Evaluating Parallelization Potential for a SystemC/TLM-based Virtual Platform

ZEESHAN HAYAT
Evaluating Parallelization Potential for a SystemC/TLM-based Virtual Platform

Zeeshan Hayat
December 2018

The thesis work was carried out in coordination with Ericsson AB, R&D ASIC and FPGA Stockholm, Sweden

Company Supervisor: Dr. Ola Dahl
Senior Specialist Model-based System Development
Ericsson AB

Supervisor: Yuan Yao
School of EECS
KTH Royal Institute of Technology

Examiner: Professor Dr. Zhonghai Lu
School of EECS
KTH Royal Institute of Technology
Acknowledgement

I would like to sincerely thank my industry supervisor, Dr. Ola Dahl for his unconditional support, expert guidance and consistent encouragement throughout the duration of this thesis work. Ola went through the technical issues patiently and promptly, and provided overall support and guidance for my thesis work.

I am greatly thankful to my line manager Pierre Rohdin G for giving me this opportunity of working at Ericsson and helping me out with the administrative issues. Also, I would like to thank my colleagues, Udayan Prabir Sinha and Roman Birca, for providing all the necessary technical assistance.

I express my humble gratitude to my examiner Dr. Zhongahi Lu for his continuous support and valuable feedback during the development of this thesis project.

Furthermore, I would like to express my deepest gratitude to my parents and my siblings, Sidra and Sameed, for their unconditional love and support.

Zeeshan Hayat
Stockholm
December 2018
System on chip (SoC) solutions, with integrated hardware and embedded software, are increasing in size and complexity. To cope with the market demand for complex SoC, the abstraction level used during development is raised to allow co-development of software (SW) and hardware (HW). Functional and bit-accurate simulators, referred to as Virtual Platforms, play a vital role in co-development of HW and SW. A virtual platform supports early development, testing, and verification of the embedded SW. However, as the complexity of SoC is increasing so does the complexity of virtual platforms, which is a major bottleneck in the performance of the virtual platforms.

SystemC is an industry standard supporting development of hardware models. SystemC uses co-routine semantics, also known as co-operative multitasking, for the control of simulation. A single process is active at any time. This means that the potential for parallelism, by executing a SystemC simulation on multiple cores in a modern multi-core processor, is not utilized.

This thesis work proposes a parallelization algorithm for SystemC simulations, where one SystemC thread controls a set of parallel host threads. A proof of concept trace-driven simulator is developed to verify the results from the proposed algorithm. Also, an optimized algorithm is proposed which improves the simulation speed. Furthermore, the behavior of the simulator is analyzed by looking into traces, from the Linux kernel and user application level traces, with the help of an open source tracing framework known as LTTng.

The trace-driven simulator is used for evaluation of the parallelization potential for SVP, a virtual platform used at Ericsson. The evaluation makes it possible to determine, for the ideal case when the threads execute independently, the maximum possible speedup for a given test case. Using test cases from production usage, an evaluation of the possible performance improvements for SVP can be done.
System på chip (SoC) -lösningar, med integrerad hårdvara och inbyggd programvara, ökar i storlek och komplexitet. För att klara av marknadens efterfrågan på komplexa SoC höjs den abstraktionsnivå som används under utveckling, för att möjliggöra samutveckling av programvara (SW) och hårdvara (HW). Funktionella och bitexakta simuleratorer, benämnda virtuella plattformar, spelar en viktig roll vid samutveckling av HW och SW. En virtuell plattform stöder tidig utveckling, samt testning och verifiering av den inbäddade programvaran. Eftersom komplexiteten i SoC ökar så ökar även komplexiteten hos virtuella plattformar, vilket begränsar prestandan för de virtuella plattformarna.


**Sammanfattning**

System på chip (SoC) -lösningar, med integrerad hårdvara och inbyggd programvara, ökar i storlek och komplexitet. För att klara av marknadens efterfrågan på komplexa SoC höjs den abstraktionsnivå som används under utveckling, för att möjliggöra samutveckling av programvara (SW) och hårdvara (HW). Funktionella och bitexakta simuleratorer, benämnda virtuella plattformar, spelar en viktig roll vid samutveckling av HW och SW. En virtuell plattform stöder tidig utveckling, samt testning och verifiering av den inbäddade programvaran. Eftersom komplexiteten i SoC ökar så ökar även komplexiteten hos virtuella plattformar, vilket begränsar prestandan för de virtuella plattformarna.


Contents

Abstract i
Abstract in Swedish i
List of Figures viii
List of Tables ix
Abbreviations xi

1 Introduction 1
  1.1 Background .................................................. 2
  1.2 Problem ..................................................... 2
  1.3 Purpose ...................................................... 2
  1.4 Goal ........................................................ 2
  1.5 Benefits, Ethics and Sustainability ......................... 3
  1.6 Methodology ................................................ 3
  1.7 Delimitations ............................................... 3
  1.8 Outline ..................................................... 3

2 SystemC/TLM 5
  2.1 SoC Design .................................................. 5
     2.1.1 SoC Design Flow ........................................ 6
     2.1.2 SoC Bottlenecks ........................................ 6
     2.1.2.1 Increasing Complexity and Cost ................... 6
     2.1.2.2 Time To Market .................................... 6
     2.1.3 TLM ...................................................... 7
  2.2 SystemC/TLM ............................................... 7
     2.2.1 SystemC Processes ...................................... 8
     2.2.1.1 SC_THREAD .......................................... 8
     2.2.1.2 SC_METHOD ........................................ 8
     2.2.2 SystemC kernel ......................................... 8
     2.2.3 Notion of Time ......................................... 10
     2.2.4 TLM ..................................................... 11
     2.2.4.1 LT ..................................................... 12
     2.2.4.2 AT ..................................................... 12
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Abstraction Level For Different System Design Languages</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>SoC Sequential Design Flow</td>
<td>6</td>
</tr>
<tr>
<td>2.3</td>
<td>RTL vs TLM</td>
<td>7</td>
</tr>
<tr>
<td>2.4</td>
<td>SoC Parallel Design Flow</td>
<td>7</td>
</tr>
<tr>
<td>2.5</td>
<td>SystemC Simulation Kernel</td>
<td>9</td>
</tr>
<tr>
<td>2.6</td>
<td>Processes Queues</td>
<td>10</td>
</tr>
<tr>
<td>2.7</td>
<td>Simulation Time and Wall-clock time</td>
<td>11</td>
</tr>
<tr>
<td>2.8</td>
<td>A basic TLM system</td>
<td>12</td>
</tr>
<tr>
<td>3.1</td>
<td>SW execution on target hardware vs on a Virtual Platform</td>
<td>15</td>
</tr>
<tr>
<td>3.2</td>
<td>System Virtualization Platform</td>
<td>16</td>
</tr>
<tr>
<td>3.3</td>
<td>System Virtualization Platform Architecture</td>
<td>17</td>
</tr>
<tr>
<td>3.4</td>
<td>EMCA Architecture</td>
<td>18</td>
</tr>
<tr>
<td>3.5</td>
<td>Memory Layout</td>
<td>18</td>
</tr>
<tr>
<td>5.1</td>
<td>Parallel and sequential execution of tasks</td>
<td>26</td>
</tr>
<tr>
<td>5.2</td>
<td>Co-routine multitasking</td>
<td>26</td>
</tr>
<tr>
<td>5.3</td>
<td>SVP execution on single host thread</td>
<td>27</td>
</tr>
<tr>
<td>5.4</td>
<td>Single-threaded process (many-to-one model)</td>
<td>27</td>
</tr>
<tr>
<td>5.5</td>
<td>SVP execution on multiple host threads</td>
<td>28</td>
</tr>
<tr>
<td>5.6</td>
<td>Mapping between SystemC threads and Linux threads</td>
<td>28</td>
</tr>
<tr>
<td>5.7</td>
<td>Algorithm Description</td>
<td>29</td>
</tr>
<tr>
<td>5.8</td>
<td>PARA_SIM synchronization</td>
<td>29</td>
</tr>
<tr>
<td>5.9</td>
<td>SC_DURING</td>
<td>30</td>
</tr>
<tr>
<td>5.10</td>
<td>PARA_SIM synchronization for optimized algorithm</td>
<td>31</td>
</tr>
<tr>
<td>6.1</td>
<td>SC_SIM architecture</td>
<td>35</td>
</tr>
<tr>
<td>6.2</td>
<td>Processes execution flow in SC_SIM</td>
<td>35</td>
</tr>
<tr>
<td>6.3</td>
<td>DSP workflow</td>
<td>36</td>
</tr>
<tr>
<td>6.4</td>
<td>SC_PARA_SIM architecture</td>
<td>37</td>
</tr>
<tr>
<td>6.5</td>
<td>Notion of time in SC_PARA_SIM</td>
<td>38</td>
</tr>
<tr>
<td>6.6</td>
<td>SC_PARA_SIM simulation flow</td>
<td>39</td>
</tr>
<tr>
<td>6.7</td>
<td>SC_PARA_SIM - Optimized simulation flow</td>
<td>40</td>
</tr>
<tr>
<td>6.8</td>
<td>SC_PARA_SIM with multiple DSPs per group</td>
<td>41</td>
</tr>
<tr>
<td>7.1</td>
<td>Relative performance when all threads are aligned for the whole simulation</td>
<td>44</td>
</tr>
<tr>
<td>7.2</td>
<td>Relative performance for actual tests</td>
<td>45</td>
</tr>
<tr>
<td>7.3</td>
<td>Quantization Impact on Execution Time</td>
<td>46</td>
</tr>
<tr>
<td>7.4</td>
<td>Theoretical Speed-up</td>
<td>47</td>
</tr>
<tr>
<td>7.5</td>
<td>Performance comparison of PARA_OP_SIM with other simulators</td>
<td>47</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>7.6</td>
<td>Performance comparison of PARA_OP_SIM with other simulators</td>
<td>48</td>
</tr>
<tr>
<td>7.7</td>
<td>Relative performance comparison with <code>sc_during</code></td>
<td>49</td>
</tr>
<tr>
<td>7.8</td>
<td>Relative performance comparison with <code>sc_during</code></td>
<td>49</td>
</tr>
<tr>
<td>7.9</td>
<td>Trade-off between simulation speed and accuracy</td>
<td>50</td>
</tr>
<tr>
<td>7.10</td>
<td>Control Flow of SC_SIM</td>
<td>51</td>
</tr>
<tr>
<td>7.11</td>
<td>CPU Usage of SC_SIM</td>
<td>51</td>
</tr>
<tr>
<td>7.12</td>
<td>Control Flow of PARA_SIM</td>
<td>51</td>
</tr>
<tr>
<td>7.13</td>
<td>CPU Usage of PARA_SIM</td>
<td>51</td>
</tr>
<tr>
<td>7.14</td>
<td>Control Flow of PARA_OP_SIM</td>
<td>52</td>
</tr>
<tr>
<td>7.15</td>
<td>CPU Usage of PARA_OP_SIM</td>
<td>52</td>
</tr>
</tbody>
</table>
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Thesis report outline</td>
<td>4</td>
</tr>
<tr>
<td>6.1</td>
<td>Trace format</td>
<td>33</td>
</tr>
<tr>
<td>6.2</td>
<td>Filtered traces</td>
<td>34</td>
</tr>
<tr>
<td>7.1</td>
<td>Speed-up for best case</td>
<td>44</td>
</tr>
<tr>
<td>7.2</td>
<td>Quantization and speed-up - PARA_SIM</td>
<td>46</td>
</tr>
<tr>
<td>7.3</td>
<td>Quantization and speed-up - PARA_OP_SIM</td>
<td>48</td>
</tr>
<tr>
<td>7.4</td>
<td>Quantization and speed-up - SC_DURING</td>
<td>50</td>
</tr>
</tbody>
</table>
Abbreviations

