM` is relevant to our work because it is also an attempt to deal with the problem of parallelising loosely-timed models. However, as it is completely incompatible with SystemC, it is unlikely to be used for industrial purposes.

3.4 Discussion

First of all, most of the research does not consider the potential race conditions introduced by parallel execution of threads. The most obvious form of this problem is accesses to arbitrary variables in shared memory, which are not part of the SystemC interface. This will be a problem for any kind of parallelisation technique. If the threads rely on shared variables outside of SystemC itself for any kind of communication, then parallel execution will introduce race conditions that could cause incorrect behaviour on rare occasions. However, it has been shown that avoiding these race conditions automatically requires fairly involved analysis techniques [4].

Secondly, given the existing wealth of SystemC models, an approach to parallelisation that would require a large modification effort is unlikely to be useful for the acceleration of existing models. It might, however, be useful for the development of new models. The ideal would be to have an entirely automatic approach to parallelisation that requires no changes to existing or new models. Failing that, an approach that allows the designer to parallelise



(a) Timeline showing when each process's deadlines fall

```
cpu::compute(){
  while(1) {
    crunch();
    wait(20, MS);
  }
}
```

```
net_card::work(){
  while(1){
    decode();
    wait(13, MS);
  }
}
```

```
graphics::display(){
  while(1){
    update();
    wait(23, MS);
  }
}
```

(b) 3 SystemC processes using the loosely-timed coding style

Figure 3.1: The disparate deadlines problem

only parts of the system at a time would be the next best thing, as it would enable the system designer to obtain a certain speedup without having to rewrite the entire system.

We also see that the majority of the work has been directed towards cycle-accurate models. This is natural, as cycle-accurate models contain more details about the platform and take longer to execute, and therefore are the first type of model that should be parallelised. Importantly, it has been shown that cycle-accurate simulations can be simulated with good results using conservative techniques [15] [3]. Ensuring an even load balance is typically the limiting factor in terms of speedup. This is because cycle-accurate simulations lend themselves well to a conservative approach; each thread is sensitive to a common clock and is therefore in the runnable queue on every cycle. The schedulers have to synchronise on every clock tick, but this is not necessarily expensive as the threads often represent homogeneous components or the scheduler loads are well-balanced. Furthermore, the cost of the synchronisation pales against the computation time if the threads do a meaningful amount of work. When dealing with cycle-accurate simulations, the conservative approach ensures correctness and its costs can be managed.

3.4.1 The Disparate Deadlines Problem

However, when dealing with loosely-timed simulations (see section 2.4.1), a difficult problem arises. Since each component uses a "loose" timing value, it is rare that two deadlines collide. That is to say, while in strictly-timed simulations each component is active on each clock tick, in loosely-timed simulations the CPU will be active every 67 ms, the network card controller every 22 ms, and so on, as shown in Figure 3.1. This means that at any given moment in simulation time, there is almost always only one runnable process. This problem fatally undermines the simplistic approach of simply dividing the available work among the cores: each core will be idle the majority of the time. This problem has not been addressed by the

existing research on SystemC.

The conservative parallelisation framework is unsuitable for loosely-timed models. In order to run processes in parallel, it is necessary to remove the notion of global time and bring us into the world of optimistic simulators. This introduces certain difficulties.

Chapter 4

An Optimistic Approach

Faced with the weaknesses of the conservative approach when applied to loosely-timed simulations, we decided to investigate a weakly synchronised approach. We now provide an overview of our approach and justify our design decisions.

```
1        P1:                          P2:
2            arithmetic();                arithmetic();
3            wait(10);                    wait(15);
4            access_cache();              arithmetic();
5            wait(50);                    wait(15);
6            arithmetic();                arithmetic();
7            wait(10);                    ...
8            interrupt P2;
```

Figure 4.1: Two Processors

The sequential SystemC kernel executes threads one-by-one, in a certain order. In this way, it applies a total ordering to a set which is typically only partially ordered. For example, consider the following model sketched in Figure 4.1. A sequential execution would perform lines 2 and 3 of P1, then lines 2 and 3 of P2, then lines 4 and 5 of P1, and so on. However, lines 2 and 3 of P2 do not depend on lines 2 and 3 of P1 and could be executed earlier or concurrently. In fact, the only ordering constraint given in this example is that P2 should receive the interrupt at time 70, assuming an instantaneous interrupt model. That is to say, the sequential execution order imposed by the standard SystemC kernel is only one of many legal (with respect to the SystemC semantics) orders. Furthermore, depending on the nature of the communications between the two processes, it may not even be necessary that P2 receive the interrupt at time 70 to conserve the behaviour of the program. Perhaps the program's behaviour will not change even if P2 receives the interrupt at some other time. We attempt to use these properties to enable parallelism without violating semantics.

4.1 Overview

We modified the SystemC kernel to enable several instances of the scheduler to run in parallel, each responsible for a certain subset of the program's processes. We do not enforce synchronisation after every delta cycle. This means that schedulers are not forced to wait for

one another at each delta cycle and can therefore run in their own local time warp. This is a similar principle to TLM's temporal decoupling (see Section 2.4.2), with the difference that we apply the warp to all threads managed by a scheduler, not individual threads.

We also modified an existing, simple, transport layer built on top of TLM. This transport layer is the TLM protocol used at the Ensimag engineering school [1] and is called `ensitlm`[6]. It provides some simple data types and encapsulates calls to the TLM interface. It includes an implementation of a memory-mapped bus which we also modified for our work developing the user interface to our kernel.

Clearly, if we allow groups of threads to advance in a non-uniform manner, we risk introducing time-related inconsistencies when threads communicate, such as events from the future affecting current events and events arriving from the past. We provide an interface for the user which allows him to specify constraints on the time gap between communicating components at specific program points, as well as providing an overall limitation. In this way, we attempt to make the most of information known by the user that may not be easy to infer automatically. Furthermore, we provide the property that threads assigned to the same scheduler will share a global clock and therefore interactions between these threads will still be coherent.

We said in our literature survey that there are two ways of dealing with the problem of temporal violations: avoidance and recovery. Our work would be classed as avoidance. We attempt to avoid timing violations, but we also recognise that some timing violations may not change the behaviour of the program in a significant manner; we are only interested in avoiding behaviour-changing violations. A recovery approach would not be possible on industrial platforms, which are often a patchwork in which the SystemC model is connected to other systems.

It is worth mentioning at this point that simulation correctness is not necessarily a Boolean value. Depending on the system modeled, some timing errors may be acceptable. A time difference between the video output and the MPEG decoder that results in a skipped frame could be an acceptable price to pay for an increase in simulation speed. However, time differences that cause the system to deviate wildly from its sequential behaviour may be unacceptable.

4.2 Thread Partitioning

We modified the SystemC standard OSCI kernel, separating out the scheduling logic from the rest of the kernel, in order to be able to run it in parallel. We then had to make sure that the remaining kernel code was thread-safe.

We added a macro to the SystemC API, `SC_AFFINITY`, which enables the programmer to specify logical groupings of threads. Threads in the same logical group will be executed within the same scheduler, with a coherent notion of time, and, if there are sufficient machine cores, on a separate core to all other groups. We then run, in parallel, as many SystemC schedulers as there are groups. The schedulers do not synchronise automatically, rather we provide ways for the programmer to describe the constraints that the schedulers must respect in order to maintain the desired degree of correctness.

The macro is applied to a SystemC module. Threads within a module all share the same affinity. An example of the usage of this macro is shown in Figure 4.2.

Although partitioning may seem like a difficult task, it is usually not very complicated.

```
SC_MODULE(cpu){
  SC_CTOR(cpu){
    SC_AFFINITY(1);
    SC_THREAD(compute);
     ...
  }

  void compute();
}
```

Figure 4.2: Usage of the `SC_AFFINITY` macro

A good partitioning will balance workload and minimise communications. A good heuristic is to divide the components up according to how they are divided on the chip, grouping together components with light workloads if there are insufficient cores. The component with the largest workload will become the limiting factor on speedup.

4.3 Shared-memory parallelism

We decided to use threads to parallelise the kernel, in particular the pthreads library. We chose the pthreads library because of its portability and standardised interface. The shared memory that threads provide us with enables us to preserve the existent SystemC coding style. Consider the case of communication via events, typically written in SystemC as shown in Figure 4.3. This programming style is familiar to SystemC developers and there is a great deal of existing code written in this way. In order to minimise the number of changes SystemC developers would have to make in order to use our kernel, it is important to preserve this style.

```
sc_event e;

P1:
  wait(e);

P2:
  work();
  e.notify();
```

Figure 4.3: Inter-process communication in SystemC via events

It has been said that, now that modern machines are often equipped with several processors and a hierarchy of main and cache memory, the idea of shared memory could be considered an illusion. After all, if two threads run on different processors, shared data will almost certainly be cached locally at each processor and will be kept up to date by the cache coherence protocol. Therefore the data is not really shared, but distributed. Furthermore, there may be problems of *false sharing*, when shared data is located on the same cacheline as unshared data. For example, in [15], the authors had to pay careful attention to data placement to avoid placing master and worker data on the same cachelines. These are ar-

guments in favour of using a distributed memory model with semantics that clearly describe the interactions between processes (such as the MPI standard), even on a single machine.

