Modeling Shared Memory Access in a SystemC/TLM-based Many-core Virtual Platform

TONY LUNDGREN
Modeling Shared Memory Access in a SystemC/TLM-based Many-core Virtual Platform

Tony Lundgren
2019-07-04

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

Academic Supervisor: Professor Ingo Sander
School of EECS KTH Royal Institute of Technology

Examiner: Professor Christian Schulte
School of EECS KTH Royal Institute of Technology
Acknowledgement

There are a lot of people who have supported me while working on this thesis. First of all I would like to thank my supervisor Ola Dahl, for going above and beyond in providing guidance and support throughout the entire project. I would also like to thank Mohammad Badawi and everyone else in the SVP teams at Ericsson and Tieto who always took the time to help when I had questions. Thanks to Brinda Mohan and Jinju Joy who also did their theses on SVP who provided excellent discussion partners. Outside the SVP teams I would like to thank Tomas Östlund for teaching me about EMCA. And finally I would like to thank my Examiner Christian Schulte for your patience, feedback, and your insistence on clear goals and setting deadlines.
Abstract

The market for embedded devices is fast paced and is growing quickly. To be competitive, time-to-market is important for new products. To shorten the time it takes to release new products, hardware simulators in the form of virtual platforms are developed to allow software development to start before hardware is available.

Virtual platforms model hardware at a high abstraction level, ideally a virtual platform should be as simple as possible, with functionally correct models of the hardware, but to allow complex software to run some timing must be modeled as well. The more timing that is modeled the more complicated the virtual platform becomes. This can make the virtual platform more prone to bugs. It can result in software that is sensitive to changes in the virtual platform if software is developed depending on the modeled timing. Modeling too much timing can also make the virtual platform too slow to run large programs.

This thesis investigates how accesses to shared memory resources are modeled in a virtual platform using SystemC, compared to the many-core system it emulates. This SystemC simulation uses many SystemC processes, and frequently switch between the processes. A tool to visualize these switches between SystemC processes is developed.

Changes are implemented to reduce context switching between SystemC processes. Reducing context switching is good because each switch has some penalty in performance because of the overhead invoking the SystemC kernel, fewer context switches also mean SystemC processes are synchronizing less often which makes it possible to gain performance by running multiple SystemC processes in parallel.

These changes are evaluated running production software tests. The results show that the changes reduce context switching by 94% for one test program, and reduce the total run time for the same program by 16% on average. Though one of 159 software tests fail with the changes, the simplifications of the virtual platform in this thesis show the potential benefits of a simpler simulation model.
Sammanfattning

Marknaden för inbyggda system har ett högt tempo och växer snabbt. För att vara konkurrensmässiga så är det viktigt att vara snabbt ute på marknaden. För att det ska gå snabbare för att ta fram nya produkter så utvecklas hårdvarusimulatörer i formen av virtuella plattformar som gör det möjligt för att börja skriva mjukvara innan hårdvaran är tillgänglig.


Med ändringarna så reduceras antalet byten mellan vilken process som körs med upp till 94% för de program som testades köra på den virtuella plattformen. Tiden som krävdes för att köra programmen gick också ned, i bästa fall med 16%. Tyvärr så får ett av de 159 testprogrammen som används för att testa den virtuella plattformen underkänt, men ändringarna visar ändå potentiala förbättringar med en enklare virtuell platform.
# Contents

Abstract i
Abstract in Swedish i
List of Figures v
List of Tables vi
Acronyms vii

## 1 Introduction 1
1.1 Background ................................................. 1
1.2 Problem Statement ............................................ 1
1.3 Purpose ...................................................... 2
1.4 Goal ......................................................... 2
1.5 Benefits, Ethics, and Sustainability .............................. 2
1.6 Methodology .................................................. 3
    1.6.1 Preparation ........................................... 3
    1.6.2 Development and Testing ............................... 3
1.7 Delimitations ............................................... 4
1.8 Outline ....................................................... 4

## 2 Shared Memory Access 5
2.1 Ericsson Many-core Architecture ............................... 5
2.2 Digital Signal Processor ..................................... 5
2.3 Local Program Memory ....................................... 6
2.4 Local Data Memory .......................................... 6
2.5 Shared Memory ............................................... 7
2.6 Semaphores .................................................. 9
2.7 Message Passing ............................................. 9
2.8 Job Control Unit ............................................ 10
2.9 Summary ..................................................... 10

## 3 SystemC/TLM Virtual Platform 11
3.1 System Virtual Platform ..................................... 11
    3.1.1 QEMU ................................................. 12
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1.2</td>
<td>Simulating Concurrency</td>
<td>12</td>
</tr>
<tr>
<td>3.1.3</td>
<td>SystemC/TLM and QEMU</td>
<td>14</td>
</tr>
<tr>
<td>3.2</td>
<td>Digital Signal Processor Model</td>
<td>15</td>
</tr>
<tr>
<td>3.3</td>
<td>Shared Memory Model</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>Model Analysis using Tracing and Visualization</td>
<td>17</td>
</tr>
<tr>
<td>4.1</td>
<td>SystemC Tracing</td>
<td>17</td>
</tr>
<tr>
<td>4.2</td>
<td>Post-processing</td>
<td>17</td>
</tr>
<tr>
<td>4.3</td>
<td>Visualization</td>
<td>17</td>
</tr>
<tr>
<td>4.4</td>
<td>Use Cases for Causality Graph</td>
<td>18</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Debugging Software</td>
<td>19</td>
</tr>
<tr>
<td>5</td>
<td>Shared Memory Access in a Virtual Platform</td>
<td>21</td>
</tr>
<tr>
<td>5.1</td>
<td>Shared Memory Access</td>
<td>21</td>
</tr>
<tr>
<td>5.2</td>
<td>Semaphore</td>
<td>22</td>
</tr>
<tr>
<td>5.3</td>
<td>Job Control</td>
<td>23</td>
</tr>
<tr>
<td>6</td>
<td>Simplifying the Shared Memory Access Model</td>
<td>27</td>
</tr>
<tr>
<td>6.1</td>
<td>Never use a Shared Memory Interface Thread</td>
<td>27</td>
</tr>
<tr>
<td>6.2</td>
<td>Use a Shared Memory Interface Thread for Job Control Access</td>
<td>28</td>
</tr>
<tr>
<td>6.3</td>
<td>Trusting the Quantum</td>
<td>28</td>
</tr>
<tr>
<td>7</td>
<td>Evaluation</td>
<td>31</td>
</tr>
<tr>
<td>7.1</td>
<td>Small Test Programs</td>
<td>31</td>
</tr>
<tr>
<td>7.1.1</td>
<td>Never use Shared Memory Interface Thread</td>
<td>31</td>
</tr>
<tr>
<td>7.1.2</td>
<td>Only use Shared Memory Interface Thread for Job Control</td>
<td>33</td>
</tr>
<tr>
<td>7.1.3</td>
<td>No Synchronization Before Semaphore Access</td>
<td>33</td>
</tr>
<tr>
<td>7.2</td>
<td>Production Software Tests</td>
<td>34</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Pass/Fail</td>
<td>34</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Context Switches</td>
<td>35</td>
</tr>
<tr>
<td>7.2.3</td>
<td>Performance</td>
<td>36</td>
</tr>
<tr>
<td>8</td>
<td>Conclusion</td>
<td>39</td>
</tr>
<tr>
<td>8.1</td>
<td>Future Work</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>References</td>
<td>42</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Using conditional execution with VLIW can avoid branches</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>ASIC A’s memory layout</td>
<td>7</td>
</tr>
<tr>
<td>2.3</td>
<td>ASIC B’s memory layout</td>
<td>8</td>
</tr>
<tr>
<td>3.1</td>
<td>The exact same software runs on the target hardware and the virtual platform</td>
<td>11</td>
</tr>
<tr>
<td>3.2</td>
<td>QEMU executing basic blocks of instructions</td>
<td>12</td>
</tr>
<tr>
<td>3.3</td>
<td>Flow Chart Diagram for the SystemC kernel</td>
<td>14</td>
</tr>
<tr>
<td>3.4</td>
<td>Interface between SystemC/TLM and QEMU in SVP</td>
<td>15</td>
</tr>
<tr>
<td>4.1</td>
<td>One slice of a complete causality graph</td>
<td>18</td>
</tr>
<tr>
<td>4.2</td>
<td>Supposed locking of an already locked semaphore</td>
<td>19</td>
</tr>
<tr>
<td>5.1</td>
<td>Traces with (5.1a) and without (5.1b) direct memory interface</td>
<td>22</td>
</tr>
<tr>
<td>5.2</td>
<td>Simple semaphore program causality graph for ASIC A</td>
<td>23</td>
</tr>
<tr>
<td>5.3</td>
<td>Simple semaphore program causality graph for ASIC B</td>
<td>23</td>
</tr>
<tr>
<td>5.4</td>
<td>Program requesting job from job control then cancelling request, ASIC A</td>
<td>24</td>
</tr>
<tr>
<td>5.5</td>
<td>Program requesting job from job control then cancelling request, ASIC B</td>
<td>25</td>
</tr>
<tr>
<td>6.1</td>
<td>Semaphore execution pattern with (6.1a) and without (6.1b) wait call before access, on the Old ASIC</td>
<td>29</td>
</tr>
<tr>
<td>7.1</td>
<td>Two store instructions, no shared memory interface thread, ASIC A</td>
<td>31</td>
</tr>
<tr>
<td>7.2</td>
<td>Semaphore program, no shared memory interface thread, ASIC A</td>
<td>32</td>
</tr>
<tr>
<td>7.3</td>
<td>Job request and abort, no shared memory interface thread, ASIC A</td>
<td>32</td>
</tr>
<tr>
<td>7.4</td>
<td>Semaphore, with shared memory interface thread and no semaphore synchronization, ASIC A</td>
<td>33</td>
</tr>
<tr>
<td>7.5</td>
<td>Semaphore, no shared memory interface thread and no semaphore synchronization, ASIC A</td>
<td>33</td>
</tr>
</tbody>
</table>
List of Tables

