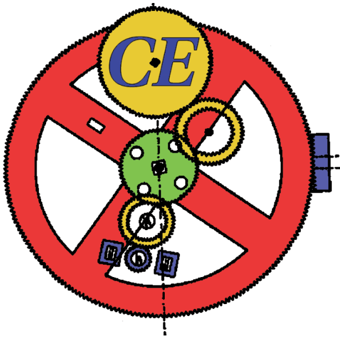


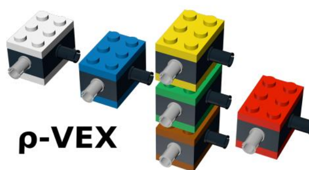
Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2018

MSc. THESIS

Operating system support for a dynamically reconfigurable VLIW processor

Anurag Kulkarni



CE-MS-2018-21

Abstract

The ρ-VEX is a dynamically reconfigurable VLIW processor, developed at TU Delft, which is capable of extracting large amounts of parallelism from applications running on it. However, without a dedicated software layer to dictate the reconfigurations, the ρ-VEX has to depend on another processor to carry out its reconfigurations meaningfully. Otherwise, this task can be left to the applications themselves, which would increase their complexity. In order to make the ρ-VEX capable of behaving as a stand-alone processor, with complex real-world applications running on it benefiting from its dynamic properties, while remaining abstract from the hardware changes, such a dedicated piece of software is needed. An operating system can equip the ρ-VEX with such functionality, as well as with other desired features like memory management support. Running applications in virtual memory is an important step towards multitasking. Therefore, the envisioned goal of this project is to make the ρ-VEX processor a truly independent and dynamic environment, capable of extracting large amounts of thread-level as well as instruction-level parallelism from programs running on it.

In this project, the Linux kernel 2.6.32 has been ported on `simrvex`: a cycle-accurate architectural simulator for the ρ-VEX processor. To test the port, three benchmarks from the Powerstone benchmark suite, *crc*, *ucbqsort* and *jpeg*, have been chosen to run as user programs on the ported Linux kernel. The timing performance of these benchmarks for the 2-issue, 4-issue and 8-issue configurations of the ρ-VEX is also presented.

TU Delft
Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Operating system support for a dynamically reconfigurable VLIW processor

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

EMBEDDED SYSTEMS

by

Anurag Kulkarni
born in Pune, India

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

Operating system support for a dynamically reconfigurable VLIW processor

by Anurag Kulkarni

Abstract

The ρ-VEX is a dynamically reconfigurable VLIW processor, developed at TU Delft, which is capable of extracting large amounts of parallelism from applications running on it. However, without a dedicated software layer to dictate the reconfigurations, the ρ-VEX has to depend on another processor to carry out its reconfigurations meaningfully. Otherwise, this task can be left to the applications themselves, which would increase their complexity. In order to make the ρ-VEX capable of behaving as a stand-alone processor, with complex real-world applications running on it benefiting from its dynamic properties, while remaining abstract from the hardware changes, such a dedicated piece of software is needed. An operating system can equip the ρ-VEX with such functionality, as well as with other desired features like memory management support. Running applications in virtual memory is an important step towards multitasking. Therefore, the envisioned goal of this project is to make the ρ-VEX processor a truly independent and dynamic environment, capable of extracting large amounts of thread-level as well as instruction-level parallelism from programs running on it.

In this project, the Linux kernel 2.6.32 has been ported on `simrvex`: a cycle-accurate architectural simulator for the ρ-VEX processor. To test the port, three benchmarks from the Powerstone benchmark suite, *crc*, *ucbqsort* and *jpeg*, have been chosen to run as user programs on the ported Linux kernel. The timing performance of these benchmarks for the 2-issue, 4-issue and 8-issue configurations of the ρ-VEX is also presented.

Laboratory	:	Computer Engineering
Codenumber	:	CE-MS-2018-21

Committee Members :

Advisor:	Dr. ir. Stephan Wong, CE, TU Delft
Chairperson:	Dr. ir. Stephan Wong, CE, TU Delft
Member:	Dr. ir. Mottaqiallah Taouil , CE, TU Delft
Member:	Dr. ir. Ioan Lager, Microelectronics, TU Delft

Dedicated to my family and friends

Contents

List of Figures vii

List of Tables ix

List of Acronyms xi

Acknowledgements xiii

1 Introduction 1
- 1.1 Context . . . 1
  - 1.1.1 Modern age computing trends . . . 1
  - 1.1.2 Multicore architectures and TLP . . . 3
  - 1.1.3 Reconfigurable Computing . . . 4
  - 1.1.4 VLIW and ILP . . . 4
  - 1.1.5 Advantages of the ρ-VEX processor . . . 5
  - 1.1.6 Current trends in operating systems supporting dynamic processors 6
- 1.2 Motivation . . . 7
- 1.3 Research Question, Goals and Methodology . . . 9
- 1.4 Overview . . . 10

2 Background 13
- 2.1 The ρ-VEX toolchain . . . 13
- 2.2 The ρ-VEX processor . . . 14
  - 2.2.1 Reconfiguration in the ρ-VEX . . . 15
  - 2.2.2 Configuration word encoding . . . 16
  - 2.2.3 Requesting a reconfiguration to the ρ-VEX processor . . . 18
  - 2.2.4 The ρ-VEX Instruction Set Architecture . . . 18
  - 2.2.5 The concept of generic binary . . . 20
  - 2.2.6 Trap handler in the ρ-VEX . . . 20
- 2.3 Simrvex: The architectural simulator for the ρ-VEX . . . 22
  - 2.3.1 TLB in simrvex . . . 23
- 2.4 Linux . . . 25
  - 2.4.1 The Linux kernel structure . . . 25
  - 2.4.2 Processes and threads in Linux . . . 27
  - 2.4.3 Scheduling in Linux . . . 28
  - 2.4.4 The Linux kernel's notion of time . . . 29
  - 2.4.5 The Linux boot process . . . 29
  - 2.4.6 Memory management in Linux . . . 30
- 2.5 Related Work . . . 31
- 2.6 Conclusion . . . 32

3 **Porting Linux to ρ-VEX** **35**
3.1 Choosing the ρ-VEX platform 35
3.2 Choosing the operating system kernel for the port 36
3.3 Porting Linux on simrvex 37
3.3.1 Modifications to the architecture-independent part of the Linux kernel 38
3.3.2 The architecture-specific kernel code 40
3.4 Conclusion 46

4 **Experimentation and Results** **47**
4.1 The Linux kernel image 47
4.1.1 Compressed kernel image 49
4.2 Booting 49
4.3 Evaluating timing performance of benchmarks from the Powerstone benchmark suite 53
4.4 Thread switch latency 57
4.5 Conclusion 59

5 **Conclusion** **61**
5.1 Summary 61
5.2 Main contributions 63
5.3 Future work 65

Bibliography **69**

List of Figures

2.1 Some configurations of the ρ-VEX processor[1] 16
2.2 A 1x4-issue and 2x2-issue configuration of the ρ-VEX processor[2] 17
2.3 The different layers of operation in a Linux-based system[3] 26
2.4 The Linux kernel structure[3] 26

3.1 Components of the Linux kernel structure that were ported[3] 44

4.1 The ELF header information for the vmlinux 48
4.2 The section headers inside vmlinux 48
4.3 The program headers inside vmlinux 49
4.4 The directory structure of the simple initramfs created to test the boot . 51
4.5 The Linux kernel boot logs observed on passing the kernel image to simrvex 52
4.6 Some of the kernel boot logs to show a shell can be started 53
4.7 Latency due to the use of Linux kernel code and virtual memory for the three benchmarks 56
4.8 Logs indicating constant thread switching between the two kernel threads created, along with the cycle counter register values 58

List of Tables

2.1 TLB configuration for simrvex 24

3.1 Register translation scheme for st200 to ρ-VEX[4] 40

4.1 Execution cycles consumed by the benchmarks when run on Linux ported on simrvex, for different configurations of simrvex 56

4.2 Execution cycles consumed by the benchmarks when run bare-metal on simrvex, for different configurations of simrvex 56

4.3 Latency in terms of percentage of increase in the execution cycle count for the three benchmarks when run on Linux ported on simrvex 57

List of Acronyms

ASIC Application Specific Integrated Circuit

BSS Block Started by Symbol

CPU Central Processing Unit

CRC Cyclic Redundancy Check

DTLB Data Translation Lookaside Buffer

ELF Executable and Linkable Format

FPGA Field Programmable Gate Array

GNU GNU's Not Unix

GDB GNU Debugger

HP Hewlett-Packard

ILP Instruction-level Parallelism

ISA Instruction Set Architecture

ITLB Instruction Translation Lookaside Buffer

JPEG Joint Photographic Experts Group

MMU Memory Management Unit

OS Operating System

RTOS Real-time Operating System

SASH Stand-alone Shell

SMP Symmetric Multiprocessing

TLB Translation Lookaside Buffer

TLP Task-level Parallelism

UTLB Unified Translation Lookaside Buffer

UNIX UNiplexed Information and Computing System

VEX VLIW Example

VLIW Very Long Instruction Word

Acknowledgements

Firstly, I would like to thank my MSc. thesis supervisor, Dr. Stephan Wong, for this project opportunity, as well as for guiding me throughout the project. I would also like to thank the thesis committee members, Dr. Ioan Lager and Dr. Mottaqialla Taouil, for agreeing to invest their valuable time on the evaluation of my work.

A very special thanks to Dr. Joost Hoozemans, my guide, for his support throughout the project. I could receive his support irrespective of his work location, and I consider myself fortunate for having gained so much through his intelligence and experience. I would also like to thank Jeroen van Straten from the Computer Engineering department at TU Delft for some important ρ-VEX toolchain fixes that he provided which were required for this project.

I would like to extend my thanks to all the EWI staff members involved in conducting the thesis formalities. My friends played a very important role in keeping my motivation high, for which I would like to thank them all from the bottom of my heart.

Most importantly, I would like to thank my parents, without whose emotional and financial support I would not have been able to reach this stage, and also my brother, Aditya, for taking care of things at home while I was away.

Anurag Kulkarni
Delft, The Netherlands
November 12, 2018

1 Introduction

This thesis documents the software implementation towards providing operating system support to the ρ-VEX processor. The intention of such an implementation is to make the ρ-VEX capable of behaving as a stand-alone system running complex, real-world applications which can benefit from its dynamic reconfigurability.

While the work involved (as would be described in detail in the following chapters) would appeal most to operating system and computer architecture enthusiasts, this chapter assumes a general computer engineering target audience. It focuses on providing an overview of the topics needed to understand the project, along with the motivation behind it, and also, the research question and the ultimate goals of this graduation project.

