Data Race Detection for Parallel Programs Using a Virtual Platform

DANIEL HAVERÁS
Data Race Detection for Parallel Programs Using a Virtual Platform

Daniel Haverås

January 31, 2018
Abstract

Data races are highly destructive bugs found in concurrent programs. Because of unordered thread interleavings, data races can randomly appear and disappear during the debugging process which makes them difficult to find and reproduce. A data race exists when multiple threads or processes concurrently access a shared memory address, with at least one of the accesses being a write. Such a scenario can cause data corruption, memory leaks, crashes, or incorrect execution. It is therefore important that data races are absent from production software.

This thesis explores dynamic data race detection in programs running on Ericsson’s System Virtualization Platform (SVP), a SystemC/TLM-2.0-based virtual platform used for running software on simulated hardware. SVP is a bit-accurate simulator of Ericsson Many-Core Architecture (EMCA) hardware, enabling software and hardware to be developed in parallel, as well as providing unique insight into software execution. This latter property of SVP has been utilized to implement SVPracer, a proof-of-concept dynamic data race detector. SVPracer is based on a happens-before algorithm similar to Google’s ThreadSanitizer v2, but is significantly different in implementation as it relies entirely on instrumenting binary code during runtime without requiring code modification during build time.

A set of test programs exhibiting various data races were written and compiled for EOS, the operating system (OS) running on EMCA Digital Signal Processors (DSPs). Similar programs were created for Linux using POSIX APIs, to compare SVPracer against ThreadSanitizer v2. Both SVPracer and ThreadSanitizer v2 correctly detect the data races present in the respective test programs.

Further work must be done in SVPracer to eliminate some false positive results, caused by missing support for some OS functionality such as semaphores. Still, the present state of SVPracer is sufficient proof that dynamic data race detection is possible using a virtual platform. Future work could involve exploring other data race detection algorithms as well as implementing deadlock/livelock detection in virtual platforms.

Keywords
Concurrency, multiprocessing, data race detection, virtual platforms, dynamic analysis
Abstract


Det här examensarbetet utforskar dynamisk detektion av datakapplöpning i program som körs på Ericssons System Virtualization Platform (SVP), en SystemC/TLM-2.0-baserad virtuell platform som används för att köra program på simulerad hårdvara. SVP är en bit-exakt simulator för hårdvara av typen Ericsson Many-Core Architecture (EMCA), vilket möjliggör parallell utveckling av hårdvara och programvara samt unik inblick i programvaruexekvering. Den senare egenskapen hos SVP har använts för att implementera SVPracer, en konceptvalidering av dynamisk detektion av datakapplöpning.

SVPracer baseras på en algoritm av typen happens-before, som liknar den i Googles ThreadSanitizer v2. Stora skillnader finns dock i SVPracers implementation eftersom den instrumenterar binärkod under köring, utan att behöva modifiera koden under kompilering.

Ett antal testprogram med olika typer av datakapplöpning skapades för (EOS), ett operativsystem som körs på EMCA:s signalprocessor (DSP). Motiverande program skrevs för Linux med POSIX-APIer, för att kunna jämföra SVPracer med ThreadSanitizer v2. Både SVPracer och ThreadSanitizer v2 upptäckte datakapplöpningarna i samtliga testprogram.

SVPracer kräver vidare arbete för att eliminera några falska positiva resultat orsakade av saknat stöd för vissa OS-funktioner, exempelvis semaforer. Trots det bedöms SVPracers nuvarande prestanda som tillräckligt bevis för att virtuella plattformar kan användas för detektion av datakapplöpning. Framtida arbete skulle kunna involvera utforskning av andra detektionsalgoritmer samt detektion av baklås.

Nyckelord

Samtidighet i programvara, multiprocessing, detektion av datakapplöpning, virtuella plattformar, dynamisk analys
# List of Figures

2.1 Example of two threads exhibiting a data race .................................. 14  
2.2 Vector clocks example ........................................................................ 18  
2.3 Example of a potential data race that happens-before testing can miss .. 21  
3.1 SW execution on virtual platform ......................................................... 28  
3.2 Simplified memory layout of ASIC ....................................................... 29  
4.1 Diagram of shadow memory ................................................................. 36  
4.2 Graphical representation of vector clock superlist ............................... 38  
4.3 The three modules of SVPracer .......................................................... 42  
4.4 Data race report example ................................................................. 43  
5.1 Simple data race .................................................................................. 45  
5.2 Correctly synchronised shared variable .............................................. 46  
5.3 Concurrent threads accessing adjacent bit fields ............................... 47  
5.4 Double-checked locking...................................................................... 48  
5.5 Simple data race that the extended happens-before algorithm can miss . 50  
5.6 Test results, deterministic tests ......................................................... 52
Contents

1 Introduction 7
  1.1 Background ................................................. 8
  1.2 Problem .................................................. 8
  1.3 Purpose .................................................. 9
  1.4 Goal ...................................................... 9
    1.4.1 Benefits, Ethics and Sustainability ......................... 9
  1.5 Methodology ............................................... 9
  1.6 Delimitations ............................................. 9
  1.7 Outline .................................................. 10

2 Parallel Computing and Data Race Detection 11
  2.1 Parallelism with processes and threads ....................... 11
    2.1.1 Processes ............................................. 11
    2.1.2 Threads ............................................... 11
    2.1.3 Alternative implementations of processes and threads ........ 12
    2.1.4 Multithreading ........................................ 13
  2.2 Time in distributed systems ................................ 15
    2.2.1 Memory locks and synchronisation ......................... 15
    2.2.2 Happens-before relation ............................... 16
    2.2.3 Vector clocks .......................................... 17
    2.2.4 Detecting happens-before relations ...................... 18
    2.2.5 Definitions ........................................... 19
  2.3 Previous work - Existing algorithms and implementations for data race detection 20
    2.3.1 Happens-before testing ................................ 20
    2.3.2 Lockset-based testing .................................. 22
    2.3.3 Hybrid race detection .................................. 23
    2.3.4 Existing data race detectors ............................ 24
  2.4 Implementing dynamic data race detection .................... 24
# Data Race Detection Using a Virtual Platform

## 3.1 The co-dependency of hardware and software

## 3.2 The case for virtual platforms

## 3.3 Ericsson Many-Core Architecture (EMCA)

### 3.3.1 EMCA Memory

### 3.3.2 Zeus DSP

## 3.4 EOS

### 3.4.1 EOS processes

### 3.4.2 EOS threads

### 3.4.3 EOS synchronisation

### 3.4.4 EOS memory management

## 3.5 SVP - System Virtualization Platform

## 3.6 SystemC and TLM

### 3.6.1 TLM 2.0

### 3.6.2 QEMU - Quick EMUlator

## 3.7 Dynamic analysis in virtual platforms

# SVPracer - a happens-before data race detector in SVP

## 4.1 The choice of algorithm

### 4.1.1 Access block entries and shadow memory

## 4.2 Differences compared to ThreadSanitizer v2

### 4.2.1 Vector clocks in SVP

## 4.3 Vector clock management for processes

## 4.4 Managing EOS synchronisation

### 4.4.1 Handling a synchronisation event

## 4.5 Detecting a data race with SVPracer

## 4.6 SVPracer schematic overview

## 4.7 Reporting data races

# Results

## 5.1 Experiments and comparison with ThreadSanitizer v2

### 5.1.1 Data race examples, deterministic

### 5.1.2 Data race examples, non-deterministic

## 5.2 Testing procedure and results, deterministic tests

## 5.3 Testing procedure and results, non-deterministic tests

## 5.4 Slowdown and memory consumption

## 5.5 Evaluation

### 5.5.1 Data race detection

### 5.5.2 Performance

# Conclusion

## 6.1 Future work

## References
Acknowledgements

I would like to sincerely thank my two Ericsson supervisors: Dr. Ola Dahl and Mikael Eriksson. Ola provided academic support and gave frequent input and mentoring to make sure that the project was moving in the right direction. Mikael patiently supported me during the technical implementation phase, sharing his tremendous SVP experience and providing valuable input on design choices for SVPracer.

I sincerely thank my KTH examiner Dr. Zhonghai Lu for accepting me as his student and providing academic input throughout the project.

A great thanks to my line manager Pierre Rohdin G, who dedicated some of his work time to help me with administrative tasks at Ericsson.

Furthermore, I would like to thank my deskmate and fellow master thesis student Roman Birca, with whom I could chat and discuss ideas with.
1. Introduction

In the decades leading up to the early 2000s, advances in computational performance were mainly made through increasing the central processing unit’s (CPU) clock frequency; the rate at which instructions are executed [1]. Advances in silicon manufacturing technology led to smaller transistors which allows more complex circuits to fit on a given size chip. In addition, the smaller transistors’ reduced capacitance allowed for faster switching and less energy consumed per switch [2]. Moore’s law from 1965 accurately predicted that transistor density would double every 2 years - an exponential growth. Similarly, a CPU’s clock frequency also increased exponentially for several years. Starting with processors around 1 MHz in 1973, a 1000x increase to 1 GHz had taken place in 2003 [3]. The industry predicted that clock frequencies would reach 10 GHz before 2010.

This was not the case, and in 2005 frequencies levelled off at 2-3 GHz. Today, a modern desktop processor operates at frequencies around 4 GHz, as is the case for the Intel Core i7 7700K [4]. Further increases in clock frequency are hindered by the power wall, as increases in consumed power would cause unmanageable heat generation [5].

Instruction level parallelism (ILP), where multiple instructions from a single thread can be reordered and issued simultaneously was exploited to increase single-threaded performance. The ILP wall was reached when further attempts to exploit ILP wouldn’t pay off with respect to increases in hardware complexity. Furthermore, memory speeds didn’t increase as fast as processor speeds which lead to the memory wall, where gains in processing speed gave little benefit as the CPU would still have to stall during memory operations.

In the mid 2000s the industry shifted its focus towards multi-core processors [5]. Single Instruction Multiple Data (SIMD) and Multiple Instruction Multiple Data (MIMD) technology enabled further growth in computing performance.

A multi-core processor, while potentially able to give large performance benefits, is difficult to program because concurrent programs must be written in such a way that parallelism is exploited while making sure that the parallel execution is deterministic and correctly synchronised. This means that parallel execution avoids unordered or simultaneous (e.g. unsynchronised) accesses to shared memory, to prevent overwriting or corrupting important data.

If two memory accesses to shared memory satisfy a strict partial order in time, we define them as synchronised. Informally, this means that one access must begin, execute, and finish before the other access can begin and that we know in which order the accesses
will be done for a given execution. This achieves what is known as a \textit{happens-before relation} \cite{6}. When two threads or processes access the same shared memory address, no happens-before relation exists, and one of the accesses is a write, we have a \textit{data race}. Data races can cause data corruption, race conditions, and other unwanted behaviour in code. They are notoriously difficult to detect, and can seem to disappear during the debugging process because of timing differences.

