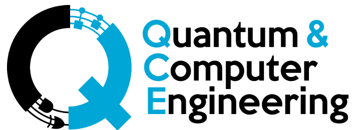
Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2018



MSc THESIS

Efficient inter-thread communication on the reconfigurable ρ-VEX

J. A. Dirks

CE-MS-2017-17

Abstract

This thesis documents the implementation of atomic instructions for the ρ-VEX (reconfigurable VEX). These instructions enable threads to communicate enabling efficient multithreading. Furthermore, we investigate the possibility to use inter-thread communication to improve performance of static ρ-VEX configurations without significantly increasing the area. Benchmark results show that the ρ-VEX can perform up to 1.33 times better because of the addition of atomic instructions.

Moreover, the combination of reconfigurability and inter-thread communication is investigated to determine the possible performance improvement resulting from this combination. A theoretical model is created which predicts that a runtime reconfigurable ρ-VEX is able to outperform any static ρ-VEX setup. Given ideal circumstances, the runtime reconfigurable ρ-VEX can be 20% to 100% faster than any static ρ-VEX.

In addition, this thesis documents the implementation of a bridge which intertwines the ρ-VEX with ARM's ZYNQ system, a single chip containing an ARM processor and a Xilinx field-programmable gate array (FPGA). This bridge gives the ρ-VEX access to a 512 MiB memory on the Basys PYNQ.

TU Delft
Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Efficient inter-thread communication on the reconfigurable ρ-VEX

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

EMBEDDED SYSTEMS

by

J. A. Dirks
born in Spijkenisse, The Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Efficient inter-thread communication on the reconfigurable ρ-VEX

by J. A. Dirks

Abstract

This thesis documents the implementation of atomic instructions for the ρ-VEX (reconfigurable VEX). These instructions enable threads to communicate enabling efficient multithreading. Furthermore, we investigate the possibility to use inter-thread communication to improve performance of static ρ-VEX configurations without significantly increasing the area. Benchmark results show that the ρ-VEX can perform up to 1.33 times better because of the addition of atomic instructions.

Moreover, the combination of reconfigurability and inter-thread communication is investigated to determine the possible performance improvement resulting from this combination. A theoretical model is created which predicts that a runtime reconfigurable ρ-VEX is able to outperform any static ρ-VEX setup. Given ideal circumstances, the runtime reconfigurable ρ-VEX can be 20% to 100% faster than any static ρ-VEX.

In addition, this thesis documents the implementation of a bridge which intertwines the ρ-VEX with ARM's ZYNQ system, a single chip containing an ARM processor and a Xilinx field-programmable gate array (FPGA). This bridge gives the ρ-VEX access to a 512 MiB memory on the Basys PYNQ.

Laboratory : Computer Engineering
Codenumber : CE-MS-2017-17

Committee Members :

Advisor:	Dr. Ir. J. S. S. M. Wong, CE, TU Delft
Chairperson:	Dr. Ir. J. S. S. M. Wong, CE, TU Delft
Member:	Dr. Ir. Z Al-Ars, CE, TU Delft
Member:	Dr. M. T. J. Spaan, Alg, TU Delft

Dedicated to my family and friends

Contents

List of Figures viii

List of Tables ix

List of Acronyms xi

1 Introduction 1
- 1.1 Context . . . 2
- 1.2 The inflexible setup . . . 2
- 1.3 The absence of inter-thread communication . . . 2
- 1.4 Problem statement and methodology . . . 3
- 1.5 Overview . . . 5

2 Background and literature survey 7
- 2.1 Concurrent Programming Control . . . 8
  - 2.1.1 Critical sections . . . 8
  - 2.1.2 Wait-free synchronization . . . 9
  - 2.1.3 Exploiting memory properties . . . 9
  - 2.1.4 Instruction set implementations . . . 12
- 2.2 The ρ-VEX . . . 14
  - 2.2.1 Context of the ρ-VEX . . . 15
  - 2.2.2 The current memory hierarchy . . . 16
  - 2.2.3 Current status of ρ-VEX w.r.t inter-thread communication . . . 16
- 2.3 The hardware: Digilents' PYNQ-Z1 . . . 18
- 2.4 Notes on performance . . . 20
- 2.5 Conclusion . . . 21

3 Interconnect implementation 23
- 3.1 Prerequisites . . . 24
- 3.2 The ρ-VEX bus . . . 25
- 3.3 The AXI4 bus . . . 26
- 3.4 Linux driver and related software . . . 28
- 3.5 Implementing the interconnect . . . 29
- 3.6 Area results . . . 32
- 3.7 Performance Results . . . 36
- 3.8 Conclusions and future improvements . . . 42

4 Enabling inter-thread communication 43
- 4.1 Choosing a suitable solution . . . 44
- 4.2 Implementation into the toolchain . . . 45
- 4.3 Implementation into hardware . . . 45
  - 4.3.1 A note about performance . . . 47

4.4 Software primitives . . . 47
4.5 Conclusion . . . 48

5 Benchmarking **49**
5.1 Prerequisites . . . 50
5.1.1 Testing dynamic solutions . . . 50
5.2 Test setup . . . 51
5.3 Quicksort . . . 52
5.3.1 A naive implementation . . . 52
5.3.2 The optimized implementation . . . 57
5.4 Matrix Multiplication . . . 61
5.5 Conclusion . . . 65

6 Model **67**
6.1 Definitions and assumptions . . . 68
6.2 Four issue, two context . . . 69
6.3 Eight issue, four context . . . 70
6.4 Conclusion . . . 72

7 Conclusions and future work **73**
7.1 Summary . . . 74
7.2 Main contributions . . . 75
7.3 Future work and future improvements . . . 76

Bibliography **79**

List of Figures

2.1 Atomic Register 10
2.2 The PYNQ-Z1 board 18

3.1 ρ-VEX Bus timing diagram 26
3.2 AXI4 handshake schematic 27
3.3 Schematic image of ρ-VEX AXI4 interconnect 30
3.4 Byte count divided by LUT count for different L2 cache configurations . 34
3.5 Byte count divided by flip-flop count for different L2 cache configurations 34
3.6 Byte count divided by BRAM count for different L2 cache configurations 35
3.7 Cache configuration results for matrix multiplication on 16x16 matrices 39
3.8 Cache configuration results for matrix multiplication on 32x32 matrices 39
3.9 Cache configuration results for matrix multiplication on 64x64 matrices 39
3.10 Cache configuration results for quicksort 39
3.11 Cache configuration results for g3fax. 39
3.12 Cache configuration results for a subset of PowerStone, single-threaded . 39
3.13 Cache configuration results for a subset of PowerStone, multithreaded . 40

4.1 Schematic image of ρ-VEX memory hierarchy 46

5.1 A schematic overview of the test setup. 51
5.2 The runtime of the different setups for naive quicksort, optimization *-Os*. 54
5.3 The runtime of the different setups for naive quicksort, optimization *-O3*. 54
5.4 The active time of the different setups for naive quicksort, optimization *-O3*. 55
5.5 The stall time of the different setups for naive quicksort, optimization *-O3*. 55
5.6 The relative performance of the different setups for naive quicksort, optimization *-O3*. 56
5.7 The runtime of the different setups for optimized quicksort. 58
5.8 The active time of the different setups for optimized quicksort. 59
5.9 The stall time of the different setups for optimized quicksort. 59
5.10 The relative performance of the different setups for optimized quicksort. 60
5.11 The runtime of the different setups for matrix multiplication, optimization *-Os*. 62
5.12 The runtime of the different setups for matrix multiplication, optimization *-O3*. 63
5.13 The active time of the different setups for matrix multiplication, optimization *-O3*. 63
5.14 The stall time of the different setups for matrix multiplication, optimization *-O3*. 64
5.15 The relative performance of the different setups for matrix multiplication, optimization *-O3*. 64

List of Tables

2.1 ZYNQ XC7Z020 properties 19

3.1 Time cost of interconnect operations, predicted by the design 33
3.2 Interconnect results common to all configurations 37
3.3 Interconnect results specific for all configurations 37

5.1 Syllables per bundle for naive -*Os* quicksort 53
5.2 Syllables per bundle for naive -*O3* quicksort 53
5.3 Syllables per bundle for the optimized quicksort algorithm 58
5.4 Syllables per bundle for -*Os* matrix multiplication 62
5.5 Syllables per bundle for -*O3* matrix multiplication 62

List of Acronyms

ALU Arithmetic Logic Unit, part of the CPU that handles arithmetic and logic

AMAT Average Memory Access Time

ASIC Application-Specific Integrated Circuit

BRAM Block RAM, on-chip FPGA RAM

CAS Compare-and-set, see 2.1.4

CMA Contiguous Memory Allocator, see 3.4

DDR Short for DDR SDRAM which is short for Double Data Rate Synchronous Dynamic Random Access Memory

ILP Instruction Level Parallelism, see 2.4

ISA Instruction Set Architecture, the language of a processor

ll/sc Load-linked store-conditional, see 2.1.4

LUT Look-Up Table

PRNG Pseudo-random number generator

SMT Simultanious Multithreading

TLP Thread Level Parallelism, see 2.4

VLIW Very Long Instruction Word, see 2.4

1 Introduction

This thesis describes the implementation of efficient inter-thread communication on the ρ-VEX in order to improve performance. Moreover, this thesis describes modifications which adapt the ρ-VEX to the Basys PYNQ-Z1, a relatively cheap development board. This chapter first explains some context and then explains the problems, the related research questions, and the goals of this project.

1.1 Context

As all programs differ from on another, one could argue that there is an ideal processor for each program. Creating a processor per program is infeasible, but creating a processor which can adapt to increase the performance of the program it executes is feasible, as demonstrated by the ρ-VEX.
The ρ-VEX is a polymorphic and dynamically reconfigurable Very Long Instruction Word (VLIW) processor, developed by the Computer Engineering laboratory of the TU Delft. The ρ-VEX can function as a single core which can execute up to eight instructions per cycle or can be split into multiple different cores, each of which can execute an independent instruction stream and a smaller amount of instructions per cycle. In other words, the ρ-VEX can exploit both task level parallelism and instruction level parallelism (ILP). The configuration can be changed during runtime so that it can even adapt while a program is running. The current implementation is a softcore processor running on a FPGA and is used in multiple courses taught by the TU Delft.

1.2 The inflexible setup

The current implementation was created by J. van Straten in 2015 and it primarily targets the ML605 and the VC707 developments boards. Busses and interconnects were created to connect the ρ-VEX to all features of these boards, most notably to the on-board DDR3 memory. Both of these boards are expensive[1] and therefore the Computer Engineering laboratory only has few of them. This limits the usability of the ρ-VEX in education and research since sharing a board between projects is a nuisance at best.
To tackle these problems the ρ-VEX was ported to the much cheaper PYNQ board[2] of which the computer engineering laboratory now has dozens. This port was never completed so that the usefulness of the ρ-VEX on the PYNQ board has remained limited. The primary problem is that there is no access to the on-board 512 MiB memory. Instead, the main memory should always be put on the FPGA chip, which limits the size to at most 512 KiB. This is not enough for many purposes, one of which is performance testing.

1.3 The absence of inter-thread communication

Currently, the ρ-VEX has limited support for multithreading and thus cannot optimally be used to exploit thread level parallelism[3] (TLP)[1]. As stated earlier, the ρ-VEX does have support for task level parallelism, the main difference between task level parallelism and thread level parallelism is the relation between the units running in parallel. Task level parallelism has independent tasks, each has their own input data, their own output data and possibly their own code. On a bare metal platform such as the ρ-VEX

[1] The VC707 is 3500 dollar (`https://www.xilinx.com/products/boards-and-kits/ek-v7-vc707-g.html`) and the ML605 is 1800 dollar (`https://www.digikey.nl/product-detail/en/xilinx-inc/EK-V6-ML605-G/122-1757-ND/`)

[2] 200 dollars, see `https://store.digilentinc.com/pynq-z1-python-productivity-for-zynq/`

[3] Notice the difference between thread level parallelism and task level parallelism

it is possible to make two tasks run in such a way that there is no communication required between the tasks. Thread level parallelism is on the level of a single application which consists of multiple related subtasks. These subtasks usually share the location of the input data or the location where the output data should be written. One of the characteristics of thread level parallelism is that there always must be some form of communication between the threads, at the very least to verify task completion.
The absence of thread level parallelism (TLP) has negative consequences for the ρ-VEX. TLP is a fundamental part of modern computing, many modern operating systems and programs depend on the existence of TLP[2]. None of these programs can currently be ported to the ρ-VEX. Moreover, it is possible to increase performance by introducing TLP. Currently, a program must have a large degree of instruction level parallelism (ILP) to be able to exploit the full potential of the ρ-VEX. Introducing inter-thread communication will allow some programs to spread its work over multiple threads and these threads together might be able to use the full potential of the ρ-VEX so that the program runs shorter.
The reconfigurability of the ρ-VEX together with inter-thread communication might be able to improve performance beyond inter-thread communication alone. This idea has been floating around the ρ-VEX research group for a while, but there is currently no information on what a program that will perform better because of reconfiguration would look like. Likewise, there is no information about upper bounds on the possible performance gain because of reconfiguration for a single program.

1.4 Problem statement and methodology

This thesis will address the lack of inter-thread communication. Inter-thread communication needs to be added and then quantified.

> How to add possibilities for inter-thread communication to the ρ-VEX?

> What is the impact of inter-thread communication on performance on the ρ-VEX?

Furthermore, the combination of runtime reconfigurable and inter-thread communication is addressed. The goal is to supply upper bounds and determine what the program the optimally exploits the ρ-VEX's properties looks like.

> What does the optimally performing program for the ρ-VEX look like and how much can runtime reconfigurability improve performance?

Finally, this thesis will address the primary limitation on the usefulness of the ρ-VEX on the PYNQ board: the lack of memory. Specifically, changes will be made to allow the ρ-VEX access to the DDR3 memory on the PYNQ board.

> How can we create a way for the ρ-VEX to get access to the on-board RAM of the PYNQ board?

The goals are the following:

- To create a ρ-VEX platform on the PYNQ which can use the on-board RAM and has inter-thread communication implemented.
- To use this platform to run some benchmarks which determine the performance improvement due to inter-thread communication alone and determine whether or not the runtime configurable property of the ρ-VEX can be used to improve performance further.
- To model some properties of the ρ-VEX to determine the optimal usage of the ρ-VEX in terms of performance.

Do note that this thesis will ignore power consumption and area, and focus on performance. The reason is that there are only very minor changes to the ρ-VEX itself so that power consumption and area remain mostly unchanged. Furthermore, the goal is to explore how performant the ρ-VEX really is and reasoning about power consumption and area lies beyond this goal.

The following method will be used to achieve these goals and answer the research questions.

1. Investigate the ρ-VEX and the current state of its implementation on the PYNQ board.
2. Investigate how the FPGA fabric can reach the on-board memory.
3. Modify the software running on the PYNQ board to allow the ρ-VEX to use the on-board memory.
4. Build an interconnect which connects the ρ-VEX PYNQ port to the on-board memory.
5. Test the performance and area of the interconnect.
6. Research what should change about the ρ-VEX to allow inter-thread communication.
7. Implement the features required to allow inter-thread communication.
8. Implement some functions which use the new features to enable inter-thread communication.
9. Build a test platform using the interconnect and the inter-thread communication features.
10. Run benchmarks on this test platform.
11. Model the ρ-VEX to determine optimal programs for dynamic configurations.

1.5 Overview

The remainder of this thesis is structured as follows. Chapter 2 provides background on inter-thread communication, the current state of the ρ-VEX, the PYNQ board and the different ways of parallelizing execution. Chapter 3 discusses the design, implementation and performance of the interconnect which gives the ρ-VEX access to the on-board memory of the PYNQ board. Chapter 4 discusses the design and implementation of inter-thread communication on the ρ-VEX, both the required hardware changes and the implementation of some functions. Chapter 5 explains the test platform and shows benchmark results for static and dynamic ρ-VEX platforms. Finally, Chapter 6 models some aspects of the ρ-VEX to provide insight in how to maximize performance on the ρ-VEX platform. Chapter 7 summarizes the work and lists possible future projects and future improvements.

2 Background and literature survey

This chapter explores the background, related work and concepts required to understand the rest of the thesis. Section 2.1 explores inter-thread communication. Section 2.2 explains the current state of the ρ-VEX, Section 2.3 gives information on the hardware platform for this project, the Basys PYNQ-Z1 and Section 2.4 provides an explanation of measuring performance, thread level parallelism and instruction level parallelism.
Inter-thread communication is explored before the actual ρ-VEX, as it is required to understand the proof presented in Section 2.2.

2.1 Concurrent Programming Control

Multiple threads can work together to perform a certain task, but the threads will need to communicate if the threads are to work together. Usually this communication manifests as a variable in a shared location and having all threads interact with this variable. For example, assume that two threads report their progress by incrementing a shared variable two times, making the desired end value 4. Without mechanisms to control access to this shared variable, the value of the shared variable will be between 2 and 4 (inclusive) when both threads have finished. The value 2 two will occur if both threads read and update the shared variable at exactly the same times, deviation leads to the values 3 and 4. This situation is called a race condition and is not desirable. Mechanisms must be put in place to ensure that the final value is always 4.

Two mechanisms to ensure that race conditions do not occur will be discussed here: critical sections and wait-free synchronization. Any implementation of either critical sections or wait-freedom consists of a property of the hardware combined with some software implementation[3]. This section will explain what critical sections and wait-free synchronization are. Furthermore, some usable hardware properties will be discussed.

2.1.1 Critical sections

The first publication on the subject of race conditions is by Dr. E. W. Dijkstra in 1965. He was considering a group of N computers, all working on some form of cyclic process in which there is a critical section. The critical section is a piece of code which is the same for all N processes in which at most one of these N processes can be active at the same time. In the example above, the critical section would have been the piece of code which increments the shared variable. If only one thread is incrementing at the same time, the end value is guaranteed to be 4.
Critical sections have to be protected, there must be a method to actually ensure that there is at most one thread in the critical section at any time. There has been some debate on what this method should look like. Dijkstra stated that the method to protect the critical section should at least have the following properties[4]:

1. There can not be a static priority between the computers, it has to be a symmetrical solution.
2. The method cannot assume anything about the relative speed of the machines.
3. If one machine stops outside of the critical section it cannot be that another machine becomes blocked in its flow.
4. If one or more machines want to get into the critical section then within finite time one machine is selected and allowed to continue into the critical section.

Property four is equivalent to "The system as a whole always makes progress", but individual threads might not make progress. D. Knuth considered this an issue and thus reworded property four to

4. If an individual computer wants access to the critical section, it will get this access within finite time, assuming the waiting is not interrupted by the computer itself or an error.

Or in other words: with Knuth's definition any thread always makes progress[5].
Property three of Dijkstra's list of properties has also been up for debate and was later modified by L. Lamport, who stated that no thread was ever required to enter the critical section and that there is no upper or lower limit on how many times a thread has to enter the critical section[3].

2.1.2 Wait-free synchronization

Critical sections are not ideal for some use cases, for example fault-tolerance. What if a process stops while it is in the critical section? A critical section implementation ignores this possibility and the entire system starts waiting forever if this situation occurs. Critical sections are also inherently single-threaded. Therefore, if a very slow process is in the critical section, faster processes will have to wait until the slow process has completed the critical section. An alternative to critical sections is proposed by M. Herlihy ([6]) and called Wait-Free Synchronization. The author defines it as follows: "A wait-free implementation of a concurrent data object is one that guarantees that any process can complete any operation in a finite number of steps, regardless of the execution speed of the other processes", or in other words: every individual thread always makes progress, unless it stops either by choice or because of component failure. A concurrent object is further defined as ".. a data structure shared by concurrent processes". This means that the shared variable from the example earlier in the text could be implemented as a concurrent object, which would then solve the race condition.

An important definition in wait-free synchronization is the *Consensus number*. If a concurrent object X has consensus number n then it allows n processes to communicate using this concurrent object in a wait-free manner. To prove that a concurrent object has a consensus number of at least n, one needs to specify a consensus protocol. This is a system in which the n processes can decide on a value using the shared object (or multiple of them). A consensus protocol has the following requirements[6]:

1. Consistency: distinct processes never decide on distinct values.
2. Wait-freeness: each process decides after a finite number of steps.
3. Validity: the common decision value is the value proposed by one of the processes.

A concurrent object can also be used to implement lock-free synchronization, which is different from wait-free synchronization in that it enforces that the system as a whole makes progress[7].

2.1.3 Exploiting memory properties

The first correct implementation of critical sections, proposed by Dijkstra, assumed a certain memory model. The model he assumed is called Atomic Memory and is defined

as follows:
First, we define the register, which is for the scope of this section a place where a single "piece" of data can be saved or loaded. A register is called atomic if it satisfies an additional set of properties.

1. Every read or write action appears as a single point in time. This point in time lies after the point where the issuee issues the action and before the issuee gets notified the action is completed.

2. Every read or write actions appears to happen at an unique time, there can never be two actions at exactly the same time.

3. Every read returns the value that was written by the closest preceding write action.

If we define a section of (shared) memory as a set of atomic registers, we call the memory atomic memory[8]. Figure 2.1 shows what atomic memory would look

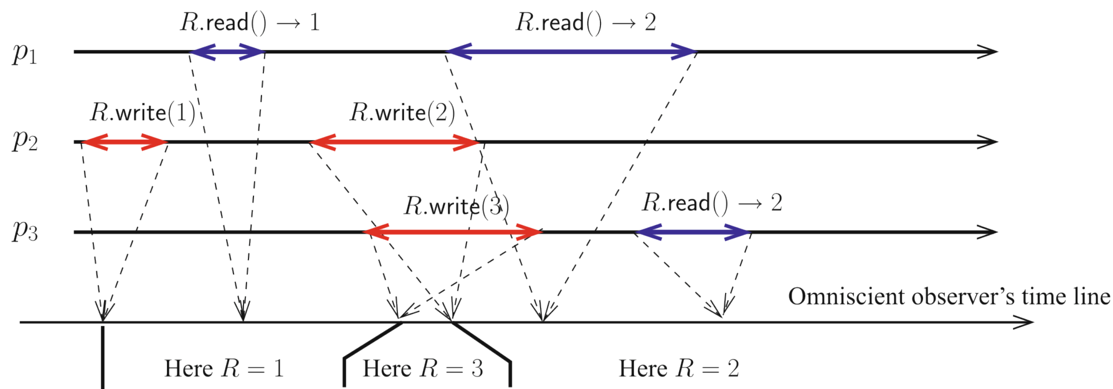


Figure 2.1: An example of an atomic register R being used by three threads at the same time.[1]

like from every observer. Note that this model is not applicable to modern memory hierarchies: property three, for example, will often be violated because changes (writes) need time to propagate trough the caches into the main memory. A read issued during this time can thus return an older value[9].

Dijkstra's original solution assumed that the machine's memory hierarchy adheres to the Atomic Memory model. On top of that, some other assumptions are made about the machines involved:

1. A machine does not stop inside the critical section.

2. The number of threads is known beforehand.

3. No thread starts in the critical section.

[1]Source: [8]

Dijkstras' solution is as follows: given N computers, two boolean arrays b and c, both of size N and an integer k. Every element of b and c is initially false, and k is initially zero. Now, the algorithm for computer i, where $0 \leq i < N$ is[4]:

Algorithm 1: Dijkstras' original solution to the Concurrent Programming Control problem

```
1  retry ← false
2  b[i] ← false
3  repeat
4  |  while k ≠ i do
5  |  |  c[i] ← true
6  |  |  if b[k] then  k ← i
7  |  end
8  |  c[i] ← false
9  |  for j ← 0 until N − 1 do
10 |  |  if j ≠ i ∧ ¬c[j] then  retry ← true
11 |  end
12 until ¬ retry
13 criticalSection ()
14 b[i] ← true
15 c[i] ← true
```

As said before, Dijkstra's algorithm only ensures that the system as a whole makes progress and Knuth wanted a solution where every individual thread makes progress. Knuth presented his solution in 1966 and added a notion of performance: The amount of turns a thread would have to wait in the worst case before entering the critical section. His own solution has a worst-case performance of $2^{N-1} - 1$ turns, where N is the amount of threads active in the algorithm[5]. This bound improved to $\frac{1}{2}N(N-1)$ in an algorithm proposed by Dr. N. G. de Bruijn [10] and finally to $N - 1$ by M. A. Eisenberg and M. R. McGuire [11].