7.1 Test cases pass fail, ASIC A ......................................................... 35
7.2 Test cases pass fail, ASIC B .......................................................... 35
7.3 Total number of suspends for each version of SVP, ASIC A .................. 35
7.4 Total number of suspends for each version of SVP, ASIC B .................. 35
7.5 Distribution of wait calls for T0, ASIC A ......................................... 36
7.6 Quantum Utilization for T0, ASIC A ................................................ 37
7.7 Run time for each SVP version for T0, ASIC A .................................. 37
7.8 Distribution of execution time in threads in milliseconds for T0, ASIC A .... 37
7.9 Time statistics for SVP4 with normal SystemC kernel compiled in release mode, ASIC A ................................................................. 38
7.10 Average run time for user-time plus system-time, ASIC A ................... 38
7.11 Average total run time for simulation, ASIC A .................................... 38
## Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMBA</td>
<td>Advanced Microcontroller Bus Architecture.</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application Specific Integrated Circuit.</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processor.</td>
</tr>
<tr>
<td>EMCA</td>
<td>Ericsson Many-Core Architecture.</td>
</tr>
<tr>
<td>FIFO</td>
<td>First In First Out.</td>
</tr>
<tr>
<td>ISA</td>
<td>Instruction Set Architecture.</td>
</tr>
<tr>
<td>LDM</td>
<td>Local Data Memory.</td>
</tr>
<tr>
<td>LPM</td>
<td>Local Program Memory.</td>
</tr>
<tr>
<td>PCI</td>
<td>Peripheral Component Interconnect.</td>
</tr>
<tr>
<td>QEMU</td>
<td>Quick Emulator.</td>
</tr>
<tr>
<td>SMI</td>
<td>Shared Memory Interface.</td>
</tr>
<tr>
<td>SVP</td>
<td>System Virtual Platform.</td>
</tr>
<tr>
<td>TCG</td>
<td>Tiny Code Generator.</td>
</tr>
<tr>
<td>TLM</td>
<td>Transaction-Level Modeling (version 2).</td>
</tr>
<tr>
<td>VLIW</td>
<td>Very Long Instruction Word.</td>
</tr>
</tbody>
</table>
1 Introduction

Virtualization of hardware can be a beneficial tool for both hardware and software developers. In hardware design it can be used to test ideas at many different abstraction levels or even to verify that high level functionality is preserved when synthesizing the design to a lower abstraction level. In software, when used in the form of hardware simulators referred to as virtual platforms, it enables developers to test their software without access to hardware, and debugging tools can be attached to a virtual platform to provide full control over the simulated hardware. This thesis looks at the virtual platform from the software developer’s perspective. The interest in the platform is from the perspective of software development, with a goal of designing the virtual platform, and when needed also the software, so that software becomes robust to changes in the virtual platform, and as a desired by-product, also to changes in the real hardware.

SystemC is a modeling language for electronic system-level design, capable of simulating hardware from very high level of abstraction down to the register transfer level. Together with transaction level modeling (TLM), a framework where hardware models can communicate using function calls instead of using pin-level interactions, for example over a memory-mapped bus, an accurate and performant hardware simulator can be developed.

1.1 Background

Ericsson is a telecommunications company and develop their own Application Specific Integrated Circuits (ASICs) for the radio base stations. For shorter time to market and to improve the testing environment, virtual platforms that can simulate the ASICs are developed. Because the virtual platforms can simulate an entire system, the virtual platform is named System Virtual Platform (SVP).

SVP is written in SystemC/TLM-2.0 and has functionally correct models of the hardware. The more similar the virtual platform is to the hardware platform the more likely it is that the same software will run and produce the same output on both the hardware and the virtual platform. In a parallel system like the Ericsson Many-core Architecture (EMCA) with a non-blocking memory interface, timing within and between different components in the system can be required in order to run software that interacts with the different components. Therefore, some timing information has been added to the SystemC models.

1.2 Problem Statement

It is important to find a level of timing modeling that makes the simulator just accurate enough for software to run functionally correct. If the timing model is more accurate than required for functional correctness, it may result in a closer representation of the real system. On the other
hand, due to the timing modeling being approximations, the timing in the virtual platform can
in the worst case create scenarios where software may work on a virtual platform but not when
run on hardware. Modeling more timing might also allow software to adapt to certain timing
characteristics in the simulator, and in this way become less tolerant to changes in the simulator,
and also less tolerant to changes in the actual hardware.

Another problem is that if the virtual platform itself is complicated enough to make bugs in
the virtual platform a frequent enough occurrence, trust in the platform can be eroded, reducing
its usefulness as a tool for the software developers. This problem of constructing simple models of
complex systems is captured in Bonini’s paradox [1].

In addition, the more complicated the models are in terms of what they do and how they
communicate can make the simulation time longer. Profiling data from production software shows
that most of the SystemC execution time in SVP is spent in the digital signal processor (DSP)
and shared memory interface threads so that is where the focus of this thesis will be.

1.3 Purpose

The purpose of this thesis is to document how the shared memory interface in SVP works so that
ideas for how to simplify the implementation can be formed. Improvements can be in the form
of shorter total simulation time, or fewer potential bugs in SVP which will make the experience
of developing software for the ASICs on the virtual platform better. Another potential benefit of
modeling less timing is that it forces software developers to not write timing dependent code. Less
timing dependent code makes for more robust software, that is a win for both the users of that
software and the developers, who will not have to update their programs when either the virtual
platform is updated or when new hardware is released.

1.4 Goal

The goal is to simplify the implementations of the shared memory interfaces for two different
versions of the EMCA ASICs, for the reasons stated in Section 1.3. To meet that end, there are a
few deliverables along the way:

- Documentation, describing current implementations of the shared memory interface.
- Test programs, where DSPs interact with shared resources so that the execution of memory
  access instructions can be analyzed.
- A simplified shared memory interface for a select number of memory access instructions.
- An evaluation of the simplified shared memory interface based on results from the tests
  programs and from production software tests.

1.5 Benefits, Ethics, and Sustainability

The benefits of this specific work is outlined in section 1.3. In more general terms virtual platform
technology can be of great boon to the environment as newer and more energy efficient products
can be released faster. The need for manufacturing early hardware development boards for software
developers can be reduced and verification of hardware design using SVP as a reference model may
catch design flaws or bugs in the Register Transfer Level code.
Ericsson is a direct beneficiary as the work is done on SVP. Ericsson has a comprehensive policy on the way they should conduct their business with consideration to their moral and environmental obligations. So this thesis work can be done with some promises that Ericsson will not use the work to do evil. However, since this work is done at a company where employees and other workers have to sign a non disclosure agreement, this thesis may lack some implementation details that would be present if the same work was done somewhere without such an agreement [2]. Care has been taken to provide an honest representation and interpretation of the results in accordance with [3].

This report attempts to be accessible to as many readers as possible, avoiding overuse of abbreviations and unnecessarily complicated language. The report and the graph drawing program developed attempts to be color blind friendly by avoiding classical use of red/green for differentiation as much as possible, and the colors are only used as a supplement; being able to differentiate between the colors is not required to understand the meaning of the graphs.

1.6 Methodology

The work was done for the virtual platforms of two different ASICs, ASIC A which is representative of older EMCA ASICs and ASIC B which is representative of the newer ASICs.

1.6.1 Preparation

First the SystemC/TLM method of simulating hardware had to be studied. Learning SystemC included reading the book SystemC from the ground up, a one week course at Cadence, and studying online material. In particular the way SystemC simulates concurrency and communication between processors and other hardware components on the board is important for this thesis. Furthermore the instruction set architecture ISA for EMCA had to be studied so that an understanding of what should happen when software runs could be formed. This included an internal Ericsson course on how to program the DSPs with their assembly language.

With the knowledge of how the hardware works, on a high level, and how to write assembly programs for the platform, the next step was to try and understand how Ericsson’s virtual platform, SVP, works in general, and how it simulates shared memory accesses. For that a study of previous work on the SVP was surveyed, in particular the modified SystemC kernel instrumented with new tracing capabilities was identified as a very good tool for analysing SVP [4].

1.6.2 Development and Testing

Small tests programs had to be written so the different types of memory accesses could be analyzed in isolation. The small test programs did things like: One load or store instruction, multiple stores right after each other, storing and loading with a dependency, storing with delays between the stores.