1.1 Context

The context of this project revolves around the advantages the ρ-VEX processor brings and the importance of porting an operating system to it, and will be discussed in detail in this section. We begin by discussing some modern age computing trends and a few techniques to exploit parallelism. These topics help understand the characteristics of the ρ-VEX processor that prove to be advantageous, which is discussed thereafter. Some modern operating systems which support dynamic processors will be discussed next. Together, the contents of this section would lead to understanding the motivation behind this project, discussed in Section 1.2.

1.1.1 Modern age computing trends

The ever-increasing demand for computing power and reduced system size and power consumption has lead to the evolution of modern computing. Improvements that have led to such an evolution can be broadly seen targeted to the following three aspects associated with a computer design[5]:

- Computer Architecture
- Computer Organization
- Computer Realization

Computer architecture refers to the conceptual structure and behaviour of a computer system as observed by its immediate user. It deals with, for example, the instructions and arithmetic a computing system is capable of executing. The logical organization of the dataflows and controls of a computer design is called computer organization, which is also its microarchitecture. Examples can include the different types of adder

units like the Manchester Carry Chain adder, for instance. Some texts use the term computer implementation for computer organization. Basically, computer organization establishes a method to achieve a desired functionality (architecture). Computer realization is the physical structure embodying the organization or implementation. It refers to the underlying technology, e.g., CMOS. Even FPGAs are computer realizations, as after manufacture (i.e., before any functionality is implemented on them), they merely exist as configurable blocks of logic gates.

Examples of improvements targeted towards computer realization or technology may include:

1. Technology scaling, which enables an increase in transistor density and speed

2. 3D-stacked memory, which reduces wire delays and maximizes throughput, and hence speeds up memory accesses multifold

Improvements in computer organization can be brought about by implementing techniques like pipelining and multi-level caches. However, the discussion which lies under the scope of this project pertains to an improvement in computer architecture.

We are, therefore, left with considering one of the most interesting themes of research in computer engineering: *exploiting parallelism in all its forms.* At the application level, parallelism can exist by virtue of multiple independent data or independent tasks. Computer organization and architecture styles can exploit these two types of parallelism in the following ways[6]:

1. *Instruction-Level Parallelism (ILP)*[1] - deals with compile-time scheduling of instructions such that data-level parallelism is exploited. It is concerned with aspects such as pipelining, speculative execution, etc.

2. *Vector Architectures and Graphic Processor Units (GPUs)* use the single instruction multiple data (SIMD) concept, and find applications in areas like Digital Image Processing.

3. *Thread-Level Parallelism (TLP)* - seeks to exploit either Task-Level or Instruction-Level Parallelism primarily through the Multiple Instructions Multiple Data (MIMD) concept. It involves identifying and executing independent "tasks" over multiple processing units. It is, hence, implied that Thread-Level Parallelism primarily focuses on multiprocessor and multicore architectures.

4. *Request-Level Parallelism (RLP)* - deals with Distributed Computing which involves executing largely decoupled tasks over computer clusters with multiple servers connected to a network.

Only ILP and TLP lie within the scope of discussion of this project. The means through which they are exploited are discussed next. It should be noted here that limitations of ILP approaches directly led to the rise of multicore architectures[6].

[1]ILP is the potential overlap between instructions

1.1.2 Multicore architectures and TLP

The notion of (truly) parallel processing is not new. It is a logical solution to physically running multiple threads (tasks[2]) in parallel, and thus exploiting thread-level parallelism. In fact, the first parallel computers were developed in the 1960s, e.g., ILLIAC IV[7]. These were, however, difficult to program and were expensive, which limited their use to supercomputers. The 1990s saw the rise of multiprocessor systems, but these also suffered from the drawbacks of programming difficulties, cache consistency maintenance difficulties[8], and size. The computer industry then shifted to multicore architectures. The first non-embedded multicore processor was IBM's POWER 4, which was released in 2001[9]. It featured two processor cores on a single chip, 4-bit PowerPC architecture and two cache levels.

Some of the reasons as to why multicore architectures became popular quickly were[3]:

1. The operating systems now had to manage a single memory hierarchy as the processor cores on the chip shared caches and MMUs.
2. Thread management was easier and so was cache synchronization.
3. Lesser inter-processor communication latency.
4. Better power performance - as they can run on lower clock frequencies than a single processor, to achieve the same performance.

Of course, they are still popular in the market, with their areas of application ranging from handheld devices to super computers. The number of processor cores is often used as a selling point.

While designing a multicore processor chip, the area of application dictates the number of cores to exist on the chip and their size. According to [10], general-purpose computers benefit from a small number of large computing cores, as they have course-grained threads which depend on the instruction-level parallelism for performance. A larger number of small processors is suited for applications which exhibit a lot of thread-level parallelism[4]. Architectures lying in between these extremes are suited only for specific workloads for which there is a granularity match.

Another point to be considered here is that the growing popularity of multicore processors has caused a focus shift towards parallelizing programs. However, sequential (single-threaded) code still remains important, and may often be difficult to parallelize. It can be argued that asymmetric multicore processors may attempt to solve this issue by dedicating a (more) powerful core to the sequential program, but the parallel applications' performance often suffers due to this: the resources dedicated to powerful cores could be used instead for more simple (small) cores for parallel execution.

It is, therefore, desirable to have a *dynamic reconfiguration* of processor cores based on the nature of the task(s) being executed. For example, Core Fusion[11] is capable of fusing multiple cores into a larger one dynamically through reconfiguring caches and

[2] Tightly-coupled tasks implied here

[3] ET4C07 Advanced Computing Systems 2017, TU Delft, Lecture 2 - Cores

[4] Popular terminology: Bulldozers, Chainsaws and Termites

other resources. Speculative multi-threading executes different portions of a single thread in parallel on different cores[12], and Intel's Turbo boost can dynamically change the frequency of processor cores based on their temperature[13]. Such reconfigurations can take place in microseconds[12].

1.1.3 Reconfigurable Computing

Reconfigurable computing aims to provide a balance between the flexibility of General Purpose Processors (GPPs) and the performance of ASICs. ASICs have the highest performance and consume the least power and area among the three, but they are not flexible. GPPs, on the other hand, are the most flexible due to their versatile instruction sets, but are not optimized for a particular application[14]. In the words of Andre DeHon, *"Reconfigurable Computing is computing via post-fabrication, spatially programmed connection of processing element."* Field-Programmable gate Arrays (FPGAs) are a popular choice for the high-speed flexible computing fabric needed to implement reconfigurable hardware. It is to be noted here that reconfigurable architecture refers to the *changing functionality*, and not the technology (like FPGAs).

In a way, reconfigurable computing enables controlling the underlying microarchitecture through software[5]. Reconfigurable computing itself is a vast space of computer designs with the ρ-VEX falling in the category of dynamic processors, with software being required to run on top of it. Reconfigurable hardware implementations are not discussed here. These implementations are suitable for a specific type of applications like streaming applications.

1.1.4 VLIW and ILP

Another way of looking at exploiting ILP is decreasing CPI (Cycles Per Instruction), or effectively, increasing IPC (Instructions per cycle). According to [6], the CPI value for a pipelined processor can be given by:

```
Pipeline CPI = Ideal pipeline CPI + Structural stalls + Data hazard stalls
+ Control stalls                                                        (1)
```

The `ideal pipeline CPI` is a measure of the maximum performance attainable by the implementation[6]. `Ideal pipeline CPI` is the only relevant parameter (among those in the R.H.S. of the above equation) for discussion from a computer architecture point of view. One of the ways to decrease the `ideal pipeline CPI` is issuing multiple instructions per cycle. Such *multiple-issue processors* can be of the following types:

1. Superscalar Processors - These issue varying number of instructions per cycle.

2. VLIW (Very Long Instruction Word) Processors - These issue fixed number of instructions per cycle.

Superscalar processors can in turn be statically scheduled (compiler intelligence required) or dynamically scheduled (reliance on hardware to discover and exploit parallelism dynamically). An example of a statically scheduled superscalar processor can be ARM

Cortex A8 while Intel Core series and ARM Cortex A9 are dynamically scheduled superscalar processors[6]. VLIW processors are inherently statically scheduled.

Statically scheduled superscalar processors witness diminishing returns with increase in issue width (above 2)[6] and dynamically scheduled superscalar processors require complex hardware circuitry[5]. VLIW processors, therefore, tend to be more area and power efficient than superscalar processors[6]. The simplicity and mutability[7] of VLIW processors work in their favour to be considered for exploiting ILP over superscalars.

1.1.5 Advantages of the ρ-VEX processor

As discussed in Section 1.1.2, it is advantageous to have processors which can be reconfigured to change their behaviour as per the nature of the application(s) executing on them. The ρ-VEX itself is a reconfigurable processor, and exists as a softcore, but it does not involve an FPGA reconfiguration[8]. This means that there is no loading of a bitstream to perform partial or full reconfiguration, as is common with FPGAs. The overhead of a reconfiguration is thus equivalent to that of a pipeline flush and is of the order of 5 cycles[15]. This proves to be an advantage as FPGA reconfiguration time can be of the order of 100ms[16].

Also, the ρ-VEX is a dynamic VLIW processor, meaning, it can execute a variable number of instructions per cycle[9], and this number can be decided runtime. Therefore, it combines the simplicity of VLIWs with the dynamism of superscalars. As per [8], the dynamic nature of the ρ-VEX can ease the implementation of hyperthreading. (Dynamic) Superscalars use this feature to save resources in case the program under execution has a lower ILP than predicted. For more information on hyperthreading, [17] can be referred. Specifically, the ρ-VEX processor is a (reconfigurable) VEX implementation[8]. VEX stands for `VLIW Example` and is a system comprising of a flexible ISA, a C compiler and a simulation system. It is introduced in [18] for the purpose of experimental use.

As per [15], the ρ-VEX can constantly adapt to the available ILP and TLP by dynamically changing the mapping between the available threads and issue slots. Thus, it can dynamically reconfigure between many small processors[10] (when more TLP is available), and one large processor (when more ILP is available). Such characteristics of the ρ-VEX make it a suitable candidate for research to further obtain performance gains for tasks running on it. The idea of adding an operating support to the ρ-VEX stems from the need to abstract "how and when the ρ-VEX is reconfigured to maximally exploit available parallelism, while saving resources" from the applications running on top of it. Section 1.2 describes in detail the motivation behind the OS support and the choice of the OS.