However, we believe that the use of a such a model would make SystemC harder for developers to use. Even if we managed to make the distributed nature of the memory transparent by means of a careful implementation, programs written assuming shared memory would almost certainly not have optimal performance. Performance-conscious developers would have to learn a new programming style.

Furthermore, the pthreads library has another advantage: that of portability. It is the POSIX standard for threads and implementations exist on most UNIX-like operating systems, and are often installed by default, being included in the `glibc` library. There is even an implementation of the standard for Windows. Furthermore, although it is not enabled by default, SystemC can be compiled to use pthreads, so the use of pthreads is in keeping with the existing multi-threading setup, as well as being portable. On the contrary, an implementation of MPI is rarely installed by default.

Since, in terms of performance, it seems there appears to be no compelling reason to use one model over the other, we decided to use the pthreads library because it is highly portable and fits well with the SystemC kernel.

4.4 Interface Requirements

Our interface should enable the user to address these two major threats to simulation correctness:

1. Temporal violations: Is it legal to execute an action that occurs on P1 at simulation time 40 *before* an action that occurs on P2 at simulation time 30? How do we know?

2. Race conditions: This problem has two facets.

   (a) As in other work, we assume accesses to shared variables which are not part of the SystemC kernel must be protected or irrelevant, or that the user has carefully placed the threads involved on the same scheduler (assigned them the same affinity).

   (b) Multi-stage operations such as read-modify-write sequences are not guaranteed to be atomic when using a parallel simulator. Consider Figure 4.4: Let us imagine that two processes want to increment d in this manner, and the two processes are run in parallel on different CPUs. In this example, even if individual accesses to the bus are serialized, by protecting the bus read and write calls with mutexes, we can still obtain a result impossible in standard SystemC, given the following sequence of executions:

      i. P1 reads d.

```
bus.read(0x1000, &d);
d = d + 1;
bus.write(0x1000, d);
```

Figure 4.4: The read-write-modify paradigm

ii. P2 reads d.
iii. Both processes increment d locally.
iv. P1 writes d.
v. P2 writes d.

The end result is that d has been incremented only once.

Our interface is made up of both global and local tools for addressing these issues. At the global level we provide a parameter which controls the maximum separation in time between any two schedulers. At the local level we provide a simple API allowing the programmer to specify certain constraints.

4.5 Global Control

Inspired by the quantum keeping approach used in TLM, we have implemented a *global quantum* parameter which can be used to control the time difference over all schedulers. For example, if the maximum distance is 30ms, no two schedulers will ever be separated by more than 30ms in time. This is not exactly the same as in TLM quantum keeping, where the distance is an offset from global time. We have no notion of global time; rather one can think of our quantum as a window that moves along the time line. It enables us to assert that at any point in time, no scheduler can be more than the global quantum ahead of or behind any other. The quantum prevents one scheduler with a lighter workload running away from the others, and can be used as a knob controlling the tradeoff between speed and accuracy.

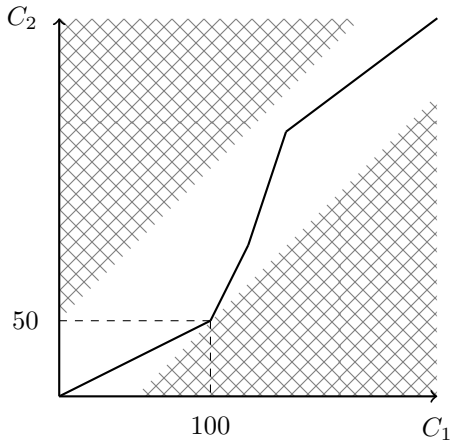


Figure 4.5: The effect of the global quantum on legal system states

Figure 4.5 illustrates this effect when we consider only two schedulers. Let us imagine that we have set the global quantum to 50. The clock of the first scheduler is on the x-axis, and the clock of the second is on the y-axis. As wall-clock time advances, we consider the clocks of each scheduler and plot one against the other. The plotted line shows one possible path. The hatched areas denote states defined as illegal when the global quantum is at 50. We can see that the global quantum constrains the possible combinations of states.

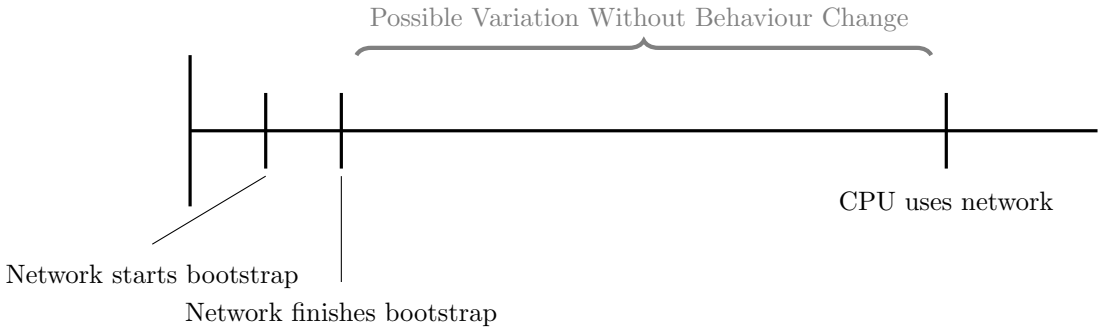


Figure 4.6: *Loose* timing

4.5.1 Notion

The global quantum tool is perfect for expressing what was said about loosely-timed models in Section 2.5.4: *Loosely-timed models, by their very nature, use little time-based synchronisation.* Little, but still some, for convenience reasons. For example, how does one model a timer in an untimed world?

Let us consider an example: We have two processes, representing a network card and a CPU. When we launch the simulation, the CPU configures the network card with some parameters and requests it to begin bootstrapping, an operation that may take around 1 ms. The CPU then continues with other work, the expected duration of which is 10 ms. The scenario is shown in Figure 4.6. Then, the CPU uses the network card directly without any synchronisation. We make the assumption that the network card is ready because our understanding of *loose* doesn't let us envisage a scenario where the network card initialisation might not finish before the CPU's work. We might revise our estimations of the timing details one way or another when the chip is finalised, but we don't expect them to vary so much that the network card does not finish before the CPU.

So, the obvious question is: *Just how much variation do we envisage?*. Another way of expressing the same idea is: *How long ago does something need to have occurred (according to the current loose timing values) for us to safely assume it has occurred?*, or *How far apart in time must events be for us to use implicit synchronisation?* The answer to all these questions is actually the ideal value for the global quantum.

Let us imagine a simulation where we know that the answer to the above questions is 5 milliseconds. For interactions occuring in less than 5 milliseconds according to the loose timing values, event-based synchronisation is used, but for any others, implicit synchronisation is used. This means that even if up to 5 milliseconds of variation should occur, our simulation will behave in the same way, because we are using explicit synchronisation. If we set the global quantum to 5 milliseconds, then the variation between different schedulers will be limited to 5 milliseconds, and the behaviour of the simulation will remain unchanged.

4.5.2 Practice

The global quantum is a simple, yet remarkably powerful, idea. It is one simple parameter that allows us to maximise parallelism without threatening correctness. The difficulty we face is choosing the right value.

If the communication pattern of the system is simple, then one only has to set the quantum

to just below the frequency at which different schedulers communicate to assure correctness and enable a degree of parallelism. However, in a more complex system, this may be difficult. In this case, an experimental approach can be used. If the global quantum is set to 0, then the schedulers will advance in lock-step with one another, as in the conservative approaches mentioned in chapter 3. This is useful because it enables us to compare our solution to a traditional conservative parallelisation approach. It also allows us to progressively increase the global quantum from a known good starting point and observe the behaviour of the system. At some value of the quantum, the system will start to deviate from its intended behaviour, and we will know that we have gone too far.

It may be that in a complex system, in general a fairly large quantum could be tolerated, but a few small parts of the system require closer synchronisation. In this case, we provide a degree of local control that can be used to enable the global quantum to be left at a larger value.

If the quantum is not set, the schedulers will not be constrained to a certain time window. The quantum can be set by a simple function call. The quantum is disabled by default, as it would be difficult to choose a meaningful default value.

4.6 Local Control

Global control is effective at partially desynchronising parallel processes and enabling a degree of parallelism without ramifications on correctness. However, it is not sufficiently flexible. The entire system need not be constrained to a certain quantum if only two minor subcomponents actually require tight temporal constraints. Furthermore, it does not solve the shared resources problem.

We have implemented *ad-hoc*, or local, synchronisation primitives, designed to address both these problems.

4.6.1 Atomicity

When executing threads in parallel, it is possible that there may be race conditions between the threads. In our kernel, this only applies to threads running on distinct schedulers, as all threads within a scheduler share co-operative semantics as in standard SystemC. One example of why atomic sections are necessary is the read-modify-write paradigm shown in Figure 4.4 - if the section of code is executed by two threads at the same time, then d may only be incremented once.