This thesis will explore integrating data race detection in Ericsson’s System Virtualization Platform (SVP), a SystemC/TLM-based virtual platform used at Ericsson to develop, integrate, and test software for baseband and radio processing.

1.1 Background

Ericsson develops hardware as well as software for many-core computer systems. The trends in hardware, as well as in software, indicate an increase in the degree of parallelism, with more cores and more software threads. This, in combination with increasing demands on flexible product development and shortened time-to-market, puts even higher requirements on correct execution of the on-board parallel software, for example in a Radio Base Station (RBS). In order to meet demands on time-to-market, a simulator can be used. The simulator contains models of hardware, and can be used for software development and integration. The simulator is referred to as a virtual platform.

At Ericsson, a virtual platform called SVP is used. It is based on SystemC/TLM 2.0 and QEMU, and it contains models of digital hardware used in baseband processing and in radio processing. From a software perspective the SVP models are bit-accurate, making them functionally identical to target hardware. Using a virtual platform, software and hardware can be developed concurrently. Furthermore, virtual platforms allow for run-time analysis of software in ways that are impractical to implement in actual hardware.

1.2 Problem

Data races can cause catastrophic errors in parallel programs and are difficult to detect through traditional debugging methods. Analysing source code (using static analysis) to detect data races has previously been shown to be imprecise as it results in too many false positives \cite{7}. There is a need to provide the software developers with data race detection tools that can be activated during run-time. Analysing a running program is known as dynamic analysis. Implementing instrumentation for dynamic analysis in hardware is costly and impractical, making it more desirable to implement it in a virtual platform simulating the hardware. Dynamic analysis has previously been used to detect Use of Uninitialized Memory (UUM) in SVP \cite{8}.
1.3 Purpose

This master's thesis will explore methods for implementing simulated hardware instrumentation for detection of data races. The instrumentation shall be used to perform analysis of program execution, by monitoring interactions between parallel processes (i.e. message passing and locks) together with analysis of concurrency-related events such as access of shared resources.

1.4 Goal

The work will be done by extending the capabilities of SVP to include data race detection. Ericsson uses SVP for software debugging in several scenarios, and it is expected that extending the capability to also include data race detection would enable developers to detect a greater number of software errors, simplifying the debugging process at Ericsson.

A proof-of-concept data race detector for SVP shall be implemented. Examples shall be selected for illustrating the capabilities of the data race detector. The thesis shall result in a written report and oral presentation at KTH, as well as a separate written report for Ericsson.

1.4.1 Benefits, Ethics and Sustainability

The project does not involve any form of social research or technology that will be directly interfaced with humans. The software designed in this project is intended to help mitigate problems with running concurrent applications. If concurrency is not handled correctly, it could lead to unexpected behaviour such as crashes and incorrect execution. If this were to happen in a Radio Base Station (RBS), it could lead to an entire cell losing cellular connection.

1.5 Methodology

1. The project will start with a literature study of existing research in the field of data race detection, as well as studying Ericsson hardware and software.

2. Research will be done to find methods for instrumentation and algorithms for data race detection. After deciding the details of implementation, a data race detector will be developed.

3. The data race detector will be empirically evaluated with a set of software use cases known to exhibit data race bugs.

1.6 Delimitations

SVP models two sets of data memories known as SM and LDM. While concurrent processes on a single DSP can share data in both LDM and SM, concurrent processes on
different DSPs can only share data in SM. In order to cover concurrency from processes on different DSPs as well as concurrency from the same DSP, while limiting the scope of the thesis, we choose to focus on data race detection for data in SM. While LDM data race detection would be beneficial, it would require large architectural changes in SVP that would consume development time. Some races will therefore be missed for concurrent accesses in LDM when running concurrent processes on a single DSP.

The developed tool will not be a feature complete product, serving instead as a first draft or proof-of-concept. The goal is to have a tool that can detect data races for a subset of use cases. In the operating system running on the DSPs (known as EOS), a few synchronisation features and types of processes will not be supported, possibly leading to false data race reports. Testing programs will be designed to avoid these use cases.

1.7 Outline

Chapter 2 will begin by introducing concepts related to parallel computing, such as processes, threads and multiprocessing. Following that, time and synchronisation in distributed systems will be introduced, followed by definitions necessary to understand concurrent systems. In Section 2.3 existing algorithms for, and implementations of, data race detection are introduced. Chapter 3 describes virtual platforms in general, Ericsson’s hardware platform and how SVP is implemented to simulate it. Chapter 4 documents the work done in SVP. Chapter 5 evaluates the results of testing the finished data race detector. Chapter 6 contains a conclusion of results and suggestions for future work.
2. Parallel Computing and Data Race Detection

2.1 Parallelism with processes and threads

2.1.1 Processes

A modern computer is expected to do multiple things at once. In a personal computer (PC), a user might require their email client to immediately notify the user when a new mail arrives. This notification must be done without disturbing the computer’s involvement in other ongoing activities, such as listening to music or reading a document. This scenario requires parallel execution of the aforementioned programs, where each program has its own code to execute, its own program counter, and its own set of variables in memory. Unless the computer has multiple Central Processing Units (CPU), such a task is not possible unless the operating system can interleave execution of each program as a separate instance, also known as a process.

In [9], a process is defined as an instance of a sequential program running on a virtual CPU. Each process has its own memory address space, protecting it from interfering with other processes. The process will run for as long as it can, which means either until completion, until it is interrupted by another process, or until it needs to wait for external events. The latter is known as a process being in a blocked state. If a process is ready to run but the CPU not yet allowed to execute, e.g. due to the CPU executing another process or an interrupt handler, the process will wait in a ready state. When the currently executing process enters a blocked state, the operating system will let the next ready process execute. When a process is interrupted it will move to the ready state waiting to continue where it left off. The state terminated applies to a process that has finished executing, either by itself or on demand from another process. Quickly switching between processes gives the illusion of parallelism.

2.1.2 Threads

The concept of processes solves the problem of executing independent operations, either by interleaving the execution on one CPU (processes execute on the same CPU), or by using separate CPUs (processes execute on different CPUs). The execution of the processes is managed by the operating system.
The operating system also manages the memory. In an operating system such as Linux, the operating system allocates separate address spaces to the processes. This means that the operating system ensures that processes either operate on separate regions of the physical memory, or on the same region of physical memory but at different time instants. If two processes want to share a part of the physical memory, this can be managed, with support of operating system functionality for shared memory.

Multitasking processes and management of process memory is expected to exist in most consumer operating systems (OS) today. However, a problem appears when a process wants to perform multiple operations on the same memory address space, and some of these operations are blocking or resource-intensive. For example, a desirable feature in a word processing program is a spell checker, where each sentence must be checked. A spelling error is connected to a single word (unlike a grammatical error where one or more sentences must be checked) and is independent of other words in a text. Therefore, multiple words can be spell-checked in parallel. Performing such an operation with parallel processes is impractical, because all the words belong to the same text file, and the separate address spaces for each processes introduce unnecessary overhead.

A better solution is to divide the different tasks in a process into threads. Much like a process, a thread is an instantiated sequence of events that can operate in an independent manner. The difference between a process and a thread lies in the degree of separation. A process operates in a its own memory address space, with a unique set of global variables, memory pointers, open files, signals and other metadata required to keep track of the process. In contrast, threads within a process usually operate within the same address space. This means that when multiple threads refer to a specific address, they refer to the same physical address as well. In this way, memory is (by default) shared between the threads, without requiring the program to explicitly use operating system functionality for shared memory (which is the case for processes). As a consequence, multiple threads can operate on the same files, global variables, and signals, having full access to each other’s resources.

The aforementioned spell-checker could for example start a large number of threads to check multiple words from the same file concurrently (although the overhead of creating threads must of course be compared against simply running the spell check sequentially in the main process. It is possible that a multi-threaded spell-check is only beneficial for very large text files). Since multiple threads share most of their properties with each other, only their thread IDs, program counters, registers, stack pointers, and operating states (same as states for processes) must be saved and restored when switching between them. A smaller performance hit is incurred compared with switching between processes, as changing the address space for a new process requires executing a task scheduler, flushing the CPU caches and potentially flushing the translation-lookaside buffer (used for translating a virtual address used by a process to a physical memory address) [9].

2.1.3 Alternative implementations of processes and threads

The above definitions of processes and threads are similar to how they are implemented in operating systems such as Windows and Unix/Linux. Processes and threads can be
implemented differently to better suit a particular application. An example is Ericsson’s EOS operating system. The EOS OS does not manage memory as in e.g. Linux, leaving EOS processes with full access to two sets of memories: the shared memory SM (Shared Memory) and the Local Data Memory (LDM). There is one LDM for each DSP. This means that a process running on a DSP can access the LDM on the DSP on which it is running (but not the LDM of other DSPs).

Unlike common “Unix-style” processes and threads, EOS processes and threads can access the entire Shared Memory (SM) simultaneously. There is no memory management unit (MMU) for SM or LDM, and as such memory is physically addressed. Since threads operate in their parent process’ address space, and EOS processes freely accesses memory as mentioned above, EOS threads can also access physical memory belonging to another process. The fact that processes and threads share the address space and have similar read/write privileges in memory make them functionally identical from a data race detection standpoint, even if differences exist in their mode of operation. Therefore, the terms process and thread will be used interchangeably from this point unless otherwise stated. Software and hardware characteristics of the Ericsson RBS platform which runs EOS are discussed more thoroughly in chapter 3.

2.1.4 Multithreading

In the above example involving the spell-checker, threading would be an apt solution to the problem. While the first thread is checking the words in its assigned text segment, a second thread could load another text segment from disk. In this scenario, thread 1 is entirely dependent on the CPU, and thread 2 is almost entirely dependent on disk storage. Assuming both the spell check in thread 1 and the data load in thread 2 take the same amount of time\(^1\), both the CPU and disk will be utilized to their full potential, as the threads take turns using them. This is an example of coarse-grained temporal multithreading. The basic idea is that a thread runs until it is blocked (usually from a memory operation), allowing the CPU to run the next thread that is in the ready queue. When the blocked thread is ready, it is moved back onto the queue.

More advanced implementations of multi-threading include simultaneous multithreading (SMT) which requires a superscalar processor that can not only issue several instructions from a given thread in a single cycle, but also issue instructions from other threads in parallel. This requires extra functional units such as Arithmetic Logic Units (ALU). Furthermore, hardware is required to keep track of the thread ID of each issued instruction. For independent threads in a single process, SMT can be used to implement true parallelism.

It is possible to have two or more CPUs on the same silicon die or package (a multicore processor, common in consumer electronics) or on the same board or system (a multiprocessor, common in servers and high-performance computing). Having entirely

\(^1\)This is almost never the case, disk operations are several orders of magnitude slower than CPU calculations
separate sets of control logic, data paths and functional units, multiprocessors and multicores can execute threads from multiple processes at once. Each processor core can also implement simultaneous multithreading, further improving parallel execution capabilities. For example, the Intel Core i7 7700k has four physical cores on the same die, each capable of running two threads simultaneously [4].