ASIC  Application Specific Integrated Circuit.
AT  Approximately Timed.
BI  Bus Interface.
CPU  Central Processing Unit.
DSP  Digital Signal Processor.
EMCA  Ericsson Many-core Architecture.
EOS  Embedded Operating System.
ES  Embedded System.
IEEE  Institute of Electrical and Electronics Engineers.
LDM  Local Data Memory.
LPM  Local Program Memory.
LT  Loosely Timed.
LTTng  Linux Trace Toolkit: next generation.
RBS  Radio Base Station.
RTL  Register Transfer Level.
SM  Shared Memory.
SMI  Shared Memory Interface.
SoC  System on Chip.
SVP  System Virtualization Platform.
TLM  Transaction-Level Modeling.
TTM  Time to Market.
**UT** Untimed.

**VHDL** VHSIC (Very High Speed Integrated Circuit) Hardware Description Language.
1 Introduction

Embedded systems (ES) are dedicated systems comprised of software and silicon hardware to perform specific sets of tasks. Advances in the semiconductor technology has evolved the System on Chip (SoC) design. SoC includes various hardware components: memories, peripherals, hardware accelerators, and Central Processing Units (CPUs). The CPUs run the embedded software. The design complexity of SoC is increasing rapidly which makes design and verification difficult. There are also constraints to meet the time to market for such products which requires co-development of hardware and software to meet the market window.

In the past, ES were in more manageable sizes, and software development was often done using a physical silicon chip. The design flow for hardware and software was sequential. However, as the SoCs are increasing in size, complexity and heterogeneity, it becomes time consuming and costly to keep the sequential design flow. Furthermore, it is difficult to consider all the details in early development stage; therefore, Electronic System Level (ESL) design flow is used to speed-up the embedded system development process by raising the abstraction level to functional and bit accurate level. ES manufacturers use hardware simulations, in the form of software models of the SoC, a functionally correct model of the hardware, to permit the software developers to develop, test and debug the software while the actual silicon chip is still in the development phase. This also provides insight into verification and exploration in the hardware design in early stage. Such hardware simulations are called virtual platforms, and they enable hardware and software co-development in a timely manner.

SystemC is an industry standard supporting hardware modeling. SystemC provides a simulation environment for event-driven simulation. SystemC consists of C++ classes and macros which provides a simulation kernel and a library for system modeling; furthermore, SystemC uses coroutine semantics, which means all processes run in sequential order. As the complexity of SoC is increasing, the simulation complexity is also increasing. As the processors are not increasing in speed because of the hard limits in physical design e.g. clock speed, the focus is moved towards multi-core systems, where several processors are placed in a single chip. Since SystemC follows co-routine semantics, it is unable to fully utilize the increasing computing power of multi-core systems. This makes it more time consuming to simulate complex hardware models using SystemC.

To cope with this issue, it is important to develop a design that can exploit the parallelism potential in SystemC-based simulation models, so that the extra computing power provided by multi-core systems is utilized. In this thesis work, an algorithm is proposed which enables the use of several host threads in a SystemC simulation. A proof of concept (PoC) simulator design is implemented which executes the simulation in parallel. The simulator performance is evaluated by running multiple test cases on the simulator.
1.1 Background

Ericsson builds software and hardware for Radio Base Station (RBS). To meet the time to market demand, the development of embedded hardware and software is carried out in parallel. The simulator used for the development and integration is a Virtual platform, referred to as System Virtual Platform (SVP). Ericsson is using SVP since 2011 for co-development of SW prior to the development of actual HW. SVP has, to some extent, been used also for architectural exploration and performance analysis. SVP is based on SystemC/TLM 2.0. SVP contains functionally correct and bit-exact models of hardware used in Baseband ASIC and Radio ASIC.

Due to increasing size and complexity of models in SVP, a more parallel approach is required to speed up the simulation process.

1.2 Problem

As the complexity of ES is increasing rapidly, so does the complexity of simulation. Major processor manufacturers have moved towards hard physical limits when increasing the performance of single core processors; it is not feasible to increase the clock speed any more due to heat dissipation. Therefore recent processors are not increasing in terms of speed (clock) but in terms of parallelism by introducing many cores in a single chip. As the SystemC kernel follows co-routine semantics, the kernel is unable to exploit the full potential of many-core system. A parallel approach is required which can run the hardware simulation in parallel while at the same time providing a functionally correct simulation.

1.3 Purpose

The purpose of this Master’s thesis is to evaluate the potential of parallelism for SystemC based virtual platforms. A parallel simulation algorithm is proposed, and verified by a proof of concept trace-driven simulator implementation. This simulator uses traces from SVP, recorded from different test cases, to exploit the potential of parallelism. It is also used to explore quantization in which the running processes will only be allowed to synchronize with the SystemC kernel at specific simulation times. The results are compared with another parallel approach [1]. Finally, the simulator behavior is explored by looking into kernel level traces using LTTng.

1.4 Goal

The goal of this thesis work is to investigate the potential for parallelism in SystemC/TLM based Virtual Platforms. A methodology for executing different SVP models in different Linux threads shall be investigated, and specific algorithms for controlling the parallel threads shall be developed. The algorithms shall be evaluated using a trace-driven simulator, where traces from SVP are used as input. The trace-driven simulator shall be possible to execute in parallel, but also in a sequential mode. The results from parallel simulation will be compared with the SystemC based sequential simulator and sc-during library [1], a parallel programming approach for loosely-timed SystemC models. Furthermore, the simulator behavior will further be explored by looking into kernel level traces by using LTTng, an open source tracing framework for Linux. The results generated from LTTng will be analyzed by a viewing and analyzing tool, Trace Compass.
1.5 Benefits, Ethics and Sustainability

The purpose of the simulator designed in this project is to investigate the parallelism potential for SystemC/TLM based virtual platforms. The parallelism can increase the simulation speed which in turn can increase the efficiency in testing and verification of embedded software. A faster simulation can also be of help in finding flaws in the design in early stages of the product development.

Due to confidentiality involved in this thesis work, some of the information is presented in an abstracted manner for easy understanding of this thesis work. Therefore, this thesis work follows all the confidentiality terms and conditions provided by Ericsson and offers the thesis work to be openly available for other students and researchers. The citations are handled carefully, some figures are reproduced with the permission of the authors. Furthermore, the report is written by keeping in mind all the ethical concerns related to this field.

1.6 Methodology

- The project will start by exploring the parallel approaches already implemented for SystemC/TLM, as well as exploration of Ericsson’s virtual platform, SVP.
- Sample tests, in assembly language, will be run on the virtual platform, and the traces from the simulation will be analyzed to understand the simulation behavior.
- A parallel simulation approach will be proposed, which will be analyzed by a proof of concept simulator implementation and the results will be compared with a SystemC based sequential simulator and the sc-during library.
- Quantization, where the SystemC kernel synchronizations are constrained to a chosen time grid, will be analyzed to further speed up the simulation.
- An improved version of the algorithm will be proposed.
- The simulator behavior will be further analyzed by tracing tools.

1.7 Delimitations

SVP is a well developed system with several hardware models. It is not feasible to implement a parallel SVP in the brief duration of this thesis. For this reason, a proof of concept trace-driven simulator is designed which mimics the actual SVP, and the parallelism is investigated for the proof of concept simulator.

1.8 Outline

This thesis work can be divided into theoretical study, proposed algorithm, implementation of trace driven simulator and evaluation of results. The organization of the thesis is described in Table 1.1.

<table>
<thead>
<tr>
<th>Chapters</th>
<th>Brief Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Introduce SoC Design flow and briefly describe concepts related to the SystemC/TLM standard, such as processes, notion of time, and scheduler. Furthermore, concepts related to synchronization are explained.</td>
</tr>
<tr>
<td>3</td>
<td>Brief explanation about Virtual Platforms.</td>
</tr>
<tr>
<td>----</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>4</td>
<td>Discussion about the related work carried out so far in this domain.</td>
</tr>
<tr>
<td>5</td>
<td>Theory about the conventional sequential execution and proposed parallel execution of DSPs.</td>
</tr>
<tr>
<td>6</td>
<td>Implementation concepts related to trace driven simulator.</td>
</tr>
<tr>
<td>7</td>
<td>Evaluation of results and comparison with other methods.</td>
</tr>
<tr>
<td>8</td>
<td>Conclude the thesis work and suggest directions for future work.</td>
</tr>
</tbody>
</table>

**Table 1.1:** Thesis report outline
In this chapter, the traditional SoC design flow will be briefly described before going into the details about SystemC.