[5] Although the advantage is that programs need not be recompiled for processors with different degrees of parallelism

[6] The disadvantage being recompilation requirement for processors with different instruction issue widths

[7] The ease of morphing an existing VLIW design to a similar design

[8] Reconfiguration mechanism is discussed in detail in Chapter 2

[9] Number of instructions (issued) per cycle can be 2, 4 or 8; to be discussed in detail in Chapter 2

[10] Actually, virtual cores - to be discussed in Chapter 2

1.1.6 Current trends in operating systems supporting dynamic processors

As discussed in Section 1.1.3, the ρ-VEX falls under the category of dynamic processors, i.e, it can change some of its properties runtime. There exist other such dynamic processors, e.g., CoreFusion[12], which can do this. Often, it is the cores and the caches that are considered for reconfiguration. It is important that an operating system maintains its state during the reconfigurations of the processor hardware, and also is able to control such reconfigurations, for multiple reasons. Some examples of modern operating systems[11], and their mechanisms, supporting such dynamic processor reconfigurations and environments are as follows:

1. **The Linux hotplug mechanism**
   Hotplugging became a standard feature in the Linux kernel from version 2.4[19]. Through the hotplug mechanism, the Linux kernel can update itself when devices are added to the processor system or removed. A classic example of such a device-level hotplug support by the Linux kernel is for USB devices. In a similar manner, Linux also supports CPU hotplugging. This means that one or more cores can be selectively disabled by the kernel, which helps in removing a dysfunctional processor or allocating processor cores from virtual machines to the host machine[12]. Hotplugging mechanisms, however, suffer from a delay of the order of 200ms while performing the activation or deactivation of a core[12].

2. **The Barrelfish OS**
   The Barrelfish is an operating system developed from scratch at ETH Zurich, for research purposes[20]. Its development is motivated by the scalability issue faced due to the rapid increase in the number of cores, and also, the need to manage the increasing heterogeneity in computer hardware resources[21]. The Barrelfish is capable of fast coupling and decoupling of the kernel state to and from the cores, thereby resulting in a highly dynamic environment. Essentially, a lot of low-level code is populated in databases, thereby doing away with driver software, which aids in fast device interrupt rerouting and thereby resulting in fast kernel decoupling[22]. Kernel coupling and decoupling happens through dynamic booting and shutdown of cores[22]. [22] reports a 2x decrease in boot time and a 57,500x decrease in the shutdown time of a core[12] when compared to the hotplug mechanism in Linux kernel version 3.13.

3. **The Chameleon OS**
   An operating system support for dynamic processors is presented in [12]. The work results in an an extension to the Linux kernel which supports rapid dynamic processor reconfigurations, called Chameleon, and basically involves the following:

   (a) Providing *processor proxies*, which are abstraction structures running on active processor cores, and representing offline processors. When the kernel

[11] Kernels mostly
[12] No load conditions

requires the presence of the processor cores for certain operations, these proxies can be helpful. The kernel, therefore, need not wait for the offline processor(s) to become active (through reconfiguration) for performing certain core-specific operations, thereby enabling fast reconfiguration.

(b) Abstracting the cores through kernel structures called *execution objects*, which act as representatives of the active cores. Basically, during scheduling, these objects can be assigned to threads, while the actual dispatching to the real hardware can be done after the assignment.

(c) Implementing a gang scheduling based technique called *cluster scheduling* in order to balance the needs of the sequential and parallel programs. Chameleons scheduler can use intelligence to decide whether to execute a thread on a single core or multiple cores via selective activation of an execution object.

[12] reports a speedup of upto 75,000 times for fusing cores (*hot unplugging* a CPU) and upto 160,000 times for splitting an execution object of two CPUs (*hot plugging* a CPU).

The ρ-VEX is a dynamic processor capable of rapid reconfigurations. Therefore, the supporting operating system should also be able to control such reconfigurations through minimum latency. Out of the kernel mechanisms discussed above, the features of the Chameleon OS are the most desirable to have as it provides the fastest reconfiguration support. Besides, it is an extension of the standard operating system kernel, Linux, which itself is modular and portable. This increases the chances of resulting in a successful port to the ρ-VEX. The Linux hotplug mechanism is too slow for the rapid runtime reconfiguration capabilities of the ρ-VEX. The Barrelfish OS is also a good option, though not as suitable as the Chameleon OS. Besides, it is an experimental operating system as of now. Also, it mainly targets heterogeneous multicore systems[21]. With these parameters taken into account, the Barrelfish OS is not the best choice for porting to the ρ-VEX processor.

1.2 Motivation

An operating system running on top of the ρ-VEX processor can provide the necessary abstraction of the reconfiguration mechanism to the applications it runs. It can take care of the software state while performing configuration changes in the ρ-VEX, dynamically, to maximize parallelism exploitation. Due to this, applications meant to run on static processors can also run on the ρ-VEX, which is not possible now in case of a reconfiguration requirement. A program running on the ρ-VEX core can indeed request a reconfiguration, but in order to decide upon a suitable configuration, a lot of efforts would be required during the development of its code. Instead, intelligent algorithms can be embed into another software layer, running on the ρ-VEX, which is capable of scheduling such programs or tasks. The algorithms, for example, can monitor

the execution cycles consumed by a running task and also the NOP[13] instructions in its instruction stream to determine whether a task indeed benefits from a configuration. An operating system can act as this intermediate layer, and can control the reconfiguration by, for example, scheduling a reconfiguration request (as a task) after identifying the suitable configuration for a task running on the operating system. Such a mechanism can abstract:

1. The algorithm to determine a suitable configuration for a task running on the ρ-VEX processor.
2. The reconfiguration request[14] procedure.
3. The saving of the (software) context of a task before a reconfiguration.
4. Restoring the software context after the reconfiguration.

The general functions of an operating system like memory management, file system management, input/output device handling, etc. are also desirable to have. If all these functions are not provided, either the applications themselves have to take care of the hardware control and finding out a suitable configuration for themselves, or the ρ-VEX needs to behave as a co-processor, with another processor, running an OS, controlling its configurations as suited for the applications meant to run on the ρ-VEX core. In the former situation, programs meant to run on the ρ-VEX processor need to include the necessary code for hardware control, and configuration determination and reconfiguration request if runtime reconfiguration is to happen, which is not logical at all. In the latter situation, the communication latency between the main processor and the co-processor or the accelerator (here, the ρ-VEX) will affect performance of applications to a great extent.

[23] also mentions the desirability of an operating system for the ρ-VEX. While [23] reports a uCLinux variant ported on the ρ-VEX softcore, this variant of Linux is without an MMU (Memory Management Unit) support. During the time of implementation of [23], the ρ-VEX did not have an MMU. It still does not have it, but its simulator, *simrvex*, has this unit included. Basically, an MMU allows programs to use virtual memories and serves as a translator between virtual and physical memories. An MMU support is important for paging and preventing illegal memory accesses by processes. Every process is given its own address space, therefore, there is a protection from accesses of address spaces/secure data between processes. Each such address space is broken into fragments called pages, which give the illusion of contiguous memory. Not all the pages have to be mapped to physical memory. If a process accesses a page which is mapped to physical memory, a direct translation is provided by the MMU. However, if the page does not lie in the physical memory, an exception is triggered by the operating system, which fetches the corresponding page by evicting some other, and re-executes the failed instruction. It is also very important that the kernel address space and the user address space is maintained (and mapped) separately, which can be achieved effectively by using virtual memory. Virtual memory also allows for memory

[13] No Operation

[14] Reconfiguration request to the ρ-VEX processor

to be swapped to an external storage, such as a disk, or a memory mapped file, thereby creating an illusion of sufficient physical memory (RAM). This is not a problem nowadays as it is common to have physical memories larger than 4GB (for systems with 32-bit address spaces, for instance) but virtual memory still makes it effective to run multiple processes simultaneously. With these advantages of using virtual memory, it becomes very much desirable to have an operating system for the ρ-VEX which supports memory management.

1.3 Research Question, Goals and Methodology

The facts discussed earlier provide a good motivation to move ahead with the project implementation. However, as with any project, the goals need to be clearly defined and as this is a research-based project, a research question needs to be formulated. Our entire discussion will therefore be along the lines of the answer to this question. The long term goal of this project is:

To make the ρ-VEX capable of behaving as a stand-alone processor, with the applications running on it benefiting from its dynamic properties, while remaining abstract from the hardware changes.

However, to limit the scope of the project to that of a graduation project, the research question which this thesis intends to answer is:

How to provide operating system support to the ρ-VEX?

With an operating system support, the ρ-VEX can result in a stand-alone dynamic environment, which can maximally exploit parallelism in applications running on it. At the same time, the necessary abstraction with regard to the (dynamic) core reconfiguration mechanism can be achieved. As we already know, without an operating system, the ρ-VEX would have to depend on the programs to take care of their contexts themselves while performing core reconfigurations, or an external processor which would decide upon the reconfigurations of the ρ-VEX and run programs on it. In the latter case, the ρ-VEX would behave as a co-processor or an accelerator.

The answer to the above-stated research question involves many considerations ranging from the choice of the OS, to ensuring whether the ρ-VEX processor and the associated toolchain support it, and also the modifications in the chosen OS to make it "fit" for the ρ-VEX. Testing the implementation for the port is implicit.

It can be realized that the task requires a thorough knowledge of the ρ-VEX architecture, functionality of the chosen microarchitecture, toolchain functionality and limitations, operating system internals, boot sequence and thread creation in the ρ-VEX. Now that the focus of this project has been identified, the following project goals, along with the methodologies to achieve them, can be demarcated:

1. Porting an operating system to the ρ-VEX.

The approach to be considered to accomplish this goal would be of a multi-faceted nature, and can be said to consist of the following:

i) Choose an appropriate microarchitecture of the ρ-VEX which would act as the platform upon which the operating system will be ported.
ii) Ensure that the toolchain is capable of supporting an operating system port.
iii) Choose the right operating system for porting.
iv) Implement the changes in the chosen operating system's kernel code in order to accomplish the port.

What is also desirable to achieve under this goal is the compression of the end product of the operating system kernel compilation. This would help achieve faster startup time[15], apart from the reduction in the requirement of external memory for storing the kernel.

2. Evaluating performance of applications run by the operating system ported to the ρ-VEX.

   The performance of applications can be evaluated by measuring the execution time taken by them while running on the OS running on the ρ-VEX processor. Appropriate infrastructure needs to be implemented to read (and naturally, output) the count of execution cycles consumed by a running application.

In order to limit the scope of this project to that of an MSc. graduation project, the above two goals have been set for the same. However, in the event of availability of further time, the following goal is highly desirable to be pursued.

Implementing modifications in the kernel as mentioned in [12] to support fast runtime reconfiguration in the ρ-VEX, thereby creating a truly dynamic environment.