These tests were analyzed using the logs produced by the tracing library. During this time the capabilities of both the modified kernel and the post processing tools were extended. Backtraces collected for wait and notify calls enabled investigations into the huge code base of SVP. Even though the programs were as small as possible, the causality between threads was difficult to determine. This was solved by adding a graphical visualization, a causality graph, that showed how the different SystemC threads triggered each other.

Because these graphs showed a lot threads starting and stopping, even for the small programs, ideas of how to reduce the number of context switches were formed. Because the backtraces showed where the wait calls and notify calls happened, it was possible to either simply remove them, or change the surrounding code to create specific circumstances for them to happen.
These changes were tested, first with the small programs, then later with the bigger test suites, to see if they had the expected effect on the simulation, and to see if the programs still worked. Again the special SystemC kernel was used to evaluate the changes. The causality graph was used for the small programs to see how much simpler the causality chain between threads became, and the raw logs were used for a statistical comparison. Analyzing the effect the changes had on the simulation in terms of performance could not be done accurately with the modified kernel, as the kernel itself had an effect on performance. However, without the modified kernel the precise information it provided was not available, so only the total run time for the entire simulation could be captured. Each test program was run five times for each platform, five was deemed enough after one test was run fifteen times without decreasing the standard deviation compared to the initial five runs. The average run time of the five runs is what was compared to make the statement of possible performance gains when using a simplified virtual platform compared to the original.

1.7 Delimitations

This thesis does not cover any of the SVP details outside of the DSPs and the shared memory and its interfaces. The small tests will verify that the changes are doing what they are supposed to, and the operating system tests and ASIC tests will tests how the changes affect more realistic use cases. If these tests fail looking into why they failed, and if software can be modified to function on the modified SVP will be done only if time is available.

1.8 Outline

In Chapter 2, Shared Memory Access, the hardware architecture is described, what different memories there are in the two ASICs covered in this thesis, and how accesses to the different memories are implemented in the hardware. The next chapter, SystemC/TLM Virtual Platform, introduces Ericsson’s virtual platform from a high level design perspective. With the platform defined Chapter 4, Model Analysis Using Tracing and Visualization, describes the modified SystemC kernel and the post processing programs used to analyze SVP. Chapter 5, Shared Memory Access in a Virtual Platform, uses the tools from the previous chapter to explain how different shared memory accesses are simulated in the virtual platform. Chapter 6, Simplifying Shared Memory Access Model, then discusses ideas for how to simplify the simulation, the focus is on reducing context switches between SystemC processes and how the simplifications can be implemented. The results of these simplifications are evaluated in Chapter 7, comparing how programs are simulated on the simplified SVP relative to the original SVP. In the last chapter, Chapter 8, the results are summarized and provides suggestions for future work.
In this chapter the hardware architecture is introduced, what different memories there are in the two ASICs covered in this thesis, and how accesses to these memories are implemented in the hardware.

2.1 Ericsson Many-core Architecture

Ericsson develops baseband and digital radio ASICs that are inside their radio base stations. Ericsson Many-core Architecture or EMCA for short is a collection name for all the different iterations of their ASICs with many DSPs. The ASICs also have a variety of accelerators, control logic, timers, and other peripherals. In this thesis two versions of EMCA ASICs are studied. The two versions are referred to as ASIC A and ASIC B. ASIC A represents the current generation of ASICs, and ASIC B represents the next generation of ASICs, currently under development. References to ASIC A and ASIC B will be used when describing features that are different between the two ASIC variants. When describing features that are common to ASIC A and ASIC B, no specific references will be used. Specifically what is of interest in this thesis is communication between DSPs and the shared memory.

2.2 Digital Signal Processor

The EMCA DSP is a very long instruction word (VLIW) processor. Being VLIW means that parallel instructions are collected into instruction bundles, and that the DSPs can execute more than one instruction at once, as long as there are functional units to handle each instruction. For example it may not be possible to fill the entire instruction bundle with memory accesses if there are not enough memory access hardware units available. Even so, with a large number of DSPs able to execute multiple instructions at once it makes for a highly parallel system.

The DSPs have a fully protected multistage pipeline. Using pipelines means it is possible to run the processor at a higher frequency than at the speed it takes to execute a full instruction, and fully protected means that from a functional point of view, a program can pack dependent instructions without consideration for instruction latency. Hardware is responsible for detecting dependencies and ensuring that dependent instructions always operate on correct input [5].

One feature of VLIW is predicated execution of instructions, this makes it possible to selectively execute only part of an instruction bundle, or not at all. One use case of this is avoiding branches, see Figure 2.1. Using this technique eliminates 27% of branches and 56% of mispredictions from the benchmarks tested in [6]. This becomes even more important as processor pipelines have become fairly long and every misprediction of a branch means flushing the pipeline to return to the processor state before the branch.
The DSP’s local memory is of the Harvard memory architecture, which means separate local program memory (LPM) and local data memory (LDM) [7].

### 2.3 Local Program Memory

The DSP fetches its instructions from the local program memory. The DSP can not execute instructions from addresses outside the range of the LPM. The local program memory is configured as a fully associative cache. If there is a cache miss, the DSP will fetch new instructions from the shared memory. In current generations of EMCA, moving instructions to or from the local program memory can happen in the background and does not cause the DSP to stall unless necessary.

In a fully associative cache, any memory address can be stored in any cache block. The logic for this is expensive since for performance reasons the tag bits of each cache block has to be checked in parallel. The benefits of a fully associative cache compared to a directly mapped or set associative cache are that you avoid cache contention where the different address ranges overwrite each other in the cache, and that it is possible to utilize the replacement policy to its fullest potential, leading to lower miss rates in the cache [8].

### 2.4 Local Data Memory

The local data memory (LDM) is a multiport fixed time access memory. The multiport part is important because it allows an instruction bundle in the DSP to have multiple memory instructions. Because the time it takes for a value written to memory to be available is fixed, the compiler can
avoid stalls by scheduling memory accesses appropriately. This performance predictability is also an important aspect when doing worst case execution time analysis in real-time systems. Unlike the local program memory, the local data memory is not a cache, because software has to explicitly move data to and from it.

### 2.5 Shared Memory

DSPs can access the shared memory concurrently. Each DSP’s local data memory and local program memory are connected to the shared memory controller through each DSP’s shared memory interface. The shared memory interface is another multiport interface. In ASIC A the interface has one port, see Figure 2.2. ASIC B has two data ports and one program port. Data port one is for the shared memory, and data port two is used when accessing special addresses that do not point to normal memory, for example hardware semaphores or memory mapped peripherals like the job control unit use this second port, which goes through the Tree interface. The third port is for the local program memory and it goes through the program memory interface, see Figure 2.3. ASIC B also has a new hardware component to decide which of the three ports to use.

There are no caches in between the local memories and the shared memory and there is no speculative execution. This makes for high and predictable bandwidth for large burst accesses to and from the shared memory, independent of the number of cores involved and the amount of data accessed.

DSP instructions interacting with the shared memory are non-blocking, this means that the
Figure 2.3: ASIC B’s memory layout
DSP will not stall waiting for the instruction to fully complete before starting the next one, the DSP will simply tell the memory interface what needs to be done and then move on. This works because the memory interface has internal FIFOs to buffer reads and writes. Accesses that are put in the different FIFOs may run out-of-order, but only if it does not violate the original program’s semantics.

Using special registers and instructions it is possible to store and load address ranges to and from the shared memory and the local memory. The instruction itself only takes a few cycles to execute, but the act of moving the data will of course take longer. At the start of the instruction the entirety of the affected area will be marked as reserved, and other accesses to the reserved area in LDM will result in the DSP stalling. The reserved area will decrease in size in steps as data is moved to, or from, its location. Instructions with accesses to the shared memory are not limited to loads and stores. Some instructions can do operations directly on the data in the shared memory.

There are some exceptions where a shared memory access is not non-blocking. When this happens the DSP that executed the instruction will stall. These exceptions are:

- Any access to the shared memory when the internal FIFOs are full cause the DSP to stall until there is room in the FIFO.
- Trying to use the result of a load instruction before it is complete also cause the DSP to stall.

### 2.6 Semaphores

Some address ranges for the shared memory provide special functionality. For example in a specific range there are hardware binary semaphores, where loads and stores result in the locking and unlocking of a semaphore. There is a small quirk with these hardware semaphores. Locking a semaphore by issuing only a load instruction will not cause the DSP to stall, even if the semaphore is locked. Since the memory interface is non-blocking, it will only stall when there is an unmet dependency. To be blocked when trying to lock a locked semaphore the value from the load instruction has to be used in a second instruction.

In ASIC A semaphore accesses and ordinary shared memory accesses go through the same shared memory interface. In ASIC B there is a new hardware component that checks what kind of memory instruction is being executed, and where it is going, and sends it to different output ports that lead to different interfaces. Semaphore accesses are regarded as special and will go through the Tree interface.