Leslie Lamport eventually comes up with a solution which requires a less strict memory model. Compared to Atomic Memory, it is now allowed for reads and writes to happen at the exact same moment. Moreover, his solution allows for components to break, as long as it is detectable that the components are broken (meaning in practice that Lamport's solution is not resilient to every component failure while wait-free synchronization is required to be resilient)[12][3]. Although interesting, these changes do not loosen the model enough to become viable for modern computer architectures[9].

The consensus number for any concurrent object that tries to exploit memory properties like atomic memory is 1, which means that those concurrent objects are useless for wait-free synchronization[6].

2.1.4 Instruction set implementations

As discussed before, known memory models which can prevent race conditions do not perform well. As an alternative, an instruction can be implemented in the instruction set architecture (ISA) to signal that special properties are required, properties which can be exploited to implement critical sections or wait-free synchronization. These instructions are referred to as atomic instructions, where atomic is a synonym to uninterruptible. Early atomic instructions moved multiple different instructions into one indivisible instruction, called an atomic sequence. Later, the idea changed to having a sequence of instructions ending with a special store instructions. The store instruction commits if there were no relevant changes to the memory and rejects if there were relevant changes to the memory. One could argue that this setup simulates an atomic sequence. Note that every atomic instruction has a description of how it is expected to work, but the exact implementation still depends on the architecture of the processor itself, the memory bus, and the memory hierachy.

The first example is the test-and-set (TAS) instruction, this is implemented in IBMs System/360 model 67 from 1965[13]. A test-and-set instruction operates on a single register which can hold 2 values, usually referred to as 1 and 0. One can think of the instruction as a function which returns the value of the register and flips the register if the current value is 0.To clarify, the value returned is the value read at the start of the function[14].

Algorithm 2: Pseudocode of test-and-set usage

1 **while** (`test-and-set`(r) $= 1$)
2 `criticalSection` ()
3 $r \leftarrow 0$

The algorithm above adheres to Dijkstra's requirements for the concurrent programming problem, showing that the introduction of this instruction does make solving the problem a lot easier than implementing a software solution on top of atomic memory. The TAS instruction is an atomic sequence, it executes a load, a compare, possibly an update and finally returns with the guarantee that no other instructions were executed on the machine during TAS.
TAS has a consensus number of two[6].

The successor to the System/360, the IBM System/370, introduced the Compare-and-Swap (CAS) and Compare-Double-and-Swap (CDAS) instructions. The Compare-and-Swap instruction takes 3 arguments: a reference to a memory location, an expected value at that memory location and a new value. If the data at the memory location is equal to the expected data, the new value is written to the memory location. The return value is a boolean-like which returns true if the memory location was updated and false if it was not. The compare-then-update sequence should be seen as a single uninterruptible operation, and no two of them can be active on the same location at the same time[15]. Implementing a critical section based on Dijkstras' requirements is as follows:

Algorithm 3: Pseudocode of compare-and-swap usage

```
1 while ( compare-and-swap(r, 0, 1) = false )
2 criticalSection ()
3 r ← 0
```

Here, the Compare-and-Swap instruction behaves almost identical to the test-and-set instruction. Compare-and-swap is a lot more versatile. The instruction allows for, for example, incrementing a value from multiple different hardware threads without the need for a critical section. This is shown in the next algorithm. [2]

Algorithm 4: An atomic increment function based on CAS.

```
1 Function atomicIncrement (memory location r)
2     b ← 0
3     while b = 0 do
4         value ← *r
5         b ← compare-and-swap(r, value, value + 1)
6     end
```

Compare-and-Swap does suffer from the so-called ABA problem, which can happen because the CAS instruction only checks the memory location right before writing. This means that it is possible that the state of the program changes in such a way that it would be the correct answer to reject the CAS instruction, but since the hardware cannot detect the state change, the CAS instruction is allowed anyway. The ABA problem does not have a one-size-fits-all solution and can lead to race conditions happening anyway[16][15]. The IBM System/370 has the Compare-Double-and-swap instruction, in which the words are twice as big compared to the Compare-and-Swap instruction, reducing the chance of the ABA problem happening in overflow situations. If the ABA problem is correctly avoided, the consensus number of CAS is infinite.

The youngest one in the family of atomic instructions was introduced in 1987: load-linked/store-conditional (ll/sc). This is a set of two instructions, the first one is a load (which never fails) and the second one is a store which might fail. The flow here is to first load data using load-linked, then operate on the data and then write back using store-conditional. Store-conditional fails if there was any writing activity on that address in between the load-linked and store-conditional. The hardware implementation of this set of instructions needs to be very thorough, since it needs to be able to detect which hardware thread did load the data, but it also needs to respond to certain interrupts, since interrupt handlers can be ran on the same hardware threads as other processes. Load-linked/store-conditional is able to avoid the ABA problem and also has an infinite consensus number[17][18].

[2]This function is based on the Wikipedia example, see `https://en.wikipedia.org/wiki/Compare-and-swap`, retrieved on 2017-11-22

2.2 The ρ-VEX

The ρ-VEX is a runtime-configurable polymorphic very long instruction word (VLIW) processor. VLIW means that the processor attempts to process multiple instructions per cycle. There are two ways to achieve this: Superscalar and VLIW. Both VLIW and superscalar have the notion of instruction slots: if there are n instruction slots, then n instructions can start every cycle.
Superscalar implementations take instruction streams which can also be executed by a processor which does not attempt to execute multiple instructions per cycle (scalar implementations) and attempts to parallelize this stream. This is usually implemented with a window: the superscalar processor looks at the next n instructions and attempts to schedule them in such a way that it takes only a few cycles to process all of these. So from the perspective of the compiler there is only one instruction slot available per cycle, but the processor itself attempts to fill all of its instruction slots. VLIW implementations have its instruction slots exposed to the compiler and the machine code for the compiler consists of larger bundles, each of which is an ordered set of instructions (called syllables in this context). Each syllable in a word maps to an instruction slot[14]. Superscalar has equal or better performance compared to VLIW[19], but VLIW consumes less power[20].

Polymorphic literally means "many shapes" and it describes one of the major properties of the ρ-VEX: its design allows for much flexibility. The current implementation allows the developer to specify how many lanes and contexts he wants. A lane is closely related to an instruction slot. Lanes can have or lack certain capabilities. Each lane supports the base group, which is a set of basic operations such as addition and binary logic. Furthermore each lane can support multiplication, memory interaction and floating-point instructions. The base, multiplication and memory interaction groups together form the basic functionality of the ρ-VEX, the floating point instructions are an addition to the instruction set. A lanegroup is a set of lanes (at least one) which together form the basic functionalities of the ρ-VEX, so that each valid ρ-VEX design contains a set of lanegroups.
Lanegroups can be attached to contexts. A context can be compared to a core in a classic homogenous multiprocessor. It is a unit which runs an independent instruction stream and has its own instruction slots, the amount of which depends on the amount of attached lanes. The ρ-VEX can contain at most eight lanes and four contexts, there can be 2, 4 or 8 lanes connected to a context.

It is possible to enable or disable the ability to reconfigure during runtime. If this option is enabled the lanegroups can be moved somewhat freely[3] between contexts and it is possible to disable contexts (by not assigning lanes to them) or lanegroups (by not assignign them to contexts)[21]. Reconfiguration takes at least 9 cycles[22].

A conventional VLIW compiler knows how many instruction slots there are and uses this information to devise a schedule. The amount of slots vary because of

[3] It is possible to make every possible configuration, but not to attach any lanegroup to any context in any order. The exact rules can be found in [1]

reconfiguration, so that there is less information available. It is possible to always compile for the smallest issue width (issue width of n means that there are n instruction slots), but then most of the lanes never get any work. Running machine code for a bigger issue width than currently available is possible, but might give unexpected results.
The solution is introduced by A. Brandon and S. Wong and called generic binaries. These binaries are created to run correctly on any issue width, but there can be situations in which the schedule is not optimal[23]. Currently the ρ-VEX toolchain has specific compilers for 2 issue, 4 issue and 8 issue and a generic compiler suitable for any configuration.

The ρ-VEX source code is publicly available and can work on many different FPGA chips. An ASIC implementation of the ρ-VEX was also created.

2.2.1 Context of the ρ-VEX

Every task has certain characteristics which are unique, and every task runs in some context. Examples of contexts are realtime, where predictability is exceptionally important or high-performance where runtime is the only thing that matters. The combination of task and context creates a unique set of demands so that optimality can only be achieved by creating a specialized platform. In some cases it is feasible to actually create such a specialized platform, usually in the form of an ASIC. Cryptomining is an example of this: there is one algorithm which needs to run on a platform with an optimal performance per watt ratio and an ASIC is created to satisfy this need[24]. But the creation of an ASIC might be too expensive or might not suit all needs so that companies usually resort to selecting a general-purpose CPU with the features they need so that their requirements are met on a non-optimal platform.

Embedded processors require no external cooling and consume little power. This is an example of how processor manufacturers started to create hardware solutions for specific contexts. The heterogeneous multiprocessor is another example of the push towards more specialized processors. A quite well-known example is the IBM Cell, which is optimized for multimedia purposes. It consists of one big core and many smaller cores[25], which is where the name heterogeneous comes from: the cores are non-homogenous. If a program is able to exploit these smaller cores, then the Cell will outperform any homogenous processor with comparable power consumption of its time[26]. There is a specific combination of context and tasks for which the Cell is the optimal choice, and also a lot of contexts and tasks for which the Cell is not.

The ρ-VEX is an embedded processor which is reconfigurable. This is attempting to be one step beyond heterogeneous: the processor can actively optimize for the tasks it is running. So in theory, there is a more broad range of tasks or task groups for which the ρ-VEX is more optimal than homogeneous or heterogeneous processors which are comparable in price, size and power consumption. In theory, because the ρ-VEX is not in a state in which it can be properly compared.

2.2.2 The current memory hierarchy

This thesis deals with the memory hierarchy of the ρ-VEX on multiple occasions, when creating the interconnect and in Chapter 5 when the results of the benchmarks are interpreted. Chapter 3 describes the implementation of the interconnect and explains details like the current implementation of the ρ-VEX bus. This subsection will present in broad strokes the necessary knowledge to understand both Chapter 3 and Chapter 5. The ρ-VEX core has a working level 1 cache at the start of this thesis, but no other caches. There are a couple of different interconnects available, such as ρ-VEX bus to AHB[4] and ρ-VEX bus to DDR3.

The current ρ-VEX implementation has an optional level 1 cache. This level 1 cache is split into blocks, one block per lanegroup. Moreover, each cache block is tightly coupled to a lanegroup so that the amount of cache a context has is determined by the amount of lanegroups connected to it. When multiple lanes are connected to the same context after a reconfiguration, the related caches are merged and duplicate entries in the cache are thrown out. This means that the cache is not cold after reconfiguration. The level 1 cache is split into a data cache and an instruction cache. The amount of lines of each can be set independently and is the same for every block.[1].

The level 1 cache and the ρ-VEX bus can have a large and negative effect on execution time. The data cache accepts at most one request (load or store instruction) per cycle per context. This limits schedulability since data processing usually means load, process and store. So the compiler has a hard time filling all eight instruction slots, since loading eight pieces of data and then processing them all gives no performance advantages. The instruction cache is capable of returning one word per connected lane per cycle, if the words are cached. If the words are not cached the data needs to be loaded word by word, since the ρ-VEX bus only allows for four byte transactions. The instruction cache utilizes the same bus as the data cache, so that data reads from context one might have to wait for instruction reads from context zero. This is in stark contrast to other modern bus implementations, which allow out-of-order transactions, variable size transactions and bursts, among others. All of these have a very positive impact on bus performance[27]. To conclude, the current implemenation limits schedulability and causes a relatively large amount memory delay.

2.2.3 Current status of ρ-VEX w.r.t inter-thread communication

The ρ-VEX has no atomic instructions builtin, so the only way to prevent race conditions is by using the guarantees from the memory hierarchy. This already limits the consensus number of ρ-VEX to 1[6]. The current design allows for two types of memory hierarchy: cached and uncached.

2.2.3.1 The uncached design

If the design is not cached there are one or multiple buses directly connected to main memory. A bus handles the reads and writes one by one and forces cores to wait for each

[4] AHB is a depricated ARM bus, see also Chapter 3

others' writes or reads. This means that the atomic memory properties as described in section 2.1.3 hold. To expand on this:
Every read or write action appears as a single point in time. This point in time lies after the point where the issuee issues the action and before the issuee gets notified the action is completed: There is no caching or buffering before the main memory, so that the instruction only completes if the operation was actually completed.
Every read or write actions appears to happen at an unique time, there can never be two actions at exactly the same time: Holds because although two operations can be issued at the same time, the memory will handle them in some order so that they appear at different times.
Every read returns the value that was written by the closest proceeding write action: Holds because there is no cache or buffer.
So the uncached design can utilize any algorithm mentioned in Subsection 2.1.3.

2.2.3.2 The cached design

The cached design introduces a cache per core with a write-back buffer attached to every cache. Whenever a read happens it is read from the cache if possible. A write will be immediately committed to the cache belonging to that context and added to a write-back buffer embedded in the level 1 cache. The write-back buffers are handled in some order with the guarantee that every write added to any write-back buffer will eventually be committed to the memory. The main memory handles all requests in such a way that a read from an address will return the latest written value to that address known to the main memory. Only when a write leaves the write buffer all relevant cache lines from other caches are invalidated. It is clear that this does not behave like atomic memory as described in section 2.1.3, a read may not return the last written value if that value has not left the write buffer of another context yet. What follows is a proof that it is impossible to prevent race conditions on the ρ-VEX when this hierarchy is used.
Assume a design with two contexts with lanes assigned, 0 and 1. Both run the same function consisting of two parts, a critical section and a noncritical section. This is a sane assumption, since it is possible to transform any situation in which a critical section is involved into such function. Assume neither context ever fails and that the critical section is finite, so that no process will be in there for infinite time. As per [5] we need a solution so that at most one context can be in the critical section at the same time and if a context wants to enter the critical section it will within finite time. The contexts do not have, or attempt, to enter the critical section, as per [3].
The first thing to note is that a read gives no valuable information, since it cannot be guaranteed that the value of the variable is the same for every context, because there might be a write stuck in some write-back buffer. Consequently, the only way to get information is to wait for a change. But a change has to be set in motion by the other context. This forces every context to poll the critical section regularly to pass information, but this violates the idea of a critical section as per Lamport [3].

Currently it is possible to implement mutual exclusion using the cached design by exploiting how it works exactly. The depth of the write-back buffer is finite and if the write-back buffer is full, the related context locks until there is space again. It is

possible to fill up the write-back buffer by repeating the same write transaction multiple times, since the write-back buffer does not check for such situations. So the solution is to repeat every write as many times as there are spaces in the write-back buffer (the software cannot know when the context locks). Now, when the context unlocks, the write has been committed to main memory. This gives enough guarantees to implement any algorithm mentioned in Subsection 2.1.3. but now the software needs an unreasonable amount of information about its hardware platform at compile time in order to do mutual exclusion correctly.

2.3 The hardware: Digilents' PYNQ-Z1

To give some context to the implementation part of this thesis, this section will give details of the hardware used and its properties. The ρ-VEX can run on many different platforms, the PYNQ-Z1 being one of the more recent ones. At the heart of the board lies

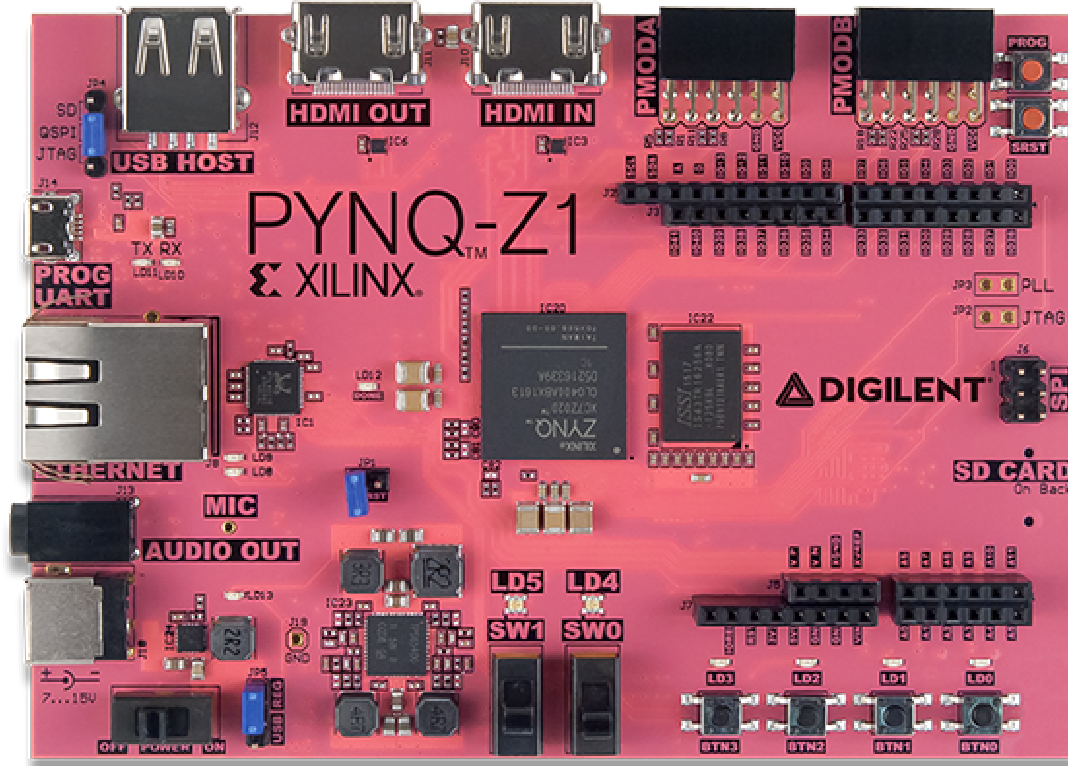


Figure 2.2: The PYNQ-Z1 board. Source: Digilent

the Xilinx ZYNQ XC7Z020, a System-on-a-Chip (SoC) consisting of an ARM processor and a FPGA, which can be interconnected to work together. The processor is a Cortex-A9 650 MHz dual-core processor. The FPGA is equivalent to the Artix-7, albeit with some additional features to interconnect with the processor. The board also contains

Object	**Count**	**Explanation**
Programmable logic cells	85000	A Cell containing multiple LUTS and Flip-Flops. "Relative measure of FPGA size"[30] Sometimes called a marketing term, not quite the same as a slice.
Look-Up Tables (LUTS)	53200	6 port LUT, meaning that is has 6 independent inputs and 1 output. It can represent any logic scheme with 6 (or less) inputs and 1 output. A third can also be used as distributed RAM, which is unclocked on-chip (rather space inefficient) RAM. They reside in groups of 8 on slices together with 16 Flip-Flops[31]
Flip-Flops	106400	D-latch Flip-Flops: used to store data Updates every rising edge. Half can be used as latch, which is an unclocked data store.
36 Kb Block RAMs	140	Dual-port RAMs which can load and/or store data at most once per cycle (every rising edge) Much more space-efficient than distributed RAM
DSP Slices	220	Digital Signal Processing Slice. Contains a lot of semi-flexible arithmetic units so that it can do more advanced math operations. Less flexible then a set of LUTs, but more space and power efficient for certain applications. Examples of capabilities include 3-input adder, wide counter and multiply-then-add[32].

Table 2.1: ZYNQ XC7Z020 properties[28]

512 MiB DDR3 RAM, UART, ethernet, SD-Card slot and many more features liked by nerds around the world[28].

The board is ruled by the processor and behaves like a microcontroller in many senses. The SD-Card slot can be used to run an operating system on the board and the board comes loaded with Ubuntu. For the project, Ubuntu 16.04 with Linux kernel 4.6 is used. The FPGA is connected trough a special Linux driver so that programming the FPGA is a simple task. When the board is powered, the OS automatically starts[29]. The FPGA chip is not very big, which is not a problem for this specific project, but it might be for other projects. Two 2-context 4-issue ρ-VEX's fit easily on the chip and one core with four contexts and eight issue fit barely on the FPGA. Per default the FPGA chip runs at 50 MHz, this default is not changed during the project[28].

Table 2.1 shows the exact contents of the FPGA.

2.4 Notes on performance

One of the major goals of this thesis is to measure, model and increase performance of the ρ-VEX. This section will explain some definitions and properties of performance, specifically tailored to ρ-VEX. This section serves as background knowledge to Chapter 5 and Chapter 6.

Performance is directly related to runtime, if a platform ϕ has a lower runtime R_ϕ than some platform ψ with runtime R_ψ, we say that ϕ outperforms ψ. Moreover we say that ϕ is $(\frac{R_\psi}{R_\phi} - 1) \cdot 100\%$ faster or $\frac{R_\psi}{R_\phi}$ times as fast.
Runtime is measured in seconds from the start until the program finishes. Due to jitter and sample time it might be hard to get the exact number by directly measuring the time between start and stop, so a shortcut is taken. Each context of the ρ-VEX keeps track on how many cycles it takes to execute, a number which is called execution cycles[21]. Because the ρ-VEX always runs on 50 MHz in this thesis, the amount of execution cycles can be translated one-to-one to a time in seconds.

The amount of execution cycles can be split into three different numbers:

- The amount of cycles stalled.
  During these cycles a context waits for the memory to respond.
- The amount of cycles active. A cycle is counted as an active cycle if the pipeline moved forward during that cycle. Usually, a new word is added to the pipeline during an active cycle.
- Pipeline flush cycles.
  There is a pipeline flush after every branch and right before a trap.

This thesis seeks to reduce the active cycles. The active time can be reduced by making sure that more instructions enter the pipeline every cycle. Each active cycle, one word is processed by a context. One word contains at most as many syllables as there are lanes attached to the context. The related metric is called syllables per word and is a metric for instruction level parallelism (ILP). Syllables per word is usually measured per context. If the average syllables per word for a program is lower than the amount of lanes (or instruction slots or issue width) it is sometimes a possibility to divide the pipelanes over multiple contexts and split the problem over all contexts. Such a setup is said to be exploiting thread level parallelism (TLP). If multiple contexts are actively working on the same problem, they have to communicate, the time spend on communication is called multithreading overhead.

Runtime reconfigurability gives the ρ-VEX more flexibility than any static setup. The idea is that a program goes trough phases and each phase allows for a certain amount of threads and a certain amount of syllables per word. By adapting, the ρ-VEX can perform optimally in all phases and thus outperform any static setup[33].

2.5 Conclusion

This chapter started with explaining how race conditions can occur due to multithreading. The chapter continues with explaining how race conditions can be prevented and the role hardware support plays. Section 2.2 explained what the ρ-VEX is and also showed that the ρ-VEX is very limited in its capabilities to avoid race conditions. Section 2.3 gave some information on the hardware used in this thesis and finally Section 2.4 gave an overview of measuring runtime, instruction level parallelism and thread level parallelism.