We have added support to initiator sockets for *atomic sections*. An atomic section is a piece of code which can only be executed by one thread at at time. If two threads try to execute the section at the same time, one will block until the other has finished all of the instructions in the section.

In our work, an atomic section locks down the bus and allows only the scheduler responsible for the thread issuing the lock to access it. This ensures that no other transactions take place on the bus while it is locked and protects the integrity of multi-stage modifications. Two examples showing the use of atomic sections are given in Figures 4.7 and 4.8. One can either make an explicit function call (`begin_atomic()` or `end_atomic()`) on the socket, or an argument can be combined with a synchronisation specifier (see following section).

Atomic sections should be used only where necessary, and with care. Omitting an `end_atomic()` or placing two `begin_atomic()`s in succession will cause deadlock. Calls to

```
void increment(addr){
  int temp;

  bus_socket.read(addr, &temp, BEGIN_ATOMIC);
  temp++;
  bus_socket.write(addr, temp, END_ATOMIC);

}
```

Figure 4.7: Declaring an atomic section with parameters

```
void increment(addr){
  int temp;

  bus_socket.begin_atomic();
  bus_socket.read(addr, &temp);
  temp++;
  bus_socket.write(addr, temp);
  bus_socket.end_atomic();

}
```

Figure 4.8: Declaring an atomic section with function calls

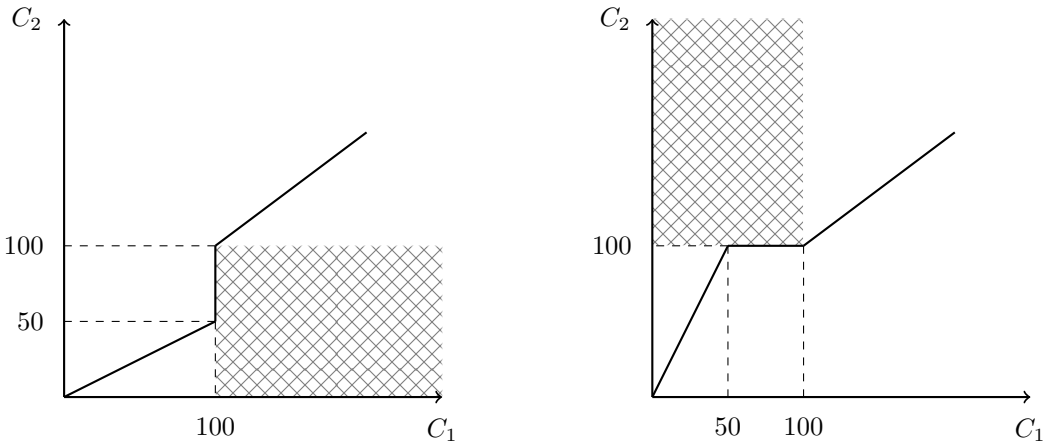
wait should be avoided within an atomic section, as yielding control to another thread while holding a lock will lead to deadlock if the second thread also attempts to begin an atomic section. Indeed, this is to be expected, as the idea of an atomic section runs contrary to the idea of yielding control of the processor.

We have included two different syntaxes for the comfort of the programmer, each one being more elegant in certain situations.

4.6.2 Transaction Timing Specifiers

We introduce *transaction timing specifiers*, arguments that can be provided when components communicate. The idea is to specify constraints on the state of the clocks of the schedulers participating in the transaction. When a transaction is performed, we provide keywords that can be provided as parameters to the read or write call to describe the relative state of the clocks of the two schedulers. Considering two processes A and B, where A performs the transaction and B is the remote module, with their scheduler clocks C_A and C_B, the keywords can be described as follows:

1. `SYNC_WAIT` : If $C_B < C_A$, block A until $C_B \geq C_A$, and then perform the transaction. If $C_B \geq C_A$, perform the transaction immediately. Figure 4.9a shows the effect this has on the clocks of the two schedulers. On the x-axis is the clock of the scheduler S_1 performing the transaction and specifying `SYNC_WAIT`. On the y-axis is the clock of the scheduler S_2 on the receiving end of the transaction. In our example, it so happens that in this run of the program when S_1 reaches the point where it wants to perform the transaction (local time 100), S_2 is only at local time 50. The `SYNC_WAIT` construct



(a) `SYNC_WAIT` (b) `SYNC_CATCH_UP`

Figure 4.9: The effect of `SYNC_WAIT` and `SYNC_CATCH_UP` on legal system states

prevents S_1 from advancing until S_2 has reached its own local time 100. The hashed area on the graph represents an illegal state implied by the use of this particular construct.

2. `SYNC_CATCH_UP` : If $C_B > C_A$, block B, insert a call to `wait(`$C_B - C_A$`)`, and then perform the transaction. This situation is described in Figure 4.9b. In this case, it so happens that when S_1 wants to execute a transaction at local time 50, S_2 already has local time 100. The `SYNC_CATCH_UP` construct blocks S_2 so that it cannot continue, until S_1's local time reaches 100. Again, the hatched area represents states disallowed by the use of this construct.

   Note that in this example the transaction is executed when both clocks are equal to 100. However, if S_2's clock had been behind S_1's when S_1 wanted to execute the transaction, the transaction would have been executed when S_1's clock was equal to 50. This means that, when using this construct, we cannot necessarily know in advance at what time the transaction will be executed, as it depends on the hardware executing the simulation.

3. `FULL_SYNC` : Perform a `SYNC_WAIT` followed by a `SYNC_CATCH_UP`, then execute the transaction. This guarantees that the clocks of the two participating schedulers will be equal at the time of the transaction, but suffers the same weakness as is inherent to `SYNC_CATCH_UP`. If the target scheduler's clock is ahead of the initiator's, the only thing that can be changed is to move the initiator's forward. This is because we do not envisage a roll-back scheme. Figure 4.10 shows a possible scenario.

4. `SYNC_INSTANT` : Perform the transaction, then block A until $C_B \geq C_A$ and B finishes its work for that time instant. Rather than verifying some constraint and then performing the transaction, the transaction is performed first, and then we wait for the constraint before continuing. This is useful when we wish to wait for the remote component to react to our transaction before continuing execution, as we will see in Section 4.8.2.

5. `NO_SYNC` : Perform the transaction regardless of the clocks.

These options allow the programmer to assert that certain properties should hold at the time of a transaction. However, the programmer should be aware of the following points:

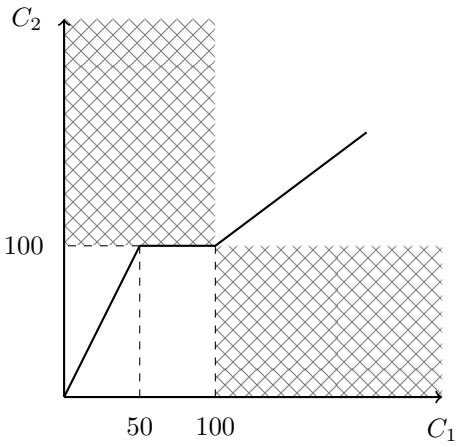


Figure 4.10: The effect of `FULL_SYNC` on legal system states

- `NO_SYNC` is the default option. This decision was not straightforward - if we do not synchronise by default then a simple oversight can wreak havoc with the model. However, if we do synchronise by default then all transactions have to be explicitly labeled with `NO_SYNC` for the user to see any speed-up using our simulator. Since we assume that most transactions do not require synchronisation, we decided to avoid synchronising by default.
- `SYNC_CATCH_UP` should be used only with care. Since it may insert a call to `wait`, it can change the semantics of a model. Furthermore, it should not be used within an atomic section, as calls to `wait` within an atomic section are dangerous. This applies to `FULL_SYNC` as well as their related window variants.
- `SYNC_WAIT` is actually very powerful. Not only does it not have the dangerous behaviour that full synchronisation has, but it can be used by two communicating threads that communicate in a ping-pong style to approximate full synchronisation. We will see this in the examples section.
- The `WAIT` and `CATCH_UP` constructs also come with a `WINDOW` variant. This variant accepts a time parameter t and asserts that the remote scheduler be no more than t behind or ahead, respectively. This allows more flexibility.

4.6.3 An implementation problem

The intuitive meaning of the local timing specifiers is that the given property should hold *when the transaction is carried out*. This causes a problem for the implementation of `SYNC_CATCH_UP` and the like. The transaction itself is arbitrary user code unknown to the TLM interface *a priori*, but the TLM interface can execute its own code just before and just after the execution of the transaction. If component A is catching up on B, and we perform the synchronisation operation just before performing the transaction, then we are forced to release the lock that blocks B from advancing just before we hand over to the user code. However, the execution order from that point is subject to the whims of the host operating system - meaning B could advance its clock before A actually executes the transaction. For correct behaviour, it is necessary to keep blocking B throughout the entire transaction.