However, multithreading comes at a cost. Since concurrent threads operate in the same address space and have full access to each others resources, a thread can overwrite or corrupt data before another thread has had chance to access it. Accessing memory in such an unsynchronised manner leads to a data race. Consider for example the following code example that could be implemented using POSIX threads:

```c
int count=0;
void *Thread1(void *x)
{
    for (int i =0; i<7;i++)
    {
        cout << count << endl;
        if (count<3)
        {
            count++;
        }
    }
    return NULL;
};
void *Thread2(void *x)
{
    for (int i =0; i<7;i++)
    {
        cout << count << endl;
        if (count>-3)
        {
            count--;
        }
    }
    return NULL;
};
```

Figure 2.1: Example of two threads exhibiting a data race

If Thread 1 were to execute until completion followed by Thread 2, one would expect the variable count to have a value of -3. However, the threads are executed concurrently! During execution, instructions from each thread will be interleaved (or even executed simultaneously in the case of a superscalar or multiprocessor or multicore computer). There is a lack of synchronisation, so both threads can access count simultaneously. Therefore, the order of code statements is undefined and we cannot guarantee that count has the value -3 after execution of the two threads. A data race exists on the variable count.
To make this code correct, synchronisation must be implemented so that only one thread attempts to perform its operations on `count` at any point in time, and in such a way that the accesses are performed in the correct order.

The above example is a simple type of data race, but like all data races it can be very difficult for a programmer to detect. If `count` is later used to decide a program’s flow of control, the non-determinism could drastically change the behaviour of the program. Detecting this kind of data race as well as many others has been an active research field for many years [10][11][12][13].

### 2.2 Time in distributed systems

Consider a single-threaded sequential program. If an event A is coded before an event B, A will happen before B in physical time. This simplicity is lost when dealing with a distributed system. Lamport [6] defines a distributed system as spatially separated distinct processes that communicate through a message passing system. This corresponds to parallel processes executing on separate DSPs in Ericsson’s baseband platforms.

Given two concurrent threads \( T_1, T_2 \) with the same event sequence \{A, B\}, we cannot always be sure that A from \( T_1 \) happens before B from \( T_2 \). This is because we cannot verify that the two threads are executing at the same rate in physical time. The developer is responsible for synchronising the processes’ events to make sure that dependent events are executed in the correct order.

#### 2.2.1 Memory locks and synchronisation

**Semaphores**

A semaphore is a type of variable used to control access to shared resources [9]. There are two kinds of semaphores. The **counting semaphore** is a positive integer value which is used to control the number of times that a shared resource can be accessed. A non-zero value indicates the number of times a resource is accessible, while a value of zero indicates that the shared resource is not available. Consider two threads sharing data through a FIFO queue, one thread pushing elements to the end (producer) and another thread popping elements from the start (consumer). The counting semaphore can then be used to control how many times the consumer is allowed to pop an element.

The **binary semaphore** is a two-valued semaphore often implemented using a boolean. A binary semaphore can be used to implement locks. A process or thread can acquire a semaphore, which means atomically checking if the semaphore is “true”, changing the value to “false” if it was true before, and then continue the process’ execution. Releasing a semaphore increments the value by one (or sets the value to true in the case of a binary semaphore). Attempting to acquire a semaphore with a zero/false value will block the process doing the acquire. It will be blocked until another process or thread releases the semaphore. It is crucial that acquisitions and releases of a semaphore are implemented as atomic operations, meaning that no other process or thread can access the semaphore while an acquire operation or a release operation on the semaphore is ongoing.
Mutexes

A mutex (short for mutual exclusion) is a binary lock that a process or thread can lock before accessing a shared resource, followed by an unlock upon completion. This makes mutexes very similar to a binary semaphore, and they can in fact share the same implementation. However there is a difference in their use; a mutex should only be unlocked by the same thread that locked it [9]. Checking the Thread ID can be done to make sure that this is the case. The purpose of a mutex is to make sure that only a single thread enters its critical section at any point in time. This is used to lock a shared resource, which makes sure that no other process or thread can access it until the critical section has finished. There is nothing hindering a thread from accessing the shared resource that is to be protected by a given mutex. It is up to the programmer to make sure that every access to a given shared variable is preceded by a mutex lock and followed by a mutex unlock.

Message passing

Consider a set of discrete computers locally connected over a network, each with an operating system supporting processes. Since there are multiple processes that execute not only in separate memory spaces, but also on separate hardware in physical space, it is not possible to synchronise an order of events using semaphores and mutexes since they are unreachable from another computer. Message passing can be used to synchronise and share data in such a system. A process can send a message to a destination, with contents such as data, process ID and timestamp. The process ID and timestamp can be useful for synchronisation. If the message is sent to a thread on the same computer, the data can be passed by reference which is faster than sending an entire copy of the data. To receive a message, a process calls a receive function. In EOS, the receive function also specifies which message type(s) to receive. The receive function can be implemented either as a blocking function, pausing execution until a message is received, or a non-blocking function that returns nothing if no message is available. In a situation where the source sends messages faster than the receiver can process them, the receiving process may block the sender while its buffer is full [9].

2.2.2 Happens-before relation

Let us assume that there exists a message passing system between processes, where Send is an event that sends a message to a destination (another process), and Receive is an event that pauses a process until it receives a message. In 1978, Leslie Lamport introduced the happens-before relation [6], denoted by $\rightarrow$ and defined as a strict partial order that satisfies:

Definition Given two events $a$ and $b$, $a \rightarrow b$ ($a$ happens-before $b$) if one or more of the following is true:

- $a$ and $b$ belong to the same process, and $a$ is ordered before $b$
• a and b belong to different processes, a is a Send event that sends a message S and b is a Receive event of S.

• a → x and x → b (happens-before relation is transitive)

If no happens-before relation exists between events a and b, i.e a ↗ b ∧ b ↗ a, we say that the events are concurrent.

2.2.3 Vector clocks

To synchronise an order of events between two discrete processes such as in an Internet-based chat application, it is necessary to keep track of the order of messages. Otherwise, the conversation might not make any sense at all! One solution could be to annotate each message with a timestamp, a variable holding the current physical time of the system. The problem is that different systems do not have perfectly synchronised clocks. Furthermore, even if two clocks were perfectly synchronised at a certain time instant, they might drift apart over time if they do not run at exactly the same rate.

A solution that does not rely on keeping track of physical time is to implement logical clocks. Time is abstracted as an integer value that is incremented by one for every event. We can then conclude that if an event A happens before an event B, A must have a lower clock value than B (Lamport’s Clock Condition [6]). Lamport showed that logical clocks alone are not enough to determine a partial order for events in a distributed system because the reverse clock condition is not always true (If A has a lower logical clock value than B, we can not be sure that A happened before B).

To determine an order of events using clocks, we need a type of clock such that when A has a lower clock value than B, we can be sure that A happened-before B. This is called the Strong Clock condition. A practical solution, known as vector clocks, was introduced in 1988 by Colin Fidge [14]. Assume a system has N processes. For each process $P_i$ in $\{P_1...P_N\}$, we assign a vector $V_i$ in $\mathbb{R}^N$. Each element in a vector is a logical clock corresponding to a process $P_i$ in $\{P_1...P_N\}$, where the logical clock is an integer that is incremented as new events occur. Consider a system with three processes $\{x, y, z\}$. The vector clocks are then managed as follows:

• Every vector $V_x, V_y, V_z$ is initialised to zero. (A)

• For every internal event in a process $P_i$, its own element with index i is incremented by one. (B)

• When sending a message to another process, the sending process’ own element is incremented by one, and a copy of the incremented vector is sent to the receiving process. (C)

$\text{\footnotesize{\textsuperscript{2}}}$Sometimes, not every internal event has to increment the vector clocks. Instead, only events that can affect other threads (such as shared memory accesses) could be taken into consideration.
Upon receiving a message, the receiving process’ own element is incremented by one. For each other element, the vector clock is updated with the maximum of the thread’s own current value and the received value. (D)

Furthermore, a vector clock’s elements must not be decremented for the entire duration of the program execution as this would break the ordering. See Figure 2.2 for an example. Here, three processes are initialised to zero. After each event, the own clock element is incremented by one. Note how the vectors become synchronised after each message (marked as arrows in the graph).

Figure 2.2: Vector clocks example. The arrows indicate happens-before arcs such as message passing.

2.2.4 Detecting happens-before relations

Let $V(a), V(b)$ be the respective vector clocks for two separate processes at respective events $a, b$. Let $V(a)_i$ be the element at index $i$ in $V(a)$. Then, the property $V(a) < V(b)$ holds if and only if:

- There exists an index $i'$ where $V(a)_{i'} < V(b)_{i'}$, and,
- For all indices $i$, $V(a)_i \leq V(b)_i$
A property of vector clocks is that $\mathbf{V}(a) < \mathbf{V}(b) \iff a \rightarrow b$ [10] (a vector clock for an event $a$ is less than a vector clock for an event $b$ if and only if $a$ happened-before $b$). In other words, vector clocks satisfy the Strong Clock condition. We can therefore keep track of happens-before relations by updating and comparing vector clocks.

### 2.2.5 Definitions

**Data races**

Two threads $T_1, T_2$ exhibit a data race if they access a memory location $m$ concurrently (no happens-before relation exists, as defined in 2.2.2), and at least one of the accesses is a write.

We define three classes of data races, named after the order in which the events are observed in memory: write-write, write-read, and read-write. Of course, this order is not bound by any ordering in code since data races are unordered in code by definition (they are concurrent). The order merely denotes in which order the memory operations actually finished.

**Locks**

A lock $l$ is a variable or object that a thread can acquire to gain exclusive access to a memory location (an address range). A lock can restrict read access, write access, or both. Releasing the lock allows another thread to acquire it. Variables are sometimes not bound by the lock as is the case with mutexes; each thread is expected to cooperatively acquire the lock before accessing shared memory. An example of strict locking of variables is the protected object in the Ada programming language, where it is impossible to access the object’s contents while it is locked.

**Locksets**

A lockset $L_s$ is a set of locks currently held by a thread $T$ [10].

**Atomicity**

An operation is atomic if it appears to be a single uninterruptible event for other threads in a system. Such an operation can for example be done by locking a mutex before accessing shared resources, followed by an unlock on the same mutex. An atomic operation will either succeed (correctly acquire the resource and perform operations on it before releasing the resource), or fail (the resource can not be acquired and nothing will be changed).
2.3 Previous work - Existing algorithms and implementations for data race detection

There are several existing implementations of dynamic detectors utilising the concepts mentioned above to detect data races. Subsections 2.3.1, 2.3.2 and 2.3.3 will introduce three different methods for this, first introduced together in [10]. In subsection 2.3.4, existing data race detectors using these methods will be outlined.