The implementation for this goal together with that for the first two will help accomplish the long term goal of this project. The third goal stated above gives rise to interesting research opportunities towards development of the ρ-VEX project. Even in case of no implementation towards it, a superficial discussion alone would provide useful insights for the next project which aims to implement it.

1.4 Overview

So far, the relevance of the project work has been discussed with an introductory flavour. Chapter 2 provides a detailed background of the ρ-VEX system essential to proceed

[15] In most processors, the total time required to read a compressed kernel image from an external storage, decompress it and then boot it, is often less than the total time required to read the uncompressed kernel from an external storage and then boot it[24]. In other words, in most processors, decompression is faster than I/O operations.

with the implementation. It discusses the study of the compiler, linker, the ρ-VEX architecture, as well as *simrvex*. Work related to this project and already been done is also discussed in Chapter 2.

Chapter 3 details the implementation of the entire project. It describes the porting in detail, including new implementations, modifications, workarounds done to achieve the goals.

Chapter 4 provides some results of the implementation, like timing performance of applications running on the ported operating system.

Chapter 5 consists of the conclusion for the entire project work, along with the main contributions towards the project. It also explains how this project lays the foundation for future work. Approaches to goal(s) not achieved are presented with the hope to provide a direction for future research.

Background

2

In the previous chapter, an overview of the project, along with the topics relevant to its context were discussed. This chapter discusses the concepts that are needed to be known in order to proceed with the implementation of the project. The discussion will now proceed from an introductory style to one which contains details of the topics with a higher relevance to this project.

The most important points to be analyzed while porting an operating system to a new processor is whether the processor can support it, and whether the toolchain is mature enough. Hence, we proceed with a discussion about the platform, which includes the compiler, the linker and the processor. Then, we move on to discussing the processor architecture details relevant for this project. Since porting an operating system involves modification of the source code, it is highly desirable to explore open source operating systems for the port. Linux is an open source operating system with rich documentation and online support available. As a result, the structure and organization of the Linux kernel will also be discussed in this chapter. Related work will be discussed after that. This chapter then concludes with a summary of the discussed topics to reinforce relevant concepts needed to understand the implementation presented in Chapter 3.

2.1 The ρ-VEX toolchain

The ρ-VEX comes with an Open64-based compiler and a GNU `binutils` port which comprises of the GNU Assembler and Linker modified for the ρ-VEX. This Open64-based compiler and `binutils` port together form part of the toolchain for the ρ-VEX[1].

Open64 is an open source and GCC compatible C/C++/Fortran compiler, suitable mainly for 32-bit architectures. A compiler, based on the Open64 compiler, was developed for the Lx architecture, which was designed by HP and STMicroelectronics[25]. Lx makes use of a scalable issue width VLIW technology and is introduced in [26]. The `st200` family of processors is based on this Lx architecture. The ρ-VEX compiler's front end gcc compatible, while its backend is modified to be compatible with the instruction set of the ρ-VEX processor. In common terminology, this is a port from an Open64-based compiler meant for Lx to one meant for the ρ-VEX. It is to be noted here that the compiler (and the binutils, which will be discussed shortly) for the ρ-VEX, themselves can run on `x86` machines. Therefore, a cross-compilation environment can be set up on the host machine which runs Linux.

The `binutils` port for the ρ-VEX is the GNU binutils open source project with the target modified for the ρ-VEX. It consists of the following tools:

[1]Actually, the ρ-VEX toolchain currently also includes a debugger (gdb modified for the ρ-VEX).

1. The ρ-VEX assembler (`rvex-elf32-as`)
2. The ρ-VEX linker (`rvex-elf32-ld`)
3. Elf manipulation tools like `rvex-elf32-objcopy`, to copy the contents of one object file to another, and `rvex-elf32-objdump` to display internal information of ρ-VEX specific object files
4. Library manipulation tools like `rvex-elf32-ar` and `rvex-elf32-ranlib`
5. The GNU debugger modified for the ρ-VEX (`rvex-elf32-gdb`)

Out of the above, only the assembler and linker will be discussed as they are visible to the programmer in the sense they are explicitly referred to in the Makefiles of the Linux kernel. The `rvex-objdump` is also useful while analyzing object files for the ρ-VEX as they provide valuable information as to what part of the code goes in which section (or segment) of the memory. The ρ-VEX toolchain must be capable of generating Linux kernel images as outputs (will be discussed later). But as a background study, it is very difficult to determine whether the kernel compilation will be successful using the toolchain without any modifications. Even though the compiler is Open64-based, and Open64 itself is gcc compatible and the Linux kernel also uses extensions specific to the GCC[23], compilation and linking incompatibilities can be expected. Therefore, the tool chain was initially tested by compiling some very basic C programs (like a "Hello World" printing application) for the ρ-VEX, and then running them on *simrvex*, while verifying the output (of *simrvex* as well as the application) observed on the terminal. The errors that were expected to be encountered while compiling the entire kernel were deemed to be resolved during the implementation stage.

2.2 The ρ-VEX processor

The ρ-VEX adopts a 32-bit big endian architecture. It currently exists as a flexible microarchitecture[23]. It has FPGA[2] and ASIC[3] implementations, and also has *simrvex* as its architectural simulator. [25] provides detailed information about the ρ-VEX processor, such as registers, instruction set architecture (ISA), debugging, etc. The characteristics of the ρ-VEX that prove to be advantageous have been discussed in Chapter 1. Here, the intention is to discuss the architectural details of the ρ-VEX, which would be helpful to understand the implementation presented in Chapter 3.

As we can recall, the ρ-VEX is a dynamically reconfigurable VLIW processor. In VLIW processors, a single instruction can actually contain multiple instructions, or rather, operations. Only independent operations are allowed to lie inside a VLIW instruction. The operations are called *syllables*, and the instructions are called *bundles*, and further discussion adheres to this terminology. In the VEX architecture, upon which the ρ-VEX is based, the boundaries between bundles are specified by means of a so-called *stop bit*[25]. Therefore, the placement of this stop bit decides the size (in terms of number

[2]ML605 and VC707 FPGA development boards[8]
[3]Application Specific Integrated Circuit

of syllables) of the next issued bundle. If stop bits are not enabled, and if the number of syllables in a bundle comes out to be less than the configured bundle size, the compiler should place NOP (no operation) instructions to align the bundle boundary with the configuration. A configuration capable of executing n syllables per cycle is referred to as an *n-way* or an *n-issue* configuration. So, basically, *reconfiguration* here refers to changing such a configuration of the processor, and the reconfiguration specific to the ρ-VEX is discussed next.

2.2.1 Reconfiguration in the ρ-VEX

In the ρ-VEX, the distribution of the *pipelanes* between different programs running in parallel is altered during reconfiguration. Pipelanes are basically groups of computational resources which execute a syllable, and their total number is fixed[25]. They are, however, not entirely reconfigurable, in the sense, there exist groups of inseparable pipelanes, which are called *lane groups*. The pipelanes in such lane groups have to remain activated together.

To be able to run multiple programs in parallel, the ρ-VEX processor supports multiple *contexts*[4]. As of now, the ρ-VEX supports 4 contexts (Context 0 to Context 3). Contexts are independent units with each having its own complete state, which consists of the program counter (PC), register files and other control registers. Contexts can run instructions independently are therefore very much like cores in a multicore processor, and [25] states that they can be considered as *virtual cores*. The terms *cores*, *contexts* and *virtual cores* will be used interchangeably henceforth.

Reconfiguration here involves changing the “connections” between the lane groups and the contexts. As per the current state of the ρ-VEX, the maximum number of execution pipelanes supported is 8 and the minimum number of pipelanes a lane group can consist of is two. The number of pipelanes a core is connected to is called its *issue width*, and such a number has to be a power of two. Therefore, some of the possible configurations of the ρ-VEX processor are: 1x8-issue, 2x4-issue, 4x2-issue, 1x4-issue and 2x2-issue. These configurations assume all contexts being activated. In fact, contexts can be deactivated by assigning no lane groups to them. Further configurations can then easily be identified. Such selective deactivation of the contexts is important for reducing power consumption, and also, the deactivated contexts can hold some extra states of the programs under execution on the active contexts, thereby enabling faster context switching. Figure 2.1 helps visualize some of the configurations of the ρ-VEX processor. As can be seen from this figure, a core of any issue width can get a cache block of appropriate size. This happens because the instruction and data cache blocks in the ρ-VEX can work together as well as independently. Also, no cache flushes are required during core reconfigurations.

As also discussed in Chapter 1, configurations with many small cores are suited for applications which exhibit a high TLP, while those with a few large cores are suited for applications with a high ILP. The word “suited”, here, refers to better performance and more efficient resource utilization. The ρ-VEX processor can therefore adapt itself to a wide variety of applications.

[4]Actually, *hardware* contexts

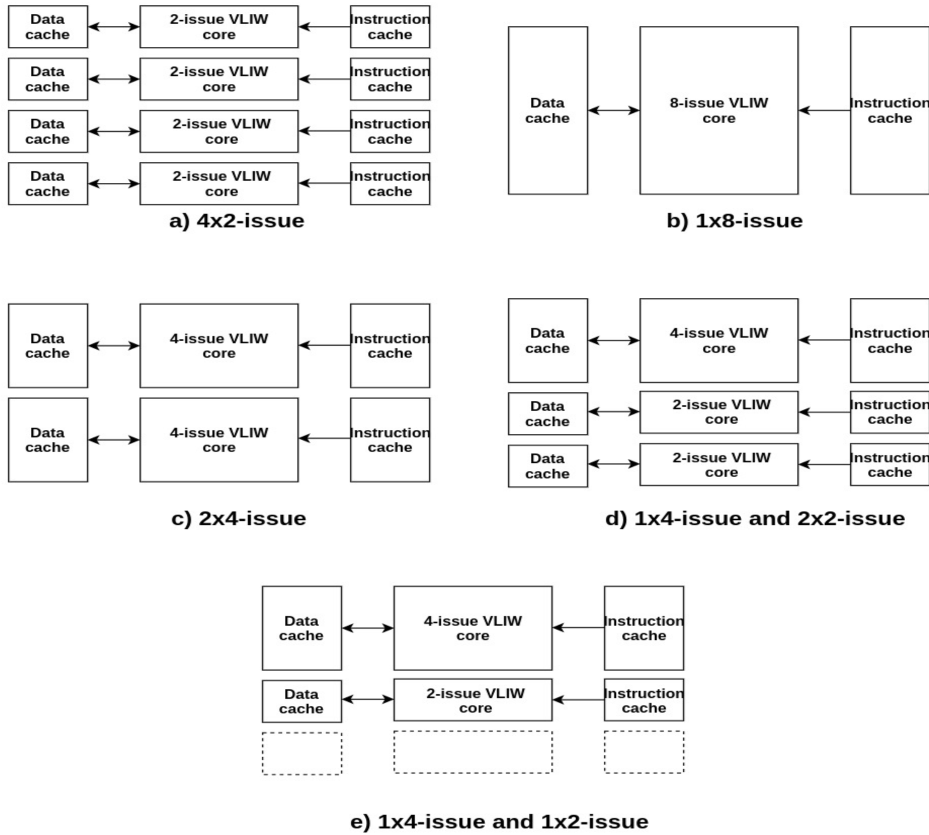


Figure 2.1: Some configurations of the ρ-VEX processor[1]

2.2.2 Configuration word encoding

In order to enable certain configuration in the ρ-VEX core, an encoding needs to be written to one of its context control registers. This encoding, referred to as *configuration word*, should follow certain rules to result in a valid ρ-VEX configuration. The configuration word consists of 32 bits, with each nibble concerning with the context a particular lane group is mapped to. For each such nibble, the most significant bit decides whether the lane group is enabled or disabled, with the value '1' standing for disabled. However, the values 9 to F for a single nibble are reserved for future use, therefore, the value a nibble can take lies within the range 0 to 8. This leaves the last three bits of a nibble to represent the context the corresponding lane group is connected to. However, as the ρ-VEX currently supports only upto 4 contexts, the 3 least significant bits of the nibble can represent values from 0 to 3. If the nibble has a value of 8, the lane group is effectively disabled. Therefore, configuration encoding like 0x7000 is an invalid configuration. Moreover, there are some other special rules governing configuration encoding. These are listed below.