### 2.7 Message Passing

There is power in a shared memory where data can be modified without being copied, and where arbitrary chunks of data can be accessed with a single instruction. However, it is not the only way to share information. In combination with the centralized structure of the shared memory, EMCA also has a more peer to peer style communication through a bus network. All DSPs and some other peripherals are connected through a bus network through which messages can be sent and passed around. The messages can be addressed to a specific receiver or a broadcast to anyone who wants to listen. DSPs can set up an inbox in their local data memory where the DSP can receive messages. In this way, a DSP can write data directly from its own LDM to another DSP’s LDM. With the inboxes DSPs can receive unsolicited information. This can be used to transfer data, but it can also be used as an alternative to semaphores for synchronization between DSPs.
2.8 Job Control Unit

The job control unit can be accessed both through the bus network and via the memory interface. In ASIC B this is through the Tree interface. Accesses to the job control unit are special in that it is possible to request data that is not there yet, so it is not predictable how long an access will take to complete. In order to notify the DSP that the transfer is done an interrupt is raised at completion. Because the accesses are of indeterminate time which blocks the shared memory interface, DSPs can issue an abort, to let other shared memory accesses through. Abort instructions cause a race between the request to abort and the request for the job, which will still try to finish until abort request is processed. The DSP can then verify if the request was completed or aborted by reading a special register.

2.9 Summary

See Figure 2.2 for a view of how the DSPs in ASIC A are connected to the shared memory and the peripherals. ASIC B is similar, but it has the three different interfaces used depending on which area is accessed. SMI is Shared Memory Interface, PMI is Program Memory Interface, BI is the Bus Interface, Tree is the memory interface with roots in many different areas. LDM and LPM are, as mentioned earlier, local data memory and local program memory.

Memory accesses are non-blocking, accessing data does not alone cause the DSP execution to stall. However, if the data is not available when the DSP tries to use it, the DSP will stall. Some memory accesses can be aborted.
3 | SystemC/TLM Virtual Platform

A virtual platform is a simulator that simulates computer hardware on a functional level from software’s point of view. The virtual platform contains models of target hardware, such as processors, peripherals, and interface units. The models are mostly register accurate and functionally correct, able to simulate all functionality of the hardware. In this chapter Ericsson’s virtual platform is introduced from a high level design perspective, with a focus on what is needed to simulate the DSPs and their memories. The virtual platform can run on a host machine like a GNU Linux x86_64 machine, and target software that would run on hardware, runs on the virtual platform, see Figure 3.1

Figure 3.1: The exact same software runs on the target hardware and the virtual platform

3.1 System Virtual Platform

Ericsson’s virtual platform is named SVP for System Virtual Platform. SVP is built upon free software technologies such as SystemC/TLM-2.0 for the hardware models and the Quick Emulator (QEMU) for instruction translation and execution. The simulation is managed by the SystemC kernel, which runs SystemC processes called threads and methods, with hardware models organized in a hierarchy of SystemC Modules.
3.1.1 QEMU

QEMU [9] is a machine emulator and virtualizer, designed to run code compiled for a target architecture on a host, making it a good candidate to use when implementing a functional instruction set simulator. One of the goals of QEMU was to enable support of as many architectures as possible, balancing the tradeoff between performance and portability [10]. Ericsson has then used this portability to add support for their own platform, EMCA.

In a QEMU simulation where all the hardware functionality is emulated inside QEMU, there is a main loop which calls a function to execute target (EMCA) code instructions, like the processor would. To achieve this it does dynamic code translation, which differs from a normal interpreter in some important ways. It does not translate and execute instructions one by one, but in so called basic blocks. A basic block is a sequence of instructions which starts with a label and ends with a branch. The translated basic blocks are saved in a cache so that if re-used they do not have to be translated again. In order to avoid the overhead of returning to the main loop unnecessarily often, the translation blocks that are in the cache are linked together if they are connected by the jumps at the end of the block and the next time the blocks execute they will run one after the other straight away, without returning to the main loop in between [11]. See figure 3.2 of how already connected blocks may execute in between the prologue and epilogue of the cpu_exec function.

![Figure 3.2: QEMU executing basic blocks of instructions](image)

Translation is done in standard compiler fashion in a backwards cross-compilation way, with a frontend and a backend, but here the frontend is for the target code and the backend is for the host machine. The frontend takes the target code and generates an intermediate format for the Tiny Code Generator (TCG) backend. TCG then generates the final translated block. Because each TCG instruction is something that should have happened on the target platform it can take many actions to emulate this behaviour on the host. To define this behaviour TCG has something called helper functions and the code generator can insert calls to these functions in the blocks [12]. In a QEMU only simulation, these functions will run as part of the simulation of the hardware models in QEMU, but since Ericsson’s EMCA model their hardware with SystemC and TLM, QEMU and the SystemC modules have to be able to communicate somehow.

3.1.2 Simulating Concurrency

The SystemC kernel manages the event driven SystemC simulation programs. Is the kernel that selects which thread or method runs next. The kernel scheduling is non-preemptive, threads and methods have to yield control back to the kernel voluntarily [13] [14]. Only the kernel can start...
processes and only one process can run at once, because the whole simulation is single-threaded. This is of course not how real hardware works, where several threads can run simultaneously.

To achieve fake concurrency the SystemC has a special time called simulation time which allows for a system where the different processes can think they are running in parallel of each other. The kernel has four different states of execution: Initialization, Evaluate, Update, and Advance Time. The kernel also keeps track of the state of all processes.

Processes have three different states: Ready, Running, and Waiting. During the initialization phase all processes that are registered with the kernel are put into either the ready list or the waiting list depending on if they are marked to initialize or not. Processes can also be spawned at any point in the program, using the SystemC spawn function.

After initialization the simulation enters the evaluation state where it selects the processes in the ready list one by one and set them to run. When the ready list is empty the kernel enters the third state, update, where all processes that have called request_update get the update done. This can result in threads being put back in the ready list, meaning threads have communicated in so called delta time, because simulation time has not yet advanced. It can also result in methods being put in the ready list. If there are any processes in the ready list the kernel will loop back to the evaluation phase. If there are no processes in the ready list the kernel advances the simulation time to the next point in time where there is a process that is going to be notified, then return back to the evaluation state. The kernel repeats this process until there are no processes in the ready list and no processes in the waiting list that are going to be notified no matter how much simulation time is advanced. This progression is shown in Figure 3.3.

The way a SystemC thread can yield control to the kernel is by calling the wait function. The argument(s) of the wait call defines what the thread will wait for. If no argument is specified the thread will wait for those events it is statically sensitive to. Allowed arguments are different units of simulated time and events. Where a special case is SC_ZERO_TIME which will yield control to the kernel and the next time the kernel is in the update phase the process will be put in the ready list.

If a process is waiting for an event, that process can be notified by a process calling the notify function. The argument of notify defines which event will be notified, it also accepts an event and a unit of simulated time for sending a delayed notification. There can only be one notification of an event at a time, therefore a new call to notify for the same event will overwrite that notify if the delay of that notification is shorter, if it is longer it will simply be ignored.

Keeping all the DSPs in sync letting each thread executing only up until time should pass, which each instruction does, would require threads to frequently yield control back to the kernel. Executing the platform like this would essentially be a cycle accurate simulator, which is not what the requirements for SVP is. SVP should be functionally accurate and as fast as possible. This is where TLM comes in. Using TLM it is possible to define interfaces allowing a SystemC Thread inside one module call functions directly in another module, and a new time tracking system with a local offset. The local offset is the difference between time in the currently executing thread, and the global SystemC time of the simulation. The limit of how far one SystemC thread is allowed to run ahead of the others is called a quantum.

SVP uses the TLM loosely timed model, which utilizes this local offset. That means that one DSP can execute many instructions, until it either hits an instruction that blocks progression, like accessing a not available resource, or if its the quantum boundary, at which point a forced synchronization happens. When the SystemC kernel advances time it is the global simulation time, each DSP thread also keeps track of how far ahead of the global simulation time it is by counting the cost in time or clock cycles to execute each task. This is the local offset, which is passed along between functions with the TLM protocol.
3.1.3 SystemC/TLM and QEMU

A framework for QEMU, that allow SystemC models to be connected to a QEMU processor via busses, like PCI (Peripheral Component Interconnect) or AMBA (Advanced Microcontroller Bus Architecture) is described in [16]. Another group started from this framework, and through extensive use of the helper functions in TCG, sent enough information to and from QEMU to create a cycle-accurate simulator, simulating the full processor pipeline [17].
The company GreenSocs has developed their own QEMU SystemC framework named Qbox [18]. In Qbox QEMU is run inside a SystemC thread. This gives the SystemC kernel full control of the program flow, and QEMU timers can be kept in sync with SystemC time. The other benefit is that other SystemC models can interface with the SystemC thread running QEMU directly, no additional wrappers needed.

In SVP, like in Qbox, QEMU is run inside each DSP’s execution thread. However, SVP does not use the Qbox library, so a fully custom interface between the C world of QEMU and SystemC is used instead. Code translation for the EMCA has been added to the compiler frontend, and new helper functions have been defined for the backend, used like described in [17]. In SVP QEMU interfaces are defined and communication to other threads are be done using TLM function calls. An overview of the interface between SystemC/TLM and QEMU in SVP is shown in Figure 3.4.