2.3.1 Happens-before testing

Happens-before testing has been used in several data-race detectors. One variation is the extended happens-before state machine, for example described in [11] as the “pure happens-before mode”. It extends the happens-before relation by also monitoring memory unlock/lock arcs. This way, a partial order of events can be established for synchronisation that does not rely on message passing.

Accumulating extended happens-before pairs

Two events $e_i, e_j$ satisfy the extended happens-before relation $e_i \xrightarrow{\text{ext}} e_j$ if one of the following is true:

- $e_i$ and $e_j$ belong to the same thread/process, and $e_i$ is ordered before $e_j$
- $e_i$ and $e_j$ belong to different threads/processes, and one of the following is true:
  1. $e_i = \text{Send}(S)$ and $e_j = \text{Receive}(S)$, for a message $S$ or,
  2. $e_i = \text{Unlock}(l)$ and $e_j = \text{Lock}(l)$ for a lock $l$, with $e_i$ being observed before $e_j$
- $e_i \xrightarrow{\text{ext}} e_x$ and $e_x \xrightarrow{\text{ext}} e_j$ (extended happens-before relation is transitive)

Note that for the Lock event $e_j$ in the Unlock/Lock arc, either Unlock or Lock must operate on a write lock (i.e. not only a read lock) to the shared resource. Multiple threads reading from a shared resource concurrently is not interesting because it is not a data race as defined in 2.2.5. Therefore we do not need to consider synchronisation between concurrent reads. For each pair of events $e_i, e_j$, we can then detect a potential data race when all of the following are true:

Data race, extended happens-before testing:

- $e_i, e_j$ belong to different threads/processes
- $e_i, e_j$ are memory access events to the same memory location $m$,
- $e_i$ and/or $e_j$ is a write event
- There exists no relation $i \xrightarrow{\text{ext}} j$ nor a relation $j \xrightarrow{\text{ext}} i$. 
An extended happens-before detector can miss some data races. Consider the following code:

```c
//missable race
pthread_mutex_t* mtx;
int foo;

void *Thread1(void *x)
{
    foo = 1;
    pthread_mutex_lock(mtx);
    //code unrelated to foo here
    pthread_mutex_unlock(mtx);
}

void *Thread2(void *x)
{
    pthread_mutex_lock(mtx);
    //code unrelated to foo here
    pthread_mutex_unlock(mtx);
    foo = 2;
}
```

Figure 2.3: Example of a potential data race that happens-before testing can miss

Here, it is clear that there is a data race because the variable `foo` is accessed outside of the mutex. However, consider this thread interleaving:

1. Thread 1 is scheduled first and writes to `foo`.
2. Thread 1 locks the mutex.
3. Thread 2 is scheduled. It halts when attempting to lock the mutex.
4. Thread 1 unlocks the mutex and finishes execution.
5. Thread 2 locks and unlocks the mutex.
6. Thread 2 writes to `foo` and finishes execution.

For this particular execution pattern, there is a happens-before arc between step 4 and 5. Therefore, no data race will be reported. Detecting the data race requires an execution pattern where Thread 2 is scheduled first. The data race will not always be reported. This can sometimes be solved by repeating the test multiple times, allowing other thread interleavings to execute and the data race to be detected. If Thread 2 were to use a delay before locking the mutex however, the data race will be very difficult to detect with happens-before testing. It is always considered a good idea to avoid relying on timing for synchronisation for other reasons than this one.
Furthermore, checking the happens-before relation for each and every memory access event is very computationally expensive if full vector clocks are compared [10]. However, happens-before detection has the advantage that every race reported is real (although possibly benign, like a counting variable that is used for approximate statistics [11]). When used in large-scale projects, this is a very important feature as a lot of time would otherwise be wasted on trying to fix perfectly working code.

An optimisation was introduced in FastTrack [15], where each event only stores its scalar clock and thread ID as a tuple known as an epoch, denoted $c@T$ where $c$ is the scalar clock and $T$ is the thread ID. It was shown that it is sufficient to compare an epoch with a new vector clock for write-write and write-read data races. This is because writes are totally ordered as long as no data race has been detected up until a given point. FastTrack therefore only guarantees detecting the first data race of a variable, which is often sufficient. However, read-write races often require a comparison between two vector clocks as in unoptimised happens-before testing. For every write at a given address (but not for every read), it is therefore sufficient to store the epoch instead of the entire vector clock.

In ThreadSanitizer v2 [16], most read-write data races can also be detected if a fixed amount of epochs are stored for each memory address (an epoch is stored for both writes and reads, and up to 4 epochs are stored for a 64-bit address range). It has a slightly larger risk of missing a data race if there is a large number of accesses to the same address [17], but has the advantage of having a fixed memory overhead and a ceiling on computational complexity. Comparing an epoch with a vector clock is done as follows:

**Definition** For an epoch $c@T$ for event $a$ (an event $a$ from thread with ID $T$ at logical time $c$) and a vector clock at event $b$ $V(b)$,

$$
c@T < V(b) \iff c < V(b)_T
$$

In other words, the epoch $c@T$ for event $a$ precedes the vector clock $V(b)$ for event $b$ iff $c$ is smaller than the element $T$ in $V(b)$. Then, $a \rightarrow b$. As mentioned above, this property is only guaranteed to hold for write-write and write-read races, until the first data races has been detected. Read-write races can be detected with acceptable precision if multiple epochs are compared.

### 2.3.2 Lockset-based testing

This type of testing is based on checking the locksets of threads involved whenever a shared resource is accessed. It was first introduced in [12] based on lock cover testing introduced in [13]. First, the locksets must be collected as described in [10]:

**Accumulating locks in the lockset**

For each thread $T_n$ at execution step $i$, we maintain and update a lockset $L_{s,T_n}(i)$ for a sequence of access events. A lock $l$ is kept in the lockset $L_{s,T_n}(i)$ if both of the following are true:

$$
There exists an access event $a$ before step $i$, where $a$ is a $\textit{Lock}$ event of lock $l$

There does not exist an access event $r$ before step $i$, where $r$ is a $\textit{Unlock}$ event of lock $l$

**Lockset Hypothesis**

In [10], Callahan defines the lockset hypothesis as follows: \textit{Whenever two different threads access a shared memory location, and one of the accesses is a write, the two accesses are performed holding some common lock.} The naive idea behind lockset-based testing is that whenever this hypothesis is violated, there is a potential data race. Given two different threads $T_i, T_j$ with sequences of events $e_i, e_j$, a data race is reported when all of the following are true:

**Possible data race, lockset-based testing:**

- $e_i, e_j$ contain respective access events $a_i, a_j$ to the same memory location $m$.
- $a_i$ or $a_j$ is a write event.
- $L_{s, T_i}(i) \cap L_{s, T_j}(j)$ is the empty set $\emptyset$. In other words, there are no common locks.

Optimisations can be implemented to ignore duplicate data race reports and reduce overhead [10]. Every race reported by a happens-before detector will also be reported by a lockset-based detector (i.e. lockset-based testing reports are a superset of happens-before reports[10]). Lockset-based testing suffers from false positives, where a data race is reported even if none exists. This can make it impractical to use for larger projects.

### 2.3.3 Hybrid race detection

With extended happens-before testing being computationally expensive and unable to detect every data race, and lockset-based testing being imprecise by over-reporting races, there is clearly a need for a middle-of-the-road “Goldilocks” solution, ideally with neither false negatives nor false positives. A solution was introduced in [10] as Hybrid Race Detection. The general idea is to start with a full lockset-based detector, and then add happens-before detection to double-check possible races. If a pair of events access a common location without locks, this will naturally be reported as a race by the lock-set based detector. However, if we can show that there also exists a happens-before relation between the accesses, no race should be reported as the events are ordered. Of course, implementing a combined lockset-based detector together with a full happens-before detector will only worsen the slow performance of the latter. One solution implemented in ThreadSanitizer v1 [11] is to combine the lockset-based detector with a limited happens-before check that only looks at message passing events, ignoring sequences of lock Unlock/Lock arcs. This is equivalent to checking for Lamport’s classic happens-before relation as defined in 2.2.2.
2.3.4 Existing data race detectors

Dynamic analysis of multithreaded applications has been a topic of research for decades. Listed below are some famous implementations using various algorithms.

- **Helgrind** is a component in the free tool Valgrind [18]. It is an extended happens-before detector, and as such it is rather slow. Like all happens-before detectors, it is also incapable of detecting every data race. However, this also leads to the benefit of every reported race being real.

- **Eraser** is credited as the first lockset-based detector, introduced in 1997 in [12]. The detector was implemented in Tru64, a now discontinued UNIX operating system which was at the time known as Digital Unix. Eraser did not need any source code to analyse dynamic behaviour. Instead, it injected code for analysis into the compiled program to produce a new binary with the same functionality, albeit with slower performance.

- In [10], an unnamed tool was introduced as the first hybrid data race detector. It was more accurate than existing lockset-based detectors and faster than existing pure happens-before detectors.

- **ThreadSanitizer v1** [11] was an evolution of Helgrind, also implemented as a tool in Valgrind. It was a configurable data race detector, and could operate in both pure happens-before mode and hybrid mode. Furthermore, it could be configured to use varying levels of precision for collecting sequences of memory access events, enabling the user to trade accuracy for speed and vice versa. Later, the hybrid mode proved to scale poorly with larger projects and the false positives could not be eliminated entirely [16]. Therefore, ThreadSanitizer v2 [17] abandoned the hybrid mode in favour of an extended happens-before implementation.

- **SimRacer** is a tool for detecting process-level races; races on resources that processes can share such as files, memory-mapped I/O, and hardware devices. While such races are out of scope for this thesis, it is still of interest because it is implemented on a virtual platform in Simics, a full system simulator sharing similarities with Ericsson’s SVP [19]

2.4 Implementing dynamic data race detection

Detecting data races requires rather deep insight into program execution, memory accesses, and OS synchronisation:

- All race detectors need full instrumentation of load and store operations in shared address spaces. Ideally, the race detector should also be made aware of variable names for each respective memory location. A data race report is hard to decipher if no information about variables and source code is given. If the source code is
available, a solution can be to compile with debugging information, as is done e.g. when using GDB with code compiled with option -g.

- Happens-before detectors need to identify happens-before arcs. This requires insight into OS synchronisation features such as message passing. A solution is to use vector clocks as they can be used to detect happens-before relations, as described in 2.2.4.

- Lockset-based detectors need to maintain locksets for each running thread, as described in 2.3.2. For each write event to shared memory, the race detector must compute the conjunction of the thread’s lockset and locksets for other threads that have previously accessed the location.