2.1 SoC Design

As SoCs are moving towards increasing complexity, it is important to decide the abstraction level used in the design phase. Factors like SoC size, complexity, heterogeneity, and shortening of development time has pushed to alter the SoC design flow.

Figure 2.1: Abstraction Level For Different System Design Languages, reproduced with permission from [2].

Figure 2.1\textsuperscript{1} shows the levels of abstraction used when designing a SoC. Several design languages are available to deal with SoC design issues. VHDL and Verilog provide Register Transfer Level (RTL) abstractions. RTL is used for design and simulation of digital hardware. SystemVerilog is

\textsuperscript{1}The Figure is reproduced from the book, SystemC: From the Ground Up, with permission from the author, Davic C. Black.

5
a hardware description and verification language which can be used in RTL verification, but also for system modeling and design. SystemC provides a broad range of abstractions. It can be used as system design language and as a hardware modeling language.

2.1.1 SoC Design Flow

SoC design flow consists of two main activities: hardware development and software development. The conventional design flow is shown in Figure 2.2, where there is no communication between hardware and software development teams, and where hardware development and software development are done in sequence. The development starts from RTL design which is a classical approach for System on Chip development.

![Figure 2.2: SoC Sequential Design Flow, reproduced with permission from [3].](image)

In RTL design, the models are described at register level where logical operations are performed, and digital signals are used to connect hardware blocks. RTL describes the system at cycle accurate level, which provides a highly accurate model of the system. In this methodology, where no hardware models are used for software development, physical test chips are required for the software development. Once the silicon chip is available, the software is integrated. If there are any errors in the chip design or software, the design process iterates until a working system is obtained. This is not only expensive but also time consuming.

2.1.2 SoC Bottlenecks

2.1.2.1 Increasing Complexity and Cost

With increasing complexity and size of an SoC, simulation performance is an important factor that should be addressed along the design cycle. A high-level modeling approach (in contrast to RTL) is required, for example by using SystemC and TLM. This enables software to run on simulated hardware, for example together with operating systems such as Linux.

2.1.2.2 Time To Market

TTM is the time from a well-developed idea to the product launch. A product success is very much dependent on the market window. A design flow using only RTL-based simulation is too time consuming because to develop embedded software, actual hardware silicon is required. Instead, a more flexible approach is required to mitigate the delay.
2.1.3  TLM

Transaction Level Modeling uses higher level of abstraction than what is used in RTL. TLM can be used to create models that are accurate enough for functional verification of software, but still fast enough to be useful in software development and integration. It gives considerable simulation speed as compared to the RTL design. TLM replaces pin level events with a single function call as shown in Figure 2.3. TLM can also be used for reference models used in RTL verification.

![Figure 2.3: RTL(left) vs TLM(right).](image)

TLM models can be used in early stages of a project, when RTL is not available, for example when developing a verification environment, or as behavioral models that are later refined to more accurate RTL models. TLM provides a design flow for SoC which enables concurrent hardware and software verification and development, as shown in Figure 2.4. Once a virtual platform with TLM models is available, it serves as a reference platform for software development.

![Figure 2.4: SoC Parallel Design Flow, reproduced with permission from [3].](image)

2.2  SystemC/TLM

SystemC is a standard which defines C++ classes, functions, and macros. SystemC is standardized by Accellera and in the IEEE 1666-2011 standard. It is a modeling language that facilitates "modeling hardware and software together at multiple levels of abstraction" [4].
The Model developers can use SystemC to build hardware models in C++. These models can be compiled to create executable models which can be used for simulation and validation. SystemC has its own simulation kernel which provides set of primitives for hardware modeling.

A complete description of SystemC is out of scope of this thesis work but a brief description of key features of SystemC is provided in this section.

2.2.1 SystemC Processes

A SystemC process is a basic execution unit. It is a small piece of code that runs concurrently with other processes. In simulation, all process are registered with the simulation kernel and are activated by the kernel. The basic SystemC processes are threads and methods.

2.2.1.1 SC_THREAD

SC_THREAD is a SystemC macro for creating a SystemC thread. It is invoked only once and runs from start to the end of the simulation. A SystemC thread has the ability “to suspend itself and potentially allow time to pass before continuing” [2, p. 26], hence giving control back to the simulation kernel. A SystemC thread can suspend its execution by specific amount of simulation time by calling \texttt{wait(simulation\_time)}, or until a specific event occurs by calling \texttt{wait(event)}. In convention, a SystemC thread runs in an infinite loop which imitates the hardware logic that performs some action at regular interval.

2.2.1.2 SC_METHOD

SC_METHOD is a SystemC macro for creating a SystemC method. A SystemC method is a process which is “a normal C++ function that happens to have no arguments, returns no value, and is repeatedly and only called by the simulation kernel” [2, p. 26]. Furthermore, a SystemC method can never suspend, instead it runs to completion and returns control to the kernel. A SystemC method is invoked whenever an event to which it is sensitive to occurs. These events could be edge of a clock, a value change of a signal, or a user defined event. In case an event happens, the kernel runs the methods which are sensitive to that event.

2.2.2 SystemC kernel

SystemC follows an event driven simulation semantics. In event driven simulation, the simulation is controlled by a central scheduler. The scheduler keeps track of time-ordered events and a global simulation time.

The SystemC Kernel follows co-operative multitasking model rather than pre-emptive model. The kernel schedules each process depending upon the state of the process. This means that the scheduler cannot pause or resume a process. Instead, a process runs until either it calls wait or it terminates. This semantics also ensures determinism in simulation which makes the simulation reproducible for the same input in every run.

The SystemC scheduler 'allows the illusion that many SystemC simulation threads are executing in parallel' [2, p. 51] but only one thread is running at any time; thus, it does not utilize the multi-cores of the host machine. With increasing complexity and size of the models, the simulation time is a major bottleneck for complex hardware models.
The SystemC scheduler follows an evaluate-update paradigm as shown in Figure 2.5.

**Figure 2.5:** SystemC Simulation Kernel, reproduced with permission from [2].

Figure 2.5\(^2\) represents a simplified version of SystemC Kernel. The simulation consists of different phases.

During elaboration phase, SystemC components are created and connected through channels and ports. Also, processes are registered and scheduled to be executed.

In initialization phase, SystemC scheduler recognizes all the simulation processes, and places them in runnable or waiting queue as shown in Figure 2.6. Most commonly, all the processes are placed in runnable queues except the processes which are explicitly not initialized by calling `dont_initialize()` which are placed in waiting queue. All the runnable processes are executed once. A SystemC method runs once until completion and a SystemC thread executes until it calls the `wait(...)` function and suspends itself to allow other runnable processes to run. The order of execution of processes is unspecified.

In evaluation phase, all runnable processes are executed once at a time. A SystemC method is executed once until completion and a SystemC thread process is executed until it yields its own execution. The processes are run until there is no process left in the runnable queue. A running process can trigger other processes in waiting queue, which then moves these processes to runnable queue during the same evaluation phase.

---

\(^2\)The Figure is reproduced from the book, SystemC: From the Ground Up, with permission from the author, Davie C. Black.
A running process can generate immediate notification, delta notification and timed notification. When there are no processes left in the runnable queue, simulation moves to update phase. In case there is a delta notification, the evaluation phase starts again. If there is no delta notification, the kernel looks for timed notification. In case there is a timed notification, the simulated time is advanced to the earliest time for a timed notification and the simulation moves to evaluation phase again. When there are no processes left, the simulation is over. The simulation also terminates if it reaches a specified simulation deadline.

2.2.3 Notion of Time

SystemC allows systems modeling on cycle accurate level and at transaction level. Cycle accurate models or models at lower abstraction level which are close to the real hardware are often modeled. Cycle accurate models imitate the behavior of each component on each clock tick. Cycle accurate modeling is mainly used for performance analysis, since it includes high level of detail and provides a very accurate simulation.

Transaction level modeling abstracts out the timing details and represents the components as independent blocks communicating via function calls.

When the SystemC simulation runs, there are three notions of time: Wall-clock time, CPU time or processor time, and simulated time.

- Wall-clock time, elapsed time or real time is the time from the start of a simulation to the end of the simulation.
- Processor time or CPU time is the amount of time for which a host CPU was used for processing of the simulation.
- Simulation time is the time being modeled by the simulation.

Simulation time represents the progress of time within simulation and has no relation with the wall-clock time. As an example, the simulation of 1 second could take 10 wall-clock seconds if the simulation performs a certain set of calculations, or only 1 wall clock second if the simulation performs less calculations. This relation is shown in Figure 2.7.
One outcome of the semantics of SystemC is that it is not possible to state that an action occurs during a certain period. It is not possible to say that certain operation took specified amount of simulation time, the simulation time is an atomic action represented by an integer quantity. So, the operation or action do not occur during certain amount of simulation time but instantaneously with regards of simulation time. A misunderstanding could arise, that SystemC time increase linearly with the elapse time; however, SystemC time advances in steps, as shown in Figure 2.7. Figure 2.7 shows the way time passes in SystemC. The intermediate time has no meaning.

2.2.4 TLM

TLM 2.0 is an Acellera standard. Transaction level modeling gives a consistent way to model transactions in systems with memory-mapped bus architectures.

TLM 2.0 supports virtual platform modeling of today’s system on chip design. TLM 2.0 makes simulation faster by replacing pin level events (clocks, signal pins), that takes lot of simulation time, with function calls. As a result, the simulation runs several orders of magnitude faster compared to RTL.

TLM is designed to support interoperability between individual functional blocks (Intellectual Property (IP)), so that IP blocks from different sources can be used together. This enables developers to create new virtual platforms effectively.