Note that this is not necessary for `SYNC_WAIT`, which guarantees only that the C_{remote} is not less than C_{local} but does not stipulate that it should be equal.

4.6.4 Persistent Events

In Section 2.5.2 we stated that SystemC events are non-persistent. Typically, if a persistent event is required, the programmer uses a Boolean variable in conjunction with the event as shown in Figure 4.11. Unfortunately, in a parallel environment, the programmer is unable to write his own persistent events in this way, as the test of the variable and the wait on the event are not atomic *vis-à-vis* the other threads that may be notifying the event. Since the wait is performed by the kernel, the programmer is unable to modify this method to build an atomic test-and-set. For this reason, our parallel kernel has to provide persistent events for the user.

```
sc_event event;
bool notified;

P1:

  notified = 1;
  event.notify();

P2:

  if(!notified)
    wait(event);

  notified = 0;
```

Figure 4.11: Using non-persistent events to build persistent events

4.7 Usage guidelines

We provide the following guidelines on partitioning and avoiding race conditions, when using our simulator.

4.7.1 Partitioning

As stated before, a good partitioning will minimise communications between schedulers and balance workload. Consider the system in question: Where are the components placed on the chip? In the SystemC model, which are the components that perform a lot of work? The priority should be the load balance, with a preference for placing communicating components together. Obviously, if one places all communicating components together, then the entire system is controlled by one scheduler, and there will be no speed-up at all! If two components have particularly sporadic or unknown communications, it may simplify things to place them on the same scheduler.

If two components are placed on the same scheduler and therefore it is not necessary to pay attention to their communication and atomic sections, this will save development time and increase simulation speed. However, it does mean that if ever it becomes desirable to separate these two components, it will be necessary to deal with these questions at that time.

4.7.2 Race conditions

As explained in Section 2.5.3, the functions that components export for remote execution will be executed in the context of the remote thread. This means that, to take an example, a component modeling main memory that exports a function *write()* allows that function to be executed by threads on other partitions, in parallel. This introduces two possible types of race conditions:

1. A race between the remote thread in the `write` function and any threads resident in the memory component, if both manipulate shared data.

2. A race between two remote threads both executing the `write` function at the same time.

Careful attention should be paid to these race conditions, and mutexes should be used where correctness is at stake.

Race 1 can turn out to be more complicated than it first seems. It may be the case that the remote thread does very little work - simply setting a flag or sentinal, and notifying an event. However, although we have made events thread safe and setting a flag may be an atomic action, the cache coherency protocols of real hardware may still mean that the actions of the remote thread are not visible to the local thread when they should be. Since the use of mutexes will ensure that memory barriers are used at the correct time, the simplest solution is just to place a mutex in the transport layer around the call to the arbitrary user code implementing the transaction.

Furthermore, it may be known that only one remote component will ever attempt to communicate, avoiding race condition 2. Where this is not the case we provide a target socket containing a mutex that serialises all transactions on the component it is bound to. However, this approach may be too heavy-handed and negatively affect performance.

4.8 Discussion

We now discuss the difficulties faced when attempting to use our interface to parallelise a common communication paradigm used in SystemC/TLM. This will clarify our design and highlight the strengths and weaknesses of our approach. We discuss the producer-consumer paradigm.

4.8.1 Producer & Consumer - time based

We first consider a purely time-synchronised producer and consumer example. For reference, Figure 4.12 shows what a classical time-based SystemC/TLM implementation might look like. First the producer writes to the main memory over the bus, and then sleeps for 10 milliseconds, to allow the consumer time to read the current value, and then it repeats. The consumer waits 5 milliseconds, to allow the producer time to write the value, and then begins sampling the memory address every subsequent 10 milliseconds.

```
void
producer::produce(){
  ensitlm::data_t x = 0;

  while(x++ < 10){
    out.write(0x0, x);
    wait(10, sc_core::SC_MS);
  }

  return;
}
```

```
void
consumer::consume(){
  ensitlm::data_t d = 0;

  wait(5, sc_core::SC_MS);

  while(d < 10){
    out.read(0x0, d);
    wait(10, sc_core::SC_MS);
  }

  return;
}
```

Figure 4.12: Producer-consumer paradigm in classical SystemC

This is not an example of the loosely-timed coding style, as there is a tight timing dependency between the two components. However, as we pointed out in Section 2.4.1, although in theory one coding style should be rigourously used throughout a given model, in practice a mix of styles can be observed. A globally LT model containing some more tightly coupled sections might require the following approach.

We now discuss two different ways of implementing this paradigm using our simulator.

Global quantum

Using our simulator, the simplest way to ensure this model was correct would be to set the global quantum to 4 milliseconds. Figure 4.13 shows the relevant code. We choose the value 4 because any value greater than or equal to 5 could cause deviant behaviour. Consider the situation where the producer has $t = 0$ and the consumer $t = 5$ at the same wall-clock time: with a global quantum of 5, the consumer could execute before the producer.

```
sc_core::sc_set_quantum_enabled(true);
sc_core::sc_time t(4, sc_core::SC_MS);
sc_core::sc_set_quantum_time(t);
```

Figure 4.13: Setting the global quantum

Local Timing Specifiers

We can achieve the same effect using local timing specifiers. In this case it is only necessary to use backward synchronisation, which is preferable. At each read or write, we insert the backward synchronisation specifier, `SYNC_WAIT`, instructing each component to wait until the other has caught up before executing the transaction. This effectively means that:

- The consumer will never read until $C_{producer} >= C_{consumer}$. This means that the consumer will never read too early, and risk reading a value as yet unwritten.

```
void
producer::produce(){
  ensitlm::data_t x = 0;

  while(x++ < 10){
    out.write(0, x,
          ensitlm::SYNC_WAIT);
    wait(10, sc_core::SC_MS);
  }

  return;
}
```

```
void
consumer::consume(){
  ensitlm::data_t d = 0;

  wait(5, sc_core::SC_MS);

  while(d < 10){
    out.read(0x0, d, ensitlm::SYNC_WAIT);
    wait(10, sc_core::SC_MS);
  }

  return;
}
```

Figure 4.14: Producer-consumer with timing specifiers

- The producer will never write until $C_{consumer} >= C_{producer}$. This means the producer will never write too early, and risk overwriting a value which has not yet been read.

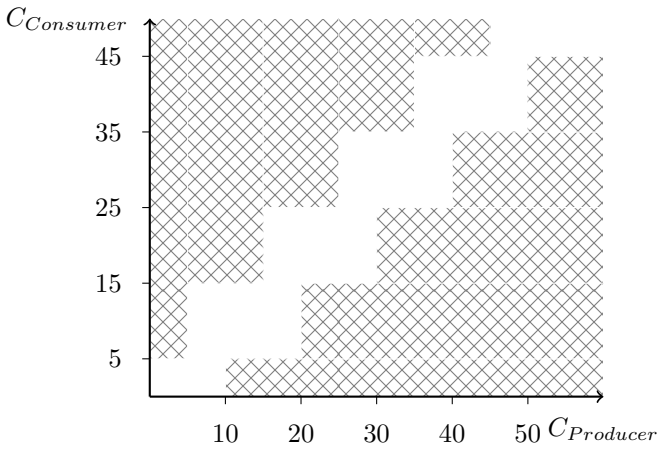


Figure 4.15: Consumer-producer paradigm with local timing specifiers

These two facts are combined to ensure correct execution. The code is shown in Figure 4.14. The state graph is shown in Figure 4.15. Although the two threads are executed in parallel, we can still see why this works by considering an example execution sequence:

1. Consumer reaches `read` before Producer reaches `write`. $C_{consumer} = 5$, as we are in the first iteration of the loop. As Producer has not yet written the value, $C_{producer} = 0$. The Consumer thread will block before executing the `read`.

2. Producer reaches the `write` call. Since $C_{consumer} = 5$ and $C_{producer} = 0$, it can execute the transaction, and does so. It then sets $C_{producer} = 10$ and loops back to the `write` call. Since $C_{consumer} < 10$, Producer blocks before carrying out the write.

3. Consumer wakes up when Producer increases its clock. Since $C_{consumer} < C_{producer}$, it executes the read and gets the value 1. It then increases its clock and loops back to

`read`. Since $C_{producer} = 10$ and $C_{consumer} = 15$, Consumer blocks before carrying out the read.

4. Producer wakes up when Consumer increases its clock. The cycle continues...

And so, by the simple addition of a parameter to each call, we have ensured correct execution order in parallel system. There is, however, a small detail that we have neglected to mention.