3 Interconnect implementation

This chapter will address the research question *How can we create a way for the ρ-VEX to get access to the on-board RAM of the PYNQ board?*
Section 3.1 will explain how the major components work together and thus what a design should look like. Furthermore, this section will explain the exact requirements of the design discussed in this chapter. Section 3.2 subsequently explains the exact inner workings of the memory bus of the ρ-VEX processor. Section 3.3 will then explain the inner workings of the AXI 4 bus, the bus internal to the ARM processor on the PYNQ chip.
Sections 3.4 and 3.5 will explain the design and implementation. Section 3.4 will explain the design and implementation of the Linux driver and related software which e.g. allows reservation of memory. Section 3.5 explain the implementation of an interconnect or a bridge which connects an AXI 4 bus to a ρ-VEX bus.
The chapter finalizes with some area measurements in Section 3.6 and performance measurements in Section 3.7.

3.1 Prerequisites

The goal of this chapter is to expand the RAM of the ρ-VEX PYNQ setup. The only memory available, aside from the memory on the FPGA, is a 512 MiB DDR3 RAM. This RAM is connected to the ZYNQ chip and the ZYNQ chip is highly configurable: the developer can configure how everything connects internally. There is an ARM processor on the ZYNQ and in the default configuration the DDR3 RAM is directly connected to this processor. The ARM processor exposes its internal trough an AXI4 slave so that designs on the FPGA can reach the DDR3 RAM trough the ARM processor via the AXI4 bus[34].
The ρ-VEX is a design on the FPGA, so that connecting it to the AXI4 bus can be a viable route. An alternative is to disconnect the DDR3 RAM from the processor and use a DDR3 to ρ-VEX bus interconnect. If this route is taken the operating system (Ubuntu) cannot function because there is no available to it. This is a problem because Ubuntu is used to run a debug server and to flash the FPGA. There are other routes available to achieve both tasks, but it would require a lot of work to set this up. An interconnect connection the ρ-VEX bus to the AXI4 bus is expected to be more trivial and thus this interconnect will be created.
If the DDR3 memory remains connected to the processor there has to be a setup to reserve memory for the ρ-VEX. The default behaviour of Ubuntu is to allocate all 512 MiB RAM to itself. When an interconnect has been created the ρ-VEX can operate freely on the RAM but as long as Ubuntu is unaware of this, Ubuntu might overwrite data placed by the ρ-VEX or vice versa. Some sort of negotiation setup has to be created to ensure that the ρ-VEX and Ubuntu do not bother each other. It is possible to exclude some memory from being allocated by Ubuntu, in such a setup Ubuntu is unable to access the memory. An alterative is to have Ubuntu allocate a chunk of memory specifically for usage with the ρ-VEX. The advantage of this implementation is that it is quite common[35]. Moreover, user space programs in Ubuntu can access this memory and write or inspect it.

The requirements for the setup involving the interconnect revolve around the idea that the ARM processor is merely a debug core to the ρ-VEX processor. The ARM core is used to initialize and check the interconnect, whilst the ρ-VEX does not even know of its existence. The following set of requirements reflect this.

1. The ρ-VEX can assume that the contents of this memory will never change without its interference.

2. The Linux instance has to assume that the contents of the memory can change at any given time.

3. The memory needs to be read and write accessible from user space on the Linux instance.

The user space requirement (3) is more of a safety issue: if root has to be acquired before every execution it is harder to detect programming errors, since root is allowed to do almost anything. Requirements one and two indicate the way the memory should be

handled from the ρ-VEX perspective: it can pretend like there is nothing else working on the same memory.

To create the interconnect, the ρ-VEX bus and the AXI4 bus need to be properly investigated. Sections 3.2 and 3.3 will detail both buses and highlight how they need to be connected.

3.2 The ρ-VEX bus

Since the ρ-VEX bus is not really documented outside of its source files, this section references source code in the footnote. The files named are relative to */lib/rvex* in the *rvex-rewrite* repository.

The ρ-VEX bus is the primary data mover for all components related to the ρ-VEX. It connects cache to main memory and UART unit to the processor. The bus is a pretty simple master/slave setup and consists of two channels, the first one is driven by the master and read by the slave, the second one is driven by the slave and read by the master. The master channel has the following signals [1]:

- Address, default 32 bit wide.
- WriteData, default 32 bit wide.
- WriteMask, byte masked so 4 bit wide by default.
- ReadEnable, signals that the channel currently contains a valid read request.
- WriteEnable, signals that the channel currently contains a valid write request.
- Flags, a set of flags whitch slightly modify the read or write request.

The slave channel has another set of signals[2]:

- ReadData, size needs to be equal to WriteData, so 32 bit default.
- Fault, a one bit flag to indicate that an error occurred.
- Busy, a one bit flag to indicate that the slave is handling the request.
- Ack, a one bit flag to indicate that the request has been handled.

It is possible to create undefined behaviour on the ρ-VEX bus, to prevent this there are the following rules:

- ReadData and writeData can never be high at the same time.
- Ack and busy can never be high at the same time.

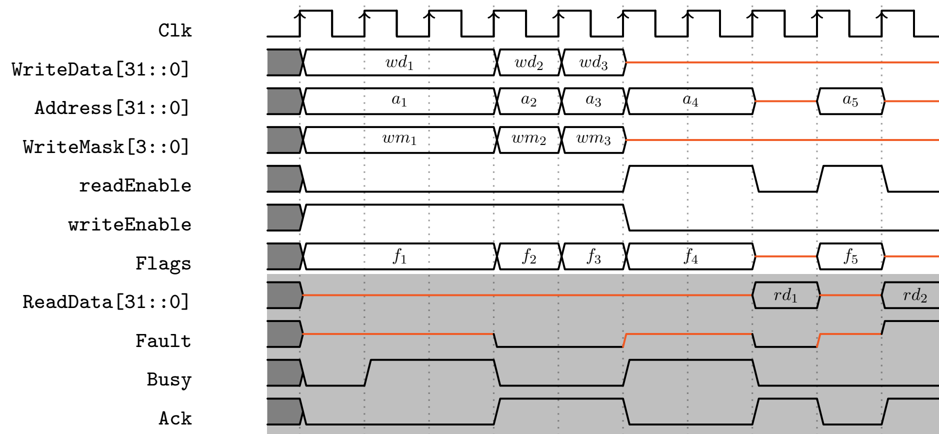


Figure 3.1: A schematic timing diagram of the ρ-VEX bus

Master nor slave needs to be capable of handling exceptions to the rules above, and a violation usually leads to a crash. Figure 3.1 shows some possible, correct interactions on the ρ-VEX bus.

Multiple masters can be connected to multiple slaves using arbiters and demuxers. This arbiter will process requests from different masters in a round-robin fashion[3]. Multiple slaves can be connected using a demuxer. The demuxer utilizes the address to forward the request to the correct slave[4]. If communication is going on it is always between one master and one slave, the bus cannot handle any other request during this time. In other words: the bus does not support multiple outstanding requests. This keeps the bus simple, but a long-running request can hold up multiple short-running requests from other masters[1][21].

3.3 The AXI4 bus

The AXI4 bus is to recent ARM Cortex processors what the ρ-VEX bus is to the ρ-VEX[36]. It is part of the AMBA (Advanced Microcontroller Bus Architecture) specification, which is an ARM Holdings licensed set of protocols for communication between chips on embedded systems. AXI4 the youngest of the AXI protocols and specifically designed for high-bandwidth on-chip communication. The bus supports any address width and any data bus width which is a multiple of 8. It is an addressed master-slave system, and much like the ρ-VEX bus there are components available to connect multiple slaves and masters on a single bus. The bus itself consists of 5 channels, each channel consists of a multitude of signals. The channels are:

---

[1] See *bus/bus_pkg.vhd*
[2] See *bus/bus_pkg.vhd*
[3] See *bus/bus_arbiter.vhd*
[4] See *bus/bus_demuxer.vhd*

- Read Address Channel
  This channel is driven by the master of the bus. It gives all details on the read request including but not limited to address, size, burst details and priority.

- Read Response Channel
  This channel is driven by the slave of the bus. It returns the requested data, among other things. If the request was a burst, this channel answers as many times as required to fulfill the burst so that one read request can lead to multiple read responses.

- Write Address Channel
  Driven by the master. It is used to set up a write request, which can consist of multiple write actions.

- Write Data Channel
  Driven by the master. It contains the data to be written, but not the address. That should be inferred from the earlier Write Address Channel request. If the write action consists of multiple writes, this channel is accessed multiple times for the same request.

- Write Response Channel
  Driven by the slave. Used to indicate the state of the write transaction, mostly success or fail. It responds once per Write Data Channel request.

Every channel has a valid/ready pair of signals. The valid signal is driven by the driver of the channel, while the ready signal is driven by the receiver of that channel. Figure

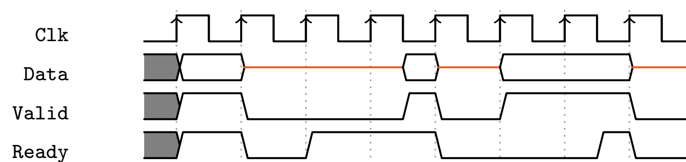


Figure 3.2: A schematic timing diagram of the AXI4 handshake

3.2 Shows the protocol of the handshake: Data transaction happens on the rising edge on which both valid and ready are high. This can allow for one transaction per channel per cycle. Note that multiple channels are involved in a data transaction so that an actual write takes at least 3 cycles and a read at least 2. To keep the thesis short, not all lines in all channels are described, but do note that undefined behaviour is not possible for this bus.

The AXI4 bus is very versatile: it supports independent parallel reading and writing and using and identification mechanism also allows out-of-order requests and it can handle multiple outstanding requests[37]. All of these features come at the cost of simplicity, the bus is a lot more complex than the ρ-VEX bus.

3.4 Linux driver and related software

This section describes the driver, library and program created to allocate some memory to the ρ-VEX and communicate this and other information to the ρ-VEX. Moreover, the implementation is created so that the memory can be flashed from a program running on top of Ubuntu so that flashing the ρ-VEX changes to dumping a file to a certain place in the memory.
The first step is to create a Linux kernel module (henceforth called the ρ-VEX driver) which reserves part of the memory for usage by the ρ-VEX. The ρ-VEX driver allows programs to map this reserved part of the memory into their address space so that a program can modify and read this memory. Furthermore, the ρ-VEX driver handles communication with the ρ-VEX. Registers are added between the ρ-VEX and the processor, these registers allow the driver to propagate the address of the reserved memory, to flush the level 2 cache and to reseed the pseudo-random number generator (PRNG)[5].
The second step is to create a library which provides a simple interface towards the kernel module. The third and final step is writing a program whose primary purpose is to flash a program to the ρ-VEX.

A Linux kernel module called the contiguous memory allocator (CMA) provides additional features to the memory allocator of Linux. It is considered to be a part of the direct memory access (DMA) feature set which is aimed towards devices which require memory buffers but work independently. Sound cards sometimes use this API so that software can prepare a buffer of sound which the soundcard will then output while the processor can do something else.
The Linux memory management workflow is based on scatter and remap. From a kernel perspective the memory consists of pages, usually 4 KiB long, although this is architecture ependent. If a certain amount of memory is requested, the kernel translates this request into an integer amount of pages (rounding up) and then tries to find those pages, but there is no guarantee that any of the returned pages are physically connected. From an user space perspective this is not an issue, since the kernel applies virtual addressing, making the pages look consecutive. But for a device this might be an issue and for the ρ-VEX it is. There are solutions where DMA-like devices can also remap physical pages into a virtual consecutive memory, but not all devices support this. The CMA module is created to accommodate for this final situation, a situation in which a large buffer is required which is physically consecutive[38][35].
Kernel parameters can be used to set the amout of memory in the CMA pool. When allocating memory from this pool, it is possible to tag this memory as uncached. Variables marked as volatile placed in an uncached pool will be reread from memory every time they are requested[39][35]. The ρ-VEX driver thus largely depends on the CMA module.

Both the library and the user space program are quite straight-forward. The library hides the somewhat complex calls required to communicate with the ρ-VEX driver behind

[5] A level 2 cache is created and includes a PRNG, see Section3.5

simple function calls. The user space program accepts one parameter, which should be a path to a SREC file. A SREC file contains the layout of a memory in some special format and can thus be used to flash a program[40]. It would arguably have been better to build a flasher which handles binary files, which are an exact memory dump and thus require little processing. By the time the author realized this, the SREC parser was already fully functional and sufficiently performant.
After flashing, the flasher request an L2 cache flush and an L2 PRNG reseed from the ρ-VEX driver.

3.5 Implementing the interconnect

The second part of the implementation is the bridge or interconnect, which poses as both an AXI4 master and a ρ-VEX slave. The ρ-VEX will send its memory requests to the interconnect and the interconnect translates those into AXI4 communication. A complication in this process is that the address should also be translated. As explained in Section 3.4, the ρ-VEX driver will reserve a section of the memory which serves as the main memory. When the ρ-VEX is powered up (or reset) it starts reading from address 0. The buffer resides at some physical address, and probably not at address 0. So all requests from the ρ-VEX need to have their address recalculated before they can be forwarded to the AXI4 bus. The parameter for this recalculation is supplied by the ρ-VEX driver and logic to achieve this needs to be included in the bridge.
The following assumptions simplify the development of the interconnect and thus will be held to during development.

1. The data bus width of the AXI bus is at least as big as the data width of the ρ-VEX bus.
2. The address bus width of the AXI bus is at least as big as the address bus width of the ρ-VEX bus.
3. No write or read request on the AXI4 bus will result in an error.

Assumption 1 is so that write requests do not have to be split into multiple requests, which would introduce a lot of complexity. Assumption 2 and 3 reduce complexity of the interconnect without removing too much of the functionality. During development, it became apparent that the introduction of a level 2 cache would have a positive influence on the performance.

The documentation on the AXI4 interface seem to imply that a large burst only takes a few more cycles than one read request[34]. This in turn means that it could be advantageous to load multiple words when a single word is requested and then cache all loaded data.
Caches attempt to exploit two tendencies in CPU behaviour: spatial and temporal locality. Spatial locality means that if a certain piece of data is requested, then it is likely that the data around that block will be requested soon. Temporal locality means that if a piece of data is requested then it is likely that the same piece of data will be

requested again soon. The instruction stream has a specific form of spatial locality, since only branches disturb the pattern of "load the next instruction". Branching is also avoided by compilers, since it could force a pipeline flush[9]. Loading multiple words surrounding the requested word on every miss allows a cache to exploit the spatial behaviour of the CPU. Being able to hold a large amount of words allows a cache to exploit the temporal behaviour of the CPU.
In an effort to create a better performing bridge, a cache is added inside the bridge. This cache is synthesis-time configurable[6] and able to hold lines larger than the size of one word. To keep transactions and logic simple, the line will be aligned, meaning that the lowest n bits of the request address will be set to zero and every word within that range will be loaded on miss. This cache will be called the level 2 cache.

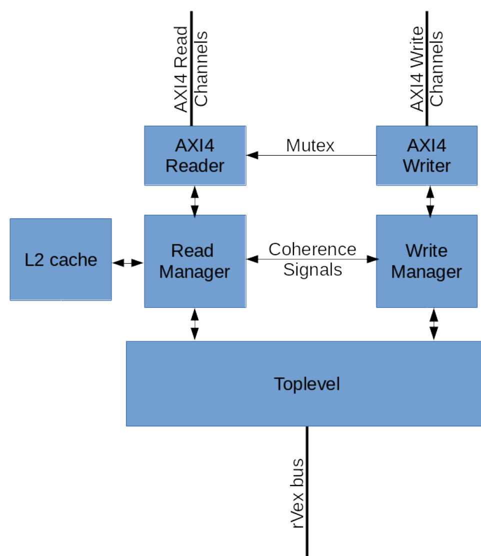


Figure 3.3: A block design of the ρ-VEX to AXI4 bridge

Figure 3.3 shows a block design of the final implementation of the interconnect. The toplevel contains little logic, it just connects all the other components. The toplevel does handle the translation discussed earlier. The AXI4 reader is connected directly to the read manager and utilizes wrapping bursts, meaning that the requested word will be the first word of its line that will be returned after a request. This speeds up transaction: the AXI4 interface can read the rest of the aligned line whilst the ρ-VEX bus can already handle the read request and start waiting for the next request.
The AXI4 read and AXI4 write part can work almost independent of each other so that the interconnect can handle writes while a line is still being written and a new read request can be issued while the AXI4 writer is still active.
The AXI4 writer assumes that there will not be errors when writing. If the AXI4 writer is idle and a write request happens, the writer will act as if the write is completed

[6]Synthesis is the first step of translating a design to a FPGA bitstream.

immediately. The next write from the ρ-VEX bus does have to wait for the completion of the previous one. Effectively, the bridge utilizes a write-back buffer of size one, as opposed to only writing data when a cache line is evicted. This design choice decreases complexity at a possible cost of performance: if a lot of write operations happen and the AXI4 bus handles the write operations slowly, the interconnect has to wait for the AXI4 write bus[9].

Both the mutex and the coherence signals exist to make sure that the cache will not desynchronize from the memory. The ZYNQ system only guarantees that a read returns the last written value, which means that a read from an address that is currently being written needs to wait until the write operation is finished. This situation can arise when a write and a read using the same address follow each other closely. The mutex signal guarantees that the AXI4 reader waits until the write is completed.
When a word from a line that is currently being moved into the cache gets written to by the ρ-VEX bus, the AXI4 writer has to wait until the read is completed. When a write occurs to a cached word, the cache needs to be updated. Both of these situations are handled by the coherence signals. The cache works simultaneous with the AXI4 writer to reduce overhead as much as possible on a cache-hitting write, but a cache-hitting write is more expensive then a write to an uncached word.

The cache allows for associativity by allowing multiple blocks. Associativity is about reducing conflicts. A cache without associativity (also called direct mapped) determines the location of a word based upon the address so that every memory address corresponds to exactly one place in the cache. The cache is smaller than main memory and since every memory block can be cached, there are so-called conflicts. A conflict is a situation where a requested word maps to the same location of another word that was requested in the past. Conflicts are resolved by removing (evicting) the earlier loaded data from the cache and replacing it with the new data. Direct-mapped caches suffer from the possibility of a cache which is not full having to evict useful data because the newly requested block is part of a line which maps to a space that is already occupied.
A n-set associative cache has n possible places for every block. If $n = 1$, the cache is direct mapped, assume from here on out that $n > 1$. If two blocks would have mapped to the same location in a direct mapped cache the same blocks have the n possible places to choose from in a n-set associative cache. If every of the n possibilities contains valid data and a new line needs to be loaded, there still needs to be an eviction. Therefore, a cache eviction policy is required to determine which line out of the n lines will be replaced. The considered options are random, FIFO (First In First Out) and LRU (Least Recently Used), since these algorithms are the most common.

There are two considerations in choosing the cache eviction policy: implementation difficulty and performance. LRU is the best performing policy of the named[41][42], but it is also hard to implement. LRU requires every block (way) of the cache to be updated on every hit and miss of the cache. The complexity also grows with the amount of blocks. FIFO is easier to implement and its complexity grows a lot slower. It does still require some updating logic which introduces additional complexity as the amount

of blocks grows. It is generally worse then LRU but better then random. Random also has its complexity issues: the generator of random bits needs to be uniform to prevent the cache from effectively becoming smaller. But the amount of cache blocks has no influence on the complexity and the random generator is the only complex part: a number comes out and the rest is straight forward. Pseudorandom number generators for VHDL are not very rare, so that there is almost no implementation complexity left. The random policy's performance is worse but quite similar to FIFO and LRU[41]. Early ARM processors also utilized random eviction policies, showing that it works well enough[43]. Random is chosen as a policy, mostly to save time and implementation difficulty. Note that the random policy is only used if every space is filled: if there are still empty spaces to choose from, those spaces are prioritized. The pseudo random number generator implementation is created by J. van Rantwijk[7] and based on the xoroshiro128+ design created by David Blackman and Sebastiano Vigna of the university of Milan[44].

Because the timing path[8] from the ρ-VEX trough the L1 cache into the L2 cache might be too long, a delay unit can be used[9]. This unit delays the input of the L2 cache with one cycle, adding 1 cycle to every transaction. Usually this unit is required so it is included in the numbers of table 3.1. This table shows the timings of the bridge according to simulations of the VHDL code. Related cache writing means that the word that needs to be loaded is in a line of which a word is currently being written. Initial AXI4 read delay is the time from offering the read command to the AXI4 bus up until the return of the first word to the interconnect.

3.6 Area results

There are three free parameters when configuring the cache: the block count (which equals amount of associativity), the line count (for every block, not in total) and the line size (in bit), all of these parameters have to be a power of two. The cache size is an easy calculation: block count $\cdot$ line count $\cdot$ $\frac{\text{line size}}{8}$ bytes. This section is only concerned with the level 2 cache, since the rest of the bridge is small and unaffected by any of the variables.

The cache is split into individual blocks which are interconnected. Every cache block has writer and reader logic implemented in LUTs and flip-flops, the amount of which is quite independent of the size of a block. A block contains a set of lines, each with a tag, a dirty bit and some data. The tag size is ependent on the amount of lines and the linesize, if either of those grow the tag size shrinks. The data and tag are stored in BRAM, the dirty bit is stored in a flip-flop. The optimal cache stores everything in BRAMs and barely uses any other logic.
Increasing the block count means that more writer and reader logic needs to be

[7]joris@jorisvr.nl
[8]A timing path is a set of components the electric signal needs to travel trough within one clock cycle
[9]See bus/bus_halfStage.vhd

Operation	**Required cycles**
Missing read bridge idle or cache writing unrelated	3 + initial AXI4 delay
Missing read, Cache reading unrelated	3 + initial AXI4 read delay + rest of AXI4 read delay
Missing read, Cache writing related	3 + initial AXI4 read delay + rest of AXI4 write delay
Read from word currently being loaded	3 + part of previous AXI4 read delay
Hitting read, any or none cache writing	3
Write, bridge idle, no cache hit	3
Write, write manager idle, cache hit, AXI4 read idle	4
Write, write manager idle, cache hit, AXI4 read working	4 + max (rest of AXI4 write, rest of AXI4 read)
Write, write manager working, no cache hit	3 + rest of AXI4 write
Write, write manager working, cache hit	3 + max (Cache + rest of AXI4 read, rest of AXI4 write)

Table 3.1: Time cost of interconnect operations, predicted by the design

implemented, which leads to an increase in LUT and flip-flop usage. It also has no impact on the amount of tag bits, meaning that more data needs to be stored if the cache size stays equal while the amount of blocks increase.
Increasing the line count means that every block gets more lines. This means that a larger part of the address maps to a specific place in the cache and thus is not part of the tag so that the amount of data that needs to be stored becomes smaller. The impact on logic is hard to predict: some logic needs to grow because they need to handle more bits (like the addressing system), whilst some other logic can become smaller because they need to handle fewer bits (like the tag comparator).
Increasing the line size increases the part of the address that is ignored, meaning that the addressing logic can stay the same size whilst the amount of tag bits that need to be stored decreases.
According to this initial analysis a cache with one block, a large line size and a large amount of lines is an optimal cache from the perspective of area. To test this hypothesis, some cache configurations are tested. Three cache sizes are chosen: 128 KiB, 256 KiB and 512 KiB. The caches are relatively large (512 KiB is the largest possible because

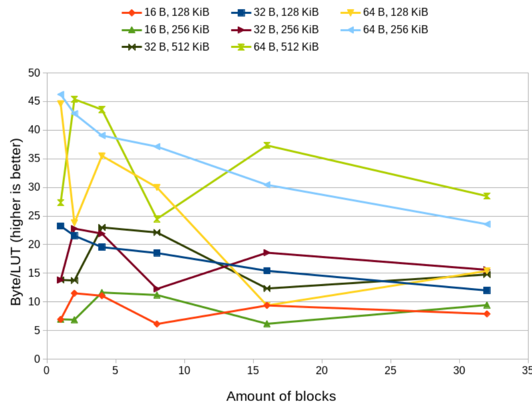


Figure 3.4: The amount of bytes divided by the amount of LUTs for different L2 cache configurations.