3.2 Digital Signal Processor Model

Section 3.1 mentions that the DSP model is in the SystemC hierarchy, with each DSP being its own SystemC module instance. Inside this module is the execution thread, which is the SystemC thread that interfaces with QEMU where the instructions are fetched, translated and the execution of the instruction initiated. All SystemC threads are managed by the SystemC kernel, so it is the SystemC kernel that decides when the execution thread that contains QEMU gets to run.

3.3 Shared Memory Model

The shared memory interface model is in its own SystemC module, with related threads. Each instance of the shared memory interface module is tied to a DSP module. The DSPs access the memory through this interface, meaning each DSP is interacting with its own shared memory interface threads.

Because ASIC B has three different interfaces with different purposes, SVP for ASIC B has three different modules for the three interfaces. The new hardware unit that is responsible for selecting which interface each instruction should use is also modeled. More on this in Chapter 5.
Chapter 4  Model Analysis using Tracing and Visualization

In this chapter the modified SystemC kernel, the post processing library, and the additions to the library is introduced. The use cases for visualization are explained.

4.1 SystemC Tracing

The reference SystemC kernel by Accellera supports extensive logging functionality for debugging purposes, and can be selectively printed using different log levels [19]. In Roman’s thesis, SystemC Tracing for Observability and Profiling, Accellera’s SystemC kernel is modified to provide simulation traces [4]. The modified SystemC Kernel emits logs for events such as a process suspend, resume, wait call, notify call, and a process being notified. The logs also include backtraces for the wait calls and the notify calls.

4.2 Post-processing

Roman’s work also include a script to convert the logs to a Python database and Python programs to process the data. Using the logs it is possible to calculate general statistics like how many times each process ran, or how long each run was. By looking at the which process calls notify and which thread is being notified it is possible to construct a causality chain of processes. This idea is mentioned in Roman’s thesis but not explored too deeply.

4.3 Visualization

A good way to try to understand something is to visualize it. While the trace logs from the modified SystemC kernel provides a lot of information, digesting that information can be difficult. Filtering the logs can make interpretation easier but identifying what should be filtered out can be hard, and even then it may be difficult to see connections between different thread runs if they are not sequentially close in the log.

To improve the situation a new Python program was written, adding one more function to the existing post-processing tools. The new program draws a causality graph, see Figure 4.1 for an example, showing which threads and methods run, when they run, and why they run, depicting the processes as nodes and the edges between them are the events that trigger the process going from the notifier process to the notified process, thus also showing what the previous process was sensitive to, or waiting for. The graph does not show everything, for example it does not draw
notifications that do not trigger anything and a thread waiting on a time or event that never happens will not be apparent from a graph. The `timed_event` edges are wait calls on a time rather than an actual event. If any node has more than one edge with the arrowhead pointing to that node, it means that the process it represents has run more than once at the same simulation time.

Even when filtered to only include DSP related processes, the graphs quickly grow large, which is why they were needed in the first place, to keep this report clean the graphs are cropped to only show the relevant nodes and edges. The simulation time on the left is censored and replaced with place holders to hide Ericsson sensitive information.

![Causality Graph Example](image)

**Figure 4.1:** One slice of a complete causality graph

### 4.4 Use Cases for Causality Graph

The causality graph program is developed to help with learning how SystemC works and in particular how SVP’s DSP and shared memory interface interact. Seeing what SystemC processes run, when they run, and why they run, is a useful thing for evaluation too, not just understanding. For example seeing a graphical representation might reveal why a simulation using the loosely timed model is not utilizing the quantum.
Another more unexpected use case is that the causality graph can be used to detect some bugs in user programs.

4.4.1 Debugging Software

Semaphores are a versatile tool for synchronization in parallel programs. However, this versatility comes with a risk. Using semaphores wrongly can cause hard to detect bugs. A causality graph can be used to detect such bugs. Imagine two processes executing in an alternating pattern and supposedly forced to do so by locking and unlocking two semaphores. Process A locks semaphore A at the start and frees semaphore B at the end of a function, Process B does the opposite. The function in between the semaphores prints "Hello X". The output of the program is 'Hello A', 'Hello B', 'Hello A', 'Hello B', 'Hello A'. The prints make it seem like the program is executing correctly, but the graph tells a different story. One of the processes is not notified by a semaphore to continue the execution after it has accessed it and is supposed to be blocked, it is notified by a timed event, see figure 4.2. This should not happen as that means that the DSP is not actually blocked by a trying to lock a locked semaphore. The programmer can then go and look at the code knowing which semaphore access is not working correctly. A causality chain for a semaphore working correctly is shown in Figure 5.2

![Figure 4.2: Supposed locking of an already locked semaphore](#)
Chapter 2 describes the memory in a DSP is non-blocking, which means that hardware can continue executing other instructions while the memory instructions are being completed. This Chapter describes how such memory instructions handled in the virtual platform. The first steps of instruction execution is covered in the previous chapter, but now it is time to look deeper and see which SystemC threads are needed, what do they do and how do they interact. To investigate this, a set of assembly programs with simple objectives were written, such as: store this, load this, store and load, lock and be blocked by a semaphore then released, and request a job from the job control unit and then abort the request.

5.1 Shared Memory Access

In both ASIC A and B executing two consecutive store instructions, with no dependency on stores close by, results the SystemC pattern shown in Figure 5.1. The the direct memory interface avoids simulating any hardware and directly moves the desired value from its source to its destination, shown in figure 5.1a.

When it is possible to do a direct memory access the virtual platform will do that, otherwise it will notify the shared memory interface and the memory interface thread will continue with executing the task of storing the values. In the case of direct memory accesses, it is hard to say when the memory instructions are executed from the graph since there is only one DSP related thread, to see this either the SVP memory related logs have to be enabled, which can then be viewed in sequence logs, or prints via a debugger interface has to be inserted in the program itself. The logs show that both stores when using the direct memory interface happen at simulation time t0 and that the cycle counter is still updated with each store instruction. Figure 5.1b shows how a store instruction will be executed when direct memory access is not possible. Notable is that the non-blocking nature of the hardware is captured in the simulation, the execution thread just sends the task to the shared memory interface thread and keeps executing, it does not wait for it to complete before continuing, this is apparent from the DSP thread which is triggered by a timed event, not a notification from the shared memory interface thread that it has completed.

There is one exception to these two patterns. If a DSP execution thread is blocked by a memory instruction the DSP may choose to complete the shared memory access through the shared memory interface directly from the DSP execution thread using the TLM interface instead of calling wait just to let the shared memory interface run to do the same thing.

The instructions in this example could be of an arbitrary number of words, changing the size does not affect these, other than the timings. Section 2.5 states that hardware storing a big
block of memory results in many memory interactions, with the reserved block of memory being unreserved over time. In SVP, simulating all the memory interactions to do the store in the same way hardware does it would be slow, so instead SVP writes the entire block to memory right away. Accesses to the reserved range will still be delayed. The delay is calculated based on the estimated time it would have taken to write up until and including the address that is being accessed.

There is an important difference between the ASIC A and B. Section 2.5 explains that, for ASIC B, different types of memory instructions are distributed by hardware into different ports. This functionality is also modeled in the virtual platform. In this case memory accesses are to non-special memory, so the shared memory interface is where the signal will pass through in both ASICs.

![Diagram](image)

**Figure 5.1:** Traces with (5.1a) and without (5.1b) direct memory interface

### 5.2 Semaphore

In this test program a semaphore is initialized, locked twice, so that the second attempt to lock is blocked by the DSP that does the lock. After some time has passed, the semaphore is released by another DSP, unblocking the first DSP’s execution. The relevant part of the SystemC execution is shown in figure 5.2 for ASIC A, and in Figure 5.3 for ASIC B.

These traces show how the SystemC threads act differently in the scenario where using a value accessed from a shared memory location, in this case a semaphore, blocks the execution of upcoming instructions, unlike in the program in Section 5.1. In Figure 5.1b each memory instruction resulted in the memory interface thread being run. At time t5 in Figure 5.2 and at time t4 in Figure 5.3 DSP 0 attempts to lock an already locked semaphore. This is a shared memory interaction, but because the DSP is not allowed to execute the next instruction, the SystemC execution thread does the memory interaction task and cancels the event that would have triggered the shared memory interface thread since it is no longer needed.

At time t7 in Figure 5.2 DSP 1 executes the instruction to free the semaphore again. This is a normal non-blocking instruction so the execution threads will hit the quantum barrier executing new instructions and the memory interface thread will be notified that it has to finish the task.
For ASIC B in figure 5.3, this part is slightly different. Apart from the timings, it also has the different ports for different instructions and destinations. In this case it is a semaphore instruction, which goes in the special data port to the tree thread instead of the shared memory thread.

Figure 5.2: Simple semaphore program causality grap for ASIC A

Figure 5.3: Simple semaphore program causality grap for ASIC B

5.3 Job Control

In test program for the job control unit the DSP sends a request for a job from the job control unit, waits for some time, then if it has not received the job, it aborts the request. Since there is no job in the job control unit for the DSP to fetch, the job should always be aborted.