- A context can only be mapped to contiguous lane groups, and the number of such

lanes groups has to be a power of two. For example, 0x0103 is invalid, but 0x0013 is valid.

- The configuration word is of length 32 bits, which means 8 nibbles, therefore, in theory, 8 lane groups can be mapped to contexts. However, as of now, testing has been performed only for design-time configurations of upto 4 lane groups[25]. The nibbles representing non-existent lane groups should be 0. For example, for a configuration of the ρ-VEX to support 4 lane groups, 0x88888800 is invalid, whereas, the configuration 0x00008800 is valid.

Based on these rules, the configuration of the ρ-VEX can be identified via the configuration word encoding. For example, an encoding 0x8800 represents a 1x4-issue configuration, 0x0 represents a 1x8-issue configuration, and so on. Figure 2.2 helps visualize a 1x4-issue and 2x2-issue configuration of the ρ-VEX, with context 3 being deactivated. The corresponding encoding would be 0x2100. It should be noted here that a context can be deactivated by not assigning any lane group to it, and a lane group itself can be deactivated by writing an '8' to the corresponding nibble in the configuration word encoding.

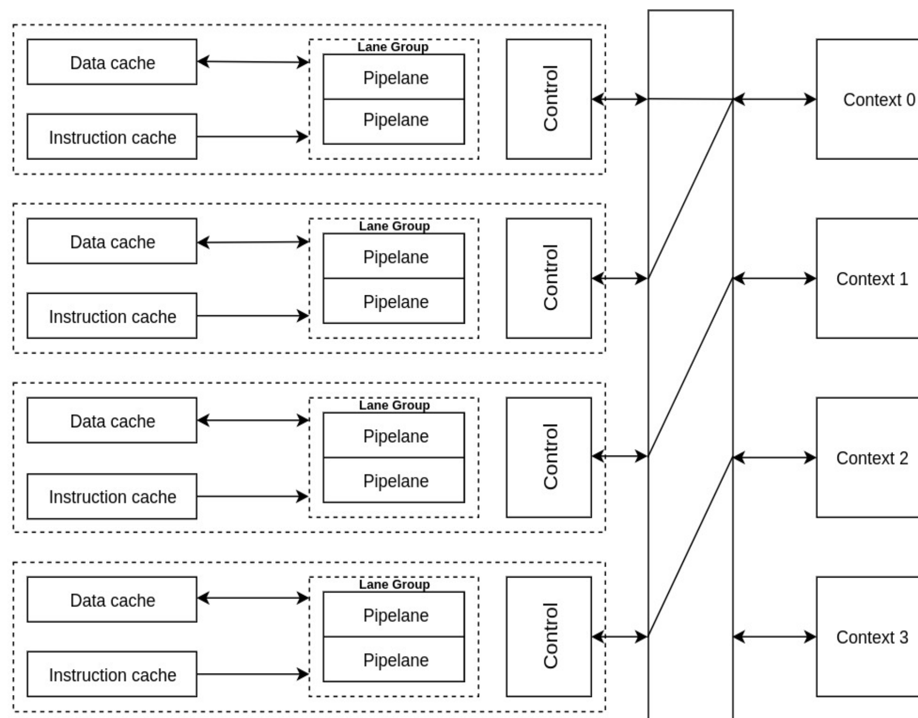


Figure 2.2: A 1x4-issue and 2x2-issue configuration of the ρ-VEX processor[2]

2.2.3 Requesting a reconfiguration to the ρ-VEX processor

There are three ways to request a reconfiguration to the ρ-VEX processor[25].

- Writing the configuration word to the context control register, CR_CRR, via a program running on a context.
- Writing the configuration word to the global context control register, CR_BCRR, via the debug bus by a program running outside the ρ-VEX core.
- Writing the configuration word to the wake-up configuration register, CR_WCFG, and setting the appropriate flag in the sleep and wake-up control register, CR_SAWC. These registers only exist on context 0. When an interrupt is pending on context 0 while it is not already active, reconfiguration as per the encoding occurs, provided there is no other reconfiguration request present. After the reconfiguration, the flag which was set in the CR_SAWC is cleared, and the old configuration word in CR_WCFG is restored. This mechanism is called *reconfiguration using the sleep and wake-up system.*

A new configuration commit process consumes execution cycles of the order of tens of cycles, depending upon how many cycles the reconfiguration controller hardware consumes to halt concerned contexts. A reconfiguration request may be rejected if the encoding does not follow the rules stated in Section 2.2.2, or when another such request from another core or the bus occurs simultaneously, and the arbitration is lost.

2.2.4 The ρ-VEX Instruction Set Architecture

The instruction set architecture of the ρ-VEX is based on the VEX architecture, which is introduced in [18]. As the ρ-VEX user manual ([25]) explains the ISA in sufficient detail, the discussion here is kept limited and only as required to understand a few parts of the implementation presented in Chapter 3.

Assembly syntax
The following piece of assembly code has been taken directly from [25] for illustration purposes.

```
start:
c0 stw 0x10[$r0.1] = $r0.53
c0 add $r0.3 = $r0.0, -32
c0 and $b0.2 = $r0.0, $r0.10
c0 call $l0.0 = interrupt
;;
```

The first line represents a label. Labels always end in a colon. Any non-empty line which is not a label, and does not begin with a semi-colon is a syllable. The 'c0' at the beginning of each syllable represents the cluster the syllable belongs to. The ρ-VEX does not support clusters, hence specifying the cluster is optional, and if specified, it

is always cluster 0. 'Stw', 'add', etc. are opcodes. The parameter list, which can be seen following the opcodes in each line has an '=' sign. The '=' sign separates the destination on the left from the source (which provides the operands) on the right. The string immediately after the '$' sign represents the register. The 'r', 'b', 'l', etc., stand for general-purpose, branch and link registers respectively. The types of registers in ρ-VEX will be discussed shortly. The '0' in '0.xx' specifies cluster 0, while the number following the decimal point represents the register number. There are restrictions on the usage of these numbers, heavily depending on the availability of registers. More on this can be found in [25]. The load and store instructions require a memory reference as one of their operands. A memory reference is represented in the form: *literal*[$r0.*index*], which means, at runtime, the *index* is added to the value in the r0.*index* to generate the required address. Finally, the ';;' marks the end of a bundle. The above piece of code, therefore, is actually one complete bundle.

Registers
The ρ-VEX currently uses 5 types of registers, as described below.

1. General-purpose registers

   There are 64 32-bit general-purpose registers in the ρ-VEX core, numbered from 0 to 63. These are intended for arithmetic. Register 0 always reads a 0 when read by the processor. It is, however, also writable. Register 1 is intended to be used as a stack pointer.

2. Branch registers

   The ρ-VEX core contains 8 1-bit branch registers. They are labeled from 0 to 7 and are used for branch conditions, select instructions, divisions, and additions of values wider than 32 bits.

3. Link registers

   The ρ-VEX core contains 1 32-bit link register (l0.0), which is used to store the return address during a function call or a jump. It can also be used as the destination address for an unconditional indirect jump or call, in cases where the branch offset field is too small or when the jump target is determined at runtime.

4. Global and context control registers

   The global control registers contain information about the processor state which is not specific to any context. The context control registers contain status information which are specific to a context. The processor can access these register files through memory operations only. A program can only read from a global register file and can only access its own hardware context control registers.

2.2.5 The concept of generic binary

A VLIW processor depends on compiler intelligence to figure out the maximum number of syllables per bundle. Also, there is a one-to-one mapping between a pipelane and a syllable in each bundle in the binary resulting after compilation for the ρ-VEX[23]. The ρ-VEX, being a VLIW processor with dynamic issue width, would then require programs to be (re-)compiled each time its configuration changes, and somehow reload the (re-)compiled programs. This would spoil the whole purpose of introducing the ρ-VEX, and the latter is actually infeasible as restoring the context of an evicted binary into the new one, compiled for a different issue width, is next to impossible.

To overcome these difficulties, the concept of *generic binaries* was introduced in [27]. Generic binaries are compiled for the largest possible bundle size. Also, they can be executed by ρ-VEX configurations of any issue width. For example, for a binary compiled for 8-issue configuration of the ρ-VEX, the eight syllables in a bundle can (theoretically) be run in parallel by an 8-issue core, and sequentially by a 2- and a 4-issue core in 4 and 2 steps respectively. There are, however, disadvantages to this approach, for example, the restriction that the branch instruction is placed as the last syllable in a bundle, and performance hampering[23]. But the reconfiguration flexibility gained outweighs the disadvantages by a huge margin.

2.2.6 Trap handler in the ρ-VEX

There are many situations in which a processor needs to pause its execution to handle another event. The nature of such an event can vary from an external interrupt to a situation which the processor cannot handle, like encountering an instruction which it cannot execute. Such events are known as *traps* in the ρ-VEX terminology. The ability of a processor to handle such exceptions plays an important role in deciding the capability of running an operating system. While there are various other aspects of the ρ-VEX processor architecture to be described, the trap handler is one of the most important features, hence, it is chosen to be presented in details. For other details of the ρ-VEX architecture, [25] can be referred.