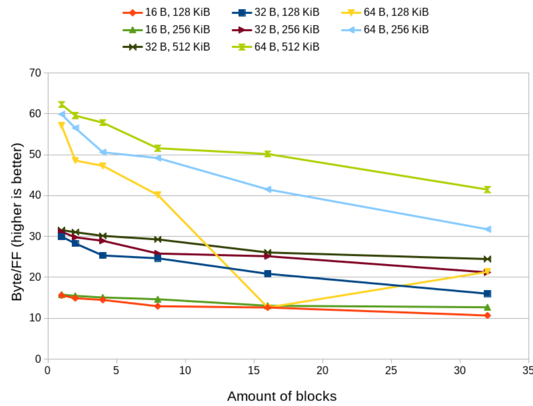


Figure 3.5: The amount of bytes divided by the amount of flip-flops for different L2 cache configurations.

of the amount of block RAMs on the chip) to force Vivado[10] to optimize properly. If the cache is sufficiently small, Vivado will not even bother using BRAMs, since the entire cache fits easily on a couple of LUTs and flip-flops. Making the cache as large as possible will also not guarantee that Vivado will optimize to the smallest possible design, since the design is supposed to lean heavily toward BRAM usage so that a large cache still does not use a lot of other resources. Five different block counts are tested: 1, 2, 4, 8, 16 and 32. Three line sizes are tested: 16 bytes, 32 bytes and 64 bytes. Line count is derived from the other parameters and varies from 16384 to 256.

The numbers shown in the graphs are exclusively the cache results, the rest of the bridge is not included. As explained in Section 2.3 a LUT can either be used as memory or as a logic block. The design of the cache avoids using LUTs as memory so that every LUT in the design is used for logic. The DSP count is ignored since it is always zero. It is not possible to generate an L2 cache of 512 KiB with 16 byte line size because the chip is too small, so there is no result for that setup. The results are represented by graphs in Figures 3.4, 3.5 and 3.6. The absolute amount of units (LUT, FF, BRAM) is divided by the effective cache size in byte ($128 \cdot 2^{10}$, $256 \cdot 2^{10}$ or $512 \cdot 2^{10}$).

The amount of used LUTs ranges from 5640 (128 KiB, 32 Byte line size, 1 block, 4096 lines) to 42486 (512 KiB, 32 byte line size, 16 blocks, 1024 lines), the total amount of LUTs on the chip is 53200[28]. Figure 3.4 shows three groups neatly above each other, if “64 B, 128” KiB is ignored. “64 B, 128” is physically the smallest configuration so that this result can be erroneous because of optimizer laziness. The results where line size is 64 bytes do deviate the most from their respective centers, which is also because of a lack of optimization.
The first conclusion that can be drawn is that a greater line size improves the efficiency

[10] Vivado is the tool that translates VHDL to a bitstream which can be loaded onto the FPGA

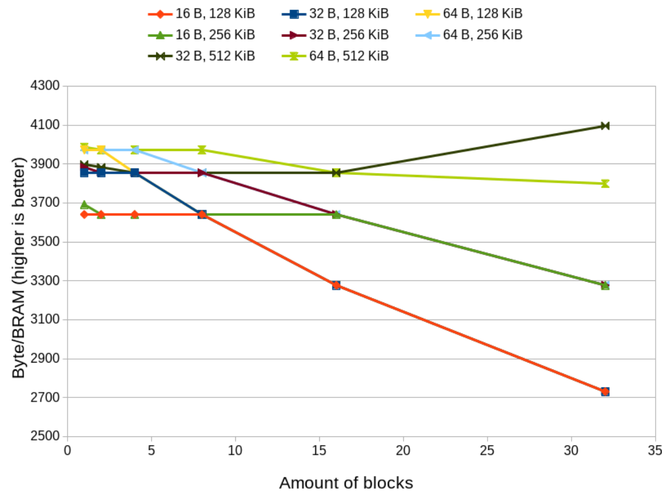


Figure 3.6: The amount of bytes divided by the amount of BRAMs for different L2 cache configurations. All 128 KiB configurations end up in the lower right corner

of LUTs, moreso than having a low amount of associtivity. This trend is even more clear in graph 3.5, which shows the relative flip-flop count for every tested cache configuration. The amount of flip-flops ranges from 2296 (128 KiB, 64 byte line size, 1 block, 4096 lines) to 21427 (512 KiB, 32 byte line size, 32 blocks, 512 lines), the total amount of flip-flops is 106400[28]. Initially, the amount of flip-flops is dominated by the amount of dirty bits: there is one per line, and the three groups at block count equals 1 all have the same amount of lines. Moreover, this number is very close to the amount of dirty bits: 8192, 4096 and 2048 (note how the configuration with the least amount of flip-flops only has 248 flip-flops more then cache lines). If the block count starts rising then the data for Figure 3.5 shows the same trent as Figure 3.4, once again the 128 KiB with 64 byte lines seems poorly optimized. This makes sense: logic LUTs (as opposed to distributed RAM LUTs) usualy feed their data to flip-flops to preserve the output value for the next clock cycle so their counts should have a relation.
Figure 3.6 does not show as much information as hoped. It just seems to prefer bigger caches over smaller caches. The amount of overlap is explained by the fact that there are only 140 blocks and only full or half blocks can be handed out, giving only 280 possible results. For each cache size, the results will be very close so that there is barely any difference left. It is interesting to reiterate that each block RAM (BRAM) is 36 Kb, making each BRAM 4500 bytes large so that 4500 would be the optimal result in Figure 3.6. The graph seems to show that the optimum amount of BRAMs is not reached for any configuration, not even when correcting for tag bits (not shown in graph). The current optimum is 512 KiB, 32 byte lines, 32 blocks and 512 lines at 4096 effective bytes per BRAM and 4384 total bytes per BRAM. That particular configuration would need a total of 561152 bytes to represent all data (tag plus data), which boils down to

127 blocks completely filled plus a rest. The design uses 128 blocks, meaning that the VHDL design allows for optimal usage of BRAMs, despite what the graph says. This optimum is farther away when multiple blocks are involved, giving an argument as to why the amount of blocks should be kept low.

The conclusions concur with the hypothesis: long lines decrease area, more than the other parameters, but block count is also relevant. Doubling the line count barely has any impact on flip-flop count and LUT count, as predicted. The BRAM count grows along with the size of the cache itself and is only marginally affected by the amount of associativity.
There is a need to keep L2 cache size modest: a 512 KiB cache tends to use about 50% of all available LUTs, with worse results if the amount of blocks increase. The ρ-VEX itself consumes a lot of LUTs as do some other components so that a large cache with many blocks will not fit on the chip together with the rest of the setup. The cache is more modest in the flip-flop department, barely ever breaking 10% of total available flip-flops.

To finalize: the bridge excluding the cache uses about 150 LUTs, 135 flip-flops and 0 BRAMs. This number was taken from the 512 KiB, 32 blocks, 32 byte line size design. The numbers exclude the earlier mentioned delay unit, since that unit is not required.

3.7 Performance Results

The performance tests are divided in two parts. First, there will be some tests to compare to the results from the simulator as presented in Table 3.1, to get an insight in the specific cycle costs of certain operations. Then some benchmarks from the Wetstone suite will be tested on a complete setup to see the real performance of the bridge.
For the cycle comparisons test 4 different 1 KiB caches are generated, each with one block and four different line sizes: 16, 32, 64 and 128 byte. This setup is chosen because the amount of blocks nor the cache size has an impact on the amount of cycles for these tests. The bridge is installed as a peripheral and a chunk of on-chip BRAM is used as the memory, so that all calls to the bridge avoid the L1 cache. The cycles of interest in this test are the stall cycles. Stall cycles are cycles in which the context is waiting for something, usually for memory. Branches also introduce a single stall cycle and a context can also be stalled from an external source, for example when reconfiguring. In Table 3.2 and Table 3.3 the stall cycles are only the ones generated by the L2 cache. This test does include the delay unit. Table 3.2 shows the set of results that are independent of linesize. It is noticeable that an uncached read is part of the linesize independent set: it shows that the ZYNQ system always takes roughly the same amount of cycles to formulate the first response to a request.
The cached read and both writes were correctly predicted by the simulator. Do note that there are three cycles overhead due to bridge internals, meaning that it could be possible to create a very simple interconnect which takes 13-14 cycles to respond to any read request.

Operation	Cycle count
Read 4 byte, uncached	16 - 17
Read 4 byte, cached	3
Write 4 byte, uncached	3
Write 4 byte, cached	4
Uncached Burst write 1 KiB	1303
Uncached Burst write 2 words	12

Table 3.2: Interconnect results common to all configurations

Line Size in byte	Burst read 8 byte seq.	Burst read line size seq.	Average cycles per read over 10e6 seq. accesses	Average cycles per read over 10e6 accesses at linesize distance
16	20	26	6.38	16.19
32	24	42	5.19 (+23%)	18.12 (-12%)
64	32	74	4.60 (+13%)	26.42 (-45%)
128	48	138	4.34 (+6%)	42.54 (-61%)

Table 3.3: Interconnect results specific for all configurations

The long write in row 5 shows that the AXI4 overhead dominates the bridge internal delay so that it virtually does not matter whether or not the written words are cached. The write 2 words statistic in row 6 also shows no difference between cached and uncached. When comparing row 1 and row 5 it is clear that the ZYNQ system handles writes faster than reads: row 5 shows an average of 5.09 cycles per write.

Table 3.3 shows the results which differ from configuration to configuration. The read delay cycles consists of two parts: an initial delay and a second word delay. The initial delay needs to be paid before the first word can be read from the AXI4 bus. If the next read request follows directly and also reads from the same line it needs to pay the second word delay, the time left on the transaction from the AXI4 bus into the cache. The second word delay only needs to be (partially) paid if the next request on the same line follows closely: it is a set time after initial delay.
The last two columns of Table 3.3 show a best and the worst case burst scenario for the L2 cache: a best case burst is a sequential one because then every cached word is read and thus the burst speedup from the AXI4 communication can be optimally used. The worst case is when only one word from every loaded line is read, which is depicted by the last column. The best case for reading would be to only read words in a cached line, but that does not supply too much information.
The results in the last two columns seem to prefer 32 byte line over 16 byte: the performance increase in the best case is more than the performance decrease in the worst case. These results also show that 128 byte lines show almost no performance increase over 64 byte linesize so that 128 byte linesize is not useful.

Cache performance is dictated by a metric called AMAT, which stands for average memory access time. AMAT is calculated as Time for a hit + Miss rate · Miss penalty. The entire development revolved around reducing hit and miss times, this subsection is about tuning the three free parameters (block count, line size and line count) to reduce miss rate.
A n-way associtative cache has n places for every block, meaning that in case of a conflict it might not be required to evict, which in turn could increase hit rate. M. D. Hill and A. J. Smith [45] showed that the returns of increasing associativity rapidly declines, with a notable performance increase when going from 0 to 2, smaller when going from 2 to 4 and barely noticeable when going from 4 to 8. This would imply that optimal is around 4.
The second parameter is the line size. Increasing linesize attempts to lower the miss rate by improving spatial locality behaviour. It is a common situation to first read memory address m, then $m + 4$ and then read memory address $m + 8$. The wordsize of this cache is eight (which is because of the AXI4 bus), so that the first and third read would cause a miss and take quite long to resolve. The idea is to slightly increase the stall time at the time of the first miss to load multiple words in order to generate more hits in the future, so that the total amount of stall cycles decreases.
Increasing the cache line size whilst keeping the total cache size the same might damage the caches' ability to exploit temporal locality, which attempts to exploit the fact that the CPU tends to load the same data more then once. For example: assume that there is only one line. Now, when a program continuously jumps between two words which are not on the same cache line, the cache always misses. This means that there is an optimum between the amount of lines and their sizes. It is also important that the extra spatial locality is actually exploited, because the amount of stall cycles does increase, as shown by Table 3.3. The same table also shows that the performance decrease when going from 16 byte to 32 byte is relatively small whilst the speedup is relatively large. So it is expected that 32 byte is the optimal line size.
The cache size has the most impact: if all the code and data fits in the cache then the linesize and associativity are irrelevant, since evictions are not required.

The test setup is a single 4 issue (2 lanegroups) 2 context ρ-VEX, a L1 cache at the default size of 64 lines (which translates to a 2 KiB instruction cache and a 256 byte data cache) and an interconnect which contains a L2 cache with 4 KiB space in differing setups. Tested block counts are 1, 2, 4 and 8. Tested line sizes are 16 byte, 32 byte and 64 byte. Every test is repeated 10 times, each time the L2 cache is flushed and reseeded. All benchmark code is optimized with *-Os*, optimize for size.
The used benchmarks are a subset of the PowerStone benchmark suite. G3fax and quicksort are particularly highlighted because their memory footprints are bigger than the cache size and thus their different behaviour can be investigated. Matrix multiplication is tested with 3 instance sizes: 16x16 matrices, 32x32 matrices and 64x64 matrices. There are also two compound tests, one single-threaded and one multithreaded. These tests contain a larger subset of the benchmark suite. The multithreaded test consists of

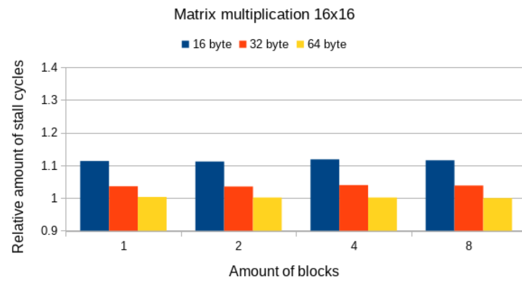


Figure 3.7

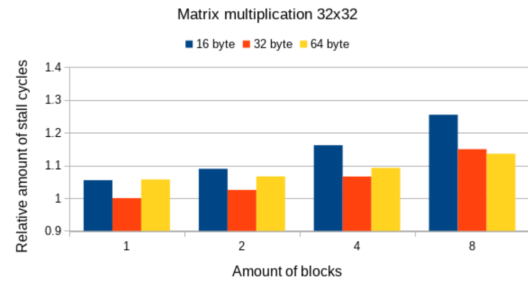


Figure 3.8

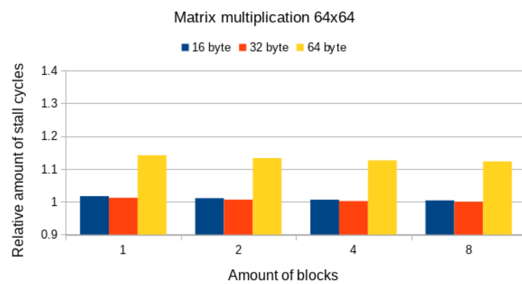


Figure 3.9

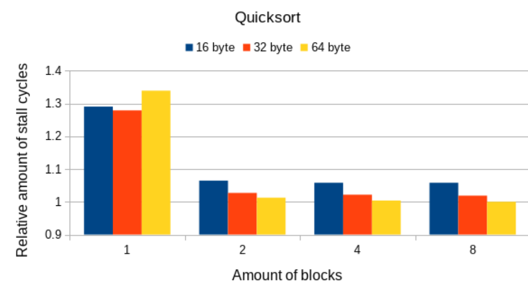


Figure 3.10

two threads, both running different benchmarks.

After all stall cycle amounts for all setups are gathered, each individual result is divided by the lowest (most optimal) result for that benchmark. This means that the optmimal setup for a benchmark gets the value one and every other setup gets a value higher than one. Figures 3.7 until 3.13 show the resulting graphs. Figure 3.7 shows the results for matrix multiplication of two matrices which are 16 by 16. This means that every matrix is $16 \cdot 16 \cdot 4 = 1024$ bytes large, so that a total of 2 KiB is required to store the data for the main loop. This fits in the cache, which explains the results. The matrices are stored in such a way that there are no collisions, so that an increase of associativity has no effect. A longer linesize helps the algorithm store its data faster

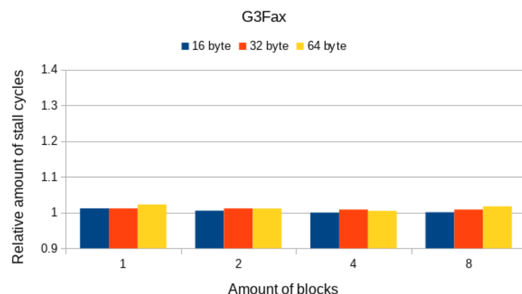


Figure 3.11

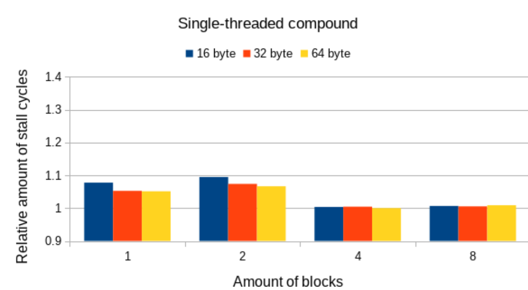


Figure 3.12

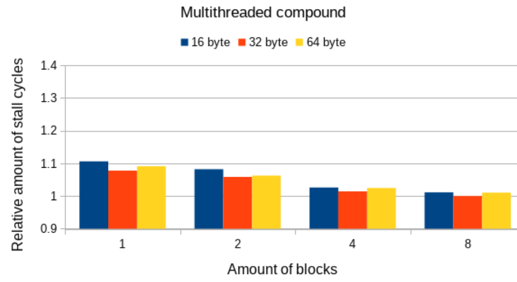


Figure 3.13

into the cache and thus leads to a better result.
Figure 3.8 shows the result for 32x32 matrix multiplication. Each of these matrices is 4 KiB large and the two matrices which are multiplied are stored directly after each other. This leads to a very interesting result: increasing the associativity leads to a decrease in performance. This result contradicts the hypothesis which stated that the hit rate should increase if associativity increases. Algorithm 5 shows the algorithm as implemented. The

Algorithm 5: Naive matrix multiplication algorithm

Input: Three matrices, A, B and Sol.
Output: True if $C = A \cdot B$ equals Sol, else false

```
1  for i ← 0 to 31 do
2  |  for j ← 0 to 31 do
3  |  |  for k ← 0 to 31 do
4  |  |  |  C[i][j] = A[i][k] · B[k][j]
5  |  |  end
6  |  end
7  end
8  for i ← 0 to 31 do
9  |  for j ← 0 to 31 do
10 |  |  if Sol[i][j] ≠ C[i][j] then
11 |  |  |  return false
12 |  |  end
13 |  end
14 end
15 return true
```

implementation never makes mistakes, so that the second loop is never aborted. The first loop dominates runtime. The matrices A and B are placed back-to-back with no padding space or other variables in between. The result is that $A[n][m]$ and $B[n][m]$ map to the same place in the cache, for every n and m. The matrices are stored row wise, so that the second row starts at place 32. The instructions all fit in the level 1 instruction cache. The value $C[i][j]$ is loaded and stored once (and initially zero) for every inner loop iteration, so that in general only $A[i][k]$ and $B[k][j]$ are loaded, multiplied, added

to a register and then the next iteration starts. The loading of C is ignored, as is the L1 data cache since the cache is so small that when the algorithm starts reusing items, these items are already evicted.

Assume that the L2 cache is directly mapped and that all relevant instructions are cached in the L1 instruction cache. Assume that the L2 cache has 16 byte (= 4 words). When looking at the situation in which $i = j = k = 0$ and the L2 cache not containing relevant information, there are two misses in a row and the second will evict the first. On the nex iteration $B[1][0]$ is loaded, which is a large memory jump and ends up in a different L2 cache line. $A[0][1]$ might or might not miss, ependent on if $A[0][0]$ was loaded before or after $B[0][0]$. Now A misses every fourth word whilst B misses all until $A[1][0]$ $B[0][0]$ is reached. B then repeats its previous pattern, which is still mostly cached, causing two misses whilst reading the entire column of 32 items. A once again misses one in four. This pattern continues until $B[4][0]$ is reached, where the initial pattern happens again. So there are two patterns for matrix B: the initial pattern in which all misses and the major pattern in which there are few misses and most hits.

If associativity is increased, the collisions change. It used to be that when B missed there would be a line from A loaded in that spot of the cache and that would be replaced. But now, there is a random choice from a set of items. A and B are usually quite far away from each other so that there usually will be one line loaded with data from A and all other lines will contain data from B. Data from B might be needed in just a couple of iterations, so that evicting the line from A is really the optimal answer, but randomness decreases this chance further when associativity increases. This explains the negative effect of increasing associativity. Increasing the linesize with little associativity has the expected effect, but when a lot of associativity is involved, hits save so many cycles on longer lines that the 64 byte configuration actually wins.

Figure 3.9 shows the result of 64x64 matrix multiplication. Its matrices are 16 KiB in size, which means that it is unlikely that a cached line is still cached whenever it is needed again. When the cache lines are increased to 64 bytes, the amount of collisions becomes so large that 64 byte linesize shows a large slowdown compared to the others. The amount of associativity is not really relevant since the amount of collisions that could be solved by it is already quite low. 32 byte linesize shows to be optimal by a small margin.
Quicksort, as depicted by Figure 3.10 shows it is helped a lot by some associativity. The items commonly used in the algorithm are stored close together by the algorithm, so that an increase in linesize tends to mean an increase in hit rate.
Figure 3.11 shows barely any response to a change in hit rate when associativity or linesize is changed. This is due to the fact that the main data array used in the algorithm is quite large so that it does not fit in the cache and its access pattern does not have enough locality to be exploited. The only real way to improve performance for this algorithm is increasing the cache size.

The two compound metrics, as seen in Figures 3.12 and 3.13. Both consists of quicksort, 32x32 matrix multiplication, crc, blit, and g3fax. The multithreaded compound runs g3fax on thread 1 and all the other benchmarks on thread 0. G3fax is by far the longest running benchmark. The single-threaded compound shows barely any difference between 4-way associativity and 8-way associativity. 4-way with 64 byte lines is optimal, according to the single-threaded compound test. The multithreaded compound test shows largely the same results, albeit with a lot more stall cycles: there are almost three times more stall cycles in the multithreaded compound as in the single-threaded.

3.8 Conclusions and future improvements

The research question *How can we create a way for the ρ-VEX to get access to the on-board RAM of the PYNQ board?* was answered in this chapter by explaining the implementation a ρ-VEX driver and an interconnect with a cache included. Together they give the ρ-VEX access to the DDR3 RAM.

The interconnect was benchmarked to determine its area usage and performance. The results on the area shows that a bigger cache is generally a better use of resources and that increasing the size of lines over the amount of blocks is preferable from an area perspective.
Performance results show that the inclusion of the cache has the potiential to reduce stall time on read from the DDR3 RAM from over 10 cycles to just above 3 cycles in the best case scenario. It was also shown that the amount of blocks, the amount of lines and the size of the lines have a large influence on the performance of the cache. The ideal cache layout is ependent on the program that is being ran, but in general it seems optimal to have about 4 blocks, a 32 byte line and as many lines as you can afford.

