Parallel Simulation of SystemC Loosely-Timed Transaction Level Models

Konstantinos Sotiropoulos
kisp@kth.se

May 2, 2016

Abstract

The vision of a connected and automated society, the Internet of Things era has promised, is depending on the industry's ability to design novel and complex electronic systems, while maintaining a short time to market. One of the first steps in the design of such systems is the in tandem simulation of hardware and software. Transaction Level Models, expressed in the SystemC modeling language, can facilitate this co-simulation. However, the sequential nature of the SystemC's Discrete Event simulation kernel is a limiting factor. Poor simulation performance often constraints the scope and depth of the design decisions that can be evaluated.

The increase in computing power, modern processing units deliver, is only available for applications that can expose parallel operations. The major obstacle one faces, when trying to parallelize a simulation, is the preservation of causality; simulation events need to be processed in a chronological order.

It is the main objective of this thesis' project to demonstrate the feasibility of parallelizing the simulation of Transaction Level Models, outside SystemC's reference simulation environment. The difficult task of achieving causal, yet parallel, processing of simulation events, is accomplished by using proper process synchronization mechanisms. Our proposed implementation does not depend on the presence of a centralized simulation moderator. It is implemented using the Message Passing Interface 3.0 framework. To demonstrate our approach and evaluate different process synchronization algorithms, we use the model of a cache-coherent, symmetric multiprocessor based on the OpenRisc 1000 Instruction Set Simulator. Our results indicate a significant speedup against the reference SystemC simulation.

Keywords: parallel discrete event simulation, conservative synchronization algorithms, transaction level models, SystemC TLM 2.0

Contents

Preface

This is a Master's Thesis project that will be carried out in Intel Sweden AB and is supervised by KTH's ICT department. Mr. Bjorn Runaker (bjorn.runaker@intel.com) is the project's supervisor from the company's side, while professor Ingo Sander (ingo@kth.se) and PhD student George Ungureanu (ugeorge@kth.se) are the examiner and supervisor from KTH. The project begun on 2016-01-16 and will finish on 2016-06-30, as dictated by the contract of employment that I, Konstantinos Sotiropoulos a Master's student at the Embedded Systems program, have signed with the company.

The scope of this project has been mutually agreed on and dialectically determined between the company's needs and the institute's research agenda. As Master's Thesis project, it will expose a scientific ground on which the engineering effort shall be rooted.

All the necessary equipment (software and hardware) has been kindly provided by the company. The exact legal context that will apply to any software produced as a result of this project is yet to be determined, but will conform to the general context dictated by the documents already signed (documents' titles: "Statement of Terms and Conditions of Fixed Term Employment" and "Employee Agreement") .

Acronyms

ASIC: Application Specific Integrated Circuit
DE: Discrete Event
DES: Discrete Event Simulator/Simulation
DMI: Direct Memory Interface
ES: Electronic System
ESLD: Electronic System-Level Design
FPGA: Field Programmable Gate Array
HDL Hardware Description Language
HPC: High Performance Computing
IC Integrated Circuit
MoC: Model of Computation
MPI Message Passing Interface
MPSoC: Multicore System on Chips
OoO: Out-of-Order
PDES: Parallel Discrete Event Simulation
SLDL: System-Level Design Language
SMP: Symmetric Multiprocessing
SoC: System on Chip
SR: Synchronous Reactive
TLM: Transaction Level Modeling
CMB: Chandy/Misra/Bryant algorithm

1 Introduction

The aim of this chapter is to present the general context of the problem statement. that is the engineering discipline of **Electronic System-Level Design (ESLD)**.

In unit 1.1 we provide a definition for the fundamental concepts of design, system, model and simulation. In units 1.2 to 1.4, using Gajski and Kuhn's Y-Chart, we determine the concept of a Transaction-Level Model, as an instance in the engineering practice of Electronic System-Level Design (ESLD). In unit 1.5 we have a rudimentary look on SystemC's role in ESLD. The structure of this document is given in unit 1.6.

1.1 The Design Process

We define the process of **designing** as the engineering art of incarnating a desired functionality into a perceivable, thus concrete, artifact. An engineering artifact is predominantly referred to as a **system**, to emphasize the fact that it can be viewed as a structured collection of components and that its behavior is a product of the interaction among its components.

Conceptually, designing implies a movement from abstract to concrete, fueled by the engineer's **design decisions**, incrementally adding implementation details. This movement is also known as the **design flow** and can be facilitated by the creation of an arbitrary number of intermediate artifacts called models. A **model** is thus an abstract representation of the final artifact. The design flow can be now semi-formally defined as a process of model refinement, with the ultimate model being the final artifact itself. We use the term semi-formal to describe the process of model refinement, because to the best of our knowledge, such model semantics and algebras that would establish formal transformation rules and equivalence relations are far from complete [1].

A desired property of a model is executability that is its ability to demonstrate portions of the final artifact's desired functionality in a controlled environment. An **executable model**, allows the engineer to form hypotheses, conduct experiments on the model and finally evaluate design decisions. It is now evident that executable models can firmly associate the design process with the scientific method. The execution of a model is also known as **simulation** [2].

1.2 Electronic Systems Design

An Electronic System (ES) provides a desired functionality, by manipulating the flow of electrons. Electronic systems are omnipotent in every aspect of human activity; most devices are either electronic systems or have an embedded electronic system for their cybernisis.

The prominent way for visualizing the ES design/abstraction space is by means of the Y-Chart. The concept was first presented in 1983 [3] and has been constantly evolving to capture and steer industry practices. Figure 1 presents the form of the Y-Chart found in [1].

The Y-Chart quantizes the design space into four levels of abstraction; system, processor, logic and circuit, represented as the four concentric circles. For each abstraction level, one can use different ways for describing the system; behavioral, structural and physical. These are represented as the three axises, hence the name Y-Chart. Models can now be identified as points in this design space.

A typical design flow for an Integrated Circuit (IC) begins with a high-level behavioral model capturing the system's specifications and proceeds non-monotonically to a lower level structural representation, expressed as a netlist of, still abstract, components. From there, Electronic Design Automation (EDA) tools will pick up the the task of reducing the abstraction of a structural model by translating the netlist of abstract components to a netlist of standard cells. The nature of the standard cells is determined by the IC's fabrication technology (FPGA, gate-array or standard-cell ASIC). Physical dimensionality is added by place and route algorithms, part of an EDA framework, signifying the exit from the design space, represented in the Y-Chart by the transition from the structural to the physical axis.