A virtual platform can be modeled on different abstraction levels depending upon the requirements of the design. In general, a virtual platform can contain models from different abstraction levels with the help of different coding styles depending upon the use cases: software development, software performance, architectural analysis and hardware verification. TLM 2.0 provides interfaces for two different coding styles: loosely-timed (LT) and approximately-timed (AT). According to the Acellera TLM 2.0 reference manual [4], users can create their own coding styles, but AT
and LT coding styles are sufficient for development of virtual platforms.

### 2.2.4.1 LT

In loosely timed (LT) coding style, we are interested in the simulation speed. In LT, only sufficient timing details are used, so that the virtual platform can boot an operating system and run application software. LT supports modeling of timers and interrupts, sufficient to boot an operating system and run arbitrary code on target machine. Furthermore, SystemC processes run ahead of time (temporal decoupling) in LT models.

### 2.2.4.2 AT

In Approximately Timed (AT) more accurate timing details are modeled which is important for architectural exploration and performance analysis. Processes run in steps with the simulation time. Each transaction between models has multiple timing details and is separated into several phases.

### 2.2.4.3 UT

In Untimed (UT), there is no notion of time. The communication is carried out only through event notification. It is also known as programmer’s view (PV), since the software developer is typically not interested in the timing of the hardware.

### 2.2.4.4 TLM Terminology

In TLM 2.0 IP blocks are classified as Initiator, Target and Interconnect components.

![Figure 2.8: A basic TLM system](image)

- **An initiator** is a component that initiates a transaction. An initiator creates a new transaction and passes it to a target component.
- **A target** is a component that receives and responds to transactions which are initiated by other components.
- **An interconnect** component acts as a router between an initiator and one or more targets.
Transactions are sent and received through TLM sockets. An initiator sends a transaction through an initiator socket and a target receives the transaction through a target socket. An interconnect component connects to both types of sockets, initiator socket and target socket. An illustration is shown in Figure 2.8.

2.2.4.5 Generic Payload

Generic payload is the default transaction type for a socket. The generic payload is an important part of TLM 2.0 for ensuring interoperability between TLM models from different vendors. A generic payload encapsulates generic attributes of a memory mapped bus communication. Each generic payload has the following set of attributes.

- **Command**: can be set as TLM_READ, TLM_WRITE and TLM_IGNORE.
- **Address**: the address to which data is to be read or written.
- **Data pointer**: points to a buffer. In case of a read command, data is copied from the target to the data buffer. In case of a write command, data is copied from the data buffer to the target.
- **Byte enable pointer**: a pointer to a byte enable array where the value 0x00 indicates that a byte is disabled, and the value 0xFF indicates that a byte is enabled).
- **Byte enable length**: specifies the length of the byte enable array.
- **Streaming width**: specifies the width of the stream (no. of bytes transferred on each beat in the bus that is simulated).
- **DMI allowed**: determines if direct memory interface is allowed or not.
- **Response Status**: indicates the response status of the transaction.
- **Extension pointers**: allows a generic payload to carry protocol specific attributes.
A virtual platform is a software system which mimics the functionality of a target system on chip board. A virtual platform contains executable models of target hardware. The models are typically register accurate and functionally complete and have loose or approximate timing information. A virtual platform is fast enough to boot operating systems and runs application software. A virtual platform is usually made available early, before RTL code is ready and accurate enough to stay in use for post-RTL.

A virtual platform can be designed to implement bit accurate models of the target hardware which allows it to run the same software, OS and drivers on both target hardware and on the virtual platform. An illustration is shown in Figure 3.1.

A virtual platform provides an alternative to a hardware prototype. It has bit accurate processor models that can be connected to other models. In this way, an entire system can be simulated.

3.1 Ericsson’s System Virtualization Platform (SVP)

Ericsson uses SystemC/TLM to develop virtual platforms for baseband ASICs and digital radio ASICs, which are used in radio base stations. SVP is a virtual platform that is used at Ericsson to accelerate the software development process. Ericsson is also developing and maintaining board-level simulators where ASIC models are combined with models for different board components. SVP allows software development to start before the actual hardware is available. Loosely timed models are used in SVP because it provides good simulation performance with sufficient hardware modeling accuracy.
Figure 3.2: System Virtualization Platform

Figure 3.2 shows a descriptive model of SVP. The platform consists of two simulators: baseband simulator and networking processor. The two simulators are standalone models and are connected via a TLM over TCP/IP protocol for communication and synchronization purposes. Both simulators run on their own pace, and synchronize using specified time quantum. As shown in Figure 3.2, external equipment can also be attached to the simulation for testing. For communication between external hardware and the simulator TCP/IP protocol is used. The synchronization works the same way as between the baseband and networking processor. The synchronization between the hardware and the simulator requires synchronization in real time and simulated time. The Board Virtual Platform in Figure 3.2 is a parallel simulator because both models have their own SystemC instances and run in parallel. One approach for creating such distributed simulations is to use CoMIX [5].

SVP is used to develop software at different software stack levels. Firstly, software that is close to the hardware, such as drivers and operating systems, are developed. Then, software at higher layers is developed with the help of SVP. RBS specific application software, which contains baseband processing software, radio processing software and control software, is also developed and tested using SVP.

SVP uses temporal decoupling, with a simulated time quantum, while running the simulation. Due to nature of the cooperative multitasking used in the SystemC scheduler, this means that each running DSP takes turns in execution. This type of simulation could be non-accurate, and can expose concurrency flaws in the software.

The SVP models are implemented to a varying degree of accuracy depending upon the requirements. There are different kinds of models representing actual hardware.

1. Processor Model
   - The processor models are instruction accurate and they model timing to some extent. A cycle accurate processor model is also used for verification.

2. Hardware Accelerator
   - Hardware accelerator models are functionally accurate and are used for ASIC verification.

3. Interconnect Model
   - Interconnect models are used for simulation of interconnects and different kinds of buses.
A descriptive view of SVP is shown in Figure 3.3. The right half of Figure 3.3 shows the virtual platform with processors and baseband accelerators. Processor and baseband accelerators communicate with each other via a bus. In the bottom part of the right half of Figure 3.3, different interfaces to the virtual platform are shown. There is an interface for the execution of the simulation, an interface for parameter passing, an interface for debugging, and an interface for linking library files to the program. The virtual platform can be connected to an external simulator which itself is another virtual platform. On the left half of Figure 3.3, the debugger is shown. The debugger which is used to debug software running on the EMCA processors, is same debugger as used for real hardware. On the top part of left half of Figure 3.3, we have different test tools, referred to as DBG Tool in the figure. Other, external debuggers and test tools can also be attached to SVP. In addition, real hardware can be attached to the simulator.

Figure 3.3: System Virtualization Platform Architecture

3.2 EMCA

EMCA is a real-time multi-core system which runs an operating system, responsible for the resource sharing among tasks and task scheduling. EMCA includes several processors (Digital Signal Processors (DSPs)) running in parallel. Each DSP has three main units: Core, Local Data Memory (LDM) and Local Program Memory (LPM), as shown in Figure 3.4. The core is responsible for program execution. LDM and LPM are used to store data and program, respectively. Furthermore, each DSP accesses Shared Memory (SM) via a Shared Memory Interface (SMI). The shared
memory contains data that can be shared between processors. Each DSP can write and read from SM concurrently. DSPs also communicate with each other with the help of message passing on a common bus via a Bus Interface (BI).

![EMCA Architecture](image)

**Figure 3.4:** EMCA Architecture

![Memory Layout](image)

**Figure 3.5:** Memory Layout

Shared memory layout is shown in Figure 3.5. SM has three parts.

1. **Data Memory Area**
   - Access to data memory can be performed concurrently from all DSPs connected to SM.
   - Read and write to SM address space is atomic. For fair access from all DSPs, an arbitration function is used which guarantees that all DSPs will get access to data memory sooner or later.
2. Semaphore Memory Area

- Before accessing memory, DSP can acquire a semaphore lock. This mechanism can be used to prevent other DSPs from accessing a certain memory region until the semaphore is released.

3. Direct Access Memory Area

- Direct access memory area is an address mapped area which has several memory blocks. It gives the possibility for the DSP to directly read and write into these memory blocks. Each read and write access must be to only one of these several memory blocks.
4 | Related Work

The conventional SystemC model uses co-operative multitasking rather than pre-emptive multitasking and runs SystemC processes in sequential order. The use of co-operative multitasking can simplify for designers, since concurrency due to true parallelism need to be taken into account. Running multiple process in parallel could also result in indeterminism in the order in which processes execute, which then would in the SystemC standard.

Parallelization approaches are aim to increase the simulation speed of SystemC/TLM simulation by running SystemC process in parallel. There has been attempts to add concurrency to SystemC, so that the SystemC processes are running in parallel. In this section the work on parallelizing SystemC has been reviewed.

Moy [1] introduces a methodology to provide parallelism in loosely timed models. Processes can be run in parallel by using a `during` task which can be run in parallel over multiple simulation cycles. The `during` primitive takes a duration $d$ and a function $f$ to be executed, as arguments, and can only be called from a SystemC thread because it calls SystemC wait internally. `sc_during` approach lets the legacy SystemC kernel execute as it is, which is sequential. The `sc_during` method is relevant to loosely timed models and is not efficient in case of approximately timed models because than there is a major overhead because of frequent synchronization with the SystemC kernel.

Other approaches propose to modify the SystemC kernel. One example is SCope [6]. In Scope, the simulation is divided into several simulation context, as compare to Accellera reference implementation where there is a single simulation context. During the elaboration phase, modules are assigned to different simulation context. Processes and events declared in a single module are assigned to a single simulation context. Each simulation context has its own event queue, process lists and runs in its own thread. Each simulation context can execute in its own thread at its own pace. However, simulation context running in different threads are not allowed to exceed a constant lookahead time and simulation engine makes sure that two simulation contexts do not differ by more than lookahead time. Lookahead time is specified carefully because it impacts the simulation performance and timing accuracy.