There are multiple possible improvements to the interconnect. First of all it is possible to reduce write on cache hit time by one more cycle, although this requires the architecture to change. Currently, the cache is just a slave to the read manager, so that there is one additional cycle of overhead whilst the request travels from the write manager to the read manager and finally to the cache. If the cache becomes a more separate entity the additional step can be removed and writes can be made one cycle faster.
A second possibility is to allow certain addresses to not be cached. The goal in this chapter was to connect to memory and a cache can improve performance in those situations. But the AXI4 bus allows more devices, and some of those devices give a different result every time they are read. Moreover, it can be important for the processor to wait for a write to complete. Currently the level 2 cache prevents direct access to such devices, making it impossible for the ρ-VEX to communicate with them.

4 Enabling inter-thread communication

This chapter will document what changes and implementations will be made to prevent race conditions from happening, so that inter-thread communication can be made possible. Therefore, this chapter will address the following research question: *How to add possibilities for inter-thread communication to the ρ-VEX?* Section 4.3 will discuss changes to the bus and will refer to Section 3.2.

Section 4.1 will discuss the required hardware changes and decide what solution will be implemented into the ρ-VEX. Sections 4.2 and 4.3 will discuss the implementation of the solution and its effects on the toolchain. Section 4.4 will finalize by discussing the software implemented to ensure that inter-thread communication becomes possible.

4.1 Choosing a suitable solution

Section 2.1 introduced a major problem in concurrent programming: race conditions. Two solutions were proposed, critical sections and wait-free synchronization. The same section then explained atomic memory, Lamport's memory model, test-and-set (TAS), compare-and-swap (CAS) and load-linked/store-conditional (LL/SC). This section will choose a suitable solution.
We have at least the following requirements:

1. The consensus number of the solution should be infinite.

2. The required changes cannot penalize every memory transaction.

Requirement one is mostly to support possible future research, this project will use critical sections to prevent race conditions. If this requirement is not included then the possibility arises that this subject has to be revisited if wait-free synchronization on the ρ-VEX is researched. The second requirement is a direct result of the focus of this thesis on performance.

Based on the second requirement, both atomic memory and Lamport's memory model are not possible. Section 2.2.3 explains that the level 1 cache needs to be removed before the memory hierarchy of the ρ-VEX can adhere to either model, but removing the level 1 cache penalizes every memory transaction.
The second requirement eliminates TAS, since its consensus number is only two. So finally, it has to be either CAS or LL/SC.

As discussed before, CAS suffers from the ABA problem. This refers to a situation in which some data is loaded as A, then changed by some other thread to become B, then changed again by some other thread to become A again and then the original thread executes the CAS instruction. The sequence should be rejected because the data loaded has changed during the manipulation, but the hardware cannot detect this issue[15]. The ABA problem can occur in any setup in which CAS is used and does not have a universal solution[16].
Load-linked/store-conditional prevents the ABA problem by introducing link registers. Load-linked is used to load the data before manipulation and it stores the address of the data together with the callee of load-linked in the so-called link register. All information is thus available to the system so that any change to this address can be detected, preventing the ABA problem. A new problem is that there cannot be an infinite amount of link registers so that false rejects become a possibility. The problem of finite link registers can be solved by making sure that LL/SC pairs are not interleaved, ie every load-linked should be followed by store-conditional before a second load-linked is issued. CAS also performs an additional load compared to LL/SC. When CAS is executed, it needs to first load the data from the memory to compare its value and then (possibly) execute a store. The store-conditional instruction does not have to load the data from the main memory first, since all required data was already stored when executing load-linked. Based on these arguments, LL/SC is chosen. The next sections will describe its implementation.

4.2 Implementation into the toolchain

The first step is to introduce both instructions into the toolchain. The load-linked instruction is introduced as *ldwl*. It loads a single, full word whilst setting a link register. The mnemonic for store conditional is *stwl*. It attempts to write a single, full word and sets a branch register to indicate whether or not the write was succesful.
There are no half-word or byte linked operations, since there is no real use case for this. Being able to operate on half-words and bytes is useful in the case of peripherals and in the case of operating on specific data structures to save memory. In the peripheral case, load-linked/store conditional makes no sense since the goal is to make sure that nothing else changed the underlying memory whilst the core operated on it. The peripheral is no memory in that sense. Atomic operations are not useful in the case of memory-saving operations because they should be so sparse and rare that it should not matter[46].
These instructions are first introduced in the simulator to validate design choices.

4.3 Implementation into hardware

The primary idea behind the implementation is to exploit the fact that the ρ-VEX bus executes all memory operations in some order and thus attach a new unit (the synchronization unit) to the bus at some point after the L1 cache, in a place where all requests from all contexts are merged. The two new memory instructions avoid the level 1 cache and stall the issuee core until the transaction is completed, much like when a core is connected to the ρ-VEX bus directly.
The synchronization until will only respond to load-linked or store-conditional. A seemingly more intuitive approach is to store the relevant data during a load-linked and then listen for any store to the same address. If such a store occurs, the link register should be invalidated so that the store-conditional gets rejected. The issue with this approach is that it is costly to ensure consistent behaviour. If a store-conditional occurs there might be a relevant store waiting in some level 1 write-back cache so that it is wise to flush these buffers first. To ensure that the flushing takes a finite amount of time the contexts should be stalled during flushing. This approach is inadvisable because of the possibility of a denial-of-service attack: if one context continuously executes the *stwl* instruction other cores can barely progress since they spend so much time stalled. An improvement to this method would be to introduce some smart system that detects relevant changes and only stalls relevant cores. But are these changes even useful?
If there is one core which considers address a to be some concurrent object, like a semaphore and a second core which considers a to be some other object, like a normal counter, the changes described become relevant. But it can (and should) be argued that this is a mistake by the programmer which the hardware cannot guard against anyway. With a proper system in place which reads all stores and somehow flushes all relevant stores from the L1 buffers before accepting any store-conditional, there are still no guarantees on the concurrent object. The store-conditional could be accepted just before the second core updates the value with total disregard for the changes the first core just made: it is just not useful to guard against it. If an address is to be defined a concurrent object, every line of code should treat it as such and only use *ldwl* and *stwl*

on that address so that only those instructions are relevant to the synchronization unit.

The synchronization unit has a statically configurable granularity, the default option ignores the two least significant bits of the address. This means that a load-link on address 1 is effectively counted as a load-linked to address 0. Having only one link register makes the unit less complex but also means that every load-linked overrides the previous data. A link register is invalidated if and only if a store-conditional to that same address (under granularity) is executed successfully. So a rejected store-conditional or a common store has no impact on the current registers. The unit only delays store-conditionals with one cycle, every other bus operation passes by as if there was no synchronization unit.
It is also possible to flush link registers for a specific context, this is usually required if the context is interrupted. An example of the new memory hierarchy can be seen in figure 4.1. To allow the synchronization unit to function, the ρ-VEX bus needs

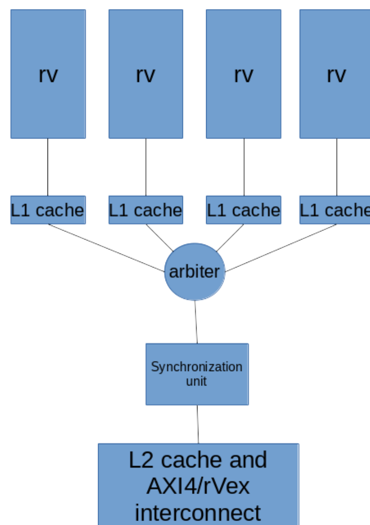


Figure 4.1: A block design of the ρ-VEX memory hierarchy, including the synchronization unit

to change[1]. First, a flag called “Synchronize” is added. This flag signals that the synchronization unit should be concerned with this bus request. Next, a new 32 bit signal is added, this signal tells the source of the request. This signal is usually set after an arbiter, but can be set at any place. The L1 cache will not set this signal, since it cannot know the correct ID of the context which transmitted the request from

[1] see figure 3.1

the perspective of the synchronization unit. Finally, the inner working of the bus is slightly changed: the readData is used to communicate from the synchronization unit to the callee core if the store conditional failed or succeeded. If the implementation is not careful enough this breaks compatibility with other units, since readData is assumed not to be read after a write request unless fault is high (a failed store conditional is not defined as a fault, since the core handles faults separately). The synchronization unit therefore overrides readData whenever fault is low after a write request.

a synchronization signal is added to the ρ-VEX, and the instructions are implemented. The new instructions are nearly identical to the existing load word and store word so that this implementation is quite trivial. Implementation in the L1 cache is slightly harder. Every memory request from the core enters the L1 cache, so the L1 cache is responsible for immediately forwarding the request and stalling the core. All of the logic required is already implemented for some other purpose so that only slight modifications are needed.

4.3.1 A note about performance

It makes sense that load-linked and store-conditional are not the best performing memory operations: they avoid the level 1 cache and thus always have to pay the overhead for the L2 cache or the main memory. A major issue is using load-linked to busy-wait, which is a common use case in an environment without an operating system. When busy-waiting, the core continuously issues *ldwl* until it gets the response it wants. Normal busy-waiting goes back and forth between the L1 cache and the core and thus has no impact on the performance of the shared ρ-VEX bus. Busy waiting with *ldwl* does cause an issue, since every read is handled by the first common part of the memory hierarchy and therefore makes other operations wait: it clutters the bus. For this reason a performing solution to, for example, waiting for a semaphore should be a combination of normal read operations and load-linked. This works because an accepted store-conditional invalidates all level 1 cache lines with that address so that a new load will read the new value.

4.4 Software primitives

The final step of the implementation is an actual solution, implemented in software. As discussed in Section 2.1, load-linked/store-conditional can be used to implement wait-free synchronization and critical sections. Critical sections are easier to implement and chosen as the way to go for this thesis.
Two primitives are introduced, the semaphore and the atomic integer. The semaphore is an idea introduced by Dijkstra in [47] and further explored by Tanenbaum and Woodhull in [46]. The semaphore is a counter which can only contain positive values and zero. Semaphores can be used to guard critical sections by initializing a semaphore to one, and only allow processes to increment the semaphore after they first decremented it. But semaphores are a lot more versatile[46].
Setting the semaphore's initial value can be done by the compiler using a macro or by calling a function. This semaphore implementation supports three operations: *post*,

wait and *trywait*. The *post* operation utilizes the hardware atomic implementation to increment the value of the semaphore by one. The *wait* operation waits until the value of the semaphore is bigger than zero, subtracts one and returns. This function is also implemented using load-linked/store-conditional so that when two threads are in the *wait* function while *post* is called only one thread will return from the *wait* function while the other one keeps waiting. The *wait* function utilizes only busy waiting and has no specific order so that it does not adhere to Knuths' requirements[5] which states that any thread entering *wait* has to have a guarantee that it eventually will leave *wait*. It is guaranteed that if there are multiple threads in the *wait* function and a *post* happens exactly one thread gets to leave the *wait* function, so that Dijkstras' requirements are fulfilled[4]. The *trywait* function attempts to grab the semaphore. If it can, it will return one, else zero. It will not wait for the semaphore.
The atomic integer is merely an integer which is incremented and decremented using special functions. Currently, it can only increment or decrement one step which is enough for the purpose of this thesis. Note that implementing just a semaphore would have been an option, but since an atomic integer is a very common case, it is decided to implement it directly using primitives.

4.5 Conclusion

The research question *How to add possibilities for inter-thread communication to the ρ-VEX?* has been answered in this Chapter. To make sure that the ρ-VEX bus would not be penalized in every transaction, load-linked/store-conditional instructions were added. These instructions perform worse then other memory instructions, but in return guarantee behaviour that is exploited by a semaphore implementation to allow critical sections. Using these critical sections, race conditions can be prevented so that efficient and correct inter-thread communication is now possible.

5 Benchmarking

This chapter will research if there is performance improvement due to thread level parallelism (TLP) on the ρ-VEX. Therefore, this chapter will address the research question *What is the impact of inter-thread communication on performance on the ρ-VEX?*. This chapter also looks into the combination of dynamic reconfiguration and TLP, so that the research question *What does the optimally performing program for the ρ-VEX look like and how much can runtime reconfigurability improve performance?* is touched upon.
Section 5.1 will introduce some definitions and explain the goals further. Section 5.2 will explain the testsetup on which the benchmarks will run. Section 5.3 will present the results of the first benchmark, quicksort. Section 5.4 will present the results of the second benchmark, matrix multiplication.

5.1 Prerequisites

Now that inter-thread communication is possible, we would like to see its effect on performance. As discussed before in Section 2.4, better performance means shorter runtime and runtime consists of three parts:

1. Time executing code (called active time).
2. Time stalled.
3. Time spend on pipeline flushes.

Each of these values is initially measured in an amount of cycles and then translated to a time in seconds. This is possible because the clock frequency is known beforehand. A cycle is counted as an active cycle (item one) if a new syllable entered the pipeline during that cycle. A cycle is counted as a stall cycle if the processor was waiting for an answer from the ρ-VEX bus during that cycle. Stall cycles happen when load and store instructions are executed and when instructions are requested. A cycle is counted as a pipeline flush cycle if the processor was not stalled but no words were processed either. Pipeline flush cycles happen after branches and when the processor is interrupted.
Section 2.4 also described that the addition of inter-thread communication attempts to affect item one (active time) the most. The general idea is to keep every lane active and thread level parallelism helps because the average syllables per word might be lower than the amount of lanes available, so that available resources cannot be used. When multiple threads can be active it might become possible to use all lanes. The primary goal of this chapter is to explore if this idea works and how it can be effectively implemented on the ρ-VEX.
It is not always possible to effectively use thread level parallelism, there are a couple of factors in play.

1. Amount of independent subtasks.
2. Average amount of syllables per word of the subtasks.
3. Relative multithreading overhead.

Item one is trivial: if threads cannot work separately from each other, it is not possible to utilize more lanes than one thread would have. Item two and three are very much related. If the subtasks have a high syllables per word and the implementation has a high amount of multithreading overhead it makes little sense to spread the task over multiple contexts.
The next sections investigate both quicksort and matrix multiplication in an attempt to quantify the impact of inter-thread communication on performance.

5.1.1 Testing dynamic solutions

As the introduction already stated, this chapter also touches upon the last research question, which asks what the performance gain of reconfiguration in combination with thread level communication is. To give some background on this research question, this

chapter will attempt to include the dynamic ρ-VEX into the graphs.
The idea behind reconfiguration is that a program has different phases and each phase has an optimal configuration. The ρ-VEX should thus reconfigure strategically to optimize for each phase. In the next sections, multiple different implementations are described. A description of the role of reconfigurability is given for each implementation.

5.2 Test setup

This section will define the exact testsetup on which all benchmarks will be run. The hardware is the PYNQ-Z1 as described in Section 2.3. The interconnect described in Chapter 3 is used to connect the ρ-VEX to the memory on the PYNQ-Z1. A total of 100 MiB is reserved for the ρ-VEX. The setup utilizes two ρ-VEX processors, each with four lanes and two contexts. Section 2.2 describes that an ρ-VEX can have at most eight lanes and four contexts, but that setup is larger than our setup. Our setup cannot attach eight lanes to one context, but previous results have shown that connecting eight lanes to one context does not perform muchbetter than connecting four lanes to one context, if at all[48]. So, choosing the setup with two ρ-VEX cores leaves more components available to use in the level 1 and level 2 cache. This particular setup has lanegroups of size two, meaning that either 0, 2 or 4 lanes are attached to each context.
Experimentation shows that 4 KiB level 1 instruction cache is the largest possible. It is possible to have 8 KiB data cache instead of 4, but then the L2 cache cannot be larger then 4 KiB. Therefore, the L1 data cache is limited to 4 KiB and the L2 cache is 256 KiB. Figure 5.1 shows a schematic overview of the testsetup. Note that this figure seems

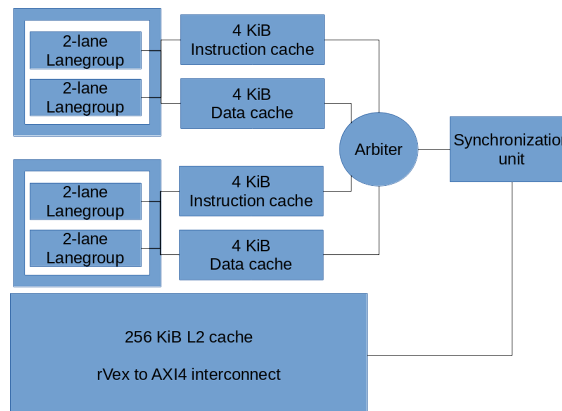


Figure 5.1: A schematic overview of the test setup.

to imply that a two-issue core can access the full 8 KiB of L1 cache, but this is not the case: it can only access 2 KiB instruction cache and 2 KiB data cache[21].
The naming scheme will be $n \times m$ for static setups, where n is the amount of contexts involved and m is the amount of lanes per context. Dynamic setups are called dn, where

n is the amount of cores involved.
Runtime is measured in active time of context 0. Context 0 is required to be active from the very start until the correct result is determined. Checking the result is not counted. The setup always runs at 50 MHz.

5.3 Quicksort

Quicksort is a well-known sorting algorithm which only requires a list of numbers as input. The algorithm will select a random number from the list (called the pivot) and then split the list in two sublists: one in which every number is smaller than the pivot and one in which every number is bigger than the pivot. Numbers equal to the pivot can be placed in any list, as long as this is done consistently. The pivot is then placed between the two lists and on its left and right two sublists emerge. The two sublists then will be sorted in the same manner, until list size one is reached, at which point the list is sorted. Quicksort relies on good splits, splits where both sublists are quite equal in size. If the list is random and a random pivot is selected it will almost always work out, making the expected runtime $\mathcal{O}(n \log n)$, where n is the size of the list. The worst case runtime is $\mathcal{O}(n^2)$. In this implementation element 0 is always chosen as pivot, which makes no difference if the lists are random[49]. This algorithm is in-place: in principle it requires no memory overhead. Any practical implementation will require some sort of a stack so that the memory overhead becomes $\mathcal{O}(\log n)$[50]. Every sublist can be handled by an independen subtask so which shows that this algorithm can be multithreaded. The algorithm starts with one list which can only be handled as one task, so that there is also an incentive to split the core during runtime, making a possible dynamic implementation feasible

Two different implementations of quicksort will be tested: a naive implementation and an optimized implementation. The difference is the way independent subtasks are handled, the single context implementations are the same for both.

The test instances are created by generating lists of predetermined sizes using a random number generator which generates numbers between 0 and 2^{30}. Twenty of these lists are generated for every size and every list is tested three times. A split L1 is two KiB and a number is four byte so that a split L1 cache can hold at most 512 numbers. A merged L1 cache is four KiB and can hold at most 1024 numbers. The L2 cache can hold at most 65536 numbers.

5.3.1 A naive implementation

The multithreading in the naive implementation works by dividing the entire list in two sublists in the first step and then passing one sublist to another thread whilst continuing on the other sublist. If there are four threads this step is repeated. From there on out each context will handle the sublist it has until completion.
The dynamic single core implementation starts as a single four-issue in the initial step and then splits into a 2x2. The biggest sublist is passed to context 0. This choice is based

on a heuristic: a bigger list is expected to generate more sublists and since execution time is counted as execution time of context 0, it is preferable to have context 0 handle the bigger part of the work. When the other context finishes, it merges its lanes back into context 0.
The dynamic dual-core implementation starts as a single four issue, which then passes the smallest sublist after the first step to the second core. After splitting this sublists both active contexts split and keep the largest sublist to themselves. The youngest contexts remerge when they have finished sorting their sublists. This implementation is called the naive implementation because it depends on the first splits being quite good in order to keep the contexts working. This setup does provide and interesting case for dynamic cores: there is now a pattern of first one task, then multiple and then one task again. The dynamic core can adapt to this situation and might be able to show better performance compared to the static setups.
The multithreading overhead is only ependent on the amount of threads for this implementation.

Two different configurations are tested, one where the code is optimized for size (-*Os*) and one with maximum optmization leven (-*O3*), the latter optimization reaches a higher average amount of syllables per word. The specific compilers delivered worse or equal performance compared to the generic compiler so that only the generic compiler is used.

Instance Size	**1x4**	**d1**	**2x4**	**d2**
64	2.29	2.17	2.14	2.18
128	2.27	2.19	2.16	2.18
256	2.25	2.20	2.13	2.18
512	2.23	2.17	2.06	2.17
768	2.24	2.14	2.15	2.17
1024	2.23	2.19	2.08	2.18
1536	2.23	2.17	2.11	2.17
2048	2.23	2.16	2.09	2.18
4096	2.22	2.15	2.09	2.12
8192	2.21	2.16	2.10	2.16
16284	2.21	2.15	2.10	2.16
32768	2.20	2.14	2.08	2.15
65536	2.20	2.16	2.10	2.15
131072	2.20	2.13	2.09	2.14
262144	2.19	2.15	2.07	2.13
524288	2.19	2.14	2.14	2.13
1048576	2.19	2.14	2.11	2.14

Table 5.1: Syllables per bundle for naive -*Os* quicksort

Instance Size	**1x4**	**d1**	**2x4**	**d2**
64	2.57	2.35	2.22	2.34
128	2.54	2.31	2.30	2.32
256	2.52	2.34	2.22	2.42
512	2.51	2.33	2.22	2.33
768	2.51	2.35	2.23	2.35
1024	2.50	2.38	2.23	2.39
1536	2.50	2.38	2.29	2.39
2048	2.50	2.37	2.28	2.33
4096	2.49	2.40	2.17	2.38
8192	2.49	2.36	2.26	2.36
16284	2.48	2.33	2.33	2.33
32768	2.48	2.36	2.29	2.32
65536	2.47	2.33	2.25	2.35
131072	2.47	2.31	2.28	2.36
262144	2.47	2.28	2.33	2.33
524288	2.47	2.35	2.21	2.32
1048576	2.46	2.34	2.22	2.32

Table 5.2: Syllables per bundle for naive -*O3* quicksort

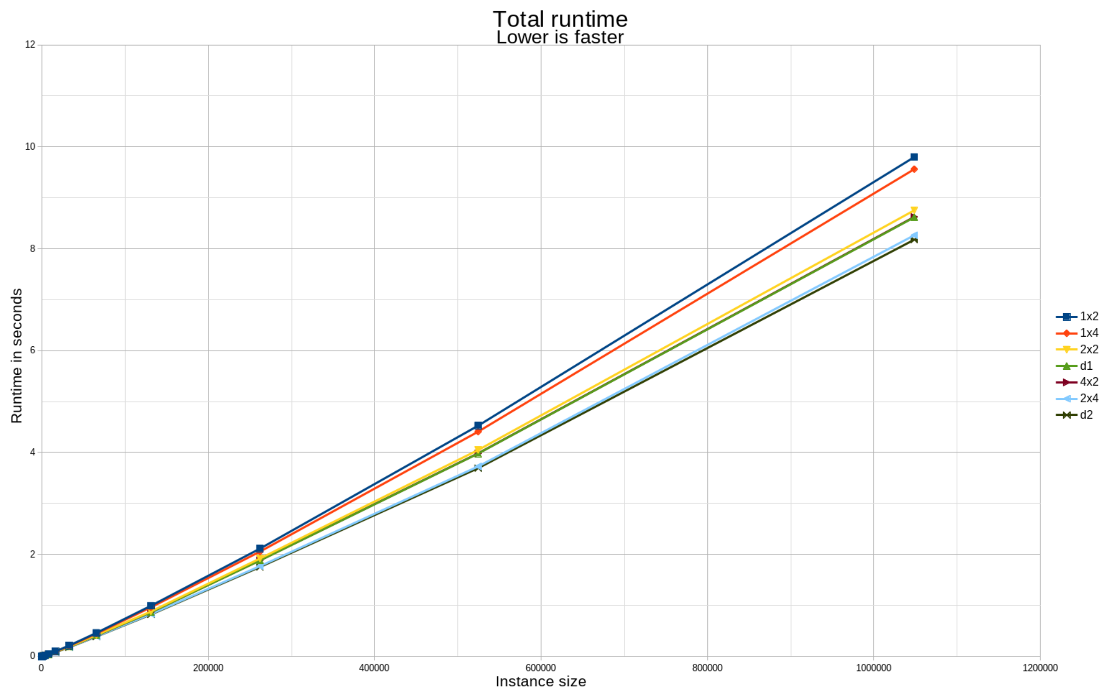


Figure 5.2: The runtime of the different setups for naive quicksort, optimization -*Os*.