We have used the adjective non-monotonic to describe the design flow, because as a movement in the abstraction space, it is iterative; design $\rightarrow$ test/verify $\rightarrow$ redesign or proceed. This cyclic nature of

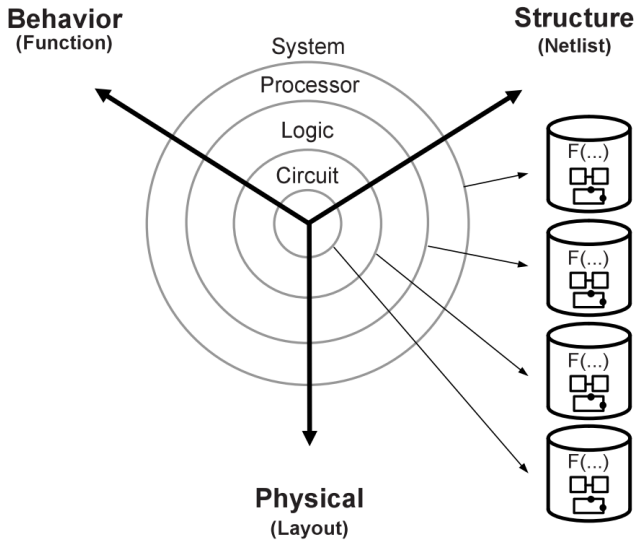

Figure 1: The Y-Chart (adopted from [1])

the design flow is implied by the errors the human factor introduces, under the lack of formal model transformation methodologies in the upper abstraction levels. The term **synthesis** is therefore introduced to describe a monotonic movement from a behavioral to a structural model, or the realization of an upper level structural model using finer components. We distinguish synthesis from the general case of the design flow, to disregard the testing and verification procedures. Therefore, the term synthesis may indicate the presence, or the desire of having, an automated design flow. Low-level synthesis is a reality modern EDA tools achieve, while high-level synthesis is still a utopia modern tools are converging to.

1.3 System-Level Design

To meet the increasing demand for functionality, ES complexity, as expressed by their heterogeneity and their size, is increasing. Terms like Systems on Chip (SoC) and Multi Processor SoC (MPSoC), used for characterizing modern ES, indicate this trend. With abstraction being the key mental ability for managing complexity, the initiation of the design flow has been pushed to higher abstraction levels. In the Y-Chart the most abstract level, depicted as the outer circle, is the system level. At this level the distinction between hardware and software is a mere design choice thus **co-simulation of hardware and software** is one of the main objectives. Thereby the term **system-level design** is used to describe design flows that enter the design space at this level.

A common practice among modern system-level design tools/methodologies, like Intel's CoFluent Studio [4], is for the designer to construct two intermediate models; An application model, that is the behavioral view of the system and a platform model, assembled using a component database of Processing Elements (PE, processors, hardware accelerators etc) and Communication Elements (CE, buses, interfaces etc). The final step towards **system-level synthesis**, that is the transition from a behavioral to a structural model on the system level, is called system mapping; the partitioning of the application to the elements of the platform.

1.4 Transaction-Level Model

A **Transaction-Level Model** (TLM) can now be defined as the point in the Y-Chart where the structural axis meets the system abstraction level. As mentioned in the previous unit, a TLM can be thought of as a platform model, or **virtual platform**, where an application can/is mapped [5]. It is the model that facilitates co-simulation of hardware and software. The notion of the transaction as an abstraction of communication will be clarified in chapter 4.

What are the pragmatic reasons that make the development of a virtual platform imperative? To begin with, an increasing amount of an ES's functionality is becoming software based. Moreover, ES related companies are facing the economical pressure of reducing new products' time to market. Thus, software engineers must be equipped with a virtual platform they can use for software development, early on in the design flow, without needing to wait for the actual silicon to arrive.

1.5 SystemC and TLM

One fundamental question, for completing the presentation of ESLD, remains; How can executable models be expressed on the system level? While maintaining the expressiveness of a Hardware Description Language (HDL), **SystemC** is meant to act as an **Electronic System Level Design Language** (ESLDL); a language with which system-level models can be expressed. It is implemented as a C++ class library, thus its main concern is to provide the designer with executable rather than EDA synthesizable models. The language is maintained and promoted by Accellera (former Open SystemC Initiative OSCI) and has been standardized (IEEE 1666-2011 [6]).

COMMENT: Why is SystemC regarded as Specific Domain Language (SDL)? In what way does SystemC provide support for Transaction Level Modeling? Through the TLM 1.0 and 2.0 API.

1.6 Document Overview

This unit be completed in the end

2 Formulating The Problem Statement

The aim of this chapter is to present a theoretical framework that will eventually lead to the formulation of the problem statement. Picking up Ariadne's thread from the introduction, this chapter begins its journey by the fact that SystemC is an Electronic System-Level Design **Language** (ESLDL) for expressing system-level models.

In unit 2.1 we link the concepts of operational semantics and Models of Computation (MoC) with that of the ESLDL. In units 2.2 and 2.3 the SystemC simulation engine or kernel is presented as an algorithm that realizes the operational semantics of a Discrete Event (DE) MoC. Units 2.4 and 2.5 introduce the concept of Parallel Discrete Event Simulation (PDES) and present the fundamental causality hazards it introduces. The prime concern of this thesis' is presented in a concise way in 2.6. Unit 2.7 introduces the objectives, that is the engineering endeavor of this project.

2.1 Models of Computation

A **language** is a set of symbols, rules for combining them (its syntax), and rules for interpreting combinations of symbols (its semantics). Two approaches to semantics have evolved: denotational and operational. **Operational semantics**, which dates back to Turing machines, gives the meaning of a language in terms of actions taken by some abstract machine. How the abstract machine in an operational semantics can behave is a feature of what we call the **Model of Computation (MoC)** [7]. This definition implies that languages are not computational models themselves, but have underlying computational models [8].

How does the concept of a MoC fit specifically in ESLDLs? Above all the engineer needs executable models. Furthermore, an ESLDL describes an electronic artifact as a system; a (hierarchical) network of interacting components. Therefore, a MoC is a collection of rules to define what constitutes a component and what are the semantics of execution, communication and concurrency of the abstract machine that will execute the model [8] [2]. To ensure meaningful simulations, the MoC of the abstract machine that simulates a model must be equivalent with that of the abstract machine that will realize the system.

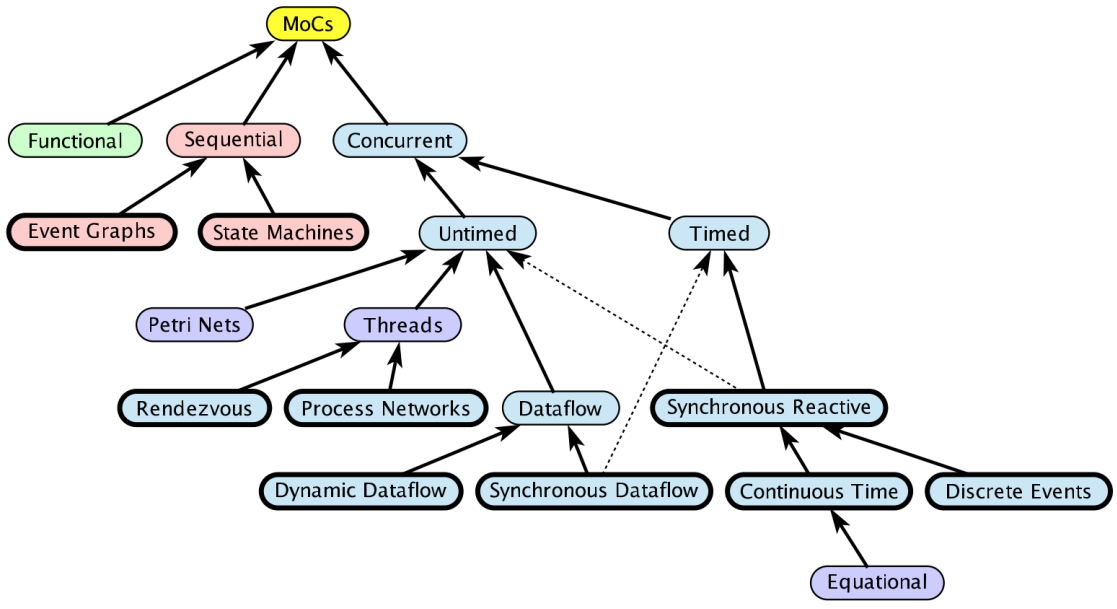

Figure 2: Categorization of three of the most explored MoCs: State Machine, Synchronous Dataflow and Discrete Event(adopted from [2])

2.2 Discrete Event Model of Computation

The dominant MoC that underlies most industry standard HDLs (VHDL, Verilog, SystemC) is the **Discrete Event (DE)** MoC. The components of a DE system are called **processes**. In this context processes usually model the behavior and functionality of hardware entities. The execution of processes is concurrent and the communication is achieved through **events**. An event can be considered as a time-stamped value.