Memory Access Synchronisation

We said that the producer and the consumer were communicating via an address in main memory. The main memory is a component that makes up part of the SystemC/TLM model, and as such must be placed on a specific partition, which will be on a specific scheduler. A read transaction over the bus with a `SYNC_WAIT` argument means *synchronise with the target component*. In our example, in both cases, the main memory is the target component, which breaks our example, as Producer and Consumer are not actually synchronising on one another, but on the main memory.

```
//producer has SC_AFFINITY(0)
bus.add_synchro_range(0x0, sizeof(ensitlm::data_t), 0);
//consumer has SC_AFFINITY(1)
bus.add_synchro_range(0x0, sizeof(ensitlm::data_t), 1);
```

Figure 4.16: Annotating ranges of memory with scheduler affinities

While the interface we have provided is technically behaving correctly, this is rather awkward. Conceptually, when the producer writes to the main memory, we imagine that we are communicating with the consumer, because it is the consumer which will subsequently read the value. We would like to avoid having to insert a dummy transaction with the consumer just for synchronisation purposes. It is for this reason that we have enabled the annotating of ranges of memory with the affinities of schedulers, so that when this memory is accessed, any timing specifiers trigger synchronisation also on these schedulers. These ranges can be added and removed both statically and dynamically. To complete our example, we used the to calls shown in Figure 4.16 to set up the ranges.

4.8.2 Producer & Consumer - event-based

There is another way of writing the producer-consumer paradigm, which we have called event-based. It is not entirely event-based, but rather a hybrid of the two approaches. The producer is the only component using time, and the consumer is notified over the bus when the producer has written a value. We refer to this as an "interrupt", even though real interrupts are not sent over the bus. For reference, Figure 4.17 shows what this might look like in classical SystemC/TLM. The consumer simply waits on the event. When it is woken up, it reads a value, and goes back to sleep. The TLM layer automatically routes the write on the memory-mapped bus on the address `CONSUMER_INT` to the consumer's `write` function. The `wait` at the beginning of the producer code is necessary, otherwise, if both components were eligible at $t = 0$, the producer might send the interrupt before the consumer started waiting. The

producer writes the value to the memory and then interrupts the consumer, then waits and begins the cycle again.

```
void producer::produce(){
  ensitlm::data_t x = 0;

  wait(5, sc_core::SC_MS);

  while(x++ < 10){
    out.write(0x0, x);
    out.write(CONSUMER_INT, 1);

    wait(10, sc_core::SC_MS);
  }
}
```

```
void consumer::consume(){
  ensitlm::data_t d = 0;

  while(d < 10){
    wait(c_int);
    out.read(0x0, d);
  }
}

void consumer::write(addr_t, data_t){
  c_int.notify();
}
```

Figure 4.17: Producer-consumer paradigm using events in classical SystemC/TLM

This style is harder to write correctly in our simulator. The introduction of SystemC events brings up some interesting questions. The first of which is the simple observation that, as we have written it, the consumer does not advance time. There is no call to `wait(time)` in its code. An obvious first problem is therefore finding the answer to the questions: *what will happen if we use `SYNC_WAIT` on the producer's side?* Since the consumer does not move time forward, how can the synchronisation specifier ever be satisfied?

An absence of time

In classical SystemC/TLM, a call to `wait(event)` removes the calling thread from the eligible queue. The thread will not be placed in the eligible queue again until a notify is performed on the event. This means that while a thread is waiting on an event the rest of the simulation proceeds normally. This is necessary in classical, sequential SystemC/TLM, as, if the call to wait were to block, the entire simulation would halt. Since there is only one notion of time and there are no external events, the waiting thread will be woken up at the same point in every simulation run and the simulation will be in the same global state.

In a parallel simulation with distributed time, there are two clocks in play when an event is notified - the clock of the waiting thread's scheduler, and the clock of the notifying thread's scheduler. We would like the thread to wake up when $C_{waiting} = C_{notifying}$ to avoid temporal violations. If $C_{notifying} > C_{waiting}$ when the `notify` is performed, we can insert the wake up as a timed event that the waiting thread's scheduler will execute in the future. However, if $C_{waiting} > C_{notifying}$, we have a problem. The damage has been done, and a roll-back scheme would be required to bring $C_{waiting}$ back to $C_{notifying}$. In this case, we can either say that we don't care, and continue the transaction anyway, or we can set $C_{notifying} = C_{waiting}$ and then perform the transaction. This corresponds to usage of the `WAIT_CATCH_UP` identifier introduced earlier.

To come back to our original question: *what will happen if we use `SYNC_WAIT` on the producer's side?*. By saying `SYNC_WAIT`, we are specifying that we want $C_{waiting} \geq C_{notifying}$ before we execute the transaction. Unfortunately, in our example, the transaction itself is what will set $C_{waiting} = C_{notifying}$, because it is the transaction that notifies the event and

inserts the timed event discussed above. Therefore, a natural implementation will deadlock at this point.

Since the transaction is arbitrary user code, unknown to the TLM interface *a priori*, it is difficult to see how to avoid this problem. We opted for an imperfect, pragmatic solution to avoid deadlock and to bring time forward on the consumer partition. It is easy to detect when the entire system is about to deadlock by checking, when the last scheduler attempts to perform a synchronisation operation, whether all other schedulers are sleeping (out of work) or synchronising. When this situation occurs we choose the synchronisation operation with the earliest deadline, move the time on the target scheduler forward, and wake up that scheduler. That scheduler will execute its work for that time instant, and then wake up any other schedulers synchronising on it, as per the usual rules.

The fundamental problem is that our local synchronisation primitives are time-based, and therefore we cannot synchronise correctly with a partition that does not make use of time. A partition that does not advance its clock requires another type of primitive, which we will discuss later.

We can, however, handle this kind of situation with the global quantum. If it is the case that all synchronisers are either asleep or blocked from advancing by the global quantum, the global quantum algorithm allows the scheduler which has the nearest deadline to advance, and advances all schedulers' clocks appropriately. This part of the algorithm allows us to execute this example correctly with a global quantum ≤ 5.

In presence of time

```
void producer::produce(){
  ensitlm::data_t = 0;

  wait(5, sc_core::SC_MS);

  while(x++ < 10){
    out.write(0, x,
              ensitlm::FULL_SYNC);
    out.write(CONSUMER_INT, 1,
              ensitlm::FULL_SYNC);

    wait(10, sc_core::SC_MS);
  }
}
```

```
void consumer::consume(){
  ensitlm::data_t d = 0;

  while(d < 10){
    wait(c_int);
    out.read(0, d, ensitlm::NO_SYNC);
  }
}
void consumer::move_time(){
  x = 0;
  while(x++ < 10000)
    wait(2, SC_MS);
}

void consumer::write(addr_t, data_t){
  c_int.notify();
}
```

Figure 4.18: Producer-consumer paradigm using events in our simulator

We have thus far assumed that the consumer and producer were alone on their respective partitions. It is interesting to see what can be done when this is not the case. If we add another thread to the consumer's partition, that simply advances time in small increments in a tight loop, then the problem of a lack of time is solved. However, we need to use a different approach to the original, time-based example. On the consumer side, we can safely

say `NO_SYNC`, because the consumer is woken up only once the producer has written the value, so the consumer cannot possibly read a value before it is written. However, on the producer side, we need a `FULL_SYNC`. Since the consumer's partition can easily be ahead of the producer, use of `SYNC_WAIT` would be insufficient, as the producer could write several values in sequence before the consumer reads them. We know that:

1. $C_{cons} = C_{prod}$ when we wake up the consumer (thanks to the full synchronisation).
2. C_{cons} cannot advance until the consumer has read the value written by the producer.
3. C_{prod} must advance before the producer comes to write the next value.
4. C_{cons} must equal C_{prod} before the next value is written.

These four facts combine to prevent the producer from overwriting an unread value, and since the consumer cannot read an unwritten value, this solution is correct. The code is shown in Figure 4.18. Furthermore, the solution remains correct in presence of another thread that advances time on the producer's partition.

Instant synchronisation

Faced with the failure of our synchronisation specifiers to deal with a partition that does not make any calls to `wait(time)`, we introduced the `SYNC_INSTANT` specifier. The goal of this specifier is to be able to say: *Block until the remote component has reacted to this transaction.* More precisely, if $C_{remote} < C_{local}$, the local component is woken up only once $C_{remote} = C_{local}$ and the remote component has finished all its work for this time instant. If $C_{remote} >= C_{local}$, then the remote component is pinned to the time instant it is in just before the user code that constitutes the transaction is executed. Then, the local component unpins the remote, and sleeps until the remote component finishes the current time instant. At this point, we are sure that any code triggered by the transaction has been completed.

```
void producer::produce(){
  ensitlm::data_t = 0;

  out.synchronise(CONSUMER_INT, ensitlm::INSTANT_SYNC);

  wait(5, sc_core::SC_MS);

  while(x++ < 10){
    out.write(0, x, ensitlm::NO_SYNC);
    out.write(CONSUMER_INT, 1, ensitlm::INSTANT_SYNC);

    wait(10, sc_core::SC_MS);
  }
}
```

Figure 4.19: Usage of instant synchronisation

This specifier can be used at the point where the producer writes the interrupt that wakes up the consumer, as shown in Figure 4.19. The initial synchronise operation is used to

bootstrap the interaction, that is, to ensure that the consumer is waiting on the event before beginning. Alternatively, we could have used time to do this as in the earlier examples.