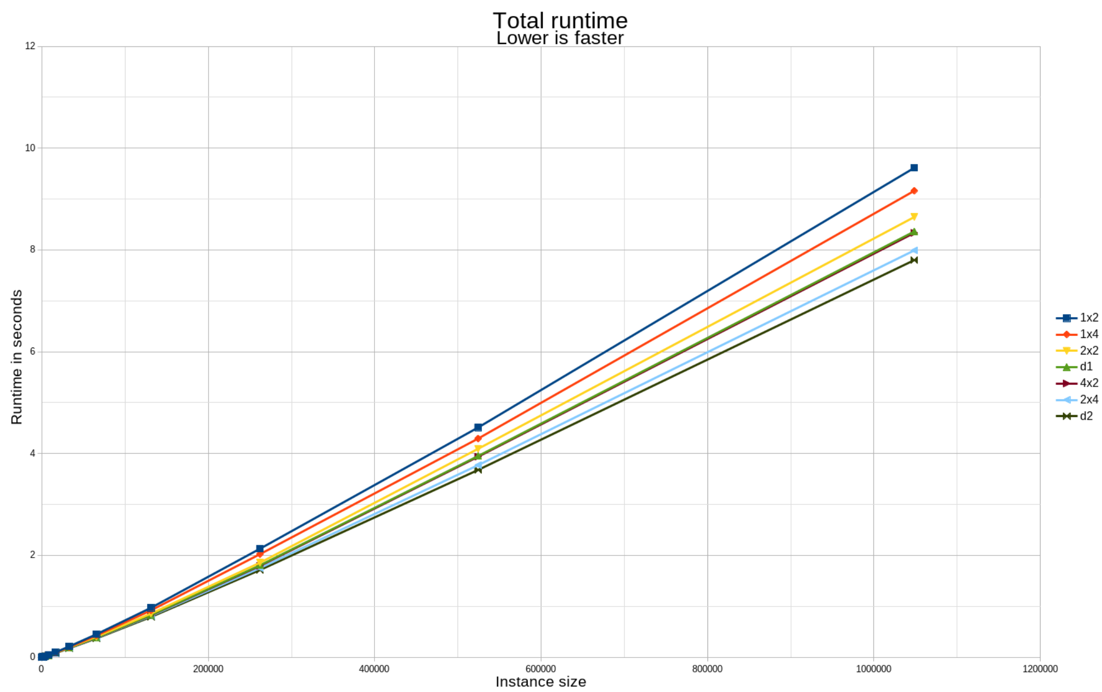


Figure 5.3: The runtime of the different setups for naive quicksort, optimization -*O3*.

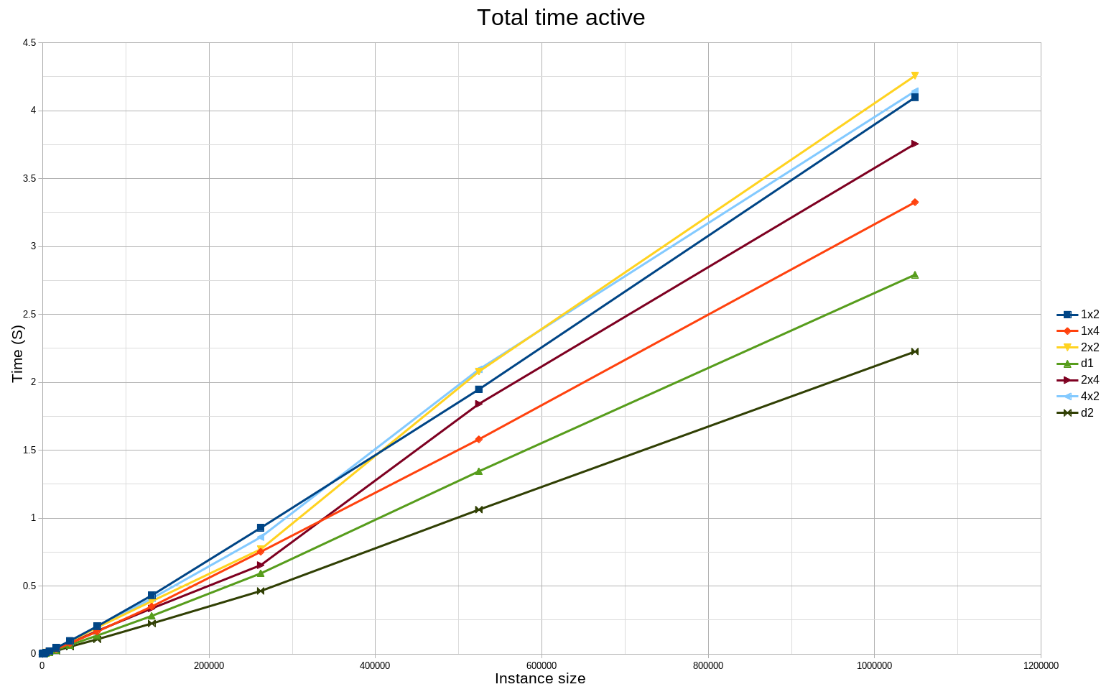


Figure 5.4: The active time of the different setups for naive quicksort, optimization -*O3*.

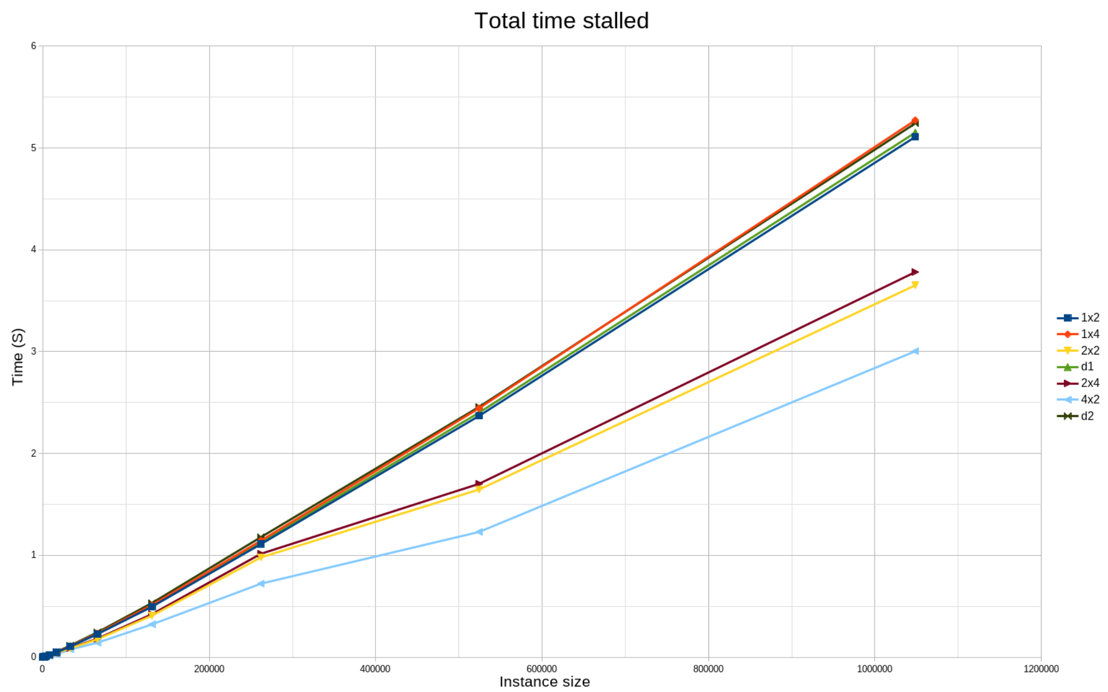


Figure 5.5: The stall time of the different setups for naive quicksort, optimization -*O3*.

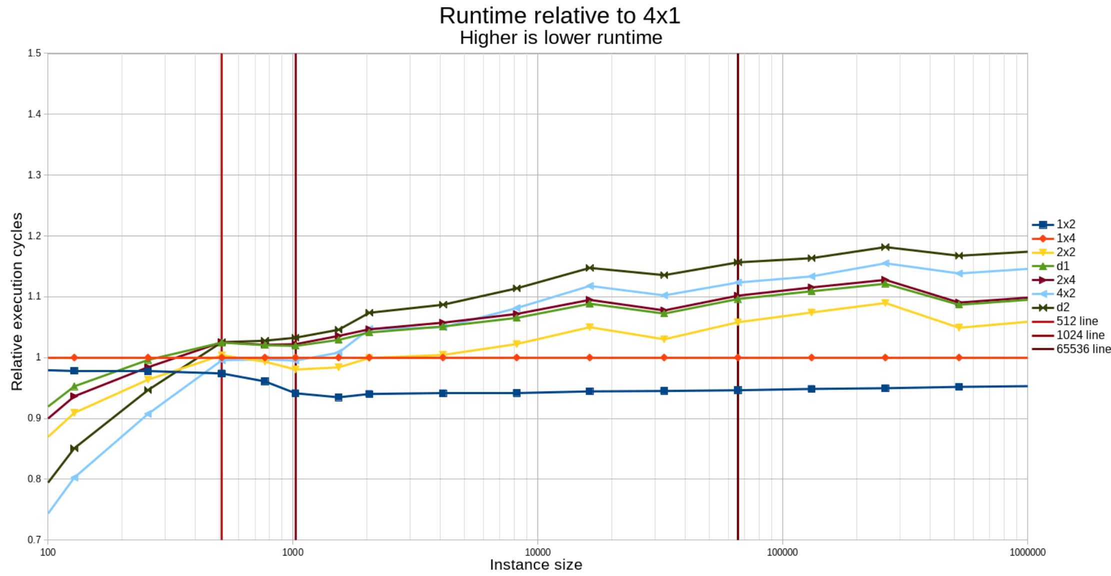


Figure 5.6: The relative performance of the different setups for naive quicksort, optimization *-O3*.

Tables 5.1 and 5.2 do not show some setups, these setups have a constant average syllables per word of two. We would like to note the following:

- Because the average syllables per word is firmly below 4 for any setup, an eight isse one context setup would not have been faster than the 1×4 setup (see Tables 5.1 and 5.2).

- The addition of atomic instructions does lead to a performance improvement: all setups utilizing more than one context outperform all setups utilizing only one context.

- The active time and stalled time graphs (Figures 5.4 and 5.5) show that the non-dynamic multi-context setups have more performance because they avoid stall cycles whilst the dynamic setups have more performance because they have less active time. Less active time implies that the work is better distributed over the lanes, but also seems to imply that there are more stall cycles.

- As expected, there is multithreading overhead (see Figure 5.6) which means that for small instances the multi-context setups perform worse then the single-context setups.

- For a setup with one ρ-VEX core d1 is the fastest setup, outperforming 2×2 by up to 4.7% and outperforming 1×4 by up to 12.1%.

- For a setup with two ρ-VEX cores d2 is the fastest setup, outperforming 2×4 by up to 3.9% and outperforming 1×4 by up to 18.2%.

5.3.2 The optimized implementation

The optimized implementation uses a task queue with a maximum size. Whenever there are two sublists and one of them is bigger than some threshold an attempt is made to put it into the task queue, this will succeed if the task queue is not full. If the task queue is full the issuee will handle the task. Initially there is one task and context 0 starts by executing this task and splitting away one subtask. Idle contexts poll the task queue waiting for either completion or a new subtask. Compared to the naive implementation the multithreading overhead increases and now also becomes a function of the instance size, since bigger instances generate more sublists in general.
The dynamic setup has the concept of a master context and a slave context. A master context will split if applicable when it places a new subtask into the task queue. A slave context will merge into the master context if it polls the workqueue and notices there are no current tasks to be executed.
The maximum amount of spaces in the workqueue and the minimum size of a list have different optimums for all setups, a set of tests is run to determine these optimums for every setup.
Note that the single context setups (1×2 and 1×4) have the same implementation as they did in the naive algorithm: they even lack the notion of the work queue.

Once again the specific compiler showed no performance improvement over the generic compiler so that the generic compiler is used for all tests. At first there did seem to be a difference between -*Os* and -*O3*, but this difference could be removed by retweaking the minimum list size and task queue size so that there is in fact no difference between the two optimization levels for this setup.

Size	**1x2**	**1x4**	**2x2**	**d1**	**2x4**	**4x2**	**d2**
64	2.00	2.29	2.00	2.28	2.28	2.00	2.28
128	2.00	2.27	2.00	2.27	2.27	2.00	2.27
256	2.00	2.25	2.00	2.25	2.25	2.00	2.25
512	2.00	2.24	2.00	2.24	2.24	2.00	2.24
768	2.00	2.23	2.00	2.23	2.23	2.00	2.23
1024	2.00	2.23	2.00	2.23	2.23	2.00	2.23
1536	2.00	2.22	2.00	2.22	2.22	2.00	2.22
2048	2.00	2.23	2.00	2.23	2.23	2.00	2.23
4096	2.00	2.22	2.00	2.08	2.16	2.00	2.11
8192	2.00	2.21	2.00	2.07	2.13	2.00	2.08
16284	2.00	2.21	2.00	2.04	2.16	2.00	2.07
32768	2.00	2.20	2.00	2.03	2.19	2.00	2.07
65536	2.00	2.20	2.00	2.02	2.19	2.00	2.06
131072	2.00	2.20	2.00	2.02	2.19	2.00	2.05
262144	2.00	2.20	2.00	2.01	2.19	2.00	2.04
524288	2.00	2.19	2.00	2.01	2.19	2.00	2.04
1048576	2.00	2.19	2.00	2.01	2.19	2.00	2.04

Table 5.3: Syllables per bundle for the optimized quicksort algorithm

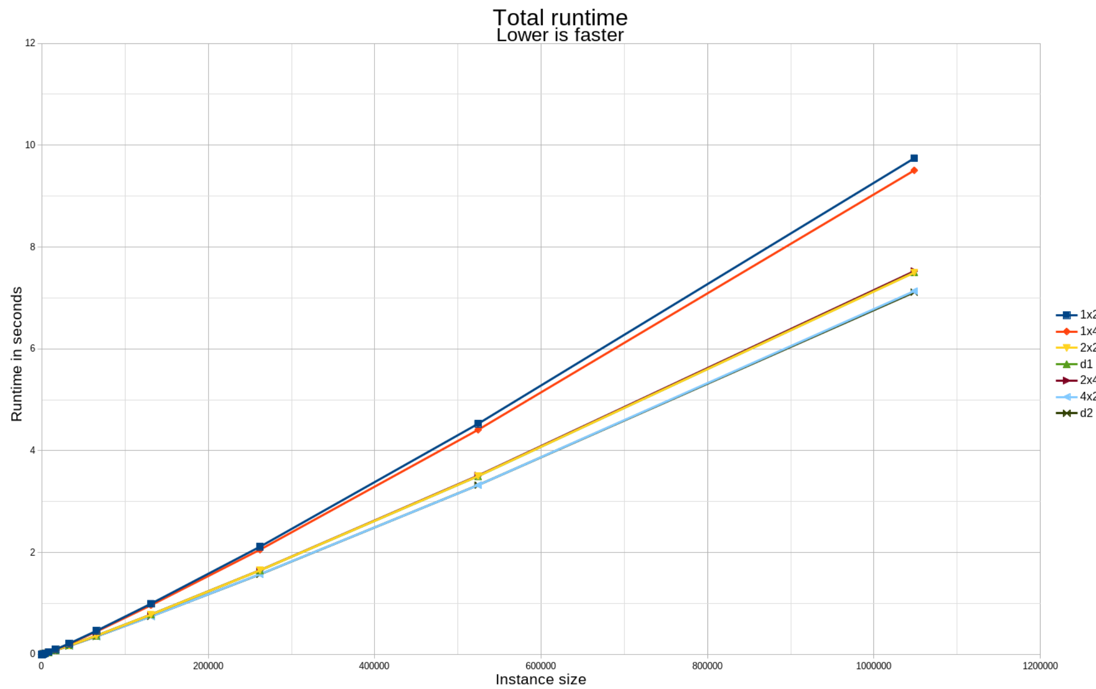


Figure 5.7: The runtime of the different setups for optimized quicksort.

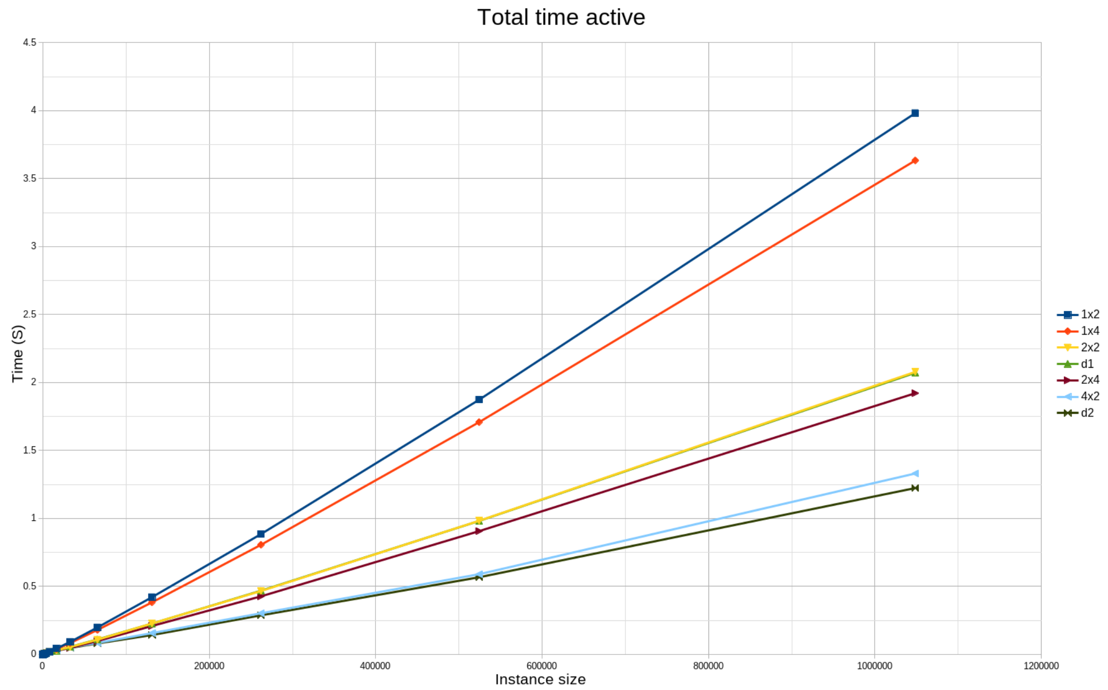


Figure 5.8: The active time of the different setups for optimized quicksort.

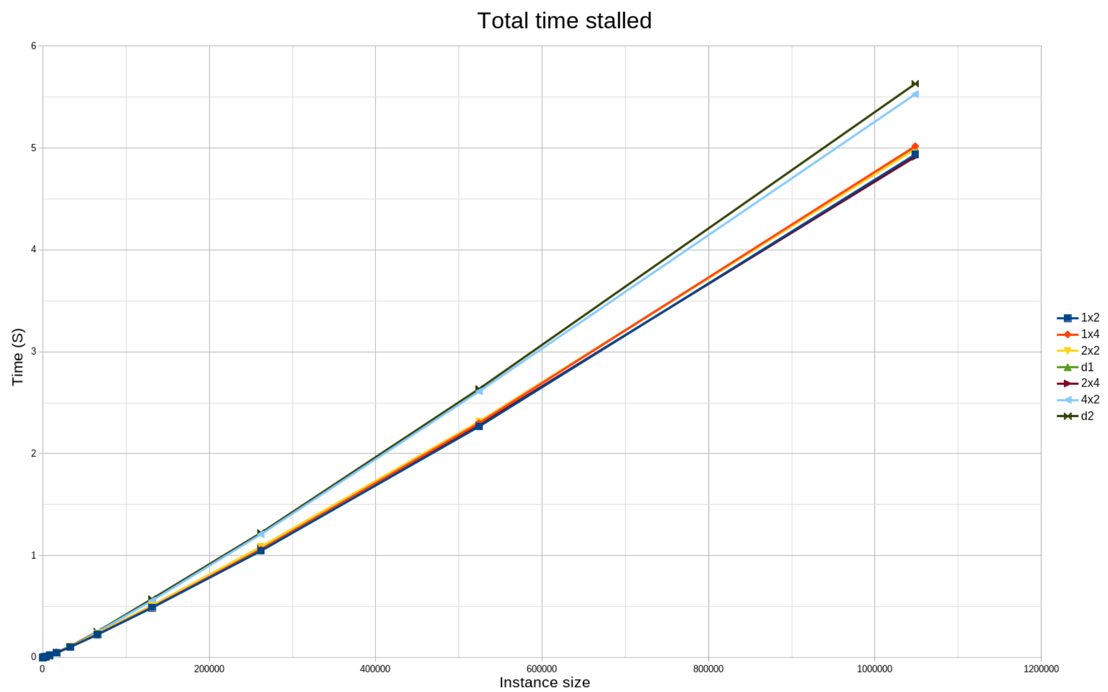


Figure 5.9: The stall time of the different setups for optimized quicksort.

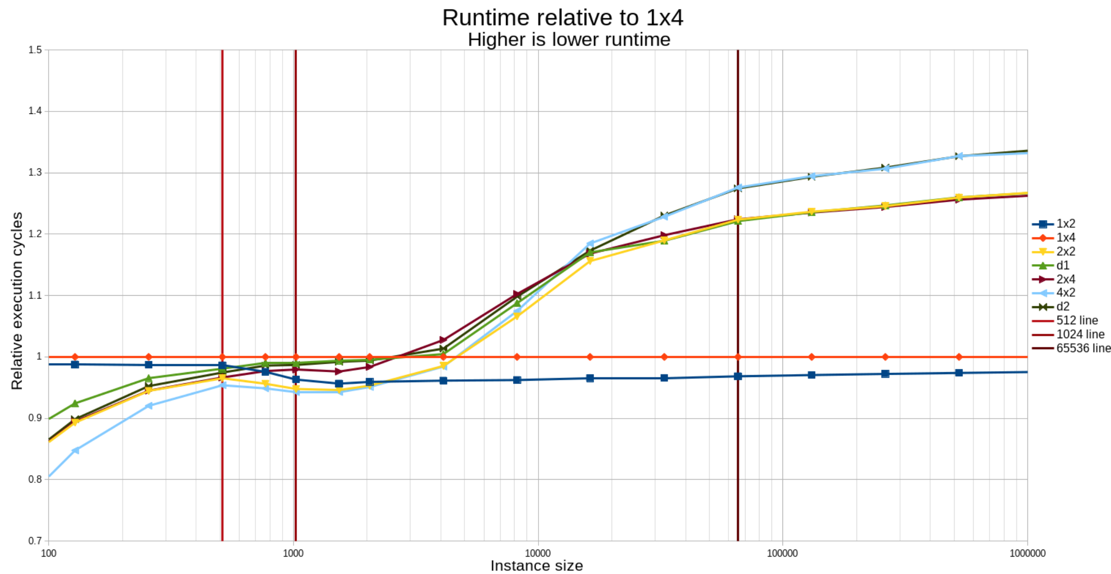


Figure 5.10: The relative performance of the different setups for optimized quicksort.