Concurrent execution does not imply parallel/simultaneous execution. The notion of **concurrency** is more abstract. Depending on a machine's computational resources, it can be realized as either parallel/simultaneous execution or as sequential interleaved execution.

Systems whose semantics are meant to be interpreted by a DE MoC, in order to be realizable, must have a **causal** behavior: they must process events in a chronological order, while any output events produced by a process are required to be no earlier in time than the input events that were consumed [2]. At any moment in real time, the model's time is determined by the last event processed.

In figure 2 one can observe that the DE MoC is also considered to be **Synchronous-Reactive (SR)**. This demonstrates the possibility of the MoC to "understand" entities with zero execution time, where output events are produced at the same time input events are consumed. We can also extend/rephrase the previous definitions and say that Synchronous-Reactive MoCs are able to handle, in a causal way, systems where events happen at the same time, instantaneously. The DE MoC handles the aforementioned situations by extending time-stamps(the notion of simulated time) with the introduction of delta delays (also referred to as cycles or micro-steps). A delta delay signifies an infinitesimal unit of time and no amount of delta delays, if summed, can result in time progression. A time-stamp is therefore represented as a tuple of values, (t, n) where t indicates the model time and n the number of delta delays that have advanced at t.

2.3 Discrete Event Simulation(or)

A realization of the DE abstract machine is called a **Discrete Event Simulator (DES)**. SystemC's reference implementation of the DES is referred to as the **SystemC kernel** [6].

Concurrency of the system's processes is achieved through the co-routine mechanism (also known as co-operative multitasking). Processes execute without interruption. In a single core machine that means that only a single process can be running at any (real) time, and no other process instance can execute until the currently executing process instance has yielded control to the kernel. A process shall not preempt or interrupt the execution of another process [6].

To avoid quantization errors and the non-uniform distribution of floating point values, time is expressed as an integer multiple of a real value referred to as the time resolution.

The kernel maintains a **centralized event queue** that is sorted by time-stamp and knows which process is **running**, which are **runnable**, and which processes are waiting for events. Runnable processes have had events to which they are sensitive triggered and are waiting for the running process to yield to the kernel so that they can be scheduled. The kernel controls the execution order by selecting the earliest event in the event queue and making its time-stamp the current simulation time. It then determines the process the event is destined for, and finds all other events in the event queue with the same time-stamp that are destined for the same process [9]. The operation of the kernel is exemplified in listing 1.

Algorithm 1 SystemC event loop, adopted from [10]

```
1:  while timed events to process exist do                ▷ Simulation time progression
2:      trigger events at that time
3:      while runnable processes exist do                 ▷ Delta cycle progression
4:          while runnable processes exist do
5:              run all triggered processes
6:              trigger all immediate notifications
7:          end while
8:          update values of changed channels
9:          trigger all delta time events
10:     end while
11:     advance time to next event time
12: end while
```

2.4 Parallel Discrete Event Simulation(or)

The previous section has made evident that the reference implementation of the SystemC kernel assumes sequential execution and therefore can not utilize modern massively parallel host platforms. The most logical step in achieving faster simulations is to realize concurrency, from interleaved process execution to simultaneous/parallel execution. By assigning each process to a different processing unit of the host platform (core or hardware thread) we enter the domain of **Parallel Discrete Event Simulation (PDES)**. After making the strategical decision that for improving a DE simulator's performance one must orchestrate parallel execution, the first tactical decision encountered is whether to keep a single simulated time perspective, or distribute it among processes.

For PDES implementations that enforce global simulation time, the term **Synchronous PDES** has been coined in [10]. In Synchronous PDES, parallel execution of processes is performed within a delta cycle. With respect to listing 1, we can say that a Synchronous PDES parallelizes the execution of the innermost loop (line 4). However, as we will see in later sections, this approach will bare no fruits in the simulation of TLM Loosely Timed simulations, since delta cycles are never triggered [11]. Therefore, we switch our interest in **Out-of-Order PDES (OoO PDES)** [12]; allowing each process to have its own perception of simulated time, determined by the last event it received.

2.5 Causality and Synchronization

The distribution of simulation time opens Pandora's box. Protecting the OoO PDES from **causality errors** demands certain assumptions and the addition of complex implementation mechanisms.

The first source of causality errors arises when the system's state variables are not distributed, in a disjoint way, among the processes [13]. A trivial realization of the above scenario is depicted in figure 3. Processes P_1 and P_2 are executing simultaneously, while sharing the system's state variable x. Events E_1 and E_2 are executed by P_1 and P_2 respectively. If we assume that in real time E_2 is executed before E_1, then we have implicitly broken causality, since E_1 might be influenced by the value of x that the execution of E_2 might have modified. Furthermore, one must observe that this kind of implicit interaction between P_1 and P_2 can not be expressed in a DE MoC. This is a meta-implication of the host platform's shared memory architecture.

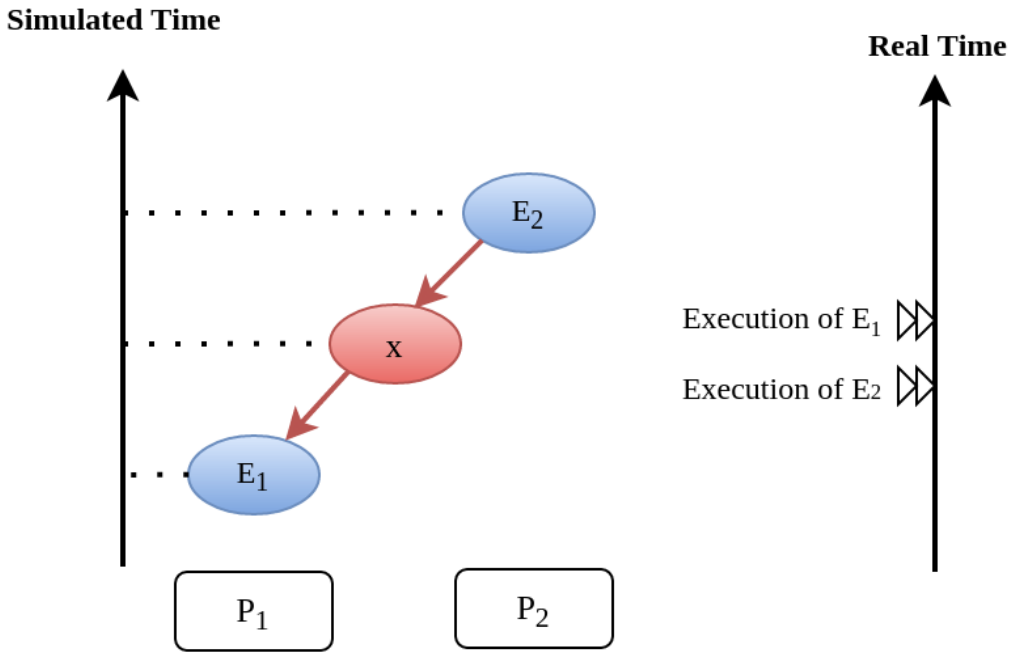

Figure 3: Causality error caused by the sharing of the system's state variable x by P_1 and P_2.