We split the study here between sources of traps and the way traps are handled. The sources of traps can be categorized into the following six types[25]:

- *Interrupts*
  When interrupts are enabled, and an interrupt occurs, a TRAP_EXT_INTERRUPT trap is generated. This trap causes the control to jump to the trap or panic handler, which has code to handle the event, like in an interrupt service routine.

- *Faults*
  A fault occurs when an instruction cannot be executed for some reason. It usually leads to halting the execution altogether, except in case of a page fault.

- *Context switch request*
  In order to help represent a change of a software context, two registers, CR_RSC

(Requested software context) and CR_CSC (Changed software context) are provided. These do not exist on context 0 and are hardwired to their equivalents on context 0.

The encoding for these registers depends on the user, provided the end result is that the memory location of the software context becomes known. When the values in these registers do not match and the context switching system is enabled by setting the C flag in CR_CCR (the main context control register), the TRAP_SOFT_CTXT_SWITCH trap is generated. After handling the context change, the trap handler routine should update CR_CSC with the value in CR_RSC.

- *Debug traps*
  These are generated when a hardware or a software breakpoint is reached. Depending on the contents of the Debug control register, the debug traps are handled in different manners.

- *The TRAP instruction*
  This instruction is used to explicitly generate a trap from a program. This is useful when a new functionality is to be implemented with the help of the trap handling mechanism.

- *The STOP instruction*
  This instruction is used to halt the execution of the core altogether. On encountering this instruction, the A TRAP_STOP trap is generated while the next execution is executing.

Usually when a trap occurs, the control jumps to the *trap handler*. A trap handler is a subroutine which contains code to handle the situation. The trap handler subroutine always ends with either a `RFI` (return from interrupt) or a `STOP` instruction. The address of the trap handler is to be stored in the CR_TH control register by the initialization code. In most of the cases of a trap occurrence, the software is responsible for storing the state of the processor's registers. After the trap is handled, it is expected that the processor state is restored by the software and the execution is resumed from the instruction before which the trap occurred.

The ρ-VEX processor identifies a trap with the help of contents of the trap cause (CR_TC) and trap argument (CR_TA) control registers. The trap cause register stores an 8-bit encoding for the trap cause and the trap argument register stores a 32-bit value depending on the cause of the trap. For a complete list of encodings for the different trap causes, [25] can be referred.

Another feature worth considering here is the *panic handler*. A major difference between the panic handler and the trap handler is that for the former it is not expected to return to the program. In case of a trap, the control can also jump to the panic handler. The *ready for trap* (R) flag in the CR_CCR register decides whether the control will jump to the trap handler or the panic handler. If the flag is set, the control jumps to the trap handler. Two such handlers are necessary to avoid loss of context for the first trap if two consecutive traps occur. The normal way of working is that the ρ-VEX clears the R flag in the CR_CCR register upon entering the trap handler, so that the next trap is handled by the panic handler.

2.3 Simrvex: The architectural simulator for the ρ-VEX

An architectural simulator is a software that models the behaviour of a computer such that its performance and output for a given input can be predicted[28]. Architectural simulators can be classified into different types depending on the following contexts[28]:

- *Scope*
  Simulators can be microarchitectural or full system simulators (emulators). A microarchitectural simulator simulates the processor only, while a full system simulator includes I/O devices, memory, drivers, etc. into its scope apart from the processor. Full system simulators can run software meant for the real system as it is.

  Microarchitectural simulators can in turn be classified into instruction set simulators and cycle accurate simulators. Instruction set simulators can simulate the behaviour of a real processor by reading instructions like the real processor and representing the registers via internal variables. Cycle accurate simulators, as the name suggests, simulate the design and behaviour of a processor at the microarchitectural level in a cycle accurate manner. The drawback is that these simulators are slow. However, they are useful for prototyping a new architecture and accurately predicting timing performance.

- *Detail*
  Under this context, simulators can be classified into functional and timing simulators. These focus on simulating as accurately as possible the functionality and timing behaviour of a processor respectively.

- *Nature of input*
  Simulators here can be classified into trace-driven and execution-driven simulators. Trace-driven simulators run a predetermined set of instructions with known inputs (like in a script), whereas execution-driven simulators allow for dynamic changes in instructions, depending on the nature of input.

Simrvex falls under the category of cycle accurate simulators. It was developed at TU Delft and is a heavily modified version of the architectural simulator for the `st200` family of processors, with all dynamic capabilities, control registers and other characteristics implemented as per the ρ-VEX concept. It is written in C, and can run on x86 machines after being compiled using `gcc`. As mentioned before, the `st200` family of processors is based on the Lx architecture, which is a 32-bit architecture and uses a scalable issue width VLIW technology. The architecture of `st200` is therefore very much similar to the ρ-VEX architecture.

Simrvex has an MMU integrated and this component is very much similar to the one used in the st200 simulator. To be specific, in simrvex the TLB (Translation Lookaside Buffer) carries out the memory translation and protection, the way an MMU does. An operating system with memory management support is what this project intends to port, hence a study of the TLB is necessary before proceeding with the implementation. This study is presented next. Only the TLB will be discussed in the context of simrvex,

as its other characteristics are similar to the ρ-VEX processor, and have already been discussed.

2.3.1 TLB in simrvex

The TLB in simrvex allows for experimenting with porting operating systems with memory management support. A TLB is like a cache for the system memory, in the sense, it stores the recent translations of virtual memory to physical memory[29].Most systems with virtual memory support resort to paging. Usually, the mapping between the (virtual) pages and page frames, which are the corresponding physical memory units, is done by a physical structure called *page table*. The more the number of accesses to the page table, the lower is the performance. This is where TLB comes into picture. It is a small hardware unit usually incorporated into the (hardware) MMU[3]. A TLB consists of a small number of entries, each containing some information about a page, and most importantly, its corresponding page frame. In most cases, the program generates the virtual address while referencing a piece of memory. This address goes to the MMU for decoding, instead of directly being sent on the memory bus, the MMU first checks whether the address is available in the TLB. If yes, and if the access is valid, the TLB returns the page frame. If not, the MMU finds the page frame from the page table and places it in the TLB after evicting another entry, following certain algorithm.

It is extremely important to note here that the discussion in the paragraph about explains TLB management in hardware by the MMU, but simrvex uses a software-managed approach. It leaves the task of adding TLB entries to the operating system. Also, in case of a TLB miss, a fault is generated (page fault) to notify the operating system, which takes the necessary page frame replacement action and then re-executes the instruction which led to the fault. When the TLB is software-managed the MMU becomes much simpler, which is why the entire memory translation and protection related operations can be carried out by the `tlb` unit itself, in simrvex. Therefore, while referring to MMU in simrvex, we are actually referring to the `tlb` unit. For TLBs with sizes greater than or equal to 64 entries, the performance of a software-based TLB management approach becomes acceptable[3]. Besides, various strategies have been researched which help improve the efficiency of the software-based TLB management approaches. It is also to be noted here that in the context of simrvex, since it is a piece of software itself, a hardware component concerning the ρ-VEX is actually a piece of (C) code which emulates the hardware.

Simrvex follows the TLB implementation very much in line with the one in the st200 simulator. Therefore, the specifics related to the simrvex TLB presented are referred from the st200 instruction set architecture manual[30] and by observing the simrvex code. Simrvex uses a small instruction TLB (ITLB), a small data TLB (DTLB) and a larger unified TLB (UTLB). The ITLB performs instruction address translations, while the DTLB performs address translation for data. Both of these act as caches for the UTLB. Changes in the UTLB are not reflected in the ITLB or DTLB, however, this is untrue for the other way round. The DTLB, ITLB and TLB control registers have to arbitrate for access to the UTLB, with the priorities of accesses being in the order: TLB control registers, DTLB and then ITLB. The TLB for simrvex currently has the

following configuration, with each 'entry' being 128 bits wide.

TLB component	**Number of entries**
ITLB	4
DTLB	8
UTLB	64

Table 2.1: TLB configuration for simrvex

As the ρ-VEX has a 32-bit architecture, the virtual address space becomes 2^{32} = 4Gbyte. The TLB in simrvex performs mapping between virtual and physical addresses with the help of pages of size 8KB. The TLB operation can be controlled by means of certain control registers. These control registers are not explicitly given any memory in the ρ-VEX architecture, but were accommodated (not under this project) in the unused memory space in simrvex to support the TLB. We discuss next these control registers and register flags which govern the operation of the TLB.

- The CCR_TLB_ENABLE flag
  This bit, on position 10 of the main context control register (CR_CCR), decides whether the TLB unit will be enabled or disabled. If this bit is 0 or disabled, the virtual addresses are treated as physical addresses.

- The TLB_ENTRYx registers
  Here, 'x' can be 0, 1, 2 or 3. These 4 registers together can map all the 128 bits of a TLB entry.

- The TLB_INDEX register
  This register is, like all other registers, 32 bits wide. Only the 8 least significant bits [7:0] are used. These bits represent the entry out of the 64 UTLB entries mapped to the TLB_ENTRYx registers.

- The TLB_REPLACE register
  This register is used by the software to decide which TLB entry to replace.

- The TLB_CONTROL register
  Only the two least significant bits of this register are used. Writing a '1' to the bit at the 0th position flushes the ITLB, while writing a '1' to the bit at the 1st position flushes the DTLB.

- The TLB_ASID register
  This register holds the 8-bit current process ID. Rest of the bits are unused.

- The TLB_EXCAUSE register
  This register stores the encoding for the cause of the exception when the TLB raises one.

TLB Exceptions

Knowledge of the TLB exceptions is vital for debugging purposes. The TLB raises an exception on multiple occasions, for example, if there is no mapping for a page available in the UTLB, or if there are multiple mappings, illegal access to a page, etc. When an exception occurs, the TLB jumps to the exception vector and updates accordingly the TLB_EXCAUSE, TLB_EXADDRESS and TLB_EXCAUSENO registers. When a DTLB exception occurs, the TLB_EXADDRESS register will contain the virtual effective address that caused the exception. When an ITLB exception occurs, the TLB_EXADDRESS register will contain the virtual address of the syllable that caused an exception. In the case of multiple ITLB exceptions, the exception with the lowest syllable address is thrown.

Simrvex, like any other open source processor simulator, offers a lot more to explore, particularly because of the source code availability. However, here the intention is to limit the discussion to the most important features. With the TLB of simrvex, the presentation of the background study of the platform concludes. We now move on to the software side of the study - the Linux kernel.