- Implementing optimisations is almost unavoidable for dynamic data race detection, to improve performance and avoid redundant reports. A race detector that is too slow can negatively affect time-sensitive applications, as was demonstrated when testing Helgrind 3.3 in [11].
3. **Data Race Detection Using a Virtual Platform**

3.1 The co-dependency of hardware and software

A computer system consists of two separate components: a physical hardware platform and a program with a set of instructions to execute. In the early days of computing, the hardware platform and the program to execute were one and the same as was the case with ENIAC (Electronic Numerical Integrator And Computer) [20]. Consisting of a set of arithmetic units, memories and other basic building blocks, the ENIAC computer had to be physically rewired to execute a new program. This process could take several weeks from start to finish. In a way, a new hardware platform was designed for every new application.

Today, the set of instructions is abstracted into computer software. The software is saved in the computer’s memory and as such it can easily be accessed, modified or replaced. Therefore, the modern computer is simple to reprogram by accessing a new program from memory. In contrast, the hardware is rarely changed apart from upgrading it for performance reasons.

When creating new software, a software developer will test the program on the hardware platform in question. If an error is found, the software developer will modify the program and test again until expected behaviour is achieved. The hardware is usually considered static; it is expected to provide all functionality necessary to run the software. This naturally requires the hardware to be finished, tested, and available to the developer. While simple when programming for existing hardware, a hardware-software dependency arises when new hardware and software is to be designed simultaneously as a complete system. This is the case when designing new baseband platforms at Ericsson, where software must be verified to have correct functionality and sufficient performance on target hardware, but target hardware must also be verified to function correctly in itself. Furthermore, a general architecture analysis must be made to make sure that the hardware platform is sufficiently capable.

Hardware is often designed in a hardware description language such as VHDL or Verilog. A component is designed as basic logic in the form of combinational components between clocked registers. Such a design is known as a Register-Transfer Level (RTL) design. The design can be tested by creating a test bench, a sequence of input signals and a clock signal. The output signals can be saved as waveforms and analysed in a
waveform viewer. An RTL simulation precisely emulates the hardware behaviour down to individual logical signal values for each clock cycle, and as such it is an important step in hardware verification. However, RTL simulation is very slow. A sequence of events that would compute in seconds in real-time can take several hours in a corresponding RTL simulation [21]. Some abstractions can be made with a cycle-accurate simulation that models the behaviour of the hardware at clock boundaries, but does not model all logical signals and all logical states. Because it still is necessary to model every clock cycle, which is very slow compared to real hardware, it is not feasible to use cycle-accurate simulation for software testing and architectural analysis. Of course, an option is to wait for the finished hardware but that may result in unwanted delays in the project plan, which in turn will prolong the time to market. The software engineers will be unable to finalise development until they receive final hardware. With RTL and cycle-accurate models being impractical and hardware being unavailable until final development stages, the software developers’ work would be stalled.

3.2 The case for virtual platforms

The solution is to take advantage of abstracted hardware models. A software engineer is not interested in the inner workings of a piece of hardware, focussing instead on its behaviour. If the general architecture of the hardware is available (instruction set architecture (ISA), I/O, interconnects etc.), it is possible to imitate the hardware in a bit-accurate software model. The software model can be interfaced with models of other hardware components to simulate an entire system. This simulator can be run on a host computer of choice, resulting in what is known as a virtual platform (see illustration in Figure 3.1). Software can be compiled and executed as if real hardware were available. This was the driving force behind the development of Ericsson’s own virtual platform SVP, described in section 3.5.
Virtual platforms using abstracted hardware models have multiple advantages compared to RTL simulations and even real hardware:

- They are faster than lower level simulations [22].
- Virtual platforms can be used to develop software before hardware is available, shortening time-to-market. Hardware issues discovered through software development can be taken into consideration, allowing for concurrent development.
- In contrast to hardware, virtual platform parameters are easily modified with respect to memory sizes, CPU core counts etc. This allows for testing of software for different configurations, increasing software robustness.
- Virtual platforms allow for increased introspection capabilities compared to hardware. The internal states of registers and memory can be monitored.
- Virtual platforms can implement dynamic binary instrumentation (DBI), allowing for dynamic analysis of code executing on the VP. This was the goal in [8] for UUM-detection, and is the goal in this thesis for data race detection. This will be discussed in-depth in section 3.7.

### 3.3 Ericsson Many-Core Architecture (EMCA)

EMCA is a highly parallel real-time system architecture running several Digital Signal Processor (DSP) cores in parallel. In this thesis, its implementation in the Olympus
ASIC (Application Specific Integrated Circuit) is detailed. Some notable differences exist compared to more conventional computer systems.

3.3.1 EMCA Memory

Three domains of memory exist in EMCA/Olympus (see illustration in Figure 3.2): Shared Memory (SM) which is shared between all DSP cores, and Local Data/Program Memory (LDM, LPM respectively) which are considered local to each DSP core. With
data and programs stored in separate memories, it is a Harvard architecture. Furthermore, the EOS operating system (see Section 3.4) does not employ virtual memory management; physical addressing is used. The LDM does not have a data cache, improving predictability of the system. The LPM can be used as an instruction cache.

SM can be used to share program code and data between DSP cores. Several cores can concurrently access SM without synchronisation. Such unsynchronised accesses have proven to be troublesome when developing EOS software, since they occasionally lead to data races.

In the scope of this thesis, we will focus on concurrent accesses to the SM, as this is the most used interface for data sharing between DSP cores.

### 3.3.2 Zeus DSP

The Zeus DSP core is the processor used in the Olympus ASIC for baseband signal processing. It has hardware support for Multiply-Accumulate (MAC) operations with dedicated registers, arithmetic operations like division and trigonometry, and a set of variable length instructions optimised for signal processing. Multiple instructions can be issued in parallel (utilizing instruction-level parallelism).

### 3.4 EOS

EOS is a lightweight operating system running on the Zeus DSP cores. EOS manages processes and resources, such as memory and dedicated hardware units. It is less strict with resource allocation compared to other real-time systems. Internal OS data structures are accessible to processes with little protection in place. Task management is non-preemptive, meaning that a process switch will not take place unless a process yields, either explicitly or by becoming blocked from waiting on a resource to become available.

#### 3.4.1 EOS processes

As was mentioned in the introduction to Chapter 2, notable differences exist in the implementation of processes in EOS, compared to a more conventional process model such as in POSIX [23]. An EOS process does not have its own heap address space, relying instead on a common heap that it must share with other processes running on the same DSP. This is similar to how threads are handled in most desktop operating systems. The shared heap, and stack areas for each process, are allocated in LDM. Processes remain on a DSP for the entire duration of their execution.

There are different types of processes in EOS. Static processes are predefined in both program code and XML files, and will start executing immediately after EOS has finished booting. Dynamic processes are created during runtime, based on a process prototype in configuration XML files. Several instances can be created of the same dynamic process, with the possibility to share a single LDM stack.
When a process yields control, it can either be done explicitly through a yield function, or from other functions that trigger a context switch. An example is when execution is blocked, such as when a process waits for the reception of a message, as a result of calling the message receive function. The OS will then perform a context switch to the process with the highest priority among the processes that are ready to run.

Processes are assigned process IDs (PIDs), which for example are used as addresses when sending messages.

3.4.2 EOS threads

EOS threads are used for jobs executed during a limited amount of time. They are used to perform short, parallelisable tasks and run on a designated set of DSPs known as the thread pool. Up to 255 instances of the same thread can be created, sharing user data context between them. An important feature of threads is that they inherit the PID of the process that spawned them. They can therefore be considered parallel forks of their parent process.

3.4.3 EOS synchronisation

The synchronisation features in EOS are of great interest in this thesis, because their use (and misuse!) must be monitored to detect data races.

Semaphores and mutexes

Hardware binary semaphores can be used in EOS. They must be used as follows:

1. A semaphore is allocated, with a semaphore ID.
2. The ID is distributed to all process that will use the semaphore.
3. A process must disable all interrupts before acquiring the semaphore.
4. The semaphore is acquired, critical code is executed, and the semaphore is released.
5. Interrupts are enabled by the same process that disabled them.
6. The semaphore is freed when no longer needed.

It is crucial that an EOS semaphore is released before it is freed, because their state is retained when another process allocates it. If a semaphore is allocated when already locked, execution will halt.

Semaphores are also used to implement EOS mutexes. Mutexes can be used directly by supplying a key, a unique number, to lock or unlock. The mutex functions take care of disabling/enabling interrupts and acquiring/releasing pre-allocated mutex semaphores automatically.
**EOS message passing**

A set of send and receive functions exist in EOS to implement message passing. They are used to distribute semaphore IDs, data, and pointers to SM between processes. Message passing is also used for ordering events. If the process ID of a receiving process is not known prior to sending (for example when replying to a message from an unknown sender), a hunt function can be called to retrieve it.

### 3.4.4 EOS memory management

As shown in Figure 3.2, each DSP has two local memories (LDM and LPM) and a Shared Memory accessible through a SM interface. Data has to be moved to LDM or a CPU register before an instruction can perform operations on it. When a process terminates, all LDM memory allocated to that specific process is freed. For processes, local and Shared Memory can be allocated and freed manually through EOS functions. In contrast, individual local heap memory addresses allocated to a set of threads can not be freed until the threads terminate. The only way to free heap memory in threads is to scratch the entire heap through a specific EOS function.

### 3.5 SVP - System Virtualization Platform

The EMCA hardware, including but not limited to DSPs, memory, I/O, and accelerators, is implemented in SVP as functionally equivalent SystemC/TLM models. The DSP models also share an interface for translating Zeus instructions into host instructions executable on the host platform. This translator is implemented using QEMU (Quick Emulator) software [24]. The TLM modules together with the QEMU translation engine are combined in SVP, resulting in a bit-accurate virtual platform that can be executed on a host computer of choice. The following subsections will give a short overview of SystemC, TLM, and QEMU and how they are used in SVP.

### 3.6 SystemC and TLM

SystemC is an extension to C++ consisting of classes, macros, and data types for event-driven system modelling. It can be used for modelling at Electronic System-Level (ESL). It supports various levels of abstraction of architecture and time. Being implemented with C++, it is object-oriented. Hardware models can be created in C++ classes called modules, comparable to entities in VHDL. These modules are declared with ports, signals, functions, and sequential processes in a manner similar to other HDL languages. Modules can be connected to model complex circuits. A SystemC model is simulated by the SystemC kernel, which models the parallel nature of hardware.
3.6.1 TLM 2.0

Transaction-Level Modelling (TLM) can be seen as a high-level abstraction for interfacing modules. The idea is to replace low-level transfer protocols, where individual ports are read and written to, with function calls. This means that TLM models the transfer of data, ignoring details on how the transaction is actually carried out in real hardware. Not only is it more convenient to use compared e.g. to RTL, it is also faster to simulate because of the reduced complexity. The TLM concept is not limited to SystemC, but is often associated with its implementation in SystemC/TLM 2.0.

Instead of ports, TLM 2.0 uses initiator sockets and target sockets for communication. A target socket is used to register a module’s callback functions, that another module can access by calling the function through its initiator socket. For example, the `b_transport()` function is used to model a blocking transport.