The second and most difficult to deal with source of causality errors is depicted in figure 4. Event E_1 affects E_2 by scheduling a third event E_3 which, for the sake of argument, modifies the state of P_2. This scenario necessitates sequential execution of all three events. Thus the fundamental problem in PDES, in the context of this scenario, becomes the question: how can we deduce that it is safe to execute E_2 in parallel with E_1, without actually executing E_1 [13]? However, one must notice that the kind of interaction that yields this problematic situation is explicitly stated in the model.

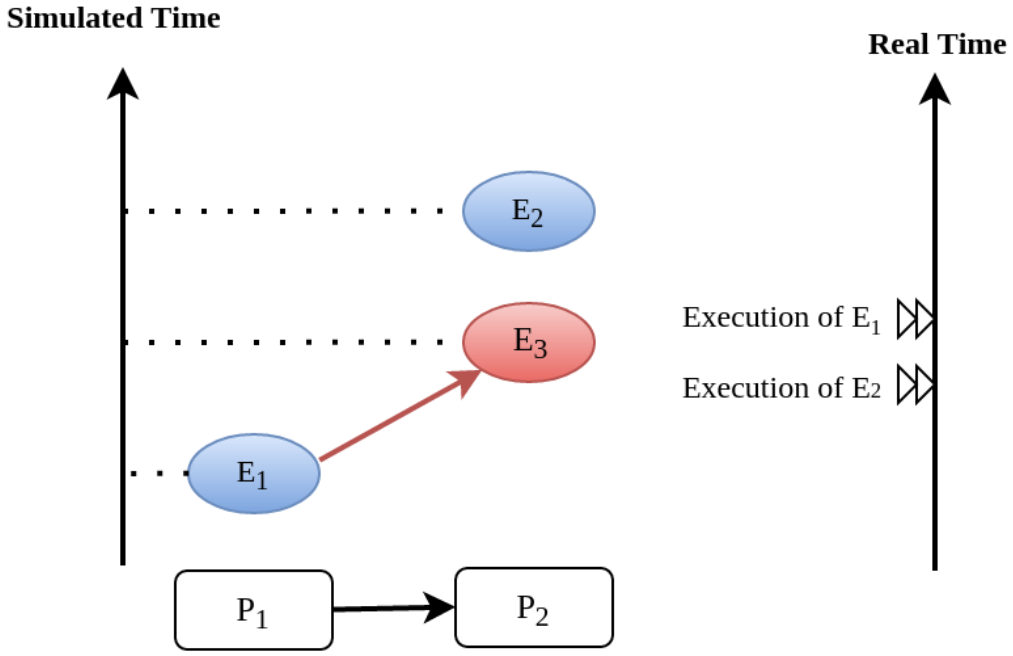

Figure 4: Causality error caused by the unsafe execution of event E_2 (adopted from [13]).

The last example makes evident the fact that the daunting task of preserving causality in the simulation is all about **process synchronization**. For example, each process must be able to communicate to each of its peers (processes that is linked with) the message: "I will not send you any event before t_1, so you can proceed with processing any event you have with time-stamp t_2 where $t_2 < t_1$".

OoO PDES synchronization algorithms, with respect to how they deal with causality errors, have been classified into two categories: **conservative** and **optimistic** [14]. Conservative mechanisms strictly avoid the possibility of any causality error ever occurring by means of model introspection and static analysis. On the other hand, optimistic/speculative approaches use a detection and recovery approach: when causality errors are detected a rollback mechanism is invoked to restore the system. An optimistic compared to a conservative approach will theoretically yield better performance in models where communication, thus the probability of causality errors, is below a certain threshold [13].

Both groups present severe implementation difficulties. For conservative algorithms, model introspection and static analysis tools might be very difficult to develop, while the rollback mechanism of an optimistic algorithm may require complex entities, such as a hardware/software transactional memory [15] .

2.6 Problem statement

The prime concern of this project can now be stated; an evaluation of the efficiency of existing conservative process synchronization algorithms when applied to the parallel simulation of Loosely-Timed Transaction Level Models.

2.7 Objectives

If the timing constraints stretched beyond the scope of a Master Thesis, the project's self-actualization would require the development/production of the following components (sorted in descending significance order):

1. At least two OoO PDE simulation mechanisms implementing proposed conservative synchronization algorithms.
2. A proof of concept application of the proposed mechanism, on a sufficiently parallel TLM model.
3. A static analysis/introspection tool for parsing the SystemC description of the model and extracting a pure representation in XML.
4. A code generation tool for realizing the model outside SystemC.

For the critical task of analyzing the model, identifying the processes and the links between them, we will follow ForSyDe SystemC's approach [16]. Using SystemC's well defined API for module hierarchy (e.g. `get_child_objects()`), along with the introduction of meta objects, the system's structure can be serialized at runtime, in the pre simulation phase of elaboration.

Given the time constraints, the primary focus falls on the first two objectives. The automation and generality the tools could deliver will be emulated by manual and ad-hoc solutions.

COMMENT: Your thesis' value (to external parties) depends highly on delivering point 4.

3 Out of Order PDES with MPI

The goal of this chapter is to present two conservative process synchronization algorithms and give their implementation using the MPI API.

In units 3.1 and 3.2 we present the conservative synchronization algorithms that will be evaluated. In unit `Semantics of point-to-point communication in MPI` and 3.4 we present the semantics of the Message Passing Interface (MPI) communication primitives. In unit `MPI Realization of CMB` we provide pseudo code for the realization of the CMB using the MPI communication primitives. In unit 3.7 we give an overview of prior art in the field of PDES in ESLD.

3.1 The Chandy/Misra/Bryant synchronization algorithm

The first conservative synchronization algorithm that will be examined originate from the work of **Chandy/Misra/Bryant (CMB)** [17] [18]. Listing 2 demonstrates how the algorithm deals with the fundamental dilemma presented in section 2.6, figure 4. Events arriving on each incoming link can be stored in a first-in-first-out (FIFO) queue.

Algorithm 2 Process event loop, adopted from [19]

```
   while process clock < some T do
2:     Block until each incoming link queue contains at least one event
       remove event M with the smallest time-stamp from its queue.
4:     set clock = time-stamp(M)
       process event M
6: end while
```

This naive realization of the individual process' event loop, however, leads to deadlock situations like the one depicted in figure 5. The queues placed along the red loop are empty, thus simulation has halted, even though there are pending events (across the blue loop).