2.4 Linux

As already mentioned, the Linux kernel code is open source, which allows for studying and understanding the code to a great extent. It is written entirely in C and assembly. Besides, there are detailed standard references available. This makes Linux a suitable candidate for the port. The entire kernel operation is complex, and it is impossible to cover everything in this thesis. This is not even the intention, as there are standard references which provide in-depth information about the kernel. The discussion here pertains to those components which are deemed important to understand the bare minimum porting of the kernel to the ρ-VEX.

Figure 2.3 shows the different layers of operation in a typical Linux-based system. The hardware at the bottom consists of the CPU, I/O devices, etc. The operating system is in charge of directly controlling the hardware, while providing a clean system call interface to the programs running over it. System calls are made by writing to the registers, or sometimes on the stack, and issuing trap instructions to switch from user mode to kernel mode[3]. Trap instructions need to be written in assembly, so, procedures implementing them are written in assembly and are provided by the libraries. A system call from C invokes the corresponding procedure. The standard utility programs are something the user interacts with.

2.4.1 The Linux kernel structure

As can be seen from Figure 2.4, the kernel subsystems can be divided into three main components: the I/O component, the memory management component and the process management component. The virtual file system in the I/O component provides the

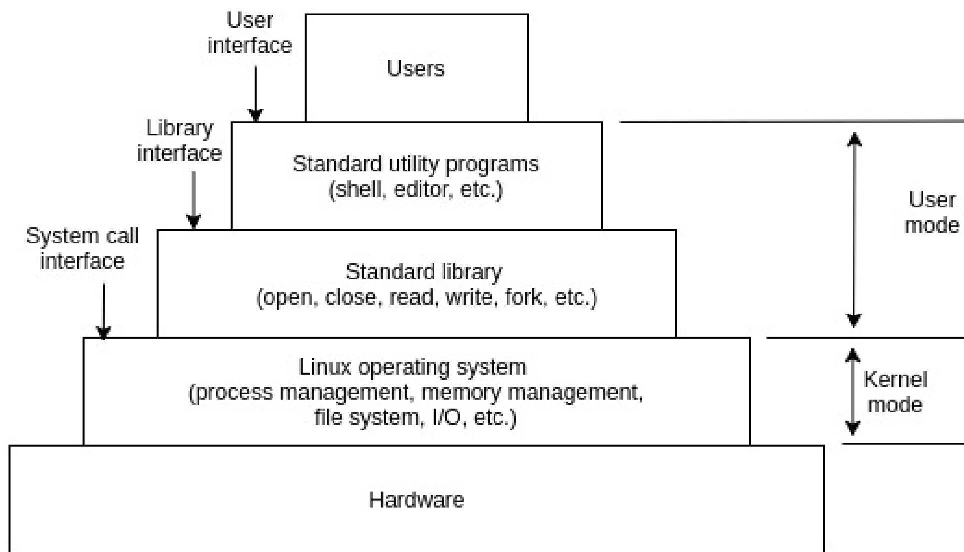


Figure 2.3: The different layers of operation in a Linux-based system[3]

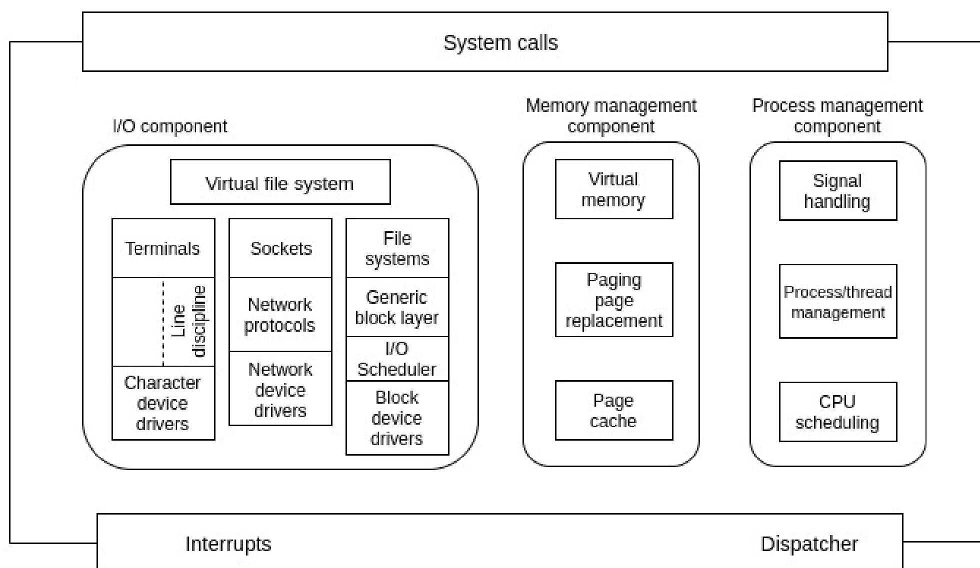


Figure 2.4: The Linux kernel structure[3]

necessary abstraction, such that the interaction with, for example, the terminal appears similar to the interaction with a file on the disk or in memory. The drivers located at the lower level take care of the necessary real interaction. All Linux device drivers are either character device drivers or block device drivers. Network device drivers are actually character device drivers but are handled differently[3]. The memory management component is responsible for maintaining the mapping between virtual and physical memory, handling page replacement and also maintaining a record of recently accessed pages. The processes management component deals with process and thread creation and destruction, scheduling of threads, etc. Signals, which are sort of notifications from the operating system, are also handled by this component. All these modules exhibit a high level of interdependency.

2.4.2 Processes and threads in Linux

A process is an instance of an executing program. It has its own address space, which means it has its own (virtual) memory[5], own set of variables and current values of the processor registers as well as open file descriptors. A copy of a process can be made using the `fork` system call[3]. In the process of *forking*, the original process is called the parent process, while the copy is called the child process. When a child process is forked from a parent process, it also gets its own address space.

Linux allows a process to create additional threads upon its creation[6]. Threads are like mini processes and share the address space of a process, and are said to be more lightweight when it comes to creation, destruction, and communication. In fact, processes can be considered to be comprised of one or more threads. Like `fork`, there exists a system call `clone` to create a new thread, either in the same or a different process.

Each thread gets its own stack and stack pointer, and registers (including the program counter) from within the memory allocated for a process, but all threads within a process share the same heap and other memory segments. The Linux kernel represents threads as *tasks*. Such tasks, with the information they contain are encoded as *task_struct* structures. A task_struct represents an entire execution context, with one object of type task_struct representing the state of and other information about a single thread. It is important to note here that Linux schedules threads and not processes, therefore, when a process is said to be running, in reality, one or more of its threads are under execution (Naturally, in a uniprocessor system, in reality, only one thread executes at a time, whereas in a multiprocessor/multicore system, multiple threads can be executed simultaneously).

A thread's program counter holds the address of the next instruction to be executed. Such program counters are actually logical program counters, provided by the operating system. When a thread executes, the contents of its logical program counter are loaded into the physical program counter available on the processor core[3].

For each process, there is a process descriptor, which is an object of type task_struct. When a process is created, a task_struct object is created, and also, memory is allocated

[5]Text segment(for instructions), data segment (for static and global data), BSS segment (for uninitialized data) and stack (for local variables, temporary variables, return addresses, etc.)[31].

[6]Upon creation, a process is said to have a single thread.

for the kernel-mode stack. Again, Linux does not differentiate between processes and threads, hence, when a thread is created, an object of type task_struct is created. However, threads share some elements of this task_struct structure if they lie within the same process. Linux can identify processes by means of process identifiers (PID), which are nothing but integers identifying a process, or rather, a *group of threads*.

We now seek to mitigate the confusion between kernel threads and user threads. The kernel and user modes of operation can be distinguished with the help of Figure 2.3. At the kernel level, there is no such thing called a process. As already mentioned, the kernel schedules threads. Every process has a user part, which runs user-level programs[3]. However, when a thread inside such a process makes a system call, it begins executing in kernel mode and thereby gets a different memory map. This is still the same thread, the *user thread*, running in kernel mode, with elevated access privileges. Threads created by Linux, and always running in the kernel mode are *kernel threads*. Such threads are not associated with any user process, but only with kernel-specific code. User threads usually map to kernel threads. This mapping can be one-to-one, many-to-one, or many-to-many.

2.4.3 Scheduling in Linux

Linux distinguishes the following three classes of threads[3], listed in the order of their priorities:

1. Real-time FIFO
2. Real-time round robin
3. Timesharing

A real-time FIFO thread can only be preempted by another real-time FIFO thread. A real-time round robin thread is similar to a real-time FIFO thread, except for the fact that the former has time quanta associated with it. If multiple real-time round robin tasks are ready, each gets a time duration for execution equal to the time slice. This list of round robin threads is maintained in a round robin fashion. For timesharing threads also, time quanta are assigned for each task, based on its priority. The priorities of real-time tasks are higher. Linux maintains 140 priority levels for all the types of tasks, numbered from 0 to 139 for highest to lowest priorities[3].

Linux uses a data structure called *runqueue*, per core, to aid in scheduling. A runqueue maintains two fields, *active* and *expired*, each of which points to an array of 140 list heads, each representing a priority level. Lists of tasks having the same priority are linked to list head representing the corresponding priority. A doubly linked list of tasks can thus be visualized. The scheduler selects the highest priority task from the active array to run. When the quantum for this task expires, it is moved to the expired list. If a task enters a blocked state, it is moved to another data structure called *waitqueue*. When the conditions for the blocked state are over, the task is brought back to the runqueue, with its time quantum adjusted to reflect the time spent during its earlier execution. The priorities for the tasks can be static or dynamic, the latter requiring an algorithm to continuously calculate its priority. The higher the priority of a task (the lower its priority level), the higher the time quantum assigned to the task. When the

queue pointed to by the *active* field becomes empty, the *active* and *expired fields* are swapped, so that the expired tasks now become active[3], and their execution continues, as Linux can only schedule runnable tasks.

2.4.4 The Linux kernel's notion of time

The kernel depends upon the hardware timer of the processor to keep track of time. To keep a record of passage of time is very important for the kernel, as many operations are time-driven. The hardware timer is usually a programmable digital clock, which works on an internal or an external crystal oscillator. It is also called *system timer*. When this timer overflows, an interrupt is generated. The kernel handles this interrupt using an *interrupt handler*[32] in a manner specific to the processor architecture. The time period between two such consecutive interrupts is called a *tick*[32]. Keeping track of such ticks helps the kernel maintain a knowledge of system up time and the wall time[7].

The frequency of the system timer is setup by the kernel during booting. This value is dependent on the processor and is called the *internal kernel time frequency*. The kernel maintains a global variable named `jiffies`, which keeps track of the number of ticks occurred since startup. This variable is important for many internal operations, like delay loop calibration, which occurs during booting.