This specifier is very convenient to use, so much so that we considered extending the principle to include a specifier whose meaning would be: *Block until the remote component has* $C \geq x$. However, we realised that such a specifier is not really an extension of the same principle, and its rôle can be fulfilled by the other time-based specifiers which we mentioned earlier.

The guidelines for using our specifiers discussed here extend naturally to multiple consumers or producers, as well as transmitters (a ring where each node consumers from the previous and produces for the following).

We have presented how our interfaces can be used in practice and some of their strengths and weaknesses. We now move on to discuss performance.

Chapter 5

Evaluation

We set out to increase the speed of SystemC/TLM simulations while preserving simulation correctness. We now discuss the performance of our simulator compared with the reference implementation of the SystemC kernel. We use a thin layer, called `ensitlm`, to implement the TLM part of the simulation. The following three major factors limit the possible speed-up that can be achieved with our simulator:

1. The relationship between the amount of time the model spends doing *real work* and the amount of time it spends in the SystemC scheduler and TLM overlay. We do not suggest a precise way of measuring this proportion but introduce it as a high-level concept that describes how much calculation a model performs. The more real work to be done, the greater the speed-up to be expected.

2. The amount of communications occurring between components. More communications usually implies more synchronisation, and synchronisation will limit the potential speed-up.

3. The load balancing that can be achieved. The fairer the load balance, the better the speed-up we might expect.

The ideal use case for our simulator would be a simulation that does a large amount of independent arithmetic operations with no synchronisation required between the components performing the calculation. Such a simulation might be a matrix multiplication algorithm or an algorithm to calculate the value of π. In these cases we might expect a speed-up close to the number of processors N.

Conversely, the worst use case would be a simulation were very little *real work* is done and components spend a large part of their time communicating with one another. An example of such a simulation might be a simulation implementing a network protocol. Since we have modified the SystemC kernel to make it thread-safe and to add additional functionality, we would expect a very low speed-up or even a slow-down in this case.

Since SystemC models are very varied and there is no system that can be described as *typical*, it would not be very instructive to present any one system here, and it would be a very long report if we attempted to present performance results for all classes of systems. Instead, we will consider one simple system with different levels of communication and evaluate the overheads imposed by our parallel kernel and by the use of our primitives.

We begin with a description of our test architecture, and then we discuss an example of the best case. This is interesting because it confirms the principle of the project and because

```
for(int i = 0; i < hw_units; i++){
    issue_work(i);
    sc_core::wait(50, sc_core::SC_NS);
}
```

Figure 5.1: Work distribution

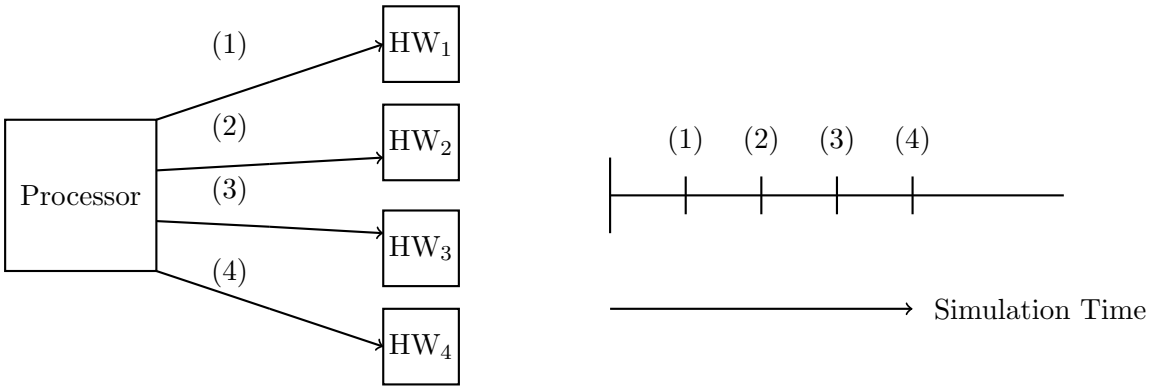


Figure 5.2: Test Bench Architecture

it provides an upper limit on the results we might reasonably expect. We then investigate how performance degrades as we make the best case progressively worse, before moving on to some conclusions.

5.1 Test Architecture

We consider a simple system which cannot be parallelised by conservative approaches, as the parallel operations are not launched at the same time. For example, consider a simple chip that calculates the value of π in parallel on several hardware units. A processor hands out work to the hardware units. We wish to model the fact that all hardware units do not receive their work at the same time, due to the serial nature of the distribution and the relative distances of each hardware unit from the processor.

We might model this situation by the code shown in Figure 5.1. Instead of issuing work to all the hardware units in the same instant, we issue work one by one with a short delay inbetween each issue. Then each hardware unit will calculate the relevant digits of π, and when the last hardware unit finishes, the simulation is over. Since we have included the time delay when distributing work, conservative parallelism is unable to parallelise this model effectively. Figure 5.2 shows the system architecture and a timeline showing when the distribution of work occurs.

The system is configurable and divides up the work to be done over a variable number of hardware units. Each hardware unit is defined to be on its own partition, by use of the `SC_AFFINITY` macro introduced earlier. Furthermore, the system can be configured to use a different level of timing accuracy - in concrete terms this means the inclusion or exclusion of calls to `wait`, as shown in Figure 5.3. The call to *out.sync* is only used if we are operating in TLM transaction specifiers, as will be explained shortly. We remind the reader that the

```
if(sync_level >= 1){
  wait(t, sc_core::SC_MS);

  if(using_trans_spec){
    out.sync(neighbour, SYNC_WAIT);
  }
}
```

Figure 5.3: Synchronisation in our test model

`sync` call generates a dummy transaction to be used just for synchronisation purposes. These levels of timing accuracy allow us to study the effect of synchronisation on speed-up, as we will use time-based synchronisation to simulate communication between the units.

We use the notation HW_n; that is hardware unit number n, and n_x; that is the x^{th} step belonging to hardware unit n, where the work to be done by each thread is divided into steps such that step n_1 is the first step of thread n , step n_2 is the second step of thread n etc. Furthermore, at the end of each step there is a call to `wait` as described in Figure 5.3. In other words, at regular intervals throughout its calculation, each hardware unit will make a call to `wait`. The higher the level of synchronisation, the more regular the intervals are.

Let us imagine a situation where we want to ensure that n_x cannot be executed before $(n-1)_{x-1}$. Put differently, HW_n, when it has finished its step n_x, must wait for HW_{n-1} to have finished step $(n-1)_x$ before proceeding. We have to achieve this effect with our interface.

If we use the global quantum, then the best approximation we can provide to this behaviour is that no hardware unit may begin step $x+1$ until all hardware units have finished working on step x. To do this, we just set the global quantum to a value smaller than the t provided to the `wait` call. If we use the TLM transaction specifiers, we can model this property precisely by inserting a synchronise operation after each call to wait, providing the neighbour as an argument.

There is a major difference between the two modes: in the global quantum mode we use a quantum value such that all hardware units must be at the same stage of the calculation at each point, that is to say they have no freedom to advance their clocks until all hardware units have finished the current step. In the transaction specifiers mode the hardware units are arranged into a ring and each unit only has to wait for its neighbour. The constraint is stronger in the global quantum mode, and we might expect to see this reflected in the results.

We include three levels of synchronisation: none, medium, and strong. The levels themselves have been chosen arbitrarily, but to give an idea of what they mean, we have observed that the level we have called medium corresponds to one call to wait per 70000 machine instructions executed by the hardware unit. The level we have called strong corresponds to one call to wait per 40000 machine instructions executed by the hardware unit.

5.2 Results

We performed our experiments on an UltraSparc T1 *Niagara*, running 64-bit Solaris. The machine has 6 processors, each supporting concurrent execution of up to 4 threads (Hyper-Threading). This gives us 24 effective processing units, which we will refer to as *cores* from

this point on, for brevity. In our tests, we test up to 128 threads. This is of interest because having spare threads to run may enable cores to hide the costs of those threads which are unable to run because they are contending for a mutex. On the other hand, by increasing the pressure on critical sections, it may cause a slow-down. Each core has 8KiB of L1 data cache, using a write-through policy, and there is a 3MiB L2 inclusive cache shared between all cores.

5.2.1 Best Case

Figure 5.4 shows our results in the best case - no synchronisation between hardware units. Each hardware unit simply performs its part of the calculation and returns. Since the SystemC kernel does a minimal amount of work in this case, there is little overhead caused by the extra code we have added to the kernel. This means that both our kernel and the OSCI version perform similarly given only 1 processor.