The TLM concept provides a transactional, socket-based, interface for models, making it possible to connect models in a standardized way. In SVP, this interface encapsulates the actual behaviour of a model. Examples of behaviour are calculations done by a hardware accelerator (e.g. performing an FFT), transfer of data in a SPI controller model, or instructions executed by a DSP.

3.6.2 QEMU - Quick EMUlator

The Zeus DSP cores execute instructions. The execution of instructions is implemented in SVP using a QEMU interface to translate Zeus machine code to host computer machine code. This is known as Dynamic Binary Translation (DBT). DBT has significant positive impact on the overall performance of SVP.

3.7 Dynamic analysis in virtual platforms

The hardware models in a virtual platform are implemented in software, e.g. using SystemC and TLM. It is therefore possible to extend these models with instrumentation capabilities, to gain insight into what happens during execution of a program [8]. In SVP, every executed EMCA instruction must be processed by the QEMU engine in SVP. Therefore, we can add instrumentation to gain insight into

- Process switches, static data memory, dynamic memory allocations, and stack allocated memory.
- Which EOS processes and threads that exists and are currently executing on a given DSP.
- Which registers and memory addresses that are accessed.

Adding Ericsson’s debugger EMCA-GDB, we also receive vital information of the source code, allowing us to see which C code line the Zeus assembler instructions correspond to through a stack trace.
In [8], SVP was successfully extended with a proof-of-concept detector of Uninitialized Use of Memory (UUM). Reading variables that have not been initialized can result in unexpected behaviour, for example when using uninitialized variables in conditional branches. By maintaining a shadow memory that indicates, for each memory address, whether the corresponding actual memory location is initialized, such UUM accesses could be detected.

A similar approach will be used in this thesis. The race detector will be implemented with a monitor that detects certain EOS function calls. For data race detection using the happens-before algorithm, it is necessary to monitor mutex locks/unlocks, semaphore acquisitions/releases, and message send/receives because these are happens-before arcs. Furthermore, a SM monitor will be implemented to track each shared memory access.
4. **SVPracer - a happens-before data race detector in SVP**

4.1 **The choice of algorithm**

Of the three algorithms specified in 2.3, the extended happens-before algorithm described in subsection 2.3.1 was deemed the best fit for SVPracer. EMCA is a distributed many-core system and synchronisation between processes can be done with message passing alone. In such a system, testing for the lockset hypothesis would potentially result in a large amount of false positives; an undesirable trait in a debugging tool. The hybrid algorithm (described in subsection 2.3.3) could alleviate this, but it is difficult to implement and fine-tune in such a way that it removes every false positive report without hiding any true positives. In contrast, the pure happens-before algorithm is rather simple to implement (although the data collection and synchronisation operations necessary to make it work is significantly more complicated). The algorithm was combined with a custom-made framework for shadow memory, vector clocks, synchronisation handling and error reporting, forming the tool SVPracer.

The SVPracer algorithm shares many functional similarities with ThreadSanitizer v2. The implementation has some notable differences, detailed in the following subsections:

4.1.1 **Access block entries and shadow memory**

For every memory access, metadata about the access is stored in a struct called an *access block entry*, with the following contents:

- **PID**, or process ID, an unsigned 32-bit integer used to identify the source of the access.
- **is_write**, an 8-bit integer with value 1 for write events and value 0 for read events.
- **clock**, a 64-bit unsigned integer storing the epoch of the accessing process. For a process P, the epoch is retrieved from element P in P:s vector clock.

This struct is nearly identical to the one used in ThreadSanitizer v2, with only the bit widths of the fields being different [17]. For every byte address in SM, SVPracer maps an access block holding up to N access block entries, with N being set to 4 at the time of writing. The access blocks are stored in a shadow memory array with one element per...
byte in SM. For each access block, the corresponding byte address in SM is also used as the access block’s index in the shadow memory array.

Additional access block entries will be assigned in proportion to the size of the transaction, because one entry is created per accessed byte. See Figure 4.1 for an overview of the shadow memory.

**Shadow memory array (indexed with byte addresses)**

| 0x1ffffa | 0x1fffb | 0x1ffffc |

Figure 4.1: Diagram of shadow memory, showing access block entry 1 contents for address 0x1fffb. The entry has logged a write access by process 1 at time 1.

If a memory access happens at an address that already has four populated access block entries, an entry that happened-before the current entry will be evicted. If no such entry exists, a random block is evicted. The number of access block entries per address can be configured. A higher number of entries should theoretically decrease the risk of missing a race when an unsynchronised access is preceded by a large number of synchronised accesses from the same thread, as the preceding offender would be evicted in such a case. This comes at the cost of greater memory overhead.

### 4.2 Differences compared to ThreadSanitizer v2

ThreadSanitizer v2 is a thread-level data race detector that detects data races within a virtual address space that belongs to a single process. SVPracer is an EOS-process-level data race detector that detects data races in a shared physical address space in a many-core system. The SVPracer shadow memory approach differs from ThreadSanitizer v2, which assigns four access block entries per 64-bit address range in memory and stores the accessed byte range in the address block entries to detect overlaps. The ThreadSanitizer v2 shadow memory and application memory operate in the same user space.

While using similar data race detection algorithms, ThreadSanitizer v2 and SVPracer use fundamentally different implementations. ThreadSanitizer v2 relies on instrumenting the code during compilation, so that the data race detection algorithm becomes interleaved with the code of the application. As such, both are simultaneously executed
at runtime. In contrast, SVPracer is independent from the compiler. SVPracer instead does the analysis at runtime by detecting translations of synchronisation functions and tracking TLM transactions. SVPracer can therefore be used to analyse any EOS binary (with some restrictions, see 5.5). However, compiling with debugging enabled will yield more informative back traces in the reports. It is important to note that SVPracer needs knowledge of EOS, including but not limited to semantics, names, and symbols. This info is hard-coded into e.g. the function call hook handler and process handler.

4.2.1 Vector clocks in SVP

Before vector clocks could be implemented, it was necessary to find a way to uniquely identify each EOS process. This was not possible using their PID alone, since parallel processes on different processors can have the same PID. By concatenating the DSP number with the local PID as ”dspXXpidXX”, a unique string was assigned to each process. Using GQuark functions from the GNOME library [25], a unique unsigned integer (a GQuark) can be assigned to each string as an identifier. Mapping from a string ID to an integer ID is helpful when indexing the process names in an array, and also makes it possible to pass IDs by value between functions.

Every process is assigned its own vector clock. It was deemed impractical to implement the vector clocks as static arrays, as this would require allocating thousands of vector clocks holding thousands of 32-bit elements each (to account for the maximum number of unique processes on the Olympus platform), resulting in a considerable memory overhead.

GLib’s datalists were used to map GQuark identifiers to values. Vector clocks were created as datalists mapping scalar clock values (an integer) to a GQuark (the identifier, representing a process). Each process is represented in every vector clock as a scalar clock, and every process owns a vector clock. The required storage space for the vector clocks can therefore be modelled by $O(N^2)$. The vector clocks are stored in a top datalist called superlist, also indexed with GQuarks. To retrieve a scalar clock, it is therefore necessary to have two GQuarks; one identifying the process we want to retrieve the vector clock for and another one to retrieve the scalar clock inside the vector.
4.3 Vector clock management for processes

EOS supports a number of different process types. A static process is created after the OS has booted, and remains on a DSP for its entire lifetime. When EOS boots, every static process will eventually be switched to once for initialisation. The first process to start is the OS process for each DSP, requiring SVPracer to initialise its vector clock. The OS process will then spawn the first application process, with SVPracer creating a corresponding vector clock. Spawning a process actually constitutes a happens-before arc, so SVPracer must increment the OS vector clock followed by copying it into the application’s vector clock. Otherwise, false races would be reported because the OS and application processes write to common locations in SM on startup.

EOS also contains dynamic processes. These are created by a ”parent” process that calls a spawn function. This is also a happens-before arc, just as when the OS process spawns static processes. SVPracer will create a vector clock for the new process which inherits its values from the parent process’ vector clock.

There are other kinds of processes in EOS that are currently not supported in
SVPracer. Some minor tweaks must be added in the SVPracer process handler to add support for these.

4.4 Managing EOS synchronisation

EOS supports semaphores, mutexes, and message passing for synchronisation. Usage of these synchronisation functions must be tracked to keep the vector clocks at the correct values. Hooks were added to SVP, to monitor the Program Counter register. This enables detection of calls to EOS functions for semaphore acquire/releases, mutex locks/unlocks, and message send/receive. This worked well for mutexes and message passing, where the exact time instant of the function calls could be intercepted. Parameters such as mutex IDs and message destinations could be acquired from function call parameters and return values, available in various DSP registers.

This did not work as expected for semaphores, because the EOS "functions" for them are actually macros for an inlined semaphore function. This means that the semaphore function’s code is copied into the calling program’s body. As such, there is no static program address that SVPracer can search for and place a hook at. Semaphore support was omitted from the scope of this project. Luckily, EOS only supports binary semaphores which are almost identical to mutexes functionality wise (as defined in 2.2.1). The test programs can in most cases be rewritten to use mutexes instead.

4.4.1 Handling a synchronisation event

Sending a message or unlocking a mutex requires SVPracer to store a snapshot of the source process’ vector clock, which can then be merged into the receiving process’ vector clock upon message reception or mutex locking. Snapshots are stored in a separate top datalist called sendlist, as to not interfere with the active vector clocks in superlist. sendlist is otherwise functionally identical to superlist. The procedures for handling synchronisation events are as follows:

**Message send receive**

1. A function call for message send is intercepted. Call parameters and the sending process’ DSP and ID are retrieved.

2. An identifying message string is created using the message type number, sending DSP, sending PID, receiving DSP, and receiving PID.

3. A GQuark ID is generated from the message string. The sending process’ vector clock is copied to a new snapshot, and stored in sendlist with the GQuark ID.

---

1 An alternative solution has been discussed, which could be implemented in future work on SVPracer. This alternative implementation would track accesses to semaphores in memory instead of monitoring function calls.
4. A receiving process calls the message receive function, which is intercepted by SVPracer. The return value (the message itself) from the receive function is retrieved from a DSP register.

5. The receiving process recreates the identifying message string and the GQuark ID. The vector clock snapshot is retrieved from sendlist.

6. The snapshot is merged into the receiving process’ vector clock according to the rules specified in 2.2.3.

**Mutex unlock/lock**

1. A function call for mutex unlock is intercepted. Call parameters and the unlocking process’ DSP and ID are retrieved.

2. The mutex ID is converted into a unique identifying string.

3. A GQuark ID is generated from the mutex string. The unlocking process’ vector clock is copied to a new snapshot, and stored in sendlist with the GQuark ID.

4. A function call for mutex lock is intercepted in another process. The mutex key is retrieved.

5. The locking process recreates the mutex string from the mutex key, and retrieves the GQuark ID. The vector clock snapshot is retrieved from sendlist using the GQuark.