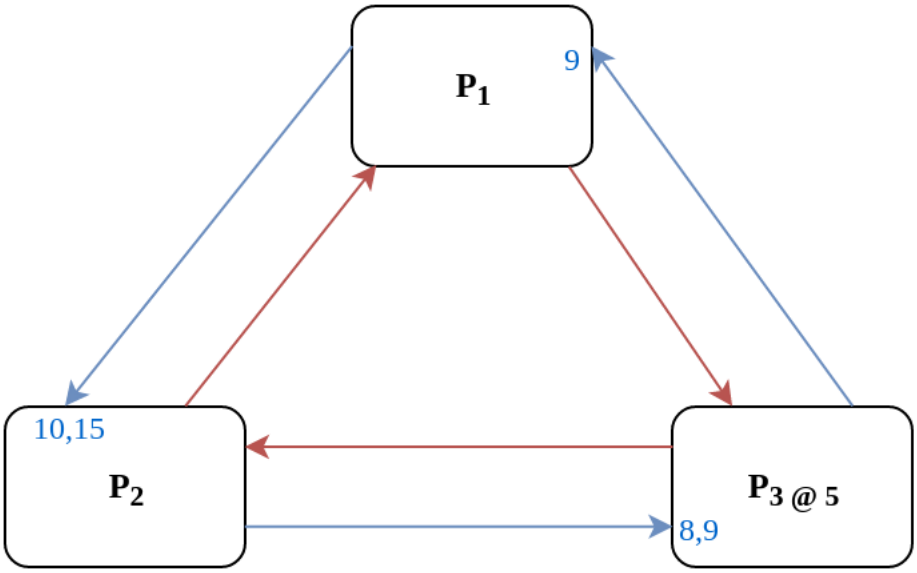

Figure 5: adopted from [19]

The deadlock avoidance mechanism that lies in the core of the CMB algorithm can be demonstrated with the following example: Let us assume that P_3 is at time 5. Furthermore, let us assume that we have the **a priori** knowledge that P_3 has a minimum event processing time of 3 (simulated). We will call this knowledge **lookahead**. P_3 could create a **null event**, with no data value, but with a time-stamp t(8) = clock(5) + lookahead(3) and place it on its outgoing links. A null event is still an event, so P_2 by processing

it would advance its clock to 8. In the same fashion, let us assume that P_2 has a lookahead of 2 and upon processing P_3's null event, it will generate a null event for P_1 with time-stamp 10. Eventually P_1 can now safely process the actual event with time-stamp 9, thus unfreezing the simulation.

Thus, the modified, for deadlock avoidance, algorithm is described in listing 3. The important points one must notice with this deadlock avoidance mechanism are that:

- Null events are created when a process updates its clock, that is upon processing an event.
- Each process propagates null events on all of its outgoing links.
- The efficiency of this mechanism is highly dependent on the designer's ability to determine sufficiently large lookaheads. The lookahead is not necessary a fixed value. It can be a function of the process' state and/or the simulation time.

Algorithm 3 Process event loop, with deadlock avoidance, adopted from [19]

while process clock < some T **do**
2: **Block** until each incoming link queue contains at least one event
remove event M with the smallest time-stamp from its queue.
4: set clock = time-stamp(M)
process event M
6: send either a null or meaningful event to each outgoing link L with time-stamp = clock + Lookahead(clock,L,...)
end while

COMMENT: This is a rather big unit. You should consider restructuring the material in a couple of shorter units. Are there any formal proofs about the properties (deadlock free, causality) of this algorithm?

3.2 On Demand Synchronization

The principal disadvantage of the CMB algorithm is that a large number of null events can be generated, particularly if the lookahead is small [19]. An alternative approach to sending a null event after processing each event is a demand-driven approach. Whenever a process is about to become blocked because an incoming link is empty, it requests an event (null or otherwise) from the process on the sending side of the link. The process resumes execution when the response to this request is achieved.

COMMENT: The description of this algorithm is not complete.

3.3 Semantics of point-to-point Communication in MPI

The framework chosen for implementing the PDES is the **Message Passing Interface** 3.0 (MPI). Events are modeled as structured messages, while event diffusion/communication as message passing. MPI is a message passing library interface specification, standardized and maintained by the Message Passing Interface Forum [4]. It is currently available for C/C++, FORTRAN and Java from multiple vendors (Intel, IBM, OpenMPI) [4]. MPI addresses primarily the message passing parallel programming model, in which data is moved from the address space of one process to that of another process through cooperative operations on each process [20].

The basic communication primitives are the functions `MPI_Send(...)` and `MPI_Recv(...)`. Their arguments specify, among others things, a data buffer and the peer process' or processes' unique id assigned by the MPI runtime. By default, message reception is blocking, while message transmission may or may not block. One can think of message transfer as consisting of the following three phases

1. Data is pulled out of the send buffer and a message is assembled
2. A message is transferred from sender to receiver

3. Data is pulled from the incoming message and disassembled into the receive buffer

Order: Messages are non-overtaking. If a sender sends two messages in succession to the same destination, and both match the same receive (a call to `MPI_Recv`), then this operation cannot receive the second message if the first one is still pending. If a receiver posts two receives in succession, and both match the same message, then the second receive operation cannot be satisfied by this message, if the first one is still pending. This requirement facilitates matching of sends to receives and also guarantees that message passing code is deterministic.

Fairness: MPI makes no guarantee of fairness in the handling of communication. Suppose that a send is posted. Then it is possible that the destination process repeatedly posts a receive that matches this send, yet the message is never received, because it is each time overtaken by another message, sent from another source. It is the programmer's responsibility to prevent starvation in such situations.

COMMENT: Why did you choose MPI?

3.4 MPI Communication Modes

The MPI API contains a number of variants, or modes, for the basic communication primitives. They are distinguished by a single letter prefix (e.g. `MPI_Isend(...)`, `MPI_Irecv(...)`). As dictated by the MPI version 3.0, the following communication modes are supported [20]:

No-prefix for standard mode: `MPI_Send(...)` In this mode, it is up to MPI to decide whether outgoing messages will be buffered. MPI may buffer outgoing messages. In such a case, the send call may complete before a matching receive is invoked. On the other hand, buffer space may be unavailable, or MPI may choose not to buffer outgoing messages, for performance reasons. In this case, the send call will not complete, blocking the transmitting process, until a matching receive has been posted, and the data has been moved to the receiver.

B for buffered mode: `MPI_Bsend(...)` A buffered mode send operation can be started whether or not a matching receive has been posted. It may complete before a matching receive is posted. However, unlike the standard send, this operation is local, and its completion does not depend on the occurrence of a matching receive. Thus, if a send is executed and no matching receive is posted, then MPI must buffer the outgoing message, so as to allow the send call to complete. A buffered send operation that cannot complete because of a lack of buffer space is erroneous. When such a situation is detected, an error is signaled that may cause the program to terminate abnormally. On the other hand, a standard send operation that cannot complete because of lack of buffer space will merely block, waiting for buffer space to become available or for a matching receive to be posted. This behavior is preferable in many situations. Consider a situation where a producer repeatedly produces new values and sends them to a consumer. Assume that the producer produces new values faster than the consumer can consume them. If buffered sends are used, then a buffer overflow will result. Additional synchronization has to be added to the program so as to prevent this from occurring.

S for synchronous mode: `MPI_Ssend(...)` A send that uses the synchronous mode can be started whether or not a matching receive was posted. However, the send will complete successfully only if a matching receive is posted, and the receive operation has started to receive the message sent by the synchronous send. Thus, the completion of a synchronous send not only indicates that the send buffer can be reused, but it also indicates that the receiver has reached a certain point in its execution, namely that it has started executing the matching receive. If both sends and receives are blocking operations then the use of the synchronous mode provides synchronous communication semantics: a communication does not complete at either end before both processes **rendezvous** at the communication point.