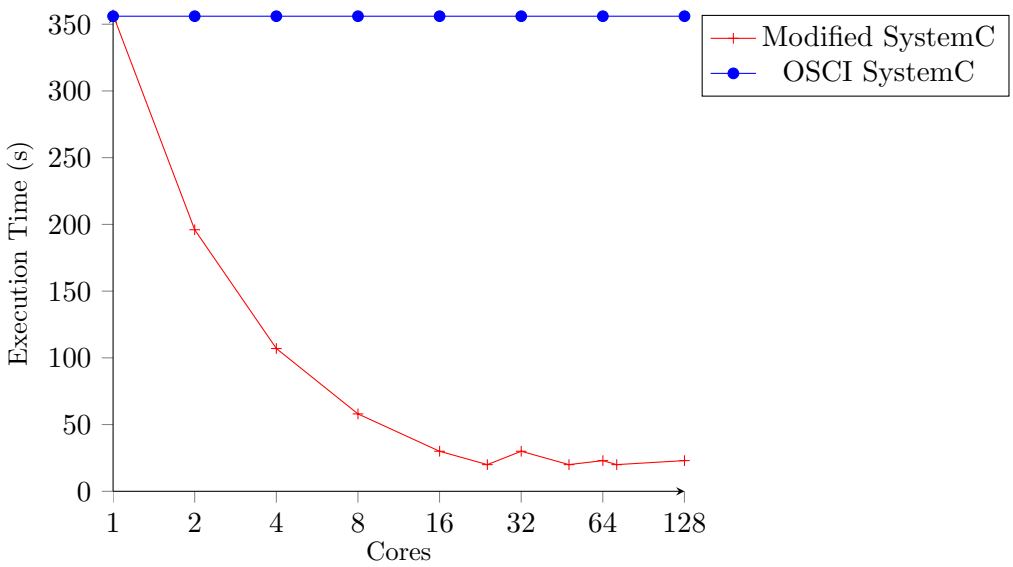


Figure 5.4: Relative Performance with no Synchronisation Requirements

Cores	Speed-up	Efficiency
2	1.81	90%
4	3.31	83%
8	6.02	75%
16	11.46	72%
24	17.42	73%
32	11.54	36%
48	17.14	36%
64	15.32	24%
72	17.14	24%
128	15.10	12%

Table 5.1: Speed-up and Efficiency on a 24-core machine

The time taken by the OSCI SystemC kernel remains constant as we add more cores. This is also to be expected - the only thing we change is that we divide the work up into smaller chunks, each run by a separate SystemC thread. This means more context switches will be made (1 per thread) but this is completely insignificant compared to the time taken by the calculation.

We see that we achieve very good speed-up as we increase the cores to 24 - the precise speed-up values can be consulted in table 5.1. The maximum speed-up we achieve is 17.42. After that point, performance begins to vary as we increase the threads used: 32 threads performs worse than 24 threads and also worse than 48 threads, and 64 threads is worse than 48 threads and 72 threads.

This effect is due to a load balancing issue: when we use 32 threads, 8 machine cores have to handle 2 threads each ($\frac{2}{32}$ of the workload), while the other 16 only handle 1 each ($\frac{1}{32}$ of the workload). Compare this with the 24-thread or 48-thread scenario where each core has precisely $\frac{1}{24}$ of the workload. Since work is pinned to cores and we must wait until all cores have finished, the execution takes longer with 32 threads than when we use 24 or 48.

We suggest that we do not reach the optimal speed-up of 24 because of the overhead incurred in our kernel in managing and creating the extra threads. There is a certain cost involved in the actual creation of pthreads, and there is also overhead in our kernel in managing these threads. They must be created when SystemC hardware units are first constructed, and then put to sleep until the processor distributes work to the hardware unit. The creation, sleeping and waking up of pthreads requires the use of expensive system calls.

Amdahl's law applies in this case: These management parts of our kernel must be executed in serial, because we are operating in a shared-memory environment. We call these parts K_s. The parallel part of the code (the calculation) we call K_p. When we run our code in parallel the time taken to execute K_p decreases, but the time taken to execute K_s remains the same. Furthermore, as we add more threads K_s actually grows as the management logic becomes more complex. Amdahl's law states that as we continue to parallelise K_p the execution time of K_p will diminish into insignificance, leaving only the execution time of K_s. This is an upper limit on the speed-up for a fixed problem size.

5.2.2 Normal case

Figures 5.5 and 5.6 show how the performance of our simulator degrades faced with an increasing level of synchronisation. In each graph we plot the performance of the global quantum technique (*Modified SystemC GQ*) and the TLM transaction specifier (*Modified SystemC TLM*) technique. We can make several useful observations.

Firstly, in the unicore case the increased number of calls to `wait` amplifies the overhead incurred by the use of our kernel. We can conclude that our more complicated logic is a serious cause for concern as even in the *medium* synchronisation example our kernel is about 20% slower than the reference implementation.

Secondly, we can see that the increased level of synchronisation decreases the speed-up we attain - 12.71 in the *medium* example and 9.94 in the *strong* example. However, we do maintain a useful degree of speed-up. It also causes the tail of the graph to rise, forming a 'U' shape. This means that after a certain point, adding more threads to the problem just worsens performance. This is often the case in parallel applications. It occurs because the bottleneck is no longer the calculation being performed but the communications between threads. Adding more threads just makes the problem worse. As we might expect, at higher

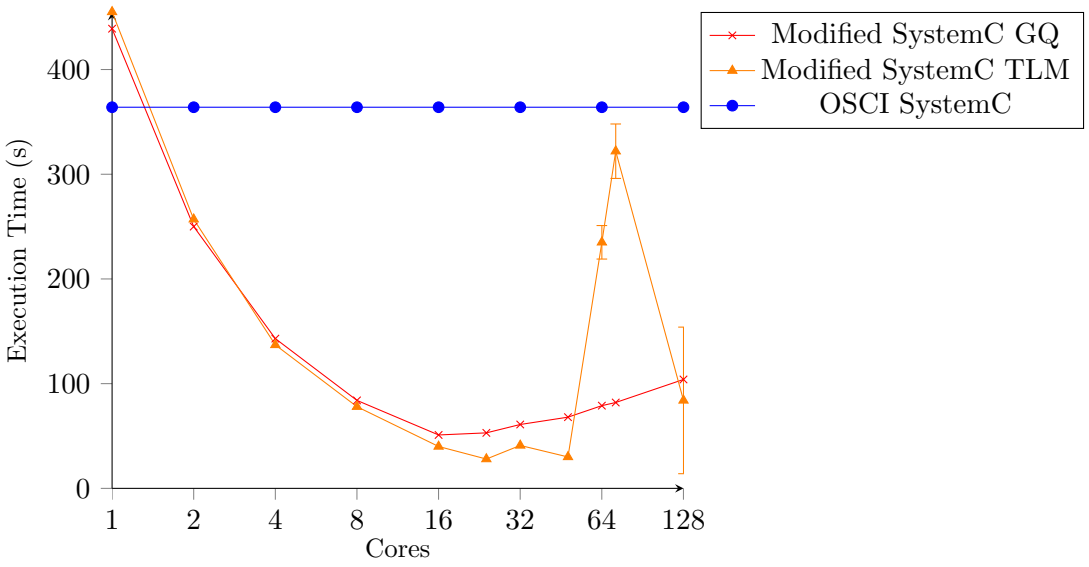


Figure 5.5: Relative Performance with Medium Synchronisation Requirements

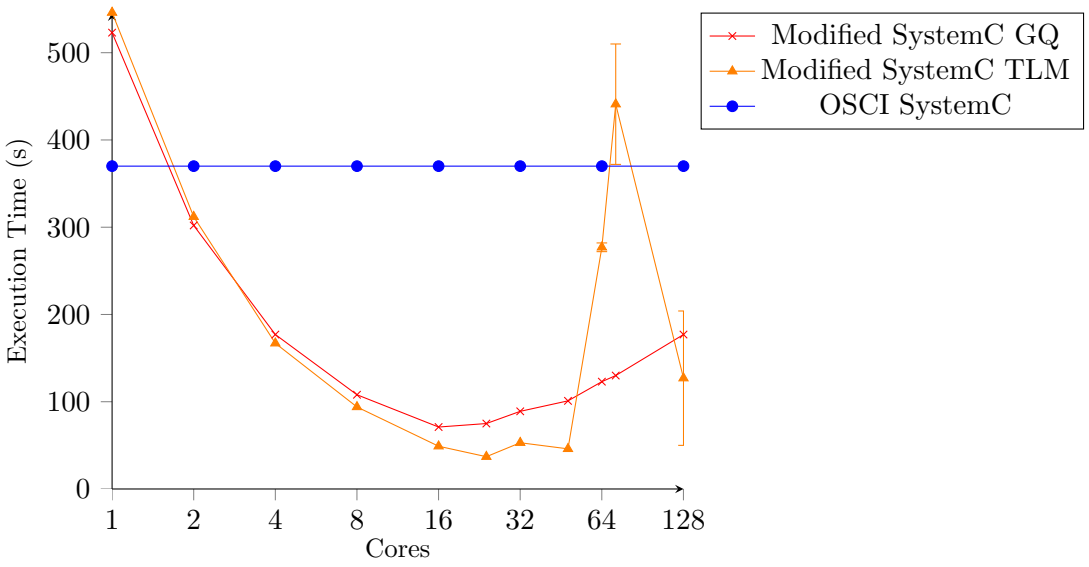


Figure 5.6: Relative Performance with Strong Synchronisation Requirements

levels of synchronisation the problem occurs earlier and the speed-ups obtained are worse.