In [7] proposed a general modelling strategy for shared memory MPSoCs called TLM-DT (Transaction Level Modeling with Distributed Time) and a parallel simulation engine called SystemC-SMP. TLM-DT works on the principle of PDES (Parallel Discrete Event Simulation) [8]. In PDES the system is described as number of logical processes where processes execute in parallel. In TLM-DT the global simulation time does not exist, each process has its own local time. Processes synchronize themselves through timed messages. For this reason, TLM-DT proposes its own coding style. There are three SystemC synchronization primitives used in TLM-DT: `wait(sc_event)`, `wait(SC_ZERO_TIME)`, and `sc_event.notify(SC_ZERO_TIME)`. In TLM-DT, each process
is allowed to increase its local time only if process has the assurance that it will not receive a timed message with local time smaller than its own. SystemC-SMP simulation engine is proposed to take advantage of TLM-DT coding style to perform parallel simulation. SystemC-SMP uses gang scheduling [9] approach where related SystemC threads are run on same processor by a local scheduler where each local scheduler of SystemC-SMP is executed in a host thread. Each host thread is associated to a physical CPU.

The order in which runnable processes are executed in SystemC kernel is undefined. A parallel SystemC Kernel [10] is proposed which modifies the SystemC kernel to execute runnable processes in parallel. In parallel SystemC kernel, at the beginning of evaluation phase, all the runnable processes are pushed into an array and executed in a separate host threads. The scheduler creates an execution environment to maintain the state information of running SystemC process and host threads. The threads are executed in parallel on multiple processors. Data inconsistency can occur when multiple SystemC processes create an update request at the same instance. A synchronization mechanism is implemented which avoids data inconsistency in case of multiple triggered immediate notifications and create new runnable process at the same instance. Furthermore, parallelization techniques like work sharing [11], work stealing [12] and manual grouping are used for load-balancing.

Thread level and data level parallelism are exploited in [13]. SystemC compiler [14] is extended which performs analysis on the source code to identify the design hierarchy and exploit thread level parallelism. Next, conflict analysis is performed to identify potential race conditions between individual threads. In addition, threads are partitioned into segments, where each segment considers executed expressions between two scheduling steps. A scheduling step is triggered with a SystemC \texttt{wait()} function call. After segmentation of all threads, data conflict analysis is performed to identify possible read-after-write and write-after-write conflicts. Conflict analysis table is passed to the parallel simulator for run time decision making. Additionally, data level parallelism is performed with the help of SystemC compiler which generates a report that list all the possible location is the source code for vectorization optimization. It is upto to the designer to decide the identified code locations for vectorization. Finally, out-of-order parallel, similar to [15], and thread-safe simulation is executed.

SimParallel [16] proposes a hierarchichal multi-threading approach. In hierarchichal multi-threading approach, there are two levels, lower level and higher level. Lower level is for parallel execution of kernel level threads and higher level is for execution of SystemC processes. A kernel level thread, which is on lower level, contains several groups of higher level processes. Each kernel thread is executed on a separate core in the host CPU. Each lower level thread has a cooperative scheduler which manages the higher level threads. To avoid race condition, the higher level threads of a module are kept in a same lower level thread.

In [17], Virtanen et al. proposed IPTLM (Inter Process Transaction Level Model) for communication between OS threads running SystemC simulation. IPTLM provides C++ library extension to SystemC without modifying the SystemC kernel. In IPTLM, SystemC simulation is manually distributed to OS threads. IPTLM implements UT or LT simulation and provides separate classes and interfaces for a master and slave bus interface.

Another similar approach is presented in [18]. A new SystemC kernel, referred to as SCale, is proposed to parallelize the evaluation phase of SystemC Kernel. SystemC processes are run in parallel inside multiple OS threads. Threads are assigned statically to different physical cores by
use of affinity. The first thread executes the SystemC kernel while other threads, referred to as workers, execute SystemC processes. SCale also provides communication primitives that can be used to prevent shared resource conflicts during simulation.

A parallel SystemC simulation framework named ParSC is proposed in [19]. During evaluation phase, SystemC processes activated in delta cycle are executed in parallel. ParSC has master OS thread which executes evaluation, notify and update phases of SystemC kernel. Also, ParSC has pool of worker OS threads which executes SystemC processes. Masters thread coordinates with the pool of worker threads before and after the evaluation phase. ParSc introduces synchronization primitives for shared data to coordinate between master and worker threads.

Another approach [20], which is not directly related to parallelism, but it improves the simulation speed by reducing the number of synchronization points in the simulator. It reduces the number of context switches between processes. The idea is to reduce the event processing latency in asynchronous events by correctly predicting the next TQ (time quantum) by using a method referred to as PTD (predictive time decoupled approach).

Parallel implementation of SystemC kernel using PDES is proposed in [21]. This approach runs each SystemC kernel copy on each distributed machine. The SystemC processes are partitioned into groups and associated with one of the scheduler. Each kernel is synchronized at each update phase. A central kernel collects the information from other SystemC kernel’s and update the simulation time.
5 Parallel Execution Of DSPs

5.1 Overview

What is limiting CPU speed? From the beginning of processor design, the focus to improve the performance of the processor was on clock speed, execution optimization, and cache performance. Firstly, increasing clock speed means running CPU faster. It has become harder and harder to keep up with the exponential speedup of the clock, because of the many physical issues; heat dissipation, power consumption, and current leakage. Secondly, an important performance factor is execution optimization, which means doing more work in each cycle. Today, a mainstream computer has several instruction execution optimization functionalities; pipe-lining, branch prediction and reorder instructions. Finally, both approaches focused on squeezing more and more work in each clock cycle. These approaches leads to speed up to only non-parallel, single-threaded application.

To boost processor performance, the objective of major processor manufacturers was to improve clock speed, execution optimization or cache capacity and performance. As of now, we hit physical limits in increasing clock speed, and the focus is more inclined towards multi-core systems. Parallelism in software can help in terms of taking full advantage of the multi-core system.

The speedup gained by parallelism depends upon the problem we are trying to solve. Some problems are inherently non-parallel due to dependencies between tasks, and hence cannot be paralleled. For some problems, where the synchronization is too often there is not much speedup in the end application and hence concurrent way is not the effective way for such tasks.

With many pros of parallelism, it has cost to it. For example, frequent synchronization increases the overall execution time. Also, the cost of development plays a major role which could be larger for a parallel program, since it could be more time-consuming to get a parallel program according to specification, compared to sequential program. The developer should keep in mind the most common problems while dealing with parallel execution. The extra effort is worthwhile because parallelism is a feasible way which enables the opportunity to exploit performance gain in processors performance.

5.2 Multi-Threading Concept

A multi-core system is designed to perform several tasks at once. A task is a basic unit of execution. When a system runs multiple tasks, a scheduler manages the order of execution of these tasks. Figure 5.1 shows sequential vs parallel task execution. A scheduler can be pre-emptive, which means that scheduler can interrupt execution of task. A non-preemptive or co-operative scheduler
cannot interrupt a task. Instead, a scheduler waits for the task to suspend itself and return control to the scheduler.

Figure 5.1: Parallel and sequential execution of tasks

A process is an instance of execution of program. A process has its own context, which may contain register values, program counter and a stack. Each process has a separate memory address space.

A process can have multiple threads. A thread is created by the process. Each thread in a process share the same memory. When multiple threads refer to a memory address, they refer to the same physical address. Thus, if a thread modifies a variable, other running threads will see the modification. Threads are also known as light-weight processes because their creation and context switching are faster than for processes.

5.3 DSPs Execution

Ericsson uses TLM-based models in a simulator, which is known as SVP. A brief introduction to SVP is presented in Chapter 3. SVP simulates the EMCA architecture, a real-time system architecture which runs several DSPs in parallel. In the simulation, we execute binary code on the DSPs. SVP has models of various other hardware components as well, like accelerators and board components. We run Embedded Operating System (EOS), together with application software on DSPs. When we run a software test on the virtual platform it runs on one or many DSPs depending on the test case. The execution flow is controlled by the SystemC kernel. If the test is running on several processors, each DSP is activated by the kernel. DSPs performs various interaction with each other via message passing and interaction with the shared memory.

5.3.1 Sequential Execution

SystemC follows co-routine semantics where each process runs one by one, in a certain order. As shown in Figure 5.2, process 1 will start and performs a task, update local time and suspend itself by giving back the control to the SystemC kernel. The SystemC kernel will run the next process present in the runnable queue.
A simplified view of the Virtual Platform is shown in Figure 5.3. The execution of the different SystemC processes is controlled by the SystemC kernel, which is executed by a single host thread on the target platform.

![Virtual Platform](image)

**Figure 5.3:** SVP execution on single host thread

Figure 5.4 shows how a typical simulation looks like. SystemC models in the simulation represent several instances of the DSPs. The simulator also contains other hardware components like accelerators, radio processing components and communication interfaces etc, which are shown on the right side of Figure 5.4. All SystemC process are elected and scheduled by the SystemC kernel which runs each process in sequential manner. The whole simulation is mapped to a single Linux thread.

![Single-threaded process](image)

**Figure 5.4:** Single-threaded process (many-to-one model)

The size and complexity of SVP is reflected in its models. This tends to make the simulation slower when more models are added. That is why increasing the simulation performance is a major research topic, and why it is important to look for speedup of simulation by exploiting parallel computation.

### 5.3.2 Parallel Execution

This section proposes a multithreaded simulation algorithm which provides a proof of concept for running a simulation in parallel. Figure 5.5 shows the concept of running the simulator on multiple Linux threads in parallel. The idea is to run Linux threads in parallel with the simulation which is controlled by a single SystemC thread. In this proposed algorithm, each DSP is executed in a separate Linux thread.
5.3.2.1 SystemC Processes and Linux Threads

The main idea is to separate the computation and data among several host threads and run them in parallel. Each thread runs separately and synchronizes with the simulated time. It is important to keep the same order of simulation cycles as compared to the sequential simulation in order to get correct results.

The simulation is divided into multiple Linux threads (each thread representing a separate DSP). We decide to use one SystemC thread, which controls the Linux threads. As shown in the left part of Figure 5.6, this represents a one-to-many model (a single SystemC process is mapped to multiple Linux threads). The other SystemC threads, which do not represent DSPs, are kept as they are. They are all run in one Linux thread. This is shown in the right part of Figure 5.6.