R for ready mode: `MPI_Rsend(...)` A send that uses the ready communication mode may be started only if the matching receive is already posted. Otherwise, the operation is erroneous and its outcome is undefined. Ready sends are an optimization when it can be guaranteed that a matching receive has already been posted at the destination. On some systems, this allows the removal of a hand-shake operation that is otherwise required and results in improved performance. A send operation that uses the ready mode has the same semantics as a standard send operation, or a synchronous send operation; it is merely that the sender

provides additional information to the system (namely that a matching receive is already posted), that can save some overhead.

I for non-blocking mode: `MPI_Isend(...)`, `MPI_Ibsend(...)`, `MPI_Issend(...)` and `MPI_Irecv(...)` Non-blocking message passing calls return control immediately (hence the prefix I), but it is the user's responsibility to ensure that communication is complete, before modifying/using the content of the data buffer. It is a complementary communication mode that works en tandem with all the previous. The MPI API contains special functions for testing whether a communication is complete, or even explicitly waiting until it is finished.

3.5 MPI realization of CMB

Using MPI's communication primitives, listing 4 provides a pseudo code describing the CMB process event loop.

Algorithm 4 CMB Process event loop in MPI

```
    while process clock < some T do
2:      post a MPI_Irecv on each incoming peer process
        post a MPI_Wait: block until every receive has been completed
4:      save each message received in a separate, per incoming link, FIFO.
        identify message M with the smallest time-stamp
6:      set clock = time-stamp(M)
        process message M
8:      post a MPI_Issend to each outgoing link L with time-stamp = clock + Lookahead(clock,L,...)
    end while
```

3.6 Evaluation Metrics

The first evaluation metric of the proposed PDES implementation will be its performance against the reference SystemC kernel. It will be measured by experimentation on the project's use case.

The simulation's size can be easily related to the duration of the simulation (in simulated time). Another way of describing the simulation's size is through the conception of a formula involving the number of system processes, the number of links, the system's topology and the amount of events generated.

The accuracy of the simulation can be measured by the aggregate number of causality errors. The detection of causality errors must be facilitated in a per process level and the aggregation shall be performed at the end of the simulation. A concrete realization of the accuracy metric comes in the form of a counter each process increments whenever it executes an event with a time-stamp lower than its clock (the time-stamp of the last processed event). Ideally, if the synchronization algorithms have been realized correctly, no causality errors should be detected.

COMMENT: This section will become more concrete when we start experimentation.

3.7 Existing PDES

The most important:
RISC: Recoding infrastructure for SystemC [21].
Miscellaneous:
SystemC-SMP [22]
SpecC [23], although the latter is not meant for SystemC.
sc_during [24]

COMMENT: This section is incomplete that should not be incomplete in an Intermediate report. Are you reinventing the wheel? Did you try at least one of these tools?

4 SystemC TLM 2.0

It is beyond the scope of this project to provide a comprehensive guide to system-level modeling in SystemC TLM 2.0. However, at the time of writing and to the best of our knowledge, we can not verify the existence of a comprehensive guide about system-level modeling with SystemC TLM 2.0. Hence, we fill obliged to provide a quick introduction into the SystemC TLM 2.0 Loosely-Timed (LT) coding style, by means of a simple example. The chapter assumes a basic understanding of C++ and SystemC.

In unit 4.1 we enumerate the features of the SystemC TLM 2.0 API. In units 4.2 and 4.3 we have a look at the fundamental notions of transaction, initiator and target components, socket and generic payload. In unit 4.4 we present the two coding styles (Loosely Timed and Approximately Timed) and give their typical use cases. In unit 4.5 we provide the implementation of a simple initiator, interconnect and target model. In unit 4.6 we present the dominant source of criticism for TLM 2.0. Finally, in unit 4.7 we provide a comparison between the dominant industry frameworks for ESLD, Simics and SystemC TLM.

4.1 Overview of SystemC TLM 2.0 API

As stated in unit 1.4, a Transaction Level Model is considered a virtual platform where an application can/is mapped. A **virtual platform** is a fully functional software model of a complete system, typically used for software development in the absence of hardware, or prior to hardware being available. To be suitable for productive software development it needs to be fast, booting operating systems in seconds, and accurate enough such that code developed using standard tools on the virtual platform will run unmodified on real hardware. [25].

The TLM 2.0 API enhances SystemC's expressiveness in order to facilitate the description and fast simulation of virtual platforms. TLM 2.0 allows **IP interoperability** for the rapid development of fast virtual platforms and facilitate the simulation under a reference simulation kernel, that of SystemC.

TLM 2.0 API [26] consists of the following features (graphically depicted in 6):

- A set of core interfaces
  - A Blocking interface which is coupled with the **Loosely-Timed (LT)** coding style.
  - A non-blocking interface, which is coupled with the **Approximately-Timed (AT)** coding style.
  - The **Direct Memory Interface (DMI)** to enable an initiator to have direct access to a target's memory, bypassing the usual path through the interconnect components used by the transport interfaces.
  - The **Debug transport interface** to allow an non-intrusive inspection of the system's state.
- The **global quantum** used by the **temporal decoupling** mechanism of the LT coding style, which facilitates faster simulations by reducing the number of context switches performed by the kernel.
- Initiator and target **sockets** to denote the links (causal dependencies) between processes.
- The **generic payload** which supports the abstract modeling of memory-mapped buses.
- A set of **utilities**, in the form of pre configured sockets and interconnect components, to facilitate the rapid development of models.

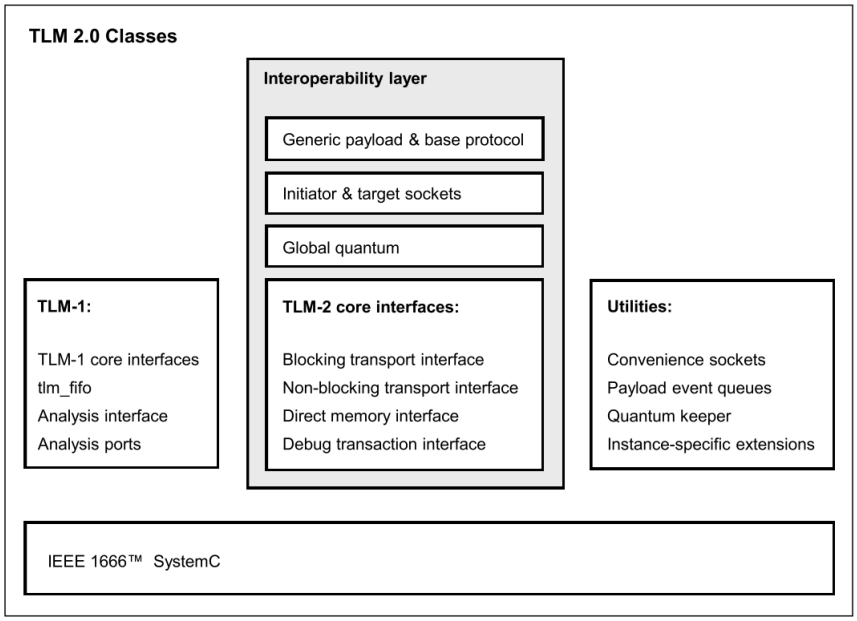

Figure 6: TLM 2.0 use cases (adopted from [26]).