SVP’s execution pattern is shown in Figure 5.4 for ASIC A and Figure 5.5 for ASIC B. Here the DSP makes a non-blocking memory access though the memory interface thread itself will block itself as it waits for the job control unit to respond. Then the DSP executes the abort instruction, causing the arbiter to cancel the request and memory interface thread is unblocked. Then the DSP
execution thread is blocked as it tries to read the register that has the information if the abort really aborted the request or if the request was done by the time the abort instruction arrived. This manifests itself differently in the two ASICs. For ASIC A the taskCompleted_event appears to come from the original DSP execution thread, while for ASIC B the completion event is sent from the tree thread.

Figure 5.4: Program requesting job from job control then cancelling request, ASIC A
Figure 5.5: Program requesting job from job control then cancelling request, ASIC B
Chapter 3 describes how SVP handles different memory accesses. This chapter introduces possible simplifications in how these scenarios can be modeled. When simplifying the model, the goal is to reduce the complexity in the simulation while maintaining correct functionality, from a software perspective. Complexity in a code base or simulation can also mean many different things, but here complexity can be read as reducing context switches between SystemC processes.

Reducing the context switches between processes is interesting because any time there is a change in which SystemC process is running, the SystemC kernel has to select which thread will run next will result in overhead. Another reason frequent context switching between threads can be a negative is that a context switch is also a synchronization between threads and the potential performance gains from temporal decoupling is limited as they can no longer run independently. This also makes efforts of parallelizing the execution of the entire simulation on multiple cores on the host machine more troublesome since the processes running on the different cores will have to synchronize frequently as well [20].

Simplifications of this kind may also cause timing-sensitive software to break as the timing the software depends on changes or disappears. If or when this occurs, the situations need to be analyzed, and a decision should be taken, to decide if the simulator is accurate enough and the software needs to change, in order to be more robust, or if the simulator needs to be modified, so that its timing accuracy is restored.

### 6.1 Never use a Shared Memory Interface Thread

One idea is to ignore the circumstances of a memory access that originally affected how the memory access would be executed in SVP as seen in the different examples in Section 5. Just unconditionally finish the task right there and then in the DSP execution thread. The change should preserve the timing information in the TLM functions, it should not replace all accesses with direct memory accesses, it should still simulate the memory interface, just from inside the execution thread.

Because there is already a decision making process if a shared memory access should be done in the execution thread or not, this code section can be changed to be very straightforward. Always do it in the DSP execution thread.
6.2 Use a Shared Memory Interface Thread for Job Control Access

The job control abort program in Section 5.3 illustrates a scenario where the shared memory interface thread is blocked while the DSP thread continues executing. Moving the shared memory interface inside the execution thread will cause it to block instead, and then it will not reach the abort instruction that causes the shared memory interface to unblock. To avoid the simulator getting stuck, this scenario has to be avoided.

The next idea is to be a bit more creative when deciding what to do with an access towards the job control unit. It is possible to program the system so that if there is an abortable access towards the job control unit, complete the task in the same way the original SVP would, but for every other access do it directly in the DSP execution thread. As with the previous change, the shared memory interfaces are still modeled, only the context switch in the SystemC simulation is removed.

There is one further special case to be covered. If there is a shared memory access that has been put in the queue to be executed in the shared memory interface thread, a later memory access in the execution thread should not be done directly in the execution thread either. The later memory access should be put in the queue as well.

6.3 Trusting the Quantum

With the shared memory interface threads mostly eliminated, the next step in simplifying the causality graph is to try and allow the DSP execution threads to run up to the quantum boundary as often as possible. This means removing wait calls in the functions accessed with TLM. The most common not quantum related wait call in the traces seen in the traces from the programs run in this thesis is when the programs use semaphores.

In the original SVP there is a wait call at the start of the function that accesses the semaphore, in order to synchronize with the other threads before accessing the semaphore. The reason for the placement of this wait call is that if a different DSP, locks the same semaphore at an earlier simulation time than the time the currently executing DSP is. This happens when there are two DSP threads starting at the same simulation time but the semaphore lock happens with different time offsets thanks to the temporal decoupling.

Figure 6.1a shows how SVP guarantees that the lock at an earlier simulation time will get the semaphore, but the number of times the threads yield control to the kernel increases fast with the number of semaphore accesses. This is with the shared memory interface thread already removed, if still in place, each semaphore access might be done through the memory thread leading to even more time spent in the SystemC kernel.

Figure 6.1b shows an example of how a timing dependent program could get stuck without synchronization. This is the timing dependent software that is not guaranteed to not get stuck on hardware, even if it does not get stuck when run with the wait call in SVP. The benefits of removing the wait are that it does not cater to this timing dependent program and the short and straight forward execution path in figure 6.1b compared to figure 6.1a will allow for higher quantum utilization.
Figure 6.1: Semaphore execution pattern with (6.1a) and without (6.1b) wait call before access, on the Old ASIC.
7 | Evaluation

In this Chapter the proposed changes from the previous chapter are put to the test and the effects of the changes are evaluated, in comparison to the original SVP.

7.1 Small Test Programs

The first step for this evaluation is to confirm that the applied changes do what they are supposed to. The small programs written to analyze how shared memory accesses work are perfect for the job.

7.1.1 Never use Shared Memory Interface Thread

Figure 7.1 shows the new causality graph for the test from Section 5.1, with direct memory access is disabled so that the memory access is forced to go through the shared memory interface.

There is no shared memory interface thread. The two store instructions successfully store data in the shared memory at the addresses specified and the simulated clock cycle for when the stores are done are the same as when the shared memory interface thread is used. This is because the only difference is the SystemC thread usage, the shared memory interface is still modeled, just from inside the DSP thread.

**Figure 7.1:** Two store instructions, no shared memory interface thread, ASIC A
The second test is the semaphore program from Section 5.2, in which one DSP should be blocked until a second DSP unblocks it. The execution pattern is shown in Figure 7.2. DSP 0 is blocked and then when the semaphore is unlocked by DSP 1, DSP 0 can continue with its execution. Since this is what the program was intended to do, it is considered a correct execution of the program.

For ASIC A the shared memory interface thread is again not used and for ASIC B the tree thread, which is the interface thread towards the semaphores, is not used. Everything is done in the execution thread, which is the intended effect of the change.

Figure 7.2: Semaphore program, no shared memory interface thread, ASIC A

The third test, the job control access and abort, fails with this change. The simulation is unable to proceed when the blocking access to the job control unit is done from inside the DSP execution thread. See Figure 7.3, note the lack of outgoing edges from the threads at time $t_2$, the simulation of the program is stuck. Compare the execution in Figure 7.3 to the original, correct, execution in Figure 5.4 from the previous chapter. The delegation of the request of the job from the job control to the shared memory interface in the original SVP allows the execution thread to issue the abort, which unblocks the shared memory interface, and the DSP can proceed as well. Whereas when there is no shared memory interface thread, the DSP requests the job from the job control unit from the execution thread, and when it is blocked it can not reach the abort. Therefore it may never continue.

Figure 7.3: Job request and abort, no shared memory interface thread, ASIC A
### 7.1.2 Only use Shared Memory Interface Thread for Job Control

With the changes Section 6.2 describes, the execution patterns for the small tests are the same as when the shared memory interface is never used. The only exception is for the special case of the job control access, where it is now done in the same way as it is done in the original SVP.

### 7.1.3 No Synchronization Before Semaphore Access

Removing the synchronization before accessing a semaphore change works well both on its own, see Figure 7.4, and in combination with the change from Section 6.2, see Figure 7.5. The execution of the program still has the functionality of one DSP being blocked by the semaphore and being unblocked by another DSP unblocking that semaphore. In the original execution the DSP it takes the simulation three context switches to complete the unlock, one for switching to the shared memory interface thread, one for the semaphore synchronization, and one to switch back to the DSP execution thread which receives the information. With these changes it takes just one, which is the notification going from one DSP execution thread to another.

![Figure 7.4: Semaphore, with shared memory interface thread and no semaphore synchronization, ASIC A](image)

![Figure 7.5: Semaphore, no shared memory interface thread and no semaphore synchronization, ASIC A](image)
7.2 Production Software Tests

The tests in section 7.1 only indicate that the changes do what they are intended to do. There may be scenarios not covered in the three small tests where the changes will cause the simulation to break. For evaluating the effect the changes has on production software, production software tests is appropriate. To test the different SVP versions the same software tests that are used by the operating system developers and ASIC developers to test their platforms are used to evaluate each SVP version. For diving deeper three tests are randomly chosen from the collection of available operating system tests.

For the rest of this chapter the different versions of SVP, and three tests from the Operating System tests are given short name to associate them with.

SVP0 Original SVP

SVP1 Never use SMI thread

SVP2 Use SMI thread for abortable access towards the job control unit

SVP3 Do not synchronize with other threads before accessing semaphore

SVP4 SVP2 and SVP3 changes together

T0 Operating system software test 0

T1 Operating system software test 1

T2 Operating system software test 2

7.2.1 Pass/Fail

The SVP1 change breaks the job control abort test, but there could be other scenarios where never using the shared memory interface threads could cause failures. Table 7.1 shows the results of running all operating system and ASIC tests on ASIC A, and table 7.2 for the operating system tests on ASIC B. The original SVP as well as the semaphore change pass all tests.