5.3.2.2 Thread Partitioning

The main purpose of thread partitioning is to divide work among multiple threads, in order to make a simulation faster. It is very important to balance work load and to minimize the communication between threads. A major limiting factors for the speed-up is the thread with the largest workload.
5.3.2.3 Synchronization

Time based synchronization is used in this algorithm. Each Linux thread corresponding to a DSP runs in its own pace and stops after each run. The SystemC thread that controls the Linux threads corresponding to the DSPs waits for all the Linux threads to stop. When all the threads are stopped, the simulation time is moved, and a selection of threads is allowed to run.

The pseudo code of the algorithm is given in Figure 5.7. In the start of the simulation, all the Linux threads are allowed to run, after which SystemC thread waits for all the Linux threads to stop. When all the Linux threads are stopped, SystemC thread will wake up and move the simulation time to the accumulated time of earliest stopped thread. Furthermore, only threads with the least accumulated time are allowed to run in the next iteration. This process will continue until there are no more threads to run.

```
1 run all threads
while threads are running
    wait for running threads to stop
    update local time
5 run earliest stopped threads
end while
```

**Figure 5.7:** Algorithm Description

5.3.2.4 Notion of Time

Each thread keeps track of its own local time, accumulated local time and wall time. Each running threads moves its own local time and accumulated time and performs its task for the specified wall time. This notion of local simulation time is important for the synchronization of the thread with the simulation.

![Figure 5.8: PARA_SIM synchronization](image)

As shown in Figure 5.8, in the start of the simulation, each Linux thread runs in parallel. Each thread\(^1\) stops at its own local time, thread 0 at 1 ms, thread 2 at 2 ms, and thread 3 at 3 ms. In the first run, Thread 0 is the thread which has least local accumulated time. SystemC thread

\(^1\)The term Linux thread is often used as thread.
will wake up at this point of the simulation and update the simulation time. In this case, the simulation time is moved to 1. In the second run, only thread 1 is allowed to run, which results in thread one stopping at simulation time 2. Again, simulation time is moved to earliest accumulated local time of the stopped threads, which in this case is 2 ms. Now, thread 0 and thread 1 have the least and same local accumulated time, therefore thread 0 and thread 1 will execute in the 3rd run and stop at accumulated time 3. In the 4th run, all threads are aligned (all threads have same accumulated time) so all threads will run in the next cycle in parallel.

5.3.3 Parallel Execution with SC_DURING

Another parallel simulation approach, called \textit{sc\_during} is presented in [1]. This approach is discussed briefly in Chapter 4. In \textit{sc\_during}, tasks with a duration are executed in parallel. Each SystemC process is mapped to a Linux thread, as shown in Figure 5.9.

![Figure 5.9: SC\_DURING](image)

5.3.4 Optimized Parallel Execution

The major limiting factor in the proposed algorithm is the alignment of the threads after each iteration. The threads run in parallel only if they are aligned in simulation time. When more the threads are aligned, the simulation becomes more parallel.

A promising approach would be to move the simulation time not only when all threads are stopped. For example, if the local simulated time of a running thread matches with the local simulation time of a stopped thread, the SystemC thread could be woken up. The SystemC thread could then update the simulation time and allow the stopped thread to run. This approach is shown in Figure 5.10.
The running thread moves its local simulation time and wall time as before, as shown in Figure 5.9. As the running thread moves its simulation time, it is compared with the threads which are not running in that cycle. If the local time of the moving thread coincides with the local time of a stopped thread, the running thread will awake the SystemC thread which will update the simulation time and activate the stopped thread.

As shown in Figure 5.10, in the first cycle, all threads run and stop at their local time. In the next cycle, when thread 0 passes the local accumulated time of thread 1, it will wake-up the SystemC thread which will update the simulation time and activate thread 1.

With this algorithm, as shown in Figure 5.10, the requirement on alignment between the threads is relaxed. Now, threads can run in parallel even if they are not completely aligned.

5.3.5 Shared Data Synchronization

In multi-core systems, where multiple processors are running on separate physical hardware, it is important to synchronize accesses to share data among different processors. Message passing can be used to communicate data between DSPs. A DSP can send a message to a destination, with a message that contains, data, local time and the DSP ID. On the receiving side, a DSP could have a blocking function, which blocks the DSP until the message is received, or a non-blocking function that returns nothing if no message is available. Also, the sender could be blocked if the buffer is full.

In this thesis work, the interaction between DSPs, for example by using message passing is neglected. The purpose is to create an environment where the maximum possible speedup using parallelization can be evaluated. In future work, the effect of DSP interaction needs to be taken into account.
6 | Trace Driven Simulator for Parallelization

Trace driven simulation is a technique which takes the trace instructions generated by running an application. In this thesis work, an instrumented SystemC kernel [22] is used that supports detailed and selective simulation tracing.

As the SVP is well developed system, it was not feasible to apply the proposed parallelism algorithm to SVP because it requires lot of factors to consider. Instead, a trace driven simulator is developed, which mimics the SVP to evaluate the potential for parallelism. It is observed from looking into trace files generated by running several tests on SVP that DSPs consume most of the wall-time in a simulation. DSPs perform computation, communicate with the shared memory, and communicate with other DSPs. Hence, for convenience a trace driven approach is used to verify potential for parallelism by running the DSPs in separate Linux threads.

In Section 6.1, an overview of our instrumented SystemC kernel [22] is provided for understanding of the reader. In Section 6.2, detailed information is provided about the implementation of the trace driven simulators.

### 6.1 SystemC Traces

Some useful simulation events (not to be confused by SystemC event, i.e the variable of sc_event class) to trace are process creation, suspension and resumption. Firstly, process creation represents when the the process was created in simulation time. This happens in the elaboration phase. When processes are created they are executed one by one by the kernel in evaluation phase. Secondly, SystemC threads can suspend themselves by calling SystemC wait. Finally, the process is resumed by the kernel when the simulation time is incremented. This provides the information about each process activity in the simulation.

The output format of the traces is shown in Table 6.1. First column indicates the event: creation, suspension or resumption. Second column indicates the simulation time. Third and fourth column indicate process wall time in seconds and nano seconds.

<table>
<thead>
<tr>
<th>event</th>
<th>sc_time_stamp</th>
<th>name</th>
<th>tv_sec</th>
<th>tv_nsec</th>
</tr>
</thead>
<tbody>
<tr>
<td>create_DSP</td>
<td>0 s</td>
<td>execute_DSP</td>
<td>1</td>
<td>1000000000</td>
</tr>
</tbody>
</table>

Table 6.1: Trace format
For the trace driven simulator we filtered out the traces and only kept the information of DSP process creation, suspension and resumption into separate files according to the DSP name, which are further filtered in order to keep the information of only the simulation time and wall time between each suspension and resumption of each DSP. The sample data file is shown in Table 6.2. First column indicates the relative local time (simulation time) and second column indicates the wall time.

<table>
<thead>
<tr>
<th>sc_time (ps)</th>
<th>wall_time (us)</th>
</tr>
</thead>
<tbody>
<tr>
<td>15432</td>
<td>26</td>
</tr>
<tr>
<td>26472</td>
<td>27</td>
</tr>
<tr>
<td>29801</td>
<td>98</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 6.2: Filtered traces

### 6.2 Sequential Trace Driven Simulator (SC_SIM)

The sequential trace driven simulator mimics the virtual platform running a hardware model with co-routine multitasking. In SC_SIM\(^1\), the simulator reads the trace data from different files depending upon the number of DSPs the test runs on. For example a test with three DSPs is shown below.

<table>
<thead>
<tr>
<th>DSP-0</th>
<th>DSP-1</th>
<th>DSP-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>sc_time</td>
<td>wall_time</td>
<td>sc_time</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

To illustrate SystemC trace driven simulator, the model of the simulator is shown in Figure 6.1. SC_SIM starts the simulation by creating the instance of SC_DSP. SC_DSP registers and executes processes. Each DSP consumes wall time and calls wait to yield its own execution to allow other processes to run.

\(^1\)SC_SIM is referred to as sequential simulator.
To illustrate the execution flow of the example, the logical flow diagram is shown in Figure 6.2. In the start of the simulation three processes are registered and moved to runnable queue, where each process is run one by one. First, DSP-0 runs for wall time of 5 us. DSP-0 calls SystemC wait for 1 ps and suspends its execution. Second, DSP-1 is moved form runnable to running queue where it also runs for wall time of 5 us. DSP-1 calls SystemC wait of 2 ps and suspends its execution. Next, DSP-2 is executed where it consumes wall time for 5 us and calls SystemC wait for 3 ps. Now, there are no further processes to run, so SystemC scheduler moves the simulation time to the earliest called simulation time which is 1 ps and starts the execution again. DSP-0 is again scheduled and executed. This will repeat itself until there are no more DSPs to run.

6.3 Parallel Trace Driven Simulator (PARA_SIM)

PARA_SIM\(^2\) is a parallel trace driven simulator, which runs multiple Linux threads in parallel under a SystemC thread. It reads the trace files, generated by instrumented SystemC kernel, on number of Linux threads. In PARA_SIM, a single SystemC thread is mapped to several Linux threads, each Linux thread representing a separate DSP. The idea is to run test on separate Linux threads in parallel without violating the SystemC semantics.

In PARA_SIM a single SystemC process runs several Linux threads. Each Linux thread represents a separate DSP. Each Linux thread runs on its own pace until it is stopped at a barrier point, where it waits for other threads to stop and when all the threads are stopped, the earliest stopped

\(^2\)PARA_SIM is referred to as parallel simulator.
thread is checked according to the local accumulated time of each thread. After the calculation of earliest stopped thread, the earliest stopped thread wakes up the SystemC thread to update the simulation time according to the least accumulated time of the earliest stopped thread. In the next run, Linux thread with least accumulated time will run and stop at the barrier where the accumulated time is calculated again. This process will continue until there are no entries left in trace files.