4.2 Transactions, Sockets, Initiators and Targets

Transactions are non-atomic communications, normally with bidirectional data transfer, and consist of a set of messages that are usually modeled as atomic communications. In a transaction one can distinguish two roles; the **initiator**, the component which initiated the communication, and the **target**, the component which is supposed to service the initiator's request. A component is not limited to either of these two roles; it can assume both. For example, **interconnect** components encapsulate the behavior of memory-mapped buses, being responsible for routing transactions to the correct target. From the initiator's perspective, they act as targets and from the target's perspective they act as initiators.

Implementation-wise, communication in TLM 2.0 is reduced to method calls, from the initiator to the target through an arbitrary number of interconnect component, without involving any context switches from the simulation kernel.

A component's role is signified by the type of **sockets** it contains. Initiator sockets are used to forward method calls "up and out of" a component, while target sockets are used to allow method calls "down and into" a component. Socket binding is the act of connecting components together, thus defining the component whose method call will be eventually executed to service the transaction. From SystemC's viewpoint, a socket is basically a convenience class, wrapping a `sc_export` and a `sc_port`.

COMMENT: Maybe explain in more detail SystemC's export and port mechanisms? Maybe you need to adopt a more SystemC like terminology? For example change the word "component" to "module".

4.3 Generic Payload

The basic argument that is passed, by reference, in communicative method calls is called the **generic payload**. It is a structure that contains all the necessary information about the transaction. It supports the abstract modeling of memory-mapped buses, together with an extension mechanism to support the modeling of specific bus protocols whilst maximizing interoperability.

The main features/fields of the generic payload are:

- Command Is it read or write?
- Address What is the address, who is supposed to serve the transaction.
- Data A pointer to the physical data as an array of bytes.

- Phase Since a transaction is a non-atomic operation, this indicates the stage of the transaction. It is used for a detailed modeling of communication protocols.
- Response An enumeration, indicating whether the transaction was successful, and if not, what is the nature of the error.

4.4 Coding Styles

LT is suited for describing virtual platforms intended for software development. However, where additional timing accuracy is required, typically for software performance estimation and architectural analysis use cases, the AT style is employed. Virtual platforms typically do not contain many cycle-accurate models of complex components because of the performance impact.

COMMENT: This is a quite problematic section. You need to elaborate more, do not forget LT is on your thesis title.

4.5 An Example

This unit will provide a literate code listing for the model in figure 7

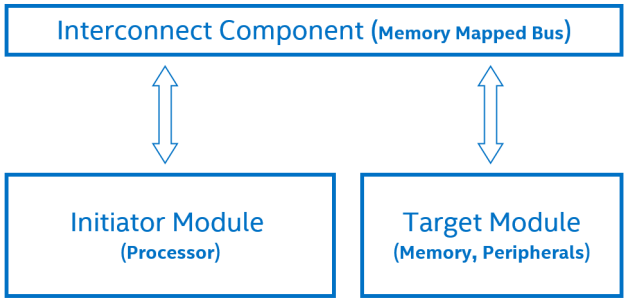

Figure 7: A simple system-level model. The initiator, for example, could model a processor, the interconnect component a memory bus and the target a memory.

4.6 Criticism

The root problem with TLM 2.0 lies in the elimination of explicit channels, which were a key contribution in the early days of research on system-level design. As most researchers agreed, the concept of separation of concerns was of highest importance, and for system-level design in particular, this meant the clear separation of computation (in behaviors or modules) and communication (in channels). Regrettably, SystemC TLM 2.0 chose to implement communication interfaces directly as sockets in modules and this indifference between channels and modules thus breaks the assumption of communication being safely encapsulated in channels. Without such channels, there is very little opportunity for safe parallel execution [21].

For the above reason some designers consider TLM 2.0 a step towards the wrong direction and revert back to TLM 1.0. Do you agree with this trend? Maybe tell us the major difference with TLM 1.0?

This is why SystemC TLM 2.0 model needs to be **recoded** to allow parallel execution. The recoding must reconstitute the separation of concerns between computation and communication. A modification of just the kernel will not suffice.

4.7 Simics and TLM 2.0

Everything you do with SystemC TLM 2.0 you can do with Simics. Simics is the main alternative to SystemC TLM 2.0 for system-level design.

COMMENT: Can you briefly outline the differences between the two tools/frameworks? Is Simics capable of PDES?

5 Use Case

In this chapter we describe the transaction level model we are going to use for conducting our experimentation. The purpose of the experimentation is twofold; verify whether we achieve better faster simulation compared to the reference SystemC kernel and evaluate the proposed process synchronization algorithms.

5.1 Platform modeling

A block diagram of the platform that will be modeled is seen in figure 8. The platform is a shared memory, cache-coherent, symmetric multiprocessor system based on the OpenRisc 1200 Instruction Set Simulator. Cache coherence is enforced by a directory residing in the inclusive L2 cache. Every component is/will be implemented in C/C++ and wrapped in SystemC modules using the TLM 2.0 API for communication. The exact number of processors is yet to be determined.

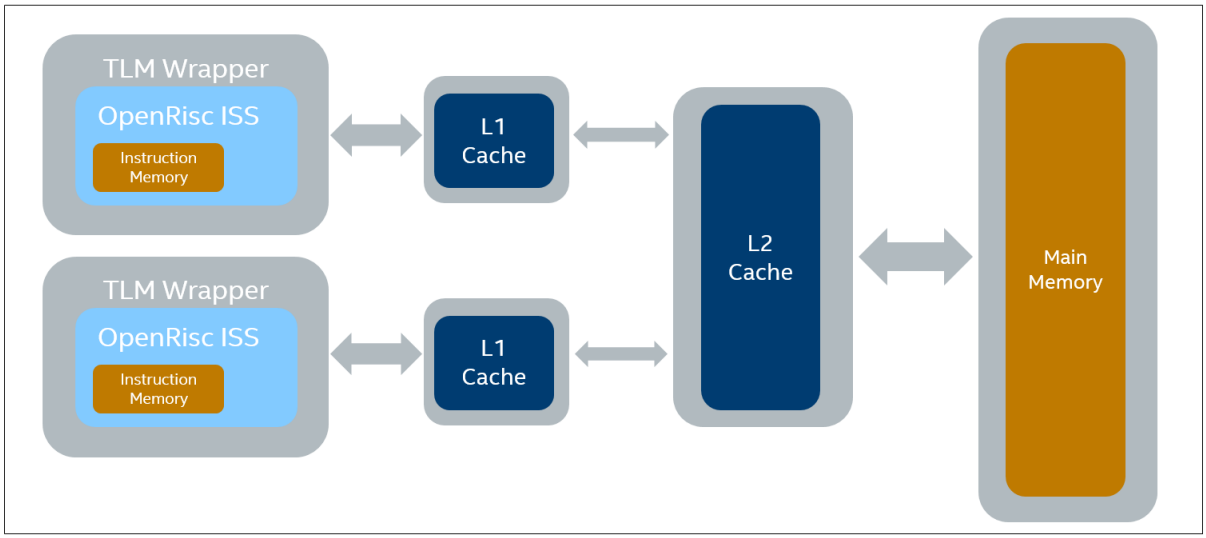

Figure 8: A model of a shared memory, cache-coherent, symmetric multiprocessor system

COMMENT: Can you be more specific about the cache coherence protocol? Maybe provide a state diagram?