6. The snapshot is merged into the locking process’ vector clock according to the rules specified in 2.2.3.

Note how the send/receive arc has an explicit sender and receiver for the snapshot ID (and therefore the direction of the vector clock synchronisation is deterministic), while the mutex ”receiver” is determined randomly by whichever process that captures the mutex first.

### 4.5 Detecting a data race with SVPracer

SVPracer detects data races by monitoring every memory access event. A memory access in SVP is implemented as a TLM transaction with a generic payload, with metadata for type of access (read,write or ignore), size in bytes, and destination address. The algorithm is implemented inside the TLM memory controller to analyse the transaction. A memory access constitutes a data race if these four properties are true:

1. The destination address range overlaps with a previous memory access’s address range

2. The previous overlapping and current memory accesses belong to different processes (different PIDs)
3. One of the accesses is a write

4. There is no happens-before relation between the two memory accesses

For each TLM transaction, a new access block entry is created for the access block at the destination address. If the access block is empty, the entry is stored and SVP continues execution. The condition of overlapping addresses is checked implicitly as there will only be a previous access block entry to check at an address if there was a previous access at the same byte address. The algorithm checks the remaining conditions as follows:

**Input:** new_entry

```plaintext
for accessed_bytes[1..M] do
  for access_block_entries[1..N] do
    old_entry ← access_block_entries[n]; // n in [1..N]
    if old_entry.PID == new_entry.PID then
      continue; // no data race, same process did both accesses
    else if old_entry.is_write == false && new_entry.is_write == false then
      continue; // no data race, both accesses were reads
    newAccsClockVal ← getClockElement(new_entry.PID, old_entry.PID)
    // Need to get the scalar clock in the new accessing
    // thread’s vector clock (new entry’s PID) that
    // corresponds to the epoch saved in the old entry
    // (old entry’s PID)
    if old_entry.clock < newAccsClockVal then
      continue; // no data race, happens-before relation exists
    else
      reportDataRace(new_entry,old_entry);
      // all conditions were met, this is a data race
  end
end
```
4.6 SVPracer schematic overview

In figure 4.3, the different domains of SVPracer are shown. The C/QEMU domain is where the binary translation for the DSP code is happening, so this is where we monitor execution with respect to the OS and synchronisation functions. Here, we also keep track of concurrency and vector clocks.

In the C++/TLM domain, we monitor SM in its TLM block. This is where the actual data race detection algorithm is performed. Here, we also handle the formatting of reports.

Because we need to dynamically access the vector clocks and OS info from the C++/TLM domain, an extension interface is in place allowing access to functions and data structures in the C/QEMU domain.

4.7 Reporting data races

When a data race is detected at a given memory address, SVPracer populates a C++ std::set with the offending access block entries and colliding byte addresses. A report handler then processes the data in the set to create a data race report string. The address where the data race occurred is stored in a variable "oldAddress", used to avoid duplicate data race reports for the same address. If the EOS program is compiled with the debugging flag enabled and SVPracer is run in conjunction with a debugger, a backtrace is printed to provide the code line at which the race occurred.
Figure 4.4: Example of SVPracer data race report. In p1.c at line 53, there was an unsynchronised write that overlapped with a previous p2 write (see Figure 5.1 for the POSIX equivalent of this code. The data race is marked with a comment)
5. Results

5.1 Experiments and comparison with ThreadSanitizer v2

5.1.1 Data race examples, deterministic

To test SVPracer, sample programs were written that exhibit different kinds of data races. These programs are written for EOS and compiled to run on EMCA hardware. Google’s ThreadSanitizer v2 was used as a control, to compare performance with an existing tool. Since the sample programs use an API that is specific to EMCA and EOS, together with the fact that ThreadSanitizer v2 is not compatible with EMCA, the programs’ were ported to POSIX and compiled for Linux. ThreadSanitizer v2 is dependent on code injection at compile time (unlike SVPracer which only monitors code during run-time), so the POSIX versions were compiled with Clang and the option -fsanitize=thread to enable ThreadSanitizer v2. The examples below show excerpts of the POSIX test programs, each with a different kind of data race. Note that all programs use pointers for shared data instead of global variables. This is done to mirror the EOS test programs as closely as possible, where pointers are used to access data in SM.

**Simple data race on two shared variables**

Two processes (or threads for the POSIX version) concurrently access shared variables, but the process p1 accesses the variables outside of the common mutex. There should be two data race reports for this program, for both foo and bar.
An alternative version of the "Simple data race" program was written, where the variables `foo` and `bar` were correctly synchronised inside the mutex. The purpose of this was to show that neither tool reports an increased number of false positives for such a shared access compared to the "Simple data race" program, proving that the tools detect the synchronisation. Ideally, there should be zero reports in this case.
unsigned int* foo;
unsigned int* bar;
pthread_mutex_t* mtx;

void* p1(void* p1_void_ptr)
{
    pthread_mutex_lock(mtx);
    *foo = 1; //these accesses are correctly synchronised
    *bar = 1;
    pthread_mutex_unlock(mtx);
    return NULL;
}

void* p2(void* p2_void_ptr)
{
    pthread_mutex_lock(mtx);
    *foo = 2; //these accesses are correctly synchronised
    *bar = 2;
    pthread_mutex_unlock(mtx);
    return NULL;
}

Figure 5.2: Correctly synchronised shared variable

Adjacent bit fields in a C struct

A much less obvious data race appears when accessing a common struct. Consider the following C struct:

typedef struct{
    unsigned int a : 4;
    unsigned int b : 4;
    unsigned int c : 4;
}dangerousStruct;

with the following access pattern:
dangerousStruct* foo;

void* p1(void* p1_void_ptr)
{
    foo->a = 1; //this one will collide with "b" in p2
    foo->c = 1; /*this one is actually safe, because the "c"
        member belongs to another byte and will not collide
        with neither "a" nor "b"*/
    return NULL;
}
void* p2(void* p2_void_ptr)
{
    foo->b = 2; //this one will collide with "a" in p1
    return NULL;
}

Figure 5.3: Concurrent threads accessing adjacent bit fields

The problem is that the bit fields are smaller than the smallest addressable range on a byte-addressable computer. When p1 loads a, it also loads b because the OS can not load less than a whole byte. This introduces a data race. The problem is worsened when the Zeus 16-bit word-addressing mode is used, meaning that the smallest possible memory access is 16 bits or two bytes. In this case, two adjacent 8-bit integers members of a struct will also race with each other.

**Double-checked locking**

Double-checked locking [26] attempts to reduce the overhead of locking a variable before initialising a shared variable by checking if the variable foo is initialised (a lazy initialisation). This is not thread-safe, as it will cause a write-read data race when the second thread to execute checks foo.
bool* foo;
*foo = false;
pthread_mutex_t* mutex;

void* p1(void* p1_void_ptr)
{
  if (!(*foo))
  {
    pthread_mutex_lock(mtx);
    if (!(*foo))
    {
      /*some initialisation function here*/
      *foo = true;
      pthread_mutex_unlock(mtx);
    }
  }
  return NULL;
}

void* p2(void* p2_void_ptr)
{
  if (!(*foo))
  {
    pthread_mutex_lock(mtx);
    if (!(*foo))
    {
      /*some initialisation function here*/
      *foo = true;
      pthread_mutex_unlock(mtx);
    }
  }
  return NULL;
}

Figure 5.4: Double-checked locking

5.1.2 Data race examples, non-deterministic

As shown in figure 2.3, there are data races that extended happens-before testing can miss (false negative). The following program was created to test the frequency of such false negatives after multiple runs. Such a test will show if it is sufficient to simply run the data race detectors a few times to make sure that this type of race is not missed. The full POSIX code is provided (including the main) to show how the effect of random thread scheduling was increased by letting the threads sleep for a random amount of time before locking the mutex. This ensures random interleaving order of the threads.
A similar approach was used for the EOS version of the code (not shown here) where a yield function was randomly called in one of the EOS processes, allowing the other process to lock the mutex first. Since SVP simulation of EOS programs is deterministic, the randomness had to be generated by a shell script that writes the random value to a global variable in SM before simulation starts.
void* p1(void* p1_void_ptr)
{
    int a = (rand() % 1000) * 1000;
    usleep(a);

    *foo = 1;
    pthread_mutex_lock(mutex);
    //code here unrelated to foo
    pthread_mutex_unlock(mutex);

    return NULL;
}

void* p2(void* p2_void_ptr)
{
    int a = (rand() % 1000) * 1000;
    usleep(a);

    pthread_mutex_lock(mutex);
    //code here unrelated to foo
    pthread_mutex_unlock(mutex);
    *foo = 2;

    return NULL;
}

int main()
{
    void* ret = malloc(4);
    mutex = malloc(sizeof(pthread_mutex_t));
    foo = malloc(4);
    srand(time(0));

    pthread_t t[2];
    pthread_create(&t[0], NULL, p1, NULL);
    pthread_create(&t[1], NULL, p2, NULL);

    pthread_join(t[0], &ret);
    pthread_join(t[1], &ret);

    return 0;
}

Figure 5.5: Simple data race that the extended happens-before algorithm can miss
5.2 Testing procedure and results, deterministic tests

The four deterministic test programs for each platform (EOS with SVPracer on SVP and Linux/POSIX with ThreadSanitizer v2 on x86) were compiled with debugging enabled, and executed. All tests produce deterministic results, as these data races will always be present regardless of thread- or process interleavings. Every execution of each test therefore produces the same result because there is no interleaving where the data race could be hidden as in the non-deterministic test program. The only difference between executions is the order in which the programs access the shared data addresses, resulting in "mirrored" data race reports ("current P1 access conflicts with previous P2 access", compared to "current P2 access conflicts with previous P1 access"). We check for three different outcomes:

- True positive - A data race exists at the given address, and the data race was reported
- False positive - No data race exists at the given address, but it was still reported
- False negative - A data race exists at the given address, but the data race was not reported
### Table 5.1: Simple data race on two variables

<table>
<thead>
<tr>
<th>Data race reports:</th>
<th>True positives</th>
<th>False positives</th>
<th>False negatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVPracer</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>ThreadSanitizer v2</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Correct result</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### Table 5.2: Synchronised access to two variables

<table>
<thead>
<tr>
<th>Data race reports:</th>
<th>True positives</th>
<th>False positives</th>
<th>False negatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVPracer</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>ThreadSanitizer v2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Correct result</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### Table 5.3: Adjacent bit fields

<table>
<thead>
<tr>
<th>Data race reports:</th>
<th>True positives</th>
<th>False positives</th>
<th>False negatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVPracer</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>ThreadSanitizer v2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Correct result</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### Table 5.4: Double checked locking

<table>
<thead>
<tr>
<th>Data race reports:</th>
<th>True positives</th>
<th>False positives</th>
<th>False negatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVPracer</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>ThreadSanitizer v2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Correct result</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 5.6: Test results, deterministic tests
5.3 Testing procedure and results, non-deterministic tests

Both the POSIX/ThreadSanitizer and EOS/SVPracer programs were tested a few times with different thread interleavings, to identify correct and incorrect data race reports beforehand. Then, automated test cases were written that would run the respective programs a thousand times each. Using information gathered from the prior smaller tests, the number of correct data race reports could be counted.