Concretely, the reasons for this effect in our kernel are as follows:

a. Every time a SystemC process attempts to move its clock forward or to synchronise on another process, it enters a critical section, and then inspects and modifies some shared state. The more threads there are, the more contention there is for this critical section and the more likely it is that threads will have to idle, waiting for access to this section. Increasing the number of calls to `wait` has the same effect.

b. When a processor core executes the serial part of our kernel it changes some shared state. To do this, it first brings the shared state into its level 1 cache. When it changes the shared state, it invalidates the copies of the state that are stored in other cores' caches. These cores will then suffer a cache miss next time they wish to access the shared state, and will have to wait for the state to be fetched from main memory.

   As the number of threads increases, the number of cores in use increases and the changes made to the shared state become more frequent. This inceases the chance of increased cache misses.

c. If there are more threads than available cores, the cores will switch from one thread to another. This context switching operation is relatively expensive as the entire state of the running thread must be saved to memory and the state of the new thread restored. Furthermore, it requires operating system intervention. The OSCI implementation of SystemC uses a userspace threading library, called QuickThreads, that implements very fast context switching without operating system intervention. Userspace threading, however, does not allow for real parallelism, that is, concurrent execution on different cores, which is why we are forced to use a heavier-weight library like pthreads.

Thirdly, we note that, at least for numbers of threads between 4 and 48, the TLM transaction solution is consistently faster than the global quantum solution. As we suggested in section 5.1, this is because the constraint that we have specified on the hardware units is weaker in the TLM solution: a hardware unit only has to wait for its neighbour to have finished the current section, instead of all other hardware units.

Fourthly, we remark that TLM performance begins to degrade significantly at around 64 threads, before recovering rather dramatically at 128 threads. This is surprising, as we would expect the results for 64 threads to sit between the results for 32 threads and 128 threads. Furthermore, the results become increasingly erratic - we have included 95% confidence intervals in the graphs where they are of significance. Why should this occur for the TLM specifiers, but not the global quantum? Although further experiments are necessary to determine the precise cause, the following factors may contribute to the effect:

a. Due to the chain of dependencies set up by the ring topology, the TLM transaction specifiers interface is sensitive to the order in which the host operating system schedules threads. (Consider an execution which schedules the hardware units in reverse order). Therefore, when we have more threads than cores, we would expect to see more variation in the results for the TLM interface. This may account for the variation in the results but not for the poor performance itself, as even the worst scheduling order should be no worse than the global quantum approach.

b. Code using the TLM transaction specifiers actually calls the kernel twice, once for the `wait()` and once for the `out.sync()`. While each of these calls executes less code than a `wait()` when the global quantum is active, this behaviour may increase contention on the critical section and cause threads to spend longer waiting on mutexes. This could potentially be a factor explaining the spike at 64 threads, but it is not clear why the results should then improve at 128 threads.

c. Load balancing problems such as those discussed in the previous section could play a rôle.

5.3 Summary

Our simulator maintains a good degree of speed-up when subjected to non-trivial amounts of inter-process communication. However, these synchronisation requirements reduce the speed-up that it is possible to reach and introduce a 'U' shaped curve. While some performance may be gained by using a machine with more than 24 cores, in our example performance gains start to stagnate around that point anyway.

The performance of the global quantum and TLM transaction specifiers interfaces is similar in this simple example, although the global quantum's performance is limited by its lack of expressive power - it can only over-approximate the constraint we wish to express.

Chapter 6

Conclusion

In this work we have described an optimistic approach to the parallelisation of SystemC/TLM models on SMP workstations, using language constructs to specify valid execution orders. We attempted to address the gap in the existing work concerning the parallelisation of loosely-timed SystemC/TLM models. Our approach goes beyond the existing conservative approaches in that it can also parallelise loosely-timed models, where there is often only one runnable thread at each timing point.

We have found that very good speed-ups can be achieved, although contention for critical sections in the kernel limits performance improvements. We have confirmed the hypothesis that the amount of inter-process communications involved in a simulation has a direct effect on its potential for speed-up.

Moreover, we have evaluated the strengths and weaknesses of our language constructs used to constrain the set of valid execution orders. We conclude that some common SystemC communication paradigms can be expressed correctly using our constructs in a natural manner. Others are more difficult to express. We have also addressed the problem of loss of atomicity which occurs in multi-stage protocols such as *read-modify-write*, although our approach is rather coarse-grain.

6.1 Future Work

Our work opens up several avenues of potential future work. It would be interesting to implement our approach in a distributed manner using a communications protocol such as MPI. This could enable the possibility of higher speed-ups being achieved as the contention for critical sections could be reduced. It could be difficult to maintain the existing SystemC programming style.

We could also attempt to improve on our existing work by using finer-grain mutexes. For simplicity, our work contains relatively few mutexes, and there is potentially room for optimisation. However, such an attempt would have to be very careful in order to avoid introducing the possibility of deadlock or data corruption.

Finally, there is more work to do be done in the domain of language constructs to support parallelism in SystemC. Our work has established semantic constructs which are effective and natural in some cases. It would be interesting to pursue this research further, in order to find constructs that allow us to express even more cases in a natural way.

Bibliography

[1] Ensimag Engineering School. http://ensimag.grenoble-inp.fr/. 4.1

[2] IEEE Standards Association. IEEE 1666-2005 Standard SystemC Language Reference Manual. 2005. 2.2, 2.6

[3] Chopard B, Combes P, and Zory J. A conservative approach to systemC parallelization. In *International Conference on Computational Science*, pages 653–660, 2006. 3, 3.1, 3.4

[4] Yussef Bouzouzou. Accélération des simulations de systèmes sur puce au niveau transactionnel. Diplôme de recherche technologique, Université Joseph Fourier, 2007. 2.1, 2.1.1, 3.4

[5] Philippe Combes, Eddy Caron, Frédéric Desprez, Bastien Chopard, and Julien Zory. Relaxing synchronization in a parallel systemC kernel. In *International Symposium on Parallel and Distributed Processing with Applications*, pages 180–187. IEEE Computer Society, Washington, DC, USA, 2008. ISBN 978-0-7695-3471-8. URL `http://portal.acm.org/citation.cfm?id=1493613.1494180`. 3.1

[6] Giovanni Funchal and Matthieu Moy. ensitlm source code. https://github.com/moy/cours-tlm/tree/master/TPs/ensitlm. 4.1

[7] 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. (to appear). 3.3

[8] Kai Huang, Iuliana Bacivarov, Fabian Hugelshofer, and Lothar Thiele. Scalably distributed SystemC simulation for embedded applications. *International Symposium on Industrial Embedded Systems*, pages 271–274, June 2008. 3.1

[9] Open SystemC Initiative. TLM-2.0 Language Reference Manual. 2007. 2.2, 2.4.2, 3.1

[10] David R. Jefferson. Virtual time. *ACM Trans. Program. Lang. Syst.*, 7:404–425, July 1985. ISSN 0164-0925. URL `http://doi.acm.org.gate6.inist.fr/10.1145/3916.3988`. 3.2

[11] Leslie Lamport. Ti clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, July 1978. ISSN 0001-0782. URL `http://doi.acm.org/10.1145/359545.359563`. 3.2

[12] A. Mello, I. Maia, A. Greiner, and F. Pecheux. Parallel simulation of systemC tlm 2.0 compliant mpsoc on smp workstations. In *Design, Automation Test in Europe Conference Exhibition*, pages 606 –609, 2010. ISSN 1530-1591. 3.1

[13] Ezudheen P, Priya Chandran, Joy Chandra, Biju Puthur Simon, and Deepak Ravi. Parallelizing systemc kernel for fast hardware simulation on smp machines. In *Workshop on Principles of Advanced and Distributed Simulation*, PADS '09, pages 80–87. IEEE Computer Society, Washington, DC, USA, 2009. ISBN 978-0-7695-3713-9. URL `http://dx.doi.org/10.1109/PADS.2009.25`. 3.1

[14] Fred B. Schneider. Synchronization in distributed programs. *Trans. Program. Lang. Syst.*, 4:125–148, April 1982. ISSN 0164-0925. URL `http://doi.acm.org/10.1145/357162.357163`. 3.2

[15] Christoph Schumacher, Rainer Leupers, Dietmar Petras, and Andreas Hoffmann. parSC: synchronous parallel systemC simulation on multi-core host architectures. In *International conference on Hardware/software codesign and system synthesis*, CODES/ISSS '10, pages 241–246. ACM, New York, NY, USA, 2010. ISBN 978-1-60558-905-3. URL `http://doi.acm.org/10.1145/1878961.1879005`. 3.1, 3.4, 4.3

[16] P Wilsey, D Martin, and K Subramani. SAVANT/TyVIS/WARPED: Components for the analysis and simulation of vhdl, 1998. 3.2