The DSP functionality can be defined by Figure 6.3. When a DSP runs it reads data form the trace file and runs for wall time. The DSP stops at barrier, and waits for other threads to stop, where all the threads synchronize with the SystemC thread.

![Figure 6.3: DSP workflow](image)

The Algorithm can also be thought of in terms of two phases: DSP execution phase and simulation time update phases. DSP execution phase runs DSPs and consume the wall time specified in the trace file. This represents the useful work done by the DSP. For convenience the wall time is consumed as it is mentioned in the trace file and not the scaled version of it. In simulation time update phase, the earliest stopped DSP wakes-up SystemC thread and updates the simulation time. The architecture of the PARA_SIM is shown in Figure 6.4. A brief description of the blocks of the simulator is given below.

1. SVP_SIM
   - Initialize and evaluate SystemC thread. The simulation time is updated by SVP_SIM.

2. DSP_Manager
   - DSP manager manages Linux threads. The decision to allow Linux threads to run is taken here.

3. DSP_Barrier
   - Dsp Barrier keeps track of all the running and stopped threads in each run. It also calculates the earliest stopped thread after each run. The information about the running and stopped threads is stored here.

4. Dsp_Group
   - Dsp Group creates Linux thread and runs it. It communicates with DSP_Barrier and informs about its state.
5. Dsp

- Dsp reads data from the trace file and consumes the wall time specified in the trace file. It keeps track of the number of runs for the whole simulation and accumulated local time.

![SC_PARA_SIM architecture](image)

**Figure 6.4: SC_PARA_SIM architecture**

### 6.3.1 Notion of Time

Each DSP keeps track of its local time, accumulated time, and wall time. Each DSP does useful work (consumes wall time) according to the wall time specified in the trace file. Simulation time can be slower or faster than wall time. Simulation time moves in a non-constant manner, where 10 ps in simulation time can take, for example, 10 us or 100 us of wall time. Figure 6.5 shows an example where 2 threads are running in parallel by PARA_SIM. Thread 0 stops at 20 ps (local time) and thread 1 stops at 30 ps (local time). In the first cycle, Thread 0 is the earliest stopped thread. It will run in the next iteration while thread 1 will remain idle. Thread 0 will start again and stops at 51 ps. Earliest stopped thread is calculated again, now, thread 1 is the earliest stopped thread because its accumulate local time is 20 ps.
6.3.2 Synchronization

In SVP SIM, Linux threads synchronize with SystemC thread after each run when all the Linux threads are stopped. SystemC thread waits for all the DSPs to stop. The earliest stopped DSP will wake-up the SystemC thread to update the simulation time.

6.3.3 Atomicity

An atomic section is the part of the code where multiple threads try to access same portion of the code. In case, if two threads want to access the same portion of the code, only one thread is allowed to access it. The shared data is protected with the help of mutex. In our case, multiple threads access DSP_Barrier. Threads access the barrier at random time. We added support to deal with race conditions. The threads which acquires the lock can access the variables and any other thread accessing the same variables is blocked until the locked is released.

6.3.4 PARA_SIM

If we run the sample trace files mentioned in Section 6.2, we get the execution flow shown in Figure 6.6. All the threads start execution in the first iteration and stop at their local simulation time. After first run, thread 0 is the earliest stopped thread which wakes-up the SystemC thread to update the simulation time to 1 ps. Thread 0 will run and stop at local accumulated time 4 ps. In third run, thread 1 is the earliest stopped thread which will wake-up the SystemC thread to update the simulation time to 2 ps. Thread 1 will run and stop at local accumulated time 4 ps. Now, thread 2 is the earliest stopped thread, which will run by moving the simulation time to 3 ps by waking-up the SystemC thread. Thread 2 will stop at 4 ps. At this point of simulation all three threads are aligned, the simulation time will be moved to 4 ps and all threads will run in parallel. Next, the threads will run in the same sequence and are aligned again at simulation time 8 ps where all threads run again in parallel. This process will continue until there are no threads to run. The red mark 'a' in Figure 6.6 represents the position in simulation where all threads run in parallel.
6.3.5 PARA_OP_SIM

The threads run in parallel in PARA_SIM only when the threads are aligned in simulation time which can be a bottleneck in simulation speed. It is observed from the traces of production usage test cases that the threads are rarely aligned. To cope with this issue, an optimized algorithm is proposed.

In optimized algorithm, PARA_OP_SIM, the synchronization between Linux threads and SystemC thread is relaxed. The local simulation time of a running thread is compared with stopped threads. If the local accumulated time of a running thread coincides with the local accumulated time of stopped thread, the SystemC thread is woken up. The SystemC thread moves the simulation time and allows the stopped thread to run.

As shown in Figure 6.7, when thread 0 runs in second iteration. Its accumulated time coincides with the stopped thread, thread 1. The simulation time is moved to 1 ps and thread 1 is allowed to run, shown by a red arrow. Thread 0 and thread 1 simulation time coincides with the thread 2 at simulation time 2 ps. The simulation time is moved to 3 ps and thread 3 is allowed to run, shown by a red arrow. At this point of the simulation all threads are running in parallel. Also, PARA_OP_SIM is referred to as parallel optimized simulator.
the accumulated time of running thread coincides with the stopped thread at simulation time 6 and 7 ps.

In optimized algorithm, the simulation is faster because even the threads are not align the running thread can trigger stopped threads by waking up the SystemC thread. This improves the execution time.

6.3.6 Parallel DSP with DSP group

The algorithm can be modified for test cases which runs on large number of DSPs (>100). The threads can be divided into groups and each thread keeps track of its local time, accumulated time and wall time. Each thread runs multiple DSPs, this can result in less context switching between threads. Each DSP group is responsible for calculating earliest stopped thread in the group after each run. This approach is shown in Figure 6.8. This idea is under consideration and is not explored in this thesis work.
Figure 6.8: SC_PARA_SIM with multiple DSPs per group
The purpose of this thesis work is to evaluate parallelization potential in SystemC/TLM-based virtual platform. Two parallelization algorithms, PARA_SIM and PARA_OP_SIM, are proposed. For evaluation of these algorithms, two trace driven simulators are implemented, also, a sequential trace driven simulator is implemented for comparison. Furthermore, the results from these simulators are compared with loosely-timed approach, sc_during [1].

The evaluation is done by running trace files generated from production usage test cases. The optimal use case for our simulator is that all the threads remain active after each run for complete simulation which gives us an insight into maximum achievable parallelization by these algorithms. Conversely, the worst use case would be a simulation where threads perform very less real work and spend most of the time in synchronization with SystemC thread.

7.1 Results

The evaluation is done by running several tests with PARA_SIM and PARA_OP_SIM. The tests are run on a multi-core system. Test cases from minimum 2 threads to maximum 65 threads are evaluated on the simulators. This is of interest because the use cases where many threads are running causes contention while acquiring shared resources; hence, slow-down in execution time.

7.1.1 Best Case

Figure 7.1 shows the results of the optimal use cases where all threads have the same accumulated time after each run which means that threads are aligned for whole simulation. For this purpose, the trace files from production usage test cases are modified so that the accumulated time of all threads is same for complete simulation. As all the threads are aligned with each other, all threads run in parallel. On the other hand, SC_SIM runs the same optimal use case in sequential manner. For evaluation, seven production usage test cases are used, each test case runs on different number of threads.
The speed-up for SC_SIM and PARA_SIM varies with the number of threads because each production usage test case is different. A good speed-up is achieved in the ideal case, as shown in Figure 7.1. It is clear from Table 7.1, which shows textual representation of Figure 7.1, that we do not reach optimal speed-up even when the threads are running in parallel for whole simulation. The overhead is because of managing threads (thread creation and thread termination) and synchronization with SystemC thread (updating simulation time after each cycle) which becomes a dominating factor in slow-down of the simulation.

Amdahl’s law applies in this case. The synchronization part of our simulator is serial where we check each thread accumulated time and wake up SystemC thread to update the local time and run the threads who are earliest stopped threads. The parallel part of the simulator are Linux threads. As we increase the number of threads we increase Tp (parallel portion of the simulation) but we also increase the synchronization overhead Ts (serial part of simulation) which is an upper limit on the speed-up.
7.1.2 Normal Case

Normal case represents the actual test case results. For evaluation, seven production usage test cases are used, each test case runs on different number of threads. It shows how the performance of the simulator degrades when threads are not aligned with each other and increase number of synchronizations with SystemC thread. For few tests the parallel simulator takes less execution time than the sequential simulator, as shown in Figure 7.2. It is also observed from inspecting the trace files and running different test cases on simulator that each test run highly parallel in the beginning of the simulation and after approximately 30% of the simulation only few threads run in parallel in PARA_SIM. In addition, after each iteration SystemC thread wakes-up to update the simulation time and simulation enters into a critical stage where all the threads are stopped until the simulation time is updated which causes extra contention.

To improve the simulation time, it is important that more threads are aligned on simulation time after each cycle which can be explored with the help of Quantization.

7.1.3 Quantization with PARA_SIM

In quantization, the Linux threads are forced to synchronize on specified time quantum (Quantization Factor). This decreases the number of context switches. In order to check the impact of quantization, we modified the trace files for different quantization factor. To evaluate impact of quantization, we choose a test case which runs on 9 threads (9 DSPs). As we increase the quantization factor, we can see improvement in the execution time, as shown in Figure 7.3 and Table 7.2.
Figure 7.3: Quantization Impact on Execution Time

<table>
<thead>
<tr>
<th>Quantization</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.03</td>
</tr>
<tr>
<td>100</td>
<td>1.63</td>
</tr>
<tr>
<td>500</td>
<td>1.64</td>
</tr>
<tr>
<td>1000</td>
<td>1.71</td>
</tr>
<tr>
<td>1500</td>
<td>2.1</td>
</tr>
</tbody>
</table>

Table 7.2: Quantization and speed-up - PARA_SIM

As quantization factor is increased, the synchronization between the Linux threads and SystmC thread is decreased. Also, the wall time is increased for each Linux thread which means that Linux threads spend more time in consuming wall time rather than synchronization which results in improved execution time.