5.2 Application modeling

We have the bare metal (newlib based) toolchain for compiling applications for the OpenRisc ISS.

COMMENT; What kind of application am I going to run on this platform? I see that most of the papers out there do some kind of mpeg2 decoding. That seems complex.

6 References

[1] D. D. Gajski, S. Abdi, A. Gerstlauer, and G. Schirner, *Embedded System Design*. Boston, MA: Springer US, 2009. ISBN 978-1-4419-0503-1. [Online]. Available: http://link.springer.com/10.1007/978-1-4419-0504-8

[2] C. Ptolemaeus, Ed., *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014. [Online]. Available: http://ptolemy.org/books/Systems

[3] Gajski and Kuhn, "Guest Editors' Introduction: New VLSI Tools," *Computer*, vol. 16, no. 12, pp. 11–14, dec 1983. doi: 10.1109/MC.1983.1654264. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1654264

[4] citation needed, *citation needed*.

[5] S. Rigo, R. Azevedo, and L. Santos, Eds., *Electronic System Level Design*. Dordrecht: Springer Netherlands, 2011. ISBN 978-1-4020-9939-7. [Online]. Available: http://link.springer.com/10.1007/978-1-4020-9940-3

[6] Open SystemC Initiative, *1666-2011 - IEEE Standard for Standard SystemC Language Reference Manual*, IEEE Std., 2012. [Online]. Available: http://ieeexplore.ieee.org/servlet/opac?punumber=6134617

[7] S. Edwards, L. Lavagno, E. Lee, and A. Sangiovanni-Vincentelli, "Design of embedded systems: formal models, validation, and synthesis," *Proceedings of the IEEE*, vol. 85, no. 3, pp. 366–390, mar 1997. doi: 10.1109/5.558710. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=558710

[8] A. Jantsch and I. Sander, "Models of computation in the design process," 2005. [Online]. Available: http://people.kth.se/~ingo/Papers/IEE2005-Book.pdf

[9] D. C. Black, J. Donovan, B. Bunton, and A. Keist, *SystemC: From the Ground Up*, 2nd ed. Boston, MA: Springer US, 2010, no. 1. ISBN 978-0-387-69957-8. [Online]. Available: http://link.springer.com/10.1007/978-0-387-69958-5

[10] C. Schumacher, R. Leupers, D. Petras, and A. Hoffmann, "parSC: Synchronous Parallel SystemC Simulation on Multi-Core Host Architectures," in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis - CODES/ISSS '10*. New York, USA: ACM Press, 2010. doi: 10.1145/1878961.1879005. ISBN 9781605589053 p. 241. [Online]. Available: http://dl.acm.org/citation.cfm?id=1878961.1879005

[11] W. Chen, X. Han, C.-w. Chang, and R. Doemer, "Advances in Parallel Discrete Event Simulation for Electronic System-Level Design," *IEEE Design & Test of Computers*, vol. 30, no. October 2012, pp. 1–1, feb 2012. doi: 10.1109/MDT.2012.2226015. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6338368

[12] W. Chen, *Out-of-order Parallel Discrete Event Simulation for Electronic System-level Design*. Cham: Springer International Publishing, 2015. ISBN 978-3-319-08752-8. [Online]. Available: http://link.springer.com/10.1007/978-3-319-08753-5

[13] R. M. Fujimoto, "Parallel discrete event simulation," *Communications of the ACM*, vol. 33, no. 10, pp. 30–53, oct 1990. doi: 10.1145/84537.84545. [Online]. Available: http://portal.acm.org/citation.cfm?doid=84537.84545

[14] ——, "Parallel and Distributed Simulation," in *Proceedings of the 2015 Winter Simulation Conference*, 2015. ISBN 978-1-4673-9743-8/15. [Online]. Available: http://www.informs-sim.org/wsc15papers/004.pdf

[15] A. Anane and E. M. Aboulhamid, "A Transaction-Based Environment for System Modeling and Parallel Simulation," *International Journal of Parallel Programming*, vol. 43, no. 1, pp. 24–58, feb 2015. doi: 10.1007/s10766-013-0303-4. [Online]. Available: http://link.springer.com/10.1007/s10766-013-0303-4

[16] S. Hosein, A. Niaki, M. K. Jakobsen, T. Sulonen, I. Sander, and D.-d. Oy, "Formal Heterogeneous System Modeling with SystemC," *Forum on Specification and Design Languages (FDL). IEEE*, pp. 160–167, 2012. [Online]. Available: http://kth.diva-portal.org/smash/record.jsf?pid=diva2%3A586216&dswid=-7079

[17] R. E. Bryant, "Simulation of packet communication architecture computer systems," Cambridge, MA, USA, Tech. Rep., 1977. [Online]. Available: http://dl.acm.org/citation.cfm?id=889797

[18] K. Chandy and J. Misra, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 5, pp. 440–452, sep 1979. doi: 10.1109/TSE.1979.230182. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1702653

[19] R. M. Fujimoto, *Parallel and Distributed Simulation Systems*, 1st ed. New York, NY, USA: John Wiley & Sons, Inc., 1999. ISBN 978-0-471-18383-9

[20] Message Passing Interface Forum, *MPI: A Message Passing Interface Standard Version 3.0*. Knoxville, Tenessee: University of Tenessee, 2012. [Online]. Available: http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf

[21] G. Liu, T. Schmidt, D. Rainer, G. Liu, T. Schmidt, and D. Rainer, "RISC Compiler and Simulator , Alpha Release V0 . 2 . 1 : Out-of-Order Parallel Simulatable SystemC Subset," Center for Embedded and Cyber-physical Systems University of California, Irvine, Irvine CA USA, Tech. Rep. 949, 2015. [Online]. Available: http://www.cecs.uci.edu/{~}doemer/publications/CECS{_}TR{_}15{_}02.pdf

[22] A. Mello, I. Maia, A. Greiner, and F. Pecheux, "Parallel simulation of systemC TLM 2.0 compliant MPSoC on SMP workstations," in *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*. IEEE, mar 2010. doi: 10.1109/DATE.2010.5457136. ISBN 978-3-9810801-6-2. ISSN 1530-1591 pp. 606–609. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5457136

[23] R. Domer, W. Chen, X. Han, and A. Gerstlauer, "Multi-core parallel simulation of System-level Description Languages," in *16th Asia and South Pacific Design Automation Conference (ASP-DAC 2011)*. IEEE, jan 2011. doi: 10.1109/ASPDAC.2011.5722205. ISBN 978-1-4244-7515-5 pp. 311–316. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5722205

[24] M. Moy, "Parallel Programming with SystemC for Loosely Timed Models: A Non-Intrusive Approach," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013*. New Jersey: IEEE Conference Publications, 2013. doi: 10.7873/DATE.2013.017. ISBN 9781467350716 pp. 9–14. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6513463

[25] R. Leupers and O. Temam, Eds., *Processor and System-on-Chip Simulation*. Boston, MA: Springer US, 2010. ISBN 978-1-4419-6174-7. [Online]. Available: http://link.springer.com/10.1007/978-1-4419-6175-4

[26] Open SystemC Initiative, *OSCI TLM-2.0 language reference manual*, OSCI Std., 2009. [Online]. Available: http://www.accellera.org/images/downloads/standards/systemc/TLM_2_0_LRM.pdf