We would like to note the following:

- Because the average syllables per word is firmly below 4 for any setup, an eight issue one context setup would not have been faster than the 1×4 setup. See Table 5.3.
- The addition of atomic instructions does lead to a performance improvement: all setups utilizing more than one context outperform all setups utilizing only one context.
- Comparing Figure 5.7 with Figures 5.2 and 5.3 shows that this implementation is faster for all but the static single-context setups, starting from instance size 8192.
- The increase in runtime for all setups which use multiple contexts (2×2, 4×2, 2×4, $d1$ and $d2$) is because the active time is drastically reduced, as seen in Figure 5.8. This means that the available lanes are used more optimally.
- The stall time is relatively high for all setups which use multiple contexts. This shows that the memory bus is the bottleneck.
- If active time would have been equal to runtime, the $d2$ setup would have been up to about 10% faster than the 4×2 setup. Incidentally, the $d2$ setup has up to 2% more stall cycles and because stall cycles are so much more dominant, this means that the runtime of $d2$ is equal to the runtime of 4×2. This shows that there is no point in using the available lanes even more efficiently than 4×2, since the memory bus cannot supply more data.
- As expected, there is a larger multithreading overhead compared to the naive implementation (compare Figure 5.6 to Figure 5.10). This means that the instances need to be even larger before multi-context setups have an advantage.

- For a setup with one ρ-VEX core, d1 and 2×2 perform equal, outperforming 1×4 by up to 26.7%.
- For a setup with two ρ-VEX cores, d2 and 2×4 perform equal, outperforming 1×4 by up to 33.7%.

5.4 Matrix Multiplication

Take two matrices, A and B. If matrix C is defined as A multiplied by B then every element of C is determined in an operation which uses a row of A and a column of B. This section performs the naive matrix multiplication algorithm, which has a runtime complexity of $\mathcal{O}(n^3)$ and a space complexity of $\mathcal{O}(n)$.
A multithreaded implementation for this algorithm is quite trivial: since every element of C can be calculated independent of every other element of C and the inputs A and B do not change during execution, every element of C can be calculated as an independent subtask. The multithreaded implementations make use of this fact by spliting the output matrix C into n sections (where n is the amount of threads), each thread then handles one section. The size difference between the biggest and smallest section will be at most one so that they are the same size in a practical sense.
The multithreading overhead is very low: only at the very end the threads need to synchronize to make sure that the task is actually completed. Therefore, while the multithreading overhead is in principe lineair in the amount of threads, it is so small that it might as well have been a small constant.

Because the amount of independent subtasks is known beforehand and because every subtask has roughly the same size, it never makes sense to reconfigure. This means that runtime reconfiguration cannot add performance, so there is neither a d1 nor a d2 configuration in the matrix multiplication test. Another consequence is that the addition of atomic instructions was not required to implement this algorithm. Recall from Section 2.2.3 that the ρ-VEX without any modifications can wait for a certain variable to flip, since any change will eventually propagate to all cores. Therefore, a solution would be to have every context except for context 0 flip its own boolean on completion. Context 0 listens for these booleans and thus knows within finite time after completion that every thread has completed. This benchmark is very useful to see the possible performance improvement because of TLP, but it does not answer any of the research questions.

There is no advantage of specific compilers over the generic compiler so that the generic compiler is used and there is a performance difference between *-O3* and *-Os*. All matrices used in testing are square matrices. The implementation works with interleaved rows of matrix A, so that a n thread setup first handles the first n rows of matrix A and combines them with all columns of matrix B, etcetera. The disadvantage of this setup is that the amount of rows has to be a multiple of n or else one or more threads will have significantly less work than the others. This mistake occurred in testing, since multiples of 50 rows where tested and the 4×2 setup has four threads active.

Size	1x4	2x4
50	2.45	2.45
100	2.45	2.45
150	2.45	2.45
200	2.45	2.45
250	2.45	2.45
300	2.45	2.45
350	2.45	2.45
400	2.45	2.45
450	2.45	2.45
500	2.45	2.45

Table 5.4: Syllables per bundle for -*Os* matrix multiplication

Size	1x4	2x4
50	2.88	2.88
100	2.92	2.92
150	2.91	2.91
200	2.92	2.92
250	2.92	2.92
300	3.08	3.08
350	2.73	2.73
400	3.08	3.08
450	3.07	3.07
500	3.08	3.08

Table 5.5: Syllables per bundle for -*O3* matrix multiplication

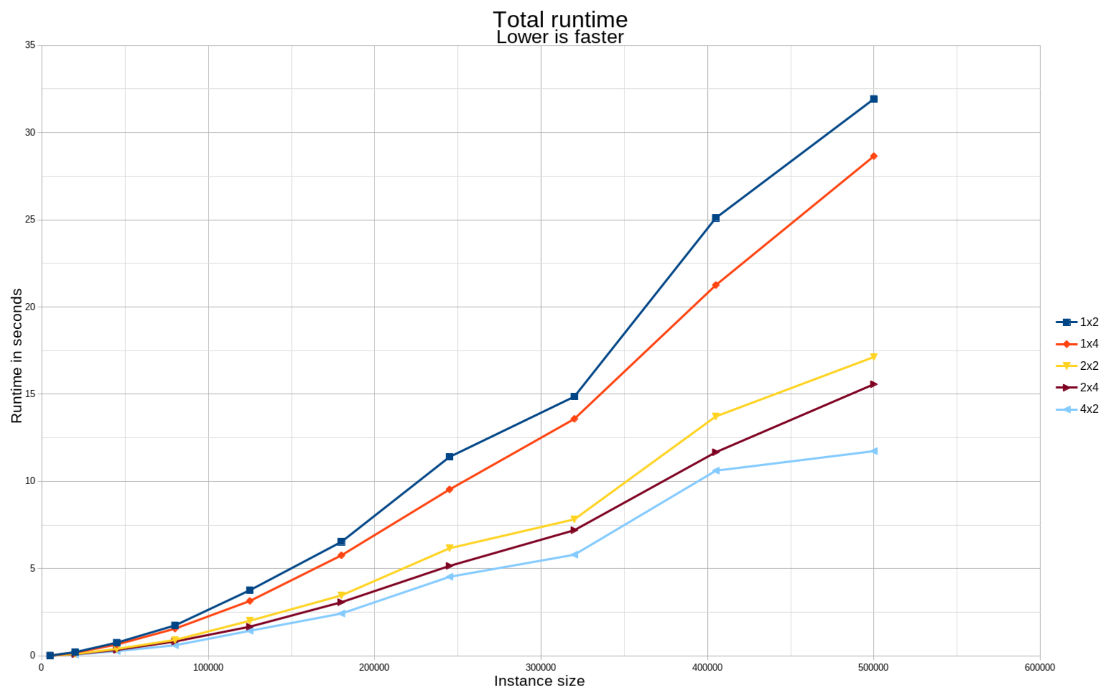


Figure 5.11: The runtime of the different setups for matrix multiplication, optimization -*Os*.

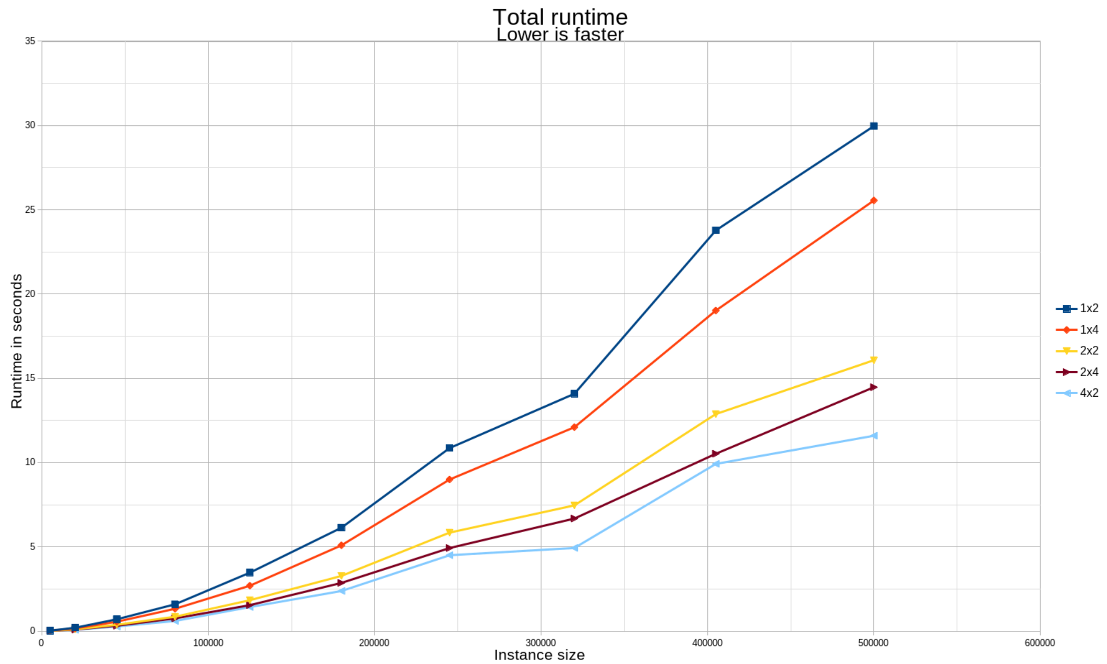


Figure 5.12: The runtime of the different setups for matrix multiplication, optimization *-O3*.

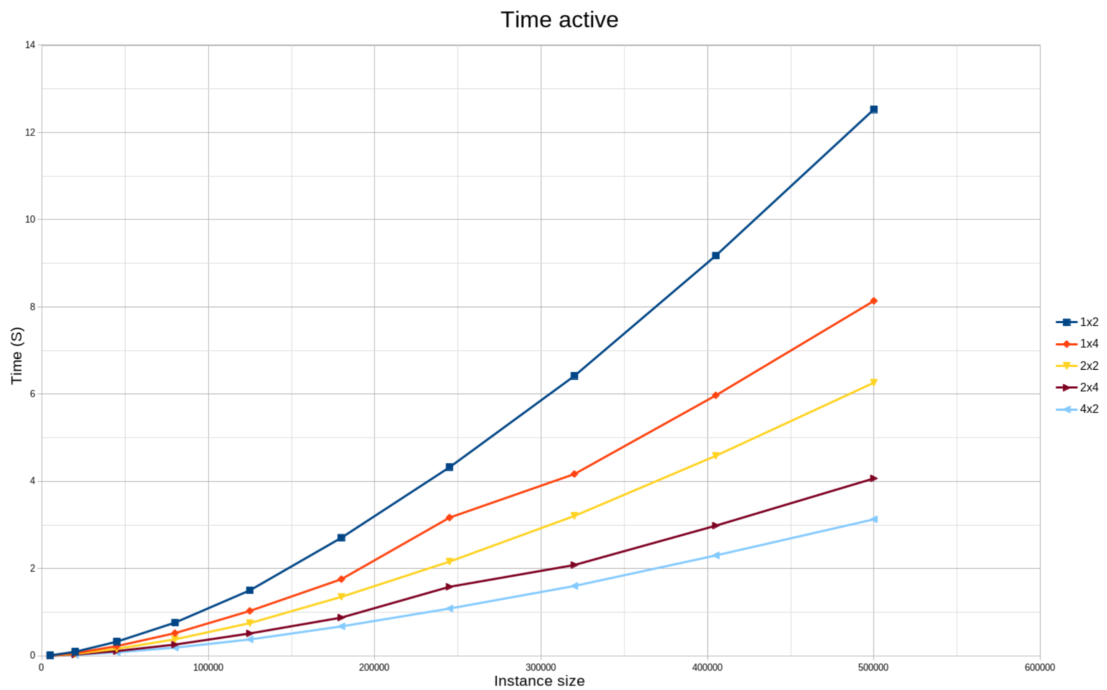


Figure 5.13: The active time of the different setups for matrix multiplication, optimization *-O3*.

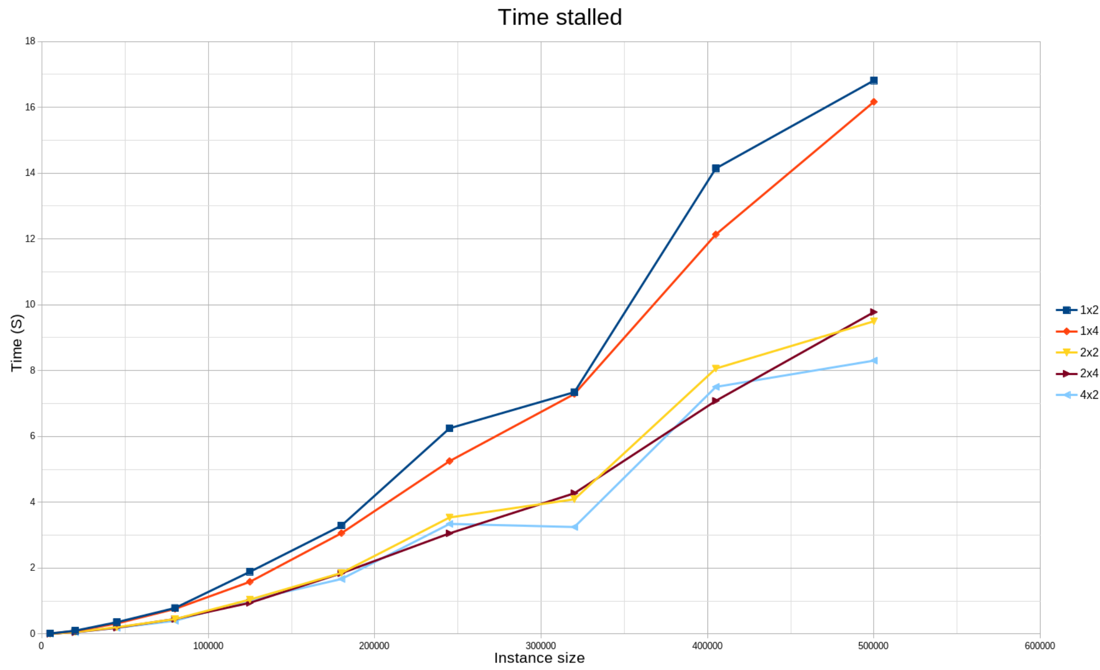


Figure 5.14: The stall time of the different setups for matrix multiplication, optimization *-O3*.

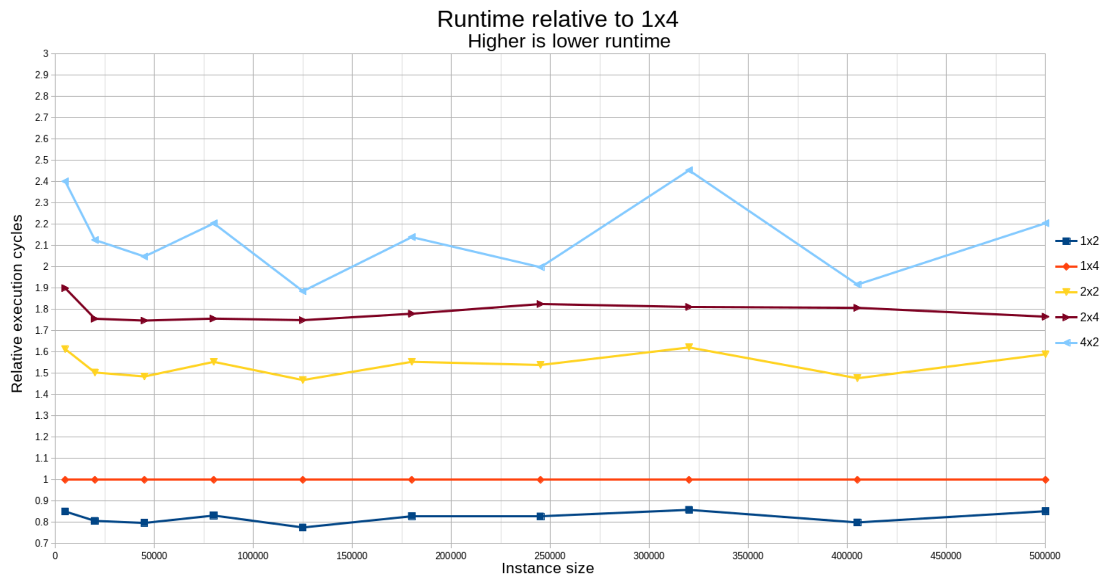


Figure 5.15: The relative performance of the different setups for matrix multiplication, optimization *-O3*.

Tables 5.4 and 5.5 do not show some setups, these setups have a constant average syllables per word of two. We would like to note the following:

- Because the average syllables per word is firmly below 4 for any setup, an eight isse one context setup would not have been faster than the 1×4 setup (see tables 5.4 and 5.5).
- The stall time is very high in general, being at least 43.6% of total runtime. The 4×2 setup has the peak relative stall time: it stalls 75.6% of total runtime for instancesize = 405000.
- The 2×2 setup spends a lower percentage of its runtime stalling and has a lower active time compared to 1×4. This means that the 2×2 setup utilizes both the available lanes and the memory hierarchy more efficient.
- The multithreading overhead is very small, as is shown in Figure 5.15. Because the multithreading overhead is so small, the active time of 2×2 is almost exactly half the active time of 1×2. The same goes when comparing 2×4 to 1×4 and comparing 4×2 to 2×2.
- For a setup with one ρ-VEX core, 2×2 is the fastest setup, outperforming 1×4 by up to 62.1%.
- For a setup with two ρ-VEX cores, 4×2 is the fastest setup, outperforming 2×4 by up to 35.4% and outperforming 1×4 with up to 145.3%.

5.5 Conclusion

This chapter showed first and foremost that the addition of atomic instructions can lead to a performance increase of up to 33.7%. Additionally, a performance increase of 145.3% is possible, but this increase does not require any changes to the ρ-VEX core. Both quicksort and matrix multiplication were tested and in both cases the addition of thread level parallelism (TLP) allows for a more efficient use of the available resources on the ρ-VEX, and in some cases, more efficient usage of the memory hierarchy. This chapter also showed that after these changes it no longer pays off to focus even more on active cycles but instead the focus should shift to the reduction of stall time. The results show that the performance can increase a lot more if the amount of stall cycles is reduced. This answers the research question *What is the impact of inter-thread communication on performance on the ρ-VEX?*

The secondary goal was to show what the combination of runtime reconfigurable and thread level parallelism can achieve. The naive implementation of quicksort does show that the dynamic setups can outperform any static setup, but this implementation is syntetic and not optimal. The optimized quicksort implementation showed no difference between a static approach and the dynamic approach when considering total runtime, but did show a difference when only considering active time. Therefore, this chapter did imply that the runtime reconfigurable ρ-VEX can outperform any static ρ-VEX given the correct circumstances, but failed to actually show it.

6 Model

This chapter will address with the final research question: *What does the optimally performing program for the ρ-VEX look like and how much can runtime reconfigurability improve performance?* A model of the ρ-VEX will be created and optimized so that the characteristics of the ideal program can be investigated.
Section 6.1 will introduce some definitions and the assumptions surrounding the model. Sections 6.2 and 6.3 will introduce the models, determine their optima and discuss the viability of the results.

6.1 Definitions and assumptions

The final goal of this thesis is to get some information on how to exploit the properties of the ρ-VEX so that performance is optimized. To get some insight, a model will be created and optimized. The goal is to find the values of multiple variables so that the reconfigurable ρ-VEX is maximally faster than any static setup, or in other words: to prove an upper bound. As explained in Section 2.4 the idea behind reconfiguration is that a program can be split into different sections and for each section there is a different static ρ-VEX configuration which is optimal[33]. Therefore, the model will model a program as a set of sections, each of which has an optimal static ρ-VEX configuration. The variables include fraction of runtime that is sequential (cannot be multithreaded), fraction of runtime that can be parallelized on two cores, fraction of runtime that can be parallelized on four cores and syllables per word.
This model will be subject to the following assumptions:

1. Every setup executes the same amount of syllables.
2. Active time equals runtime.
3. Reconfiguration takes no time.

The first assumption ignores multithreading overhead. Results in Chapter 5 have shown that multithreading overhead as a fraction of total execution time goes to zero as the instance grows and therefore this is not included in the model. The amount of times reconfiguration happens shows so far to be mostly ependent on the implementation of an algorithm and not as much on instance size. Therefore, whatever the number is, it becomes a diminishing constant as instance size grows and thus can be ignored by the model, explaining assumption three.

Assumption two removes stall cycles and pipeline flush cycles from the total. Pipeline flush cycles are a small part of total runtime according to Chapter 5 and thus can be ignored without much impact. Stall cycles are a large part, but also misleading. One of the results of Chapter 5 is that the fraction of runtime that is stall cycles grows very fast, which could point to problems in the memory hierarchy. Indeed, Section 2.2 explains that the ρ-VEX bus is not performing well by modern standards. This results in the expectation that a model including the current ρ-VEX bus will not yield useful information. Therefore, an ideal memory hierarchy is assumed, one which never stalls.
A disadvantage of this approach is that the model now becomes at best a loose upper bound and at worst only an indication. Results from Chapter 5 showed that setups with more active contexts tend to have more stall cycles, which means that dynamic setups, setups with few active contexts and setups with many active contexts are each affected differently by the memory hierarchy.

The modeling of syllables per word is the next issue. The execution of an algorithm usually involves doing a simple task many times until the problem is solved and the program can end. In that case it makes sense to say that the syllables per word is a

constant, which remains the same during the entire runtime. But not all programs run only one algorithm, so that the syllables per word may vary from program section to program section. Both of these situations will be investigated by creating two different models, one in which the program is split in multiple sections and each section has its own syllables per word and one in which the syllables per word is a constant troughout the entire program.

The next two sections will model two physical ρ-VEX processors, one with four lanes and two contexts and one with eight lanes and four contexts. As in Chapter 5, every lanegroup has a size of two so that every context can have a multiple of two lanes attached to it. This choice is made to closely reflect the setup of Chapter 5. The first model directly relates to the 2×2, 1×4 and d2 setups of Chapter 5. The second model relates to the maximum size ρ-VEX and somewhat relates to the 2×4, 4×2 and d2 setups of Chapter 5, altough the model also allows for 1×8.
Finally, defining some notation used troughout this section: let 1 be the amount of syllables that is required to execute the hypothetical program. Note that any number could be used, it has to be some constant. Furthermore, let $s \leq 1$ denote the set of syllables that cannot be run simultaneous to any other set of syllables, or in other words the sequential part of the program. Let $p_n \leq 1$ denote the set of syllables that cannot be run simultaneous with more than $n - 1$ other sets of syllables, or in other words, the set of syllable that can efficiently be run on at most n threads. For every model $s + p_a + p_b + ... + p_z = 1$ needs to hold, or in other words, every syllable needs to be executed. Finally, let I denote the syllables per word and thus I_x the syllables per word for section x.