SVP1 which never uses the shared memory interface thread does fail some tests, both for ASIC A and B. A few of the tests that fail do not fail under the same conditions as that in the job control unit abort program where the execution gets stuck never reaching the abort instruction. Some of the other tests that fail finish their execution but produce the wrong values, causing asserts to fail the tests.

SVP2 which uses the shared memory interface threads only when accessing the job control unit allows all operating system tests to pass, for the ASIC tests however, one of the tests that failed in SVP1 still fails.

SVP4 which combines the SVP3 and SVP4 fails the same ASIC test as the other simplified SVPs. Furthermore, SVP4 fails an operating system test, but only when using an older branch of SVP that was checked out at the start of the thesis. Using a newer branch from the main SVP branch and applying the same changes again allows the failing test to pass, so the final result is that all operating system tests to pass.
### Table 7.1: Test cases pass fail, ASIC A

<table>
<thead>
<tr>
<th></th>
<th>SVP0</th>
<th>SVP1</th>
<th>SVP2</th>
<th>SVP3</th>
<th>SVP4</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>100/100</td>
<td>97/100</td>
<td>100/100</td>
<td>100/100</td>
<td>99/100 (old)</td>
</tr>
<tr>
<td>ASIC</td>
<td>59/59</td>
<td>56/59</td>
<td>58/59</td>
<td>59/59</td>
<td>58/59</td>
</tr>
</tbody>
</table>

### Table 7.2: Test cases pass fail, ASIC B

<table>
<thead>
<tr>
<th></th>
<th>SVP0</th>
<th>SVP1</th>
<th>SVP2</th>
<th>SVP3</th>
<th>SVP4</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>92/92</td>
<td>88/92</td>
<td>92/92</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

### 7.2.2 Context Switches

One of the goals and expected results of the changes is the reduction in the number of context switches in SystemC when executing a program. A context switch is when one process suspends and the kernel resumes or starts up another one. Counting the number of times threads suspend shows how these changes impact the simulation. In table 7.3 the total number of suspends is shown for the different versions of SVP for ASIC A. The three tests in the table are three different tests from the operating system tests.

The dominance of the execution thread and the shared memory interface thread is apparent as the number of suspends from SVP0 to SVP1 is more than halved just by removing the shared memory interface thread. For ASIC A the shared memory interface threads make up 68% of all suspends.

Re-introducing the shared memory interface thread in SVP1 increases the number of suspends, but not by a lot. SVP3 shows that the test uses semaphores frequently, compared to SVP0 the number of suspends are about two thirds. SVP4 shows that combining the the changes of SVP2 and SVP3 results in 94% fewer context switches for test T2 when compared to SVP0.

The same results for ASIC B are in table 7.4, which only has one of the three tests available. The decrease in context switches is relatively smaller compared to for ASIC A. This is partly because there are more SystemC processes running, but mostly because for ASIC B’s SVP the shared memory interface threads only make up around 8% of the total number of suspends.

### Table 7.3: Total number of suspends for each version of SVP, ASIC A

<table>
<thead>
<tr>
<th></th>
<th>SVP0</th>
<th>SVP1</th>
<th>SVP2</th>
<th>SVP3</th>
<th>SVP4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>104,072</td>
<td>49,945</td>
<td>50,063</td>
<td>67,297</td>
<td>13,304</td>
</tr>
<tr>
<td>T1</td>
<td>106,933</td>
<td>52,263</td>
<td>52,369</td>
<td>69,772</td>
<td>15,201</td>
</tr>
<tr>
<td>T2</td>
<td>409,713</td>
<td>183,043</td>
<td>183,165</td>
<td>247,071</td>
<td>26,044</td>
</tr>
</tbody>
</table>

### Table 7.4: Total number of suspends for each version of SVP, ASIC B

<table>
<thead>
<tr>
<th></th>
<th>SVP0</th>
<th>SVP1</th>
<th>SVP2</th>
<th>SVP3</th>
<th>SVP4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>123,668</td>
<td>102,134</td>
<td>102,248</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

The way threads can suspend is either by calling wait, or a thread finishing its execution never to be run again in that simulation. Looking at what happens to wait calls for the different SVP versions can give some more information about why the suspend changes happened, Table 7.5 shows the wait statistics for test T0 on SVP A.
The original SVP has a lot of waits both for zero time and non-zero time. Removing the shared memory interface thread in SVP1 results in most wait zero calls to disappear, but wait calls for some time increases by about half of that reduction. Going back to Figure 5.2 and the knowledge that an unblocked execution thread will execute until the quantum barrier, then let the shared memory interface thread execute the non-blocking memory access. If the semaphore synchronization happens inside the execution thread, then it will happen at the quantum barrier, thus the wait will be for zero time. However, in Figure 7.2 when the semaphore access is executed right away inside the DSP execution thread, the wait time to synchronize will no longer be zero. This explains the decrease in wait zero and the increase in wait not zero in SVP1, and SVP2, only in SVP2 some accesses are through the memory interface.

SVP3 has both fewer wait zeroes and fewer wait non-zeroes. This is because sometimes a semaphore will be a blocking access, and thus executed in the execution thread, and the synchronization wait call will no longer be with time zero. This change does not have the same decrease in the number of wait for events. This is because with only the semaphore change, the execution thread still notifies the shared memory interface thread using events.

In SVP4 both the number of wait for time calls and the number of wait for event calls are fewer than those for SVP0.

Table 7.5: Distribution of wait calls for T0, ASIC A

<table>
<thead>
<tr>
<th></th>
<th>wait()</th>
<th>wait(t == 0)</th>
<th>wait(t != 0)</th>
<th>wait(events)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVP0</td>
<td>382</td>
<td>35,009</td>
<td>28,057</td>
<td>40,624</td>
</tr>
<tr>
<td>SVP1</td>
<td>384</td>
<td>347</td>
<td>43,854</td>
<td>5,360</td>
</tr>
<tr>
<td>SVP2</td>
<td>384</td>
<td>328</td>
<td>43,948</td>
<td>5,403</td>
</tr>
<tr>
<td>SVP3</td>
<td>384</td>
<td>296</td>
<td>25,998</td>
<td>40,631</td>
</tr>
<tr>
<td>SVP4</td>
<td>384</td>
<td>301</td>
<td>7,207</td>
<td>5,412</td>
</tr>
</tbody>
</table>

The title of Section 6.3 is trusting the quantum. To investigate if the changes do in fact increase the quantum utilization for the DSP execution thread, the average execution time of the threads is divided by the quantum time. Comparing the quantum utilization of SVP0 to SVP1 and SVP2 in table 7.6 where all memory accesses are moved to the execution thread. Only moving the memory accesses to the DSP execution thread lowers the quantum utilization. This is because the memory interface thread is only used if there are outstanding memory operations when the execution thread hits the quantum barrier. However, each semaphore access has a synchronization which does limit quantum utilization, and this synchronization is moved from the shared memory interface in SVP0 to the DSP execution thread in SVP1 and SVP2. In SVP3 where this synchronization is removed, the quantum utilization is slightly higher, but as most semaphore interactions are in the shared memory interface thread again the increase in quantum utilization is not that high.

It is in SVP4 when both changes are combined that the quantum utilization really spikes, as the benefits of the semaphore change always occur in the DSP execution thread rather than the shared memory interface thread.

7.2.3 Performance

Another benefit of simplification is often that simpler is faster. With a decrease in context switches seen in section 7.2.2, some performance gain is to be expected, even if the combined work done inside the threads stay largely the same. Table 7.7 shows the time spent executing threads and the total simulation time for test T0. The statistics are collected from the logs provided by the modified SystemC kernel. SVP1 spends almost as much time in threads as the original SVP 0,
Table 7.6: Quantum Utilization for T0, ASIC A

<table>
<thead>
<tr>
<th></th>
<th>dsp0.exec_thread</th>
<th>dsp1.exec_thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVP0</td>
<td>20%</td>
<td>21%</td>
</tr>
<tr>
<td>SVP1</td>
<td>13%</td>
<td>13%</td>
</tr>
<tr>
<td>SVP2</td>
<td>13%</td>
<td>13%</td>
</tr>
<tr>
<td>SVP3</td>
<td>21%</td>
<td>24%</td>
</tr>
<tr>
<td>SVP4</td>
<td>71%</td>
<td>106%</td>
</tr>
</tbody>
</table>

but almost a whole second less in the total simulation time. Perhaps a bit surprising is that SVP2 is faster than SVP1, spending 1.3 seconds less in threads and 1.6 seconds less in total, despite doing 100 more context switches than SVP0. SVP3 shows how expensive the synchronization is, decreasing total execution time by slightly more than two seconds. SVP4 needs 58% of the time SVP0 needs executing in threads, resulting in almost a two time speed-up in total execution time.