7.1.3.1 Theoretical Speed-up

Figure 7.4 shows the theoretical execution time comparison with SC_SIM and PARA_SIM. In theoretical speed-up, the maximum wall time consumed by the Linux threads in each iteration is added. This gives us the total wall time consumed by DSPs in a simulation. As we increase the quantization factor, we observe that the gap between theoretical and actual execution time decreases which shows that in our actual simulation major reason of slow-down of simulation is due to synchronization overhead.
7.2 Optimized Algorithm (PARA_OP_SIM)

In optimized algorithm, the simulator does not wait for all threads to stop but runs other threads if running thread’s accumulated time exceeds the accumulated time of the idle or stopped threads. For comparison with other simulators, the test case with 9 threads is used as used in Section 7.1.3. The optimized algorithm shows higher speed-up as compare to other simulators because the requirements on the alignment between the threads are relaxed.
7.3 Quantization with Optimized Algorithm (PARA_OP_SIM)

Quantization impact on optimized simulator is also tested. The same use case is used as mention in Section 7.1.3. This gives us a higher speed-up because more threads are running in parallel in optimized algorithm as shown in Figure 7.6.

![Figure 7.6: Performance comparison of PARA_OP_SIM with other simulators](image)

We observe that we achieve a good speed-up as we increase the quantization factor, shown in Table 7.3. The difference between both simulators speed-up increases as we increase quantization factor.

<table>
<thead>
<tr>
<th>Quantization</th>
<th>Speed-up (PARA_SIM)</th>
<th>Speed-up (PARA_OP_SIM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.03</td>
<td>1.56</td>
</tr>
<tr>
<td>100</td>
<td>1.63</td>
<td>2.48</td>
</tr>
<tr>
<td>500</td>
<td>1.64</td>
<td>3.65</td>
</tr>
<tr>
<td>1000</td>
<td>1.71</td>
<td>4.06</td>
</tr>
<tr>
<td>1500</td>
<td>2.1</td>
<td>6.10</td>
</tr>
</tbody>
</table>

Table 7.3: Quantization and speed-up - PARA_OP_SIM

7.4 Comparison with *sc_during*

*sc_during* provides parallelism in loosely timed models. *sc_during* creates a new OS thread when during task is called and threads are reused for further invocation. Figure 7.7 shows the results from running use case from section 7.3.1 on *sc_during.*
7.5 Quantization with *sc_during*

As shown in Figure 7.8, *sc_during* have greater execution time than SC_SIM when no quantization is applied but the performance tends to improve as we increase the quantization. Furthermore, it performs better than PARA_SIM after certain quantization point.

We can observe from Table 7.4 that *sc_during* performs almost similar to PARA_SIM.
<table>
<thead>
<tr>
<th>Quantization</th>
<th>PARA_SIM</th>
<th>PARA_OP_SIM</th>
<th>SC_DURING</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.03</td>
<td>1.56</td>
<td>0.97</td>
</tr>
<tr>
<td>100</td>
<td>1.63</td>
<td>2.48</td>
<td>1.08</td>
</tr>
<tr>
<td>500</td>
<td>1.64</td>
<td>3.65</td>
<td>1.29</td>
</tr>
<tr>
<td>1000</td>
<td>1.71</td>
<td>4.06</td>
<td>1.56</td>
</tr>
<tr>
<td>1500</td>
<td>2.1</td>
<td>6.10</td>
<td>2.32</td>
</tr>
</tbody>
</table>

Table 7.4: Quantization and speed-up - SC_DURING

Quantization improves simulation’s execution time by decreasing synchronization between threads but this approach has its impact on correctness. The execution speed is gained on cost of accuracy as shown in Figure 7.9. The trade-off between speed and accuracy depends upon the simulation. A simulation can be executed in a very fast abstract manner with reduced accuracy or in more accurate way that results in slower simulations. It is responsibility of the developer to know to what extent the quantization could be applied.

![Figure 7.9: Trade-off between simulation speed and accuracy](image)

7.6 LTTng

LTTng is an open source logging framework which can be used to simultaneously trace Linux kernel, user application and user libraries. LTTng provides "effective tracing and debugging toolset for Linux systems" [23]. For performance analysis of our simulators, LTTng toolset is used. In order to profile kernel and user application level events, root privileges are required, which were not available at the time of this thesis work. For testing purposes, we used a processor with two cores, each core supporting concurrent execution of 2 threads (Hyper-Threading). We performed a simple test which runs on three threads on our simulators: SC_SIM, PARA_SIM and PARA_OP_SIM. The traces are visualized with the help of Trace Compass [24] [25]. The test case from Section 6.2 is used for running simulators and to analyze the traces from LTTng.

Figure 7.10 shows kernel and user level traces from running test case on SC_SIM. Thread Identifier (TID) 8099 shows SystemC thread which runs only one thread at a time. Once the running
thread yield its execution SystemC thread allows waiting thread to run. The execution time for this simulation is 15 seconds. Figure 7.11 shows the CPU usage while running the test case on SC_SIM.

Figure 7.10: Control Flow of SC_SIM

Figure 7.11: CPU Usage of SC_SIM

Figure 7.12 shows result from running the test case on PARA_SIM. Thread identifier 12893, 12894 and 12895 are Linux thread. TID 12892 is a SystemC thread. In this simulation, only one Linux thread is active at a time except the highlighted part in red circles, where the threads are aligned and run in parallel. Furthermore, Figure 7.13 shows CPU usage for three Linux threads. The CPU is fully utilized when three threads are active, hence utilizing 300% of CPU. The execution time for this simulation is 11 seconds.

Figure 7.12: Control Flow of PARA_SIM

Figure 7.13: CPU Usage of PARA_SIM

Figure 7.12 shows result from running the same test case on PARA_OP_SIM. Thread identifier 8341, 8342 and 8343 are Linux thread. TID 8340 is a SystemC thread. For most of the simulation, all Linux thread are active. The running thread awakes SystemC thread to update simulation time when threads local simulation time is equal to other idle threads local simulation time which is represented by red circles. By the optimization, we can fully utilize CPU. CPU usage is represented in Figure 7.15, which shows that CPU usage is maximum as compare to other simulators for most
part of the simulation. The execution time for this simulation is 6.5 seconds.

**Figure 7.14:** Control Flow of PARA_OP_SIM

**Figure 7.15:** CPU Usage of PARA_OP_SIM

It can also be observed from Figure 7.15 that in start of the simulation there is increased number of context switches between three Linux threads which is represented by multiple black lines which is because of the limited cores in the target hardware.
8 Summary and Conclusion

8.1 Summary

The main focus of this thesis was to analyze potential for parallelism in SystemC/TLM-based virtual platforms. SVP, a virtual platform used at Ericsson, is increasing in size, complexity and heterogeneity. It requires more and more time to execute hardware models. SystemC kernel follows co-routine semantics which performs execution of processes in sequential order which is a major hurdle for execution time. Modern processors are not increasing in speed but in parallelism because of several hardware limitations. This is a major problem because the execution time increases as we add hardware models to virtual platform.

In this thesis work, we proposed an algorithm for parallelization of SystemC/TLM-based virtual platforms. The algorithm runs in physical concurrency on a multi-core system while keeping the simulation correctness. The algorithm runs the simulation on a number of Linux threads, each thread runs separately and synchronize with the SystemC thread over simulation time. Threads only run in parallel when they are aligned in accumulated time with other threads. Otherwise, Linux threads which have least accumulated time are allowed to run and other threads wait for the simulation time to be moved. In order to improve execution time, an optimized algorithm is proposed. In optimized algorithm, we run the stopped thread when running thread accumulated time gets equals to the accumulated local time of stopped thread.

In order to verify the performance of proposed algorithms, proof of concept trace driven simulators are implemented. It is observed by running several tests on the simulator that in most cases frequent synchronization is a bottleneck for speed-up. The trace driven simulator mimics the SVP by running the processes in the same order as it was executed in actual simulation. This approach gives us an accurate analysis of the SVP.

Through running tests with different number of threads, we have observed that a good speed-up is possible if we introduce quantization in our simulation. The speed-up is directly related to the quantization values but it has a trade off between simulation accuracy and speed-up.

Furthermore, the trace driven simulation is analyzed with the help of kernel level traces by using LTTng. The traces from LTTng are viewed with the help of a tool, Trace Compass which is an open source tool for viewing and analyzing logs and traces.
8.2 Future Work

This thesis work gave us several research questions regarding parallelism in SVP.

- It is observed from trace driven simulator that frequent synchronization slows down the simulation. To deal with the frequent synchronization, quantization is used where threads are allowed to synchronize only on specified time quantum. It would be interesting to analyze the effects of quantization on simulation and how it degrades simulation accuracy.

- Running Linux thread for each DSP creates overhead of creating and terminating thread. The idea of multiple DSPs for each Linux thread could be explored.

- The kernel level traces were analyzed on a virtual machine which has limited number of cores. The trace driven simulator could be analyzed on a multi-core system and then traces can be analyzed with the help of LTTng and it would be interesting to see the overhead caused in the simulation and to mitigate this overhead in order to improve the simulation speed.

8.3 Reflection

The results of this thesis work exemplify the parallelism potential in SystemC/TLM-based virtual platform. The results show that a significant simulation speed-up is achievable. However, considering production usage test cases, it is vital to determine the balance between simulation speed and accuracy. In addition, it is important for the future of SystemC that it should embrace true parallelism, otherwise as said by Professor Rainer Dömer, "it will go down the same path as the dinosaurs"\(^1\).

\(^1\)Presentation by Professor Rainer Dömer of his paper titled as 'Seven obstacles in the way of standard-compliant parallel SystemC simulation' [26] during SystemC Evolution Day 2016. The presentations from SystemC Evolution day are available at: [http://accellera.org/news/events/systemc-evolution-day-2016](http://accellera.org/news/events/systemc-evolution-day-2016)
Bibliography