6.2 Four issue, two context

The first hardware setup that will be modeled is the four lanes, two context ρ-VEX core. This core can have the following setups:

- 1×4
- 2×2
- d1

Note that 1×2 is ignored because this model is about performance and 1×4 always performs equal or better than the 1×2 setup. Also note that d1 is at any moment either a 1×4 or a 2×2 setup. Any program running consists of a sequential part s and a non-sequential part p. All p_n are summed up in p. Since $p + s = 1$, we can rewrite p to $1 - s$. We start with a program which has the same syllables per word for all parts of the program, I. Do note that because there are four lanes, $I \leq 4$.

$$
\begin{array}{ll}
\text{Maximize} & \min(E_{1\times4}, E_{2\times2}) - E_{d1} \\
\text{Where} & \\
& E_{1\times4} = \frac{1}{I} \\
& E_{2\times2} = \frac{s}{\min(2,I)} + \frac{1-s}{\min(4,2I)} \\
& E_{d1} = \frac{s}{I} + \frac{1-s}{4} \\
\text{Subject to} & \\
& 1 \leq I \leq 4 \\
& 0 \leq s \leq 1
\end{array}
\tag{6.1}
$$

Solving Equation 6.1 gives $I = \frac{8}{3} = 2.\overline{6}$ and $s = \frac{1}{2}$, at which point $E_{1\times4} = E_{2\times2} = \frac{3}{8} = 0.375$ and $E_{d1} = \frac{5}{16} = 0.3125$. The speedup of E_{d1} is 20% over the fastest static setup.
I is remarkebly close to some values reached by results from Chapter 5, specifically those shown in Table 5.2. It should be clear from the results in Section 5.3.1 that the fraction of syllables in s and p is distributed differently and thus that the final speedup of $d1$ over 2×2 is only a few percent.
Twenty percent is also not a very large speedup, considering that Chapter 5 shows that 2×2 can outperform 4×1 with over 25%.

Assume that the program still consist of two different sections, one strictly sequential and one that can be multithreaded. Assume now that for each part there is a different constant syllables per word, I_s for the sequential part and I_p for the parallel part.

$$
\begin{array}{ll}
\text{Maximize} & \min(E_{1\times4}, E_{2\times2}) - E_{d1} \\
\text{Where} & \\
& E_{1\times4} = \frac{s}{I_s} + \frac{1-s}{I_p} \\
& E_{2\times2} = \frac{s}{\min(2,I_s)} + \frac{1-s}{\min(4,2I_p)} \\
& E_{d1} = \frac{s}{I_s} + \frac{p}{\min(4,2I_p)} \\
\text{Subject to} & \\
& 1 \leq I_s \leq 4 \\
& 1 \leq I_p \leq 4 \\
& 0 \leq s \leq 1
\end{array}
\tag{6.2}
$$

Solving Equation 6.2 gives $s = \frac{1}{2}$, $I_s = 4$ and $I_p = 2$. At those values $E_{1\times4} = E_{2\times2} = \frac{3}{8} = 0.375$ and $E_{d1} = \frac{1}{4} = 0.25$. Therefore, E_{d1} is at most 50% faster than the fastest setup.
The strategy of the optimizer is to make sure that every lane is always active for the $d1$ setup and to penalize both static programs by making sure that they can only use half their lanes for 50% of the syllables. The requirement that $I_p = 2$ is easy to reach, although many programs tend to overshoot this. Having the syllables per word reach four is a somewhat rare sight[48].

6.3 Eight issue, four context

An eight issue four context ρ-VEX can have the following setups

- 1×8
- 2×4
- 4×2
- d2

Once again, all setups that can be created but are outperformed by some other setup are ignored. Since this core layout has eight lanes, $I \leq 8$

Let the syllables consist of set s, the set of syllables in the sequential part of execution, set p_2 which is the set of syllables in the part of the execution that can efficiently run on 2 threads and set p_4, which is the part of the execution that can efficiently run on 4 threads. All other p_n can be split into these three sets.

$$\begin{array}{ll}
\text{Maximize} & \min(E_{1\times 8}, E_{2\times 4}, E_{4\times 2}) - E_{d2} \\
\text{Where} & \\
& E_{1\times 8} = \frac{s+p_2+p_4}{I} = \frac{1}{I} \\
& E_{2\times 4} = \frac{s}{\min(4,I)} + \frac{p_2+p_4}{\min(8,2I)} \\
& E_{4\times 2} = \frac{s}{\min(2,I)} + \frac{p_2}{\min(4,2I)} + \frac{p_4}{\min(8,4I)} \\
& E_{d2} = \frac{s}{I} + \frac{p_2}{\min(8,2I)} + \frac{p_4}{\min(8,4I)} \\
\text{Subject to} & \\
& 1 \leq I \leq 8 \\
& s + p_2 + p_4 = 1 \\
& s, p_2, p_4 \geq 0
\end{array} \tag{6.3}$$

Solving Equation 6.3 gives $I = \frac{16}{3} = 5.\overline{3}$, $s = \frac{1}{2}$ and $p_2 + p_4 = \frac{1}{2}$ or in other words: it does not matter how the parallelization part is distributed. Filling in these numbers gives $E_{1\times 8} = E_{2\times 4} = \frac{3}{16} = 0.1875$, $E_{4\times 2}$ ranges from $\frac{3}{8} = 0.375$ to $\frac{5}{16} = 0.3125$. $E_{d2} = \frac{5}{32} = 0.15625$, which means that E_{d2} is 20% faster than the fastest static setup. $5.\overline{3}$ is a very rare sight, making it improbable that there exists a real world program that can reach the 20%.

Assume that the program still consists of the same sets of syllables, but the syllables per word is different for every set. This introduces the variables I_s, I_{p2} and I_{p4}.

$$\begin{array}{ll}
\text{Maximize} & \min(E_{1\times 8}, E_{2\times 4}, E_{4\times 2}) - E_{d2} \\
\text{Where} & \\
& E_{1\times 8} = \frac{s}{I_s} + \frac{p_2}{I_{p2}} + \frac{p_4}{I_{p4}} \\
& E_{2\times 4} = \frac{s}{\min(4,I_s)} + \frac{p_2}{\min(8,2I_{p2})} + \frac{p_4}{\min(8,2I_{p4})} \\
& E_{4\times 2} = \frac{s}{\min(2,I_s)} + \frac{p_2}{\min(4,2I_{p2})} + \frac{p_4}{\min(8,4I_{p4})} \\
& E_{d2} = \frac{s}{I_s} + \frac{p_2}{\min(8,2I_{p2})} + \frac{p_4}{\min(8,4I_{p4})} \\
\text{Subject to} & \\
& 1 \leq I_s, I_{p2}, I_{p4} \leq 8 \\
& s + p_2 + p_4 = 1 \\
& s, p_2, p_4 \geq 0
\end{array} \tag{6.4}$$

Solving Equation 6.4 gives $s = \frac{26}{25} = 0.64$, $p_2 = 0$, $p_4 = \frac{9}{25} = 0.36$, $I_s = 8$, $I_{p4} = 2$ and I_{p2} is free because p_2 is zero. Filling in these numbers gives $E_{1\times8} = \frac{13}{50} = 0.26$, $E_{2\times4} = \frac{1}{4} = 0.25$, $E_{4\times2} = \frac{73}{200} = 0.365$ and $E_{d2} = \frac{1}{8} = 0.125$, which means that E_{d2} is 100% faster than the fastest static setup.

6.4 Conclusion

The previous sections showed an idle speedup ranging from 20% to 50% for a ρ-VEX with four lanes and two contexts and an ideal speedup ranging from 20% to 100% for a ρ-VEX with eight lanes and four contexts. We would like to say that these numbers are loose upper bounds, but that is unsure, since the models ignore stall cycles.

If the models provide a loose upper bound, the research question *What does the optimally performing program for the ρ-VEX look like and how much can runtime reconfigurability improve performance?* has been answered by this chapter. Work on the memory model and memory hierarchy is required to be able to give a more reliable answer to the research question.

7 Conclusions and future work

Section 1.4 defined the problem statement and the goal for this project. This chapter will discuss the results of the project and evaluate whether or not all goals were reached and all research questions were answered.
Section 7.1 will summarize the thesis. Subsequently, Section 7.2 will explain the main contributions of this thesis and reflect on the goals and research questions. The chapter finalizes with recommendations for future work and possible future improvements in Section 7.3.

7.1 Summary

Chapter 2 first introduces a common problem in inter-thread communication: race conditions. Two possibile solutions to this problem are subsequently presented, critical sections and wait-free synchronization. The chapter continues with exploring possible hardware adaptations and software implementations, all of which can be used to implement either solution to race conditions.
The chapter then introduces the ρ-VEX, a polymorphic VLIW processor which can reconfigure during runtime. This processor can reassign lanes during runtime so that it can, for example, transform from a single core to a dual core whilst the program is running. A proof is also provided which shows that the ρ-VEX in its current state does not allow for efficient inter-thread communication, since no efficient solution to the problem of race conditions can be implemented. It is even argued that the ρ-VEX cannot support inter-thread communication at all, and a weaker form where every thread needs to call some function every so often to make sure that the system keeps making progress is supported, but inefficiently.
The chapter then introduces the hardware which is used troughout the project, the PYNQ-Z1 development board, and finalizes with defining the notion of performance and providing explanations for instruction level parallelism (ILP) and thread level parallelism (TLP).

Chapter 3 is the first of two implementation chapters and describes the development process and performance and area tests of an interconnect. The goal of this chapter is to connect the ρ-VEX to the DDR3 RAM which is available on the PYNQ-Z1 board. This RAM is already physically connected to the chip on which the ρ-VEX is loaded and also already in use by Ubuntu which also runs on the PYNQ-Z1 board, alongside the ρ-VEX. A driver called the ρ-VEX driver is created to reserve some memory for use by the ρ-VEX and an interconnect with an internal cache is created which connects the ρ-VEX bus, the bus used to move data within the ρ-VEX to the AXI4 bus, a bus which is interfaced with the DDR3 RAM. The cache internal to the interconnect is configurable and the chapter finalizes with testing the impact of the variables involved with configuring this cache on both area and performance.
Chapter 4 is the second of two implementation chapters and describes the changes made so that the ρ-VEX can properly support inter-thread communication. Two new instructions, load-linked and store-conditional, are added to the instruction set and a synchronization unit is placed in the memory hierarchy. This adds a new feature, the atomic sequence, which is subsequently exploited by the newly implemented semaphore. This is a software construct which can be used to implement critical sections.

Inter-thread communication allows different contexts to work together, which allows for TLP. Chapter 5 investigates the impact of TLP on performance, utilizing two algorithms: quicksort and matrix multiplication. Both algorithms benefit from TLP, showing performance improvements of up to 33.7%. Moreover, Chapter 5 shows that there are larger performance gains possible, but the current memory hierarchy does not allow for a larger performance gain. Moreover, Chapter 5 also implied that the runtime reconfig-

urable ρ-VEX could outperform any static setup given the correct circumstances. The results did show that the reconfigurable ρ-VEX needed less time active than any static setup, but this did not translate in less runtime. Chapter 6 introduces a model of the physical ρ-VEX to investigate what the possible performance improvement of the combination of runtime reconfigurable and TLP is. This model only focusses on active time, ignoring the fact that a processor also spends time waiting for the memory hierarchy. According to this model, a dynamic ρ-VEX is anywhere from 20% to 100% faster than any static ρ-VEX setup. It is unclear whether or not these numbers represent loose upper bounds, more investigation into the role of the memory hierarchy is required.

7.2 Main contributions

Section 1.4 introduced the research questions and the goals, this section addresses all of these again to conclude if they were fulfilled. Furthermore, this section lists the main contributions of the project described by this thesis.

How can we create a way for the ρ-VEX to get access to the on-board RAM of the PYNQ board?

This question is answered in Chapter 3. There are multiple ways, the one chosen by this project is to write a driver which reserves some RAM and build an interconnect which allows the ρ-VEX to access the reserved RAM.

How to add possibilities for inter-thread communication to the ρ-VEX?

This research question is partially answered in Chapter 2 which provides information on how inter-thread communication can be made possible. The process of adding inter-thread communication to the ρ-VEX is described in Chapter 4, which describes first the implementation of the load-linked and store-conditional instructions, then the implementation of semaphores on top of these instructions and finally how the semaphore can be used to implement a critical section, which allows inter-thread communication.

What is the impact of inter-thread communication on performance on the ρ-VEX?

This question is answered in Chapter 5. The impact is positive, showing a performance improvement of up to 30%.

What does the optimally performing program for the ρ-VEX look like and how much can runtime reconfigurability improve performance?

This question remains unanswered, although both Chapters 5 and 6 provided some insight in what the answer could be.

- *To create a ρ-VEX platform on the PYNQ which can use the on-board RAM and has inter-thread communication implemented.*
  This goal has been achieved, foundations are described in Chapter 3 and Chapter 4 and the finalization of the goal is described in Section 5.2.
- *To use this platform to run some benchmarks which determine the performance improvement due to inter-thread communication alone and determine whether or not the runtime configurable property of the ρ-VEX can be used to improve performance further.*
  This goal is achieved in Chapter 5, which showed the performance improvement due to inter-thread communication to be up to 30% and showed that runtime reconfiguration in combination with inter-thread communication could improve performance even further, although the test that shows this is somewhat syntetic.
- *To model some properties of the ρ-VEX to determine the optimal usage of the ρ-VEX in terms of performance.*
  The modelling is described in Chapter 6, but the results were not as conclusive as was hoped. Further research is required to determine the optimal usage of the ρ-VEX in terms of performance.

The main contribuntions of this project:

- Added a bridge so that the ρ-VEX bus can be used as a master on the AXI4 bus.
- Created a ρ-VEX driver, library and Linux program so that the ρ-VEX in combination with the bridge can use large amounts of memory on the PYNQ.
- Implemented load-linked/store-conditional, a set of instructions which allows for efficient multithreading on the ρ-VEX, into the toolchain.
- Added the synchronization unit and modified the ρ-VEX to support load-linked/store-conditional.
- Added support to load-linked/store-conditional to the ρ-VEX simulator.
- Benchmarked the addition of load-linked/store-conditional to measure the performance improvement.
- Created a performance model to predict the effects of runtime reconfigurability in combination with efficient multithreading.
- Numerous fixes to the ρ-VEX toolchain.

7.3 Future work and future improvements

This project has been a stepping stone in improving the performance of the ρ-VEX and reasoning about the upper limits of the performance of the ρ-VEX. Future work could take both a step further by starting work on the memory hierarchy.
The following work is proposed for the level 1 cache:

- Section 5.2 has shown that the L1 cache could be revisited so that its size could be increased more easily.
- Section 2.2.2 explained that the ρ-VEX can currently only issue one memory instruction per context per cycle, which is partially because of the properties of the L1 cache. Removing this barrier could improve instruction level parallelism (ILP) scheduling so that the average syllables per word could rise and performance could improve.

The ρ-VEX bus can also be improved:

- Section 3.2 explained how the bus can be put in an illegal state so that master nor slave know how to continue. This can be avoided, as shown by the ρ-VEX bus. Fixing this issue can prvent bugs and make implementation easier.
- Section 2.2.2 explained and Chapter 5 showed that the ρ-VEX bus is a bottleneck for performance. We suggest to investigate how the performance can be improved and implement those changes.

Section 3.8 explained possible future work on the interconnect:

- Revisit the archtitecture and reimplement the bus so that the cache becomes a separate entity. This can reduce the time of a cache-hitting write by one cycle.
- Add the possibility to exclude addresses from the cache. This allows the ρ-VEX to access peripherals on the AXI4 bus instead of only memory-like devices.

Finally, the model proposed in Chapter 6 can be expanded and put to use. The following work is proposed on expanding, validating and using the model:

- Investigate whether or not the models proposed in Chapter 6 is a loose upper bound.
- Model the memory and add it to the models in Chapter 6.
- Formulate a tight upper bound.
- Write testprograms which replicate the numbers from the models and test whether or not they behave as predicted.
- Create real-world programs which do solve a problem and effectively use the ρ-VEX's properties.

Bibliography

[1] J. van Straten, "A dynamically reconfigurable vliw processor and cache design with precise trap and debug support," Master's thesis, Delft University of Technology, the Netherlands, 2016.

[2] H. Dietz, "Smp linux," https://www.tldp.org/HOWTO/Parallel-Processing-HOWTO-2.html, accessed: 2018-03-15.

[3] L. Lamport, "The mutual exclusion problem: Part i: a theory of interprocess communication," *J. ACM*, vol. 33, no. 2, pp. 313–326, Apr. 1986.

[4] E. W. Dijkstra, "Solution of a problem in concurrent programming control," *Commun. ACM*, vol. 8, no. 9, pp. 569–, Sep. 1965.

[5] D. E. Knuth, "Additional comments on a problem in concurrent programming control," *Commun. ACM*, vol. 9, no. 5, pp. 321–322, May 1966.

[6] M. Herlihy, "Wait-free synchronization," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 1, pp. 124–149, Jan. 1991.

[7] A. Alexandrescu, "Lock-free data structures," *C/C++ users Journal*, Oct. 2004, revised in 2007.

[8] M. Raynal, *Concurrent Programming: Algorithms, Principles, and Foundations.* Springer Publishing Company, Incorporated, 2012.

[9] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design, Fourth Edition*, 4th ed. Morgan Kaufmann Publishers Inc., 2011.

[10] N. G. de Bruijn, "Additional comments on a problem in concurrent programming control," *Commun. ACM*, vol. 10, no. 3, pp. 137–138, Mar. 1967.

[11] M. A. Eisenberg and M. R. McGuire, "Further comments on dijkstra's concurrent programming control problem," *Commun. ACM*, vol. 15, no. 11, pp. 999–, Nov. 1972.

[12] L. Lamport, "A new solution of dijkstra's concurrent programming problem," *Commun. ACM*, vol. 17, no. 8, pp. 453–455, Aug. 1974.

[13] W. T. Comfort, "A computing system design for user service," in *Proceedings of the November 30–December 1, 1965, Fall Joint Computer Conference, Part I*, ser. AFIPS '65 (Fall, part I). New York, NY, USA: ACM, 1965, pp. 619–626.

[14] J. L. Hennessy and D. A. Patterson, *Computer Architecture, A Quantative Approach.* Morgan Kaufmann, 2012.

[15] *IBM System/370 Principles of Operation*, IBM, september 1975, fourth Edition.

[16] D. Dechev, P. Pirkelbauer, and B. Stroustrup, "Understanding and effectively preventing the aba problem in descriptor-based lock-free designs." in *ISORC*. IEEE Computer Society, 2010, pp. 185–192.

[17] E. H. Jensen, G. W. Hagensen, and J. M. Broughton, "A new approach to exclusive data access in shared memory multiprocessors," no. Technical Report UCRL-97663, Nov 1987.

[18] J. H. Anderson and M. Moir, "Universal constructions for multi-object operations," in *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '95. New York, NY, USA: ACM, 1995, pp. 184–193.

[19] D. Talla, L. K. John, V. Lapinskii, and B. L. Evans, "Evaluating signal processing and multimedia applications on simd, vliw and superscalar architectures," in *Proceedings 2000 International Conference on Computer Design*, 2000, pp. 163–172.

[20] C. Kozyrakis and D. Patterson, "Vector vs. superscalar and vliw architectures for embedded multimedia benchmarks," in *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002. (MICRO-35). Proceedings.*, 2002, pp. 283–293.

[21] *pVex User Manual*, TU Delft, 2017, version a7549bc.

[22] J. Hoozemans, J. van Straten, and S. Wong, "Using a polymorphic vliw processor to improve schedulability and performance for mixed-criticality systems," in *Proc. 23rd IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Hsinchu, Taiwan, August 2017.

[23] A. Brandon and S. Wong, "Support for dynamic issue width in vliw processors using generic binaries," in *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2013, pp. 827–832.

[24] S. van Voorst, "Samsung produceert asics voor cryptomining," https://tweakers.net/nieuws/134681/samsung-produceert-asics-voor-cryptomining.html, accessed: 15-03-2018.

[25] H. P. Hofstee, "Power efficient processor architecture and the cell processor," in *11th International Symposium on High-Performance Computer Architecture*, Feb 2005, pp. 258–262.

[26] N. Kapre and A. DeHon, "Performance comparison of single-precision spice model-evaluation on fpga, gpu, cell, and multi-core processors," in *2009 International Conference on Field Programmable Logic and Applications*, Aug 2009, pp. 65–72.

[27] V. Cuppu and B. Jacob, "Organizational design trade-offs at the dram, memory bus, and memory controller level: Initial results," 1999.

[28] *Zynq-7000 All Programmable SoC Data Sheet: Overview*, Xilinx, june 2017, v1.11.

[29] *PYNQ-Z1 Board Reference Manual*, Digilent, april 2017.

[30] U. X. Employee), "Forumpost: Re: nomenclature of xilinx fpga," https://forums.xilinx.com/t5/General-Technical-Discussion/nomenclature-of-Xilinx-FPGA/m-p/47419/highlight/true#M2148, Xilinx, accessed: 15-03-2018.

[31] A. Percey, "Advantages of the virtex-5 fpga 6-input lut architecture," Xilinx, Tech. Rep., december 2007, wP284 (v1.0).

[32] *7 Series DSP48E1 Slice User Guide*, Xilinx, september 2016, v1.9.

[33] S. Wong and F. Anjam, "The delft reconfigurable vliw processor," in *Proc. 17th International Conference on Advanced Computing and Communications*, Bangalore, India, December 2009, pp. 244–251.

[34] *Zynq-7000 All Programmable SoC Technical Reference Manual*, Xilinx, september 2016, v1.11.

[35] J. E. Bottomley, "Dynamic dma mapping using the generic device (linux kernel documentation)," https://www.kernel.org/doc/Documentation/DMA-API.txt, accessed: 2017-11-25.

[36] *Arm Cortex-A75 Core Technical Reference Manual*, ARM, 2017.

[37] *AMBA AXI and ACE Protocol Specification*, ARM, 2013.

[38] D. S. Miller, R. Henderson, and J. Jelinek, "Dynamic dma mapping guide (linux kernel documentation)," https://www.kernel.org/doc/Documentation/DMA-API-HOWTO.txt, accessed: 2017-11-25.

[39] Unknown, "When is a volatile object accessed? (gcc online documentation)," https://gcc.gnu.org/onlinedocs/gcc/Volatiles.html, accessed: 2018-05-31.

[40] ——, "Motorola s-records," http://www.amelek.gda.pl/avr/uisp/srecord.htm, accessed: 2018-05-31.

[41] H. Al-Zoubi, A. Milenkovic, and M. Milenkovic, "Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite," in *Proceedings of the 42Nd Annual Southeast Regional Conference*, ser. ACM-SE 42. New York, NY, USA: ACM, 2004, pp. 267–272. [Online]. Available: http://doi.acm.org/10.1145/986537.986601

[42] M. Chrobak and J. Noga, "Lru is better than fifo," *Algorithmica*, vol. 23, no. 2, pp. 180–185, Feb 1999.

[43] *Cortex-R4 and Cortex-R4F*, ARM, 2011.

[44] S. Vigna and D. Blackman, "xoroshiro+ / xorshift* / xorshift+ generators and the prng shootout," http://xoroshiro.di.unimi.it/, accessed: 2018-03-07.

[45] M. D. Hill and A. J. Smith, “Evaluating associativity in cpu caches,” *IEEE Transactions on Computers*, vol. 38, no. 12, pp. 1612–1630, Dec 1989.

[46] A. Tanenbaum and A. Woodhull, *Operating Systems: Design and Implementation*, 3rd ed. Pearson Prentice Hall, 2009.

[47] E. W. Dijkstra, “Over seinpalen,” n.d., circulated privately.

[48] A. Brandon, J. Hoozemans, J. van Straten, and S. Wong, “Exploring ilp and tlp on a polymorphic vliw processor,” pp. 177–189, 03 2017.

[49] P. R. Rajeev Motwani, *Randomized Algorithms*. Cambridge University Press, 1995.

[50] R. Tamassia and M. H. Goldwasser, *Data Structures and Algorithms in Java*, 4th ed. John Wiley and Sons, Inc, 2014.