Table 7.7: Run time for each SVP version for T0, ASIC A

<table>
<thead>
<tr>
<th></th>
<th>Total time in threads</th>
<th>Total simulation time</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVP0</td>
<td>7,805 ms</td>
<td>13,156 ms</td>
</tr>
<tr>
<td>SVP1</td>
<td>7,718 ms</td>
<td>12,011 ms</td>
</tr>
<tr>
<td>SVP2</td>
<td>6,423 ms</td>
<td>11,395 ms</td>
</tr>
<tr>
<td>SVP3</td>
<td>6,903 ms</td>
<td>10,993 ms</td>
</tr>
<tr>
<td>SVP4</td>
<td>4,591 ms</td>
<td>7,474 ms</td>
</tr>
</tbody>
</table>

Breaking down the execution time further for individual threads. Test T0 uses two DSPs and table 7.8 shows the DSP execution threads and the shared memory interface threads. Adding up the five threads in table 7.8 for SVP0 comes to 7,675.123 which is more than 98% of the total execution time spent in threads.

SVP0 spends a lot of time both in the DSP execution threads as well as in the shared memory interface threads. SVP1 which moves the execution of the shared memory interface thread into the execution thread results in new execution times that are pretty close to SVP0’s split execution added together. This is not the case for SVP2 where there is a clear performance benefit compared to SVP0. Again for SVP3 the synchronization cost is more visible in the shared memory interface thread as that is where the mandatory wait call is. SVP4 gets both the benefits of SVP2 and SVP3, with lower execution times for each thread compared to SVP2.

Table 7.8: Distribution of execution time in threads in milliseconds for T0, ASIC A

<table>
<thead>
<tr>
<th></th>
<th>dsp0.exec_thread</th>
<th>dsp0.SMI_thread</th>
<th>dsp1.exec_thread</th>
<th>dsp1.SMI_thread</th>
<th>dsp1.SMI_abort</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVP0</td>
<td>2979.978 ms</td>
<td>1786.148 ms</td>
<td>1192.876 ms</td>
<td>1716.251 ms</td>
<td>0.049 ms</td>
</tr>
<tr>
<td>SVP1</td>
<td>4701.910 ms</td>
<td>-</td>
<td>2885.118 ms</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SVP2</td>
<td>4040.477 ms</td>
<td>5.816 ms</td>
<td>2248.137 ms</td>
<td>0.213 ms</td>
<td>0.049 ms</td>
</tr>
<tr>
<td>SVP3</td>
<td>2831.491 ms</td>
<td>1441.956 ms</td>
<td>1131.689 ms</td>
<td>1377.650 ms</td>
<td>0.046 ms</td>
</tr>
<tr>
<td>SVP4</td>
<td>3090.429 ms</td>
<td>5.731 ms</td>
<td>1301.383 ms</td>
<td>0.183 ms</td>
<td>0.053 ms</td>
</tr>
</tbody>
</table>

The above performance investigation is using the modified SystemC kernel, which is what allows this insight. However, by logging information about context switches the performance penalty for each switch is also increased, which leads to a simplification reducing context switches showing better performance gains than when used with the normal SystemC kernel. Another shortcoming of this performance check is that the platform is compiled with debug mode which has minimal
optimization. What is possible to do is to run test programs with the program GNU time, version 1.7. The time program measures three different times. Real-time, that is the total time from start to finish. User-time, which is the time spent executing the launched program, in user mode. The last time is system-time which is the time spend executing code in the kernel mode, for example allocating new memory and such. The times for test T0 on ASIC A is shown in Table 7.9.

Table 7.9: Time statistics for SVP4 with normal SystemC kernel compiled in release mode, ASIC A

<table>
<thead>
<tr>
<th>Run</th>
<th>real</th>
<th>user</th>
<th>system</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15.29 s</td>
<td>10.85 s</td>
<td>1.56 s</td>
</tr>
<tr>
<td>2</td>
<td>21.77 s</td>
<td>10.68 s</td>
<td>3.84 s</td>
</tr>
<tr>
<td>3</td>
<td>17.59 s</td>
<td>10.90 s</td>
<td>2.46 s</td>
</tr>
<tr>
<td>4</td>
<td>20.86 s</td>
<td>11.06 s</td>
<td>1.62 s</td>
</tr>
<tr>
<td>5</td>
<td>22.46 s</td>
<td>10.94 s</td>
<td>2.61 s</td>
</tr>
</tbody>
</table>

Adding together user time and system time gives the total time spent executing the program. The average of five runs for SVP0 and SVP4 for the three different tests are listed in Table 7.10. Note that the execution time for the tests are longer than the total simulation time in Table 7.7, even though one is compiled in debug mode and does a lot of logging. This is because the time measured by the time program is the total execution time of running the test, not just the wall time for running the simulation.

Table 7.10: Average run time for user-time plus system-time, ASIC A

<table>
<thead>
<tr>
<th></th>
<th>SVP0</th>
<th>SVP4</th>
<th>diff</th>
<th>% diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>13.790 s</td>
<td>13.304 s</td>
<td>0.486 s</td>
<td>3.5%</td>
</tr>
<tr>
<td>T1</td>
<td>16.648 s</td>
<td>14.936 s</td>
<td>1.712 s</td>
<td>10.2%</td>
</tr>
<tr>
<td>T2</td>
<td>17.872 s</td>
<td>17.398 s</td>
<td>0.474 s</td>
<td>2.7%</td>
</tr>
</tbody>
</table>

The performance gains in Table 7.10 do not correlate with the decrease in context switches for each test in Table 7.3. The expected result should have a bigger performance gain for test T2, not T1.

To as accurately as possible measure the improvements of the section of the program that is affected by the change, without changing the simulation itself, time recording has to start just before the simulation starts and end just after the simulation finishes. The results are summarized in Table 7.11, where the run times are the averages for five runs again and std is the standard deviation. These results are more in line what would be expected from the reduction in context switches in Table 7.3. The performance gains are not as big as in the debug version, which is to be expected considering the logging in the context switches is no longer there to increase the cost of each context switch. There is still some noise, the slowest run of T1 for SVP4 takes longer than the fastest run of T1 for SVP0, but on average the modified SVP is faster for all three tests.

Table 7.11: Average total run time for simulation, ASIC A

<table>
<thead>
<tr>
<th></th>
<th>SVP0</th>
<th>% std SVP0</th>
<th>SVP4</th>
<th>% std SVP4</th>
<th>avg diff</th>
<th>% diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0</td>
<td>2958.8 ms</td>
<td>3.3%</td>
<td>2735.8 ms</td>
<td>1.5%</td>
<td>223 ms</td>
<td>8%</td>
</tr>
<tr>
<td>T1</td>
<td>2943.4 ms</td>
<td>3.8%</td>
<td>2734.4 ms</td>
<td>4.2%</td>
<td>209 ms</td>
<td>7%</td>
</tr>
<tr>
<td>T2</td>
<td>4619.75 ms</td>
<td>1.7%</td>
<td>3869 ms</td>
<td>2.8%</td>
<td>750.75 ms</td>
<td>16%</td>
</tr>
</tbody>
</table>
8 Conclusion

The goal of this thesis was to simplify the models of the shared memory interfaces for the two ASICs. The two changes to how SVP model memory accesses result in a considerable decrease of the number of context switches in the simulation. The performance gain was varied for the three tests, from 7% to 16%, but that is not quite the end of that story. With these two changes the temporal decoupling between DSPs is increased, allowing for higher quantum utilization. This should be useful for the efforts of making SystemC thread execution parallel, which can really increase performance [20]. The change of removing the synchronization for semaphore accesses was only implemented for ASIC A’s SVP, but should have similar impact when done for ASIC B’s SVP.

No operating system tests failed when only removing the synchronization before accessing semaphore or when only using the shared memory interface thread for abortable accesses to the job control unit. However, together they did cause one failure. The exact cause of this failure was not identified, but it does seem likely to fall under the tight coupling between software and platform, as reasoned in section 6.3. This and the case of the other failing test has to be solved before considering incorporating the SVP changes into the real SVP.

The visualization program that was developed uses the existing trace logs to draw causality graphs showing what SystemC processes run by drawing nodes, when they run by having a time axis, and placing the nodes at the corresponding simulation time. The graphs also show why each process ran by drawing edges showing events pointing from the notifying process node to the notified process node. This idea of visualizing how SVP executes a program also turned out to be of some use for catching software concurrency bugs.

The current implementation of the visualization program has some SVP specific code, in particular with how the node names are formatted, but the SVP specific code can either be removed or adapted so that any SystemC simulation using the modified SystemC kernel can view the graphs with meaningful node names.

8.1 Future Work

The changes done to SVP in this thesis do not address the problem of complexity in the code base, only the complexity of the simulation. With more time it would be desirable to attempt to try to simplify the memory interface model itself. With the current changes, memory accesses no longer stack up in the internal queues since the accesses are done right away. It could be possible to then remove these internal queues, maybe even model all memory accesses as direct memory accesses.

In the final version of the modified SVP one test case still fails. The exact reason for these failures was not uncovered, future work can be done analyzing why. The fact that the operating system test started working when using a newer branch of SVP speaks to the problem of how tightly coupled the software is to the platform. However, the ASIC test always failed. Either way,
looking at failing test could be interesting work.

There was some work done extending the functionality of the modified SystemC kernel, but more can be done on this front as well. In particular the size of the logs are a problem when running larger programs. A more efficient way of storing the logs is something valuable to consider working on in the future.
Bibliography