For both tools, the expected result is that 50% of the tests in each test case should yield a correct data race report, since there are two possible execution patterns with equal probability in the programs. An extended happens-before algorithm can only detect a data race in one of them.

<table>
<thead>
<tr>
<th>Table 5.5: Non-deterministic test program</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N = 1000 )</td>
</tr>
<tr>
<td>-----------------</td>
</tr>
<tr>
<td>SVPracer</td>
</tr>
<tr>
<td>ThreadSanitizer v2</td>
</tr>
<tr>
<td>Correct result</td>
</tr>
</tbody>
</table>

\(^{a}\text{Note that this false positive appeared every time the false negative appeared, which is why the number of results do not add up to 1000.}\)

5.4 Slowdown and memory consumption

To test slowdown and memory consumption, a new test program was written that performs 1000000 writes to an array of 10000 32-bit integers. This was done to evaluate how SVPracer scales with more memory-intensive EOS programs. Furthermore, the small "adjacent bit field" test program from Section 5.1.1 was used as a baseline reference, where most of the execution time is spent on overhead such as booting SVP. These two test programs were run on SVP with and without SVPracer activated. Execution time was measured with the linux `date` command.

<table>
<thead>
<tr>
<th>Table 5.6: SVPracer slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test program</td>
</tr>
<tr>
<td>~1 million memory accesses</td>
</tr>
<tr>
<td>&lt;100 memory accesses (adj. bitfields)</td>
</tr>
</tbody>
</table>

The test was repeated once more, to instead examine memory usage. Using `top`, the peak physical memory usage was logged for each run from the RES (resident set) column.
Table 5.7: SVPracer memory usage

<table>
<thead>
<tr>
<th>Test program</th>
<th>SVPracer on</th>
<th>SVPracer off</th>
<th>Memory overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>∼1 million memory accesses</td>
<td>231 MB</td>
<td>212 MB</td>
<td>1.09x</td>
</tr>
<tr>
<td>&lt;100 memory accesses (adj. bitfields)</td>
<td>211 MB</td>
<td>211 MB</td>
<td>1x</td>
</tr>
</tbody>
</table>

5.5 Evaluation

5.5.1 Data race detection

SVPracer correctly detects data races for applications performing unsynchronised shared memory accesses. A correctly synchronised shared variable does not yield a false positive as seen in the test with a mutex protecting the shared variable. However, further work must be done to improve SVPracer’s precision. For example, every test above produces the same false data race report caused by the processes accessing a common OS resource after being spawned by the OS. This access is correctly ordered, but the synchronisation is invisible to SVPracer. There are some other cases that can yield false positive reports, and others that yield false negatives:

- The EOS message receive function produces writes to SM (as a part of acknowledging the message) before the receive function has returned. Therefore, the receiving process’ vector clock can not be updated in time and a false data race is reported. This SM location is written to by every process using message passing, but there is no data race because the writes directly follow the happens-before arc from send/receive events. These SM accesses should be identified and ignored.

- Spawning a process results in writes to SM which are not using mutexes or message passing for synchronisation (since spawning a process followed by it starting is a happens-before arc). This results in false positives. Further instrumentation must be added when handling a dynamic process.

- EOS processes that use semaphores for synchronisation will yield false data race reports, since SVPracer is currently incapable of monitoring semaphores.

- SVPracer can not track threads. No data races will be reported when caused by EOS threads, as every thread in a set uses the same PID (the PID from the process that spawned them). SVPracer will therefore erroneously identify a set of threads as being a single process. Concurrent writes have to be identified as coming from separate processes to constitute a data race.

As for the non-deterministic program, the hypothesis of around 50% races detected was found to be accurate for both tools, which is expected since they use the same underlying algorithm. Repeating the test a few times is a viable solution for detecting this type of data race when using ThreadSanitizer, as the thread interleavings are handled
in a pseudo-random manner by the operating system. On the other hand, SVP deterministically schedules threads which makes repeated runs meaningless unless artificial randomness is introduced (as was done for this test case). SVP is based on the SystemC simulator which uses temporal decoupling, where an individual thread is allowed to run ahead of the global SystemC time up until what is known as the Thread Global Quantum, the maximum time difference allowed. A solution to the determinism problem is to change this quantum for subsequent runs, enabling different schedulings for each run.

An even better solution is to add lockset analysis to the algorithm, where the data race would be detected every time since the global variable $foo$ in the code is accessed without being inside a lock.

5.5.2 Performance

We see that SVPracer has very little impact on performance for smaller programs, where most of the time is spent on booting SVP. For the larger program, the 4.5x slowdown is in line with the performance of ThreadSanitizer V2 [27], which cites a 5-15x slowdown. It is interesting to note that the slowdown is so similar, when the virtual platform implementation of SVPracer is very different.

The test programs used only two processes, which meant that the vector clocks were rather small. Using a larger number of processes will result in synchronisation operations taking longer to complete, since vector clocks must be merged. This aspect has not been tested in the scope of this thesis. Furthermore, more time will be spent handling switching between processes, since SVP simulates the concurrency processes in a sequential simulator.

The performance of the data race detection algorithm itself will not be affected by a larger number of processes (and in turn, larger vector clocks). This is because the clock comparison always compares a single pair of scalars, making it a constant time operation.

Memory overhead of SVPracer is mostly negligible for the tested programs. While the entire shadow memory is allocated at startup, only the addresses which are actually addressed will add to the physical memory usage thanks to OS virtual memory management. Even if the entire SM were to be accessed during execution, physical memory usage will not increase significantly since the SM (and therefore the shadow memory) is rather small.
6. Conclusion

Programming for concurrent systems has been a challenge for decades, exacerbated by the past decade’s transition to multiprocessor computing. The potential increase in performance comes at the cost of having to implement complex synchronisation operations such as mutexes and message passing, to avoid data races. Data races are highly destructive bugs that can cause problems in certain interleavings of processes and threads. A programmer can not guarantee that a program is race free even after showing that the program has produced the correct output.

Using the happens-before relation [6] with some optimizations from [15] and [17], the data race detector SVPracer was implemented in SVP, Ericsson’s virtual platform for running software on simulated EMCA hardware. SVPracer’s purpose is to show that dynamic data race detection is possible using a virtual platform, without needing special binaries or access to source code. Instead, SVPracer relies on hardcoded information about operating system semantics, and the usage of a debugger to enable data race reporting with source code annotation.

To test the data race detection, test programs were written for EOS and POSIX and used as input to SVPracer and Google’s ThreadSanitizer v2 [17] respectively. Both tools correctly detected all data races, except for the non-deterministic test program which was detected in 50% of cases. SVPracer needs some extra work to eliminate a set of false races. A workaround to identify a false data race report is to examine the data addresses in the reports and compare them to the addresses of variables used inside the tested programs. Of course, this requires source code compiled with debugging information. Furthermore, a subset of EOS’s functionality is not supported in this early build of SVPracer, sometimes leading to false positives or false negatives.

To move SVPracer from the proof-of-concept stage to the deployment stage requires added support for the EOS functionality mentioned in Section 5.5, together with extensive testing to make sure that false results are eliminated. Furthermore, the current build of SVPracer only supports the Olympus platform, since some values, such as the number of DSPs, are hard-coded. These values should be replaced with preprocessor macros that apply the correct values for each platform.
6.1 Future work

- Better back traces would be a desirable feature in SVPracer. Currently, only the back trace of the current process causing a data race is printed. It would be desirable to at least print the code line of the previous access as well.

- Since the majority of the technical implementation is in place, a future project could look into modifying SVPracer to support different data race detection algorithms. They can then be evaluated against each other. The hybrid data race detection algorithm (see 2.3) has shown promising results in other data race detectors.

- In [8], an SVP detector for use of uninitialized memory was discussed. This could be combined with SVPracer into a single tool.

- A deadlock is another unpleasant type of bug in concurrent systems. Consider a situation where a thread $T_1$ attempts to acquire a lock $L_a$, while holding a lock $L_b$ that it won’t release until after $L_a$ is acquired. Meanwhile, a thread $T_2$ attempts to acquire lock $L_b$, while holding lock $L_a$ that it won’t release until $L_b$ is acquired. Here, execution will halt indefinitely. By storing a list of held and requested locks for each thread, a deadlock detection algorithm can be implemented. Also related are livelocks, where two or more threads react to each other in an infinite loop. Compare with e.g. two people attempting to pass each other in a narrow corridor by simultaneously attempting to make room for each other. Code exhibiting such behaviour does not technically stall, but the threads will be unable to leave the sections that they are at.

- There is a distinction between data races and what is known as a race condition. While often related and therefore confused, data races and race conditions are not the same thing [28]. Race conditions occur when timing affects the ordering of events in a program. Consider two threads that correctly acquire and release a lock while accessing a shared variable (thus eliminating a data race). Here, a race condition occurs if the order in which the locks are acquired is not deterministic. Another way of differentiating data races and race conditions is as follows: For a data race, no happens-before relation exists for two memory access events where one is a write. For a race condition, the order of a happens-before relation can differ between subsequent executions. Data races and race conditions are not mutually exclusive, it is possible to have one or the other, or even both. A more in-depth analysis of races is available in [29].

Race conditions will only occasionally be detected by SVPracer, when they exist in conjunction with a data race (this is often the case [28]). Detecting race conditions is generally more difficult, as it requires the tool to know the correct execution path for each run of the program (e.g. three mutexes a,b,c are locked and unlocked in order a,b,c). A solution could be to write asserts for each program, as a way to provide a race condition detector with the intended program execution path.
References


Acronyms

ALU  Arithmetic Logic Unit.
API  Application Programming Interface.
ASIC  Application-Specific Integrated Circuit.
CPU  Central Processing Unit.
DSP  Digital Signal Processor.
EMCA  Ericsson Many-Core Architecture.
EOS  The operating system running on EMCA hardware.
FIFO  First In First Out.
GDB  GNU Debugger.
I/O  Input/Output.
ILP  Instruction-Level Parallelism.
LDM  Local Data Memory.
LPM  Local Program Memory.
MIMD  Multiple Instruction, Multiple Data.
OS  Operating System.
PID  Process ID.
POSIX  Portable Operating System Interface.
QEMU  Quick EMUlator.
**RBS** Radio Base Station.

**RTL** Register-Transfer Level.

**SIMD** Single Instruction, Multiple Data.

**SM** Shared Memory.

**SMT** Simultaneous Multi-Threading.

**SVP** System Virtualization Platform.

**TLM** Transaction-Level Modeling.

**UUM** Use of Uninitialized Memory.

**VHDL** Very high-speed integrated circuit Hardware Description Language.