2.4.5 The Linux boot process

The boot process - a series of steps occurring after powering on the processor, depends on the processor, but in general can be said to consist of the following steps[3]:

1. Hardware initialization through BIOS
   BIOS stands for Basic Input Output System. It is a firmware in the flash memory of the mother board and is responsible for performing the POST (Power On Self Test) to check for hardware integrity and also to load and execute the bootloader.

2. Reading and executing MBR
   MBR stands for Master Boot Record. It is the first sector of the boot disk. MBR contents are first loaded into the memory and executed. MBR contains a small program, which loads a standalone program called *bootloader*, located on the boot disk. The bootloader copies itself to a fixed memory location such that the specific region of memory reserved for the operating system is made available.

3. Loading of the kernel image into memory
   The bootloader, for example the GRUB (GRand Unified Bootloader), should understand the file system of the boot device. It finds the kernel image on the boot device and loads it at a specific location into the main memory. The bootloader then jumps to the kernel, and the kernel then takes over.

4. Mounting of the root file system
   The kernel does the job of mounting the root file system. Often, the initial RAM disk or the *initrd*, which is a temporary root file system, is mounted before the

[7] Wall time is the time taken by a task to complete.

actual root file system is available. A root file system is a file system located on the same partition of the disk as the root directory.

The kernel also performs important initializations of the internal data structures. Finally, it starts Process 0, which is also known as the *idle task*. This marks the beginning of the program execution in user space.

5. Jumping to user space
The init program represents the first user space program the kernel can execute. This program should be available on the mounted root file system, and its location can be made known to the kernel. The init program executes as the *init process*, which is actually the child process of Process 0. The init process gets a PID of 1 and gives rise to all the other processes, which execute a variety of user space programs, including the shell.

This marks the end of the booting process.

2.4.6 Memory management in Linux

The virtual address space which each process can access is 32GB for 32-bit machines. It can be said to consist of four segments: text segment, data segment, BSS segment and stack segment. The text segment contains machine instructions. The data segment contains initialized variables, strings and constants. The BSS segment contains uninitialized data, which is initialized to zero after loading. The BSS and data segment contents can be changed, but this is not true for text segment. Furthermore, for dynamic memory allocation, the system call `brk` is often used to alter the size of the data segment. The C library function *malloc* makes use of this system call internally[3]. The stack starts near the top of the virtual address space and grows down to zero. If the stack crosses the lower limit of the stack segment, a fault occurs, which is to be handled as a trap. In situations like this, the kernel can adjust the stack segment boundaries.

The purpose of implementing memory management in Linux is to allow multiple processes to run simultaneously by creating an illusion for a process that 32GB of address space is available to it and by ensuring that each process has its own virtual address space. For the kernel, *physical pages* are the basic units of memory management. Objects of type `page` structure represent a physical page or a page frame. Such objects are called page descriptors[3], and the kernel maintains an array of such descriptors called *mem_map*. Each page descriptor has a pointer to the address space it belongs to, among other information. The memory management operations by Linux can be described in the following manner[3]:

- **Physical memory management**

  Linux differentiates three physical memory zones:

  1. ZONE_DMA - For pages that can be used for direct memory accesses
  2. ZONE_NORMAL - For pages that can be normally mapped

3. ZONE_HIGHMEM - For pages with high memory addresses and which are not permanently mapped

The layout for these zones depends on the architecture. The kernel maintains a *zone* structure for each of these zones, called the *zone descriptor*. It contains information such as the number active and inactive pages in the corresponding zone. Furthermore, the Linux kernel divides the entire physical memory into three parts. In the first two, the kernel and memory map reside, paging is not performed. It is only the remaining part of the memory that is paged, i.e, it is divided into page frames.

- **Memory allocation mechanisms**
  Linux supports many memory allocation mechanisms, with a so-called *page allocator* being the main mechanism for allocating new page frames. Details about this mechanism can be found in [3].

- **Virtual address space representation**
  The virtual address space can be said to be divided into page-aligned regions. The properties of each such region are described by objects of type *vm_area_struct*. All such objects for a process are linked together in a list and are sorted on the basis of virtual addresses. This way all pages belonging to a process can be found.

- **Paging**
  With paging, the kernel can run a process without it being completely in physical memory. Only its descriptor and the page tables need to be in the memory, the process can then be scheduled. During the execution of the process, the pages of the text, data and stack segments can be dynamically loaded as required. Another process which helps in paging is the *page daemon*. This is the process 2 which is forked from Process 0, as is the init process. The job of page daemon is to periodically check the number of free memory pages. If this number is low, it is replenished by freeing more pages. Linux adopts a so-called *PFRA (Page Frame Reclaiming Algorithm)* to make more free pages available.

With the memory management functionality, we conclude the discussion over the Linux kernel.

2.5 Related Work

This section describes some other projects of similar nature, and also highlights the differences.

1. *uCLinux on ρ-VEX*
   The ρ-VEX already has a uCLinux ported to it. This is a 2.0 Linux kernel no MMU version. The work is detailed under [23]. Some important changes were made to the ρ-VEX toolchain and also a vectored trap controller was designed to support

the Linux port. These modifications are important for experimenting with porting other kernels on the ρ-VEX as well.

However, the current project aims to port an operating system having an MMU support to the ρ-VEX. The importance of memory management by the hardware and its support in the software have already been discussed.

2. *FreeRTOS on ρ-VEX*
The ρ-VEX also has FreeRTOS ported on it. The work is detailed under [33]. FreeRTOS was ported on the ρ-VEX processor to exploit its reconfigurability for real-time robotics applications. FreeRTOS is a real-time operating system kernel developed and licensed by MIT, USA[34]. It is a highly portable kernel consisting of only three C files, along with architecture-specific assembly code.

   FreeRTOS is more of a thread library than a complete operating system kernel[34]. An operating system, besides thread management, performs many other functions like memory management, implementing device drivers, and file handling. The current project aims at making ρ-VEX suitable for a wide range of applications, which need not necessarily be real-time. The emphasis here is more on providing near full-fledged functionality to the ρ-VEX, like dynamically loading applications located on the file system, memory protection for applications running simultaneously, etc.

2.6 Conclusion

Porting an operating system to a processor requires a thorough knowledge of the processor architecture, the toolchain capabilities, and of course, the operating system code itself[8]. This is the reason a study for these three components was presented in detail in this section. The contents of this chapter were presented with the intention of making the understanding of the implementation, presented in Chapter 3, easier.

The discussion related to the toolchain part was kept short, and was emphasized more on a study of the visible (and known) capabilities, rather than internal working. Implementing toolchain modifications was not a desired component of this project. The emphasis is more on modifying the operating system code to make it suitable for the current ρ-VEX system as much as possible, before considering toolchain modifications. However, knowledge of the toolchain limitations is important to consider modifying the operating system for the port, and will be presented along with the implementation for a better reference. The functionality/characteristics of the ρ-VEX visible to the programmer, including some details of its internal working to gauge the impact of changing such functionality through program, were considered.

As far as discussion over the Linux kernel is considered, efforts were put into discussing the required details, keeping in mind audience having some basic operating system background (particularly UNIX-based operating systems). This is not to say, however, that the discussion about the Linux kernel presented in this chapter completely covers the background for the implementation, or completely lies within the scope of the

[8] At least the part interacting with the hardware

implementation, but the overlap is high. Much more detailed and extensive documentation about the Linux kernel can be found online, and in standard references. The same goes for the discussion about the ρ-VEX characteristics and the TLB unit presented in this chapter - the ρ-VEX user manual[25] and the ST231 Core and Instrucion Set Architecture manual[30], respectively, are more detailed sources of reference, and relevant topics out of these manuals were chosen for discussion.

Finally, a discussion about projects related to this one was presented, while highlighting the key differences.

3 Porting Linux to ρ-VEX

Availability of the source code of an operating system kernel is a prerequisite to porting the kernel to any architecture. Linux is the most popular open source operating system kernel available today. In fact, it is the largest open source project in the world, with a plethora of documentation available. This is not however the only reason for choosing Linux for the port. Factors which strengthen the choice of Linux, and the choice of the particular Linux kernel version will be discussed in this chapter first. As can be recalled, the ρ-VEX is implemented in hardware (on FPGAs and as an ASIC) as well as software (the simrvex architectural simulator). The choice of the ρ-VEX *implementation* and its justification also lies within the scope of this chapter.

This chapter, however, mainly describes the implementation of this project, i.e, modifications in the kernel code to make it run on the ρ-VEX. The entire implementation is in line with the goals mentioned in Chapter 1.

3.1 Choosing the ρ-VEX platform

Although in literature platform can mean different things, in this context, it strictly refers to the processor hardware or simulator as seen by the operating system running on top of it. As discussed in Section 1.2, one of the strong motivations behind this project is to make applications run in virtual memory on the ρ-VEX. Advantages of using virtual memory have been discussed before. The ρ-VEX currently has FPGA (on ML605 and VC707 FPGA development boards) and ASIC implementations. However, these lack a memory management unit. A hardware-based memory management unit for the ρ-VEX has been implemented under [35], however, it is currently not integrated in the release version of the ρ-VEX. As discussed in Chapter 2, simrvex has an MMU integrated, which is similar to the one in the architectural simulator for the st200 family of processors. The presence of an MMU in simrvex proves to be a huge advantage as it saves the time to implement an new MMU from scratch, or to integrate one in the hardware and then verifying it. The time thus saved can be effectively invested in the main objectives of this project. Apart from this, simrvex, being an architectural simulator of the ρ-VEX processor, it possesses the following advantages:

- Simrvex is written in C and its source code is available. This makes understanding its working easier through a code walk through, and also through implementing `printf` statements and tracing the execution flow via terminal output.
- If at all any changes are required in any of the components of the processor to support the port, they can be easily implemented. Moreover, such modifications are convenient to test - the simulator just needs to be compiled and rerun.

- Debugging is much easier on simulators than on hardware processors.
- Simrvex is a cycle accurate simulator for the ρ-VEX, therefore, it mimics the timing performance of the ρ-VEX accurately[1]. This way, it is possible to accurately report the execution cycles consumed by user space applications running on the ρ-VEX. Performance evaluation of applications running on the operating system after it is ported to the ρ-VEX is also one of the goals of this project.

These advantages make simrvex the most suitable candidate among the available ρ-VEX processor implementations to carry out the operating system porting, and therefore, was chosen for the task.

3.2 Choosing the operating system kernel for the port

Writing an operating system from scratch is a tedious task. An interesting alternative is studying existing and mature operating system codes and modifying them to suit the ρ-VEX platform. For this, the source code of the operating systems is desired to be free or open source. The Linux kernel is the largest open source project in the world, with a plethora of documentation online, and its availability further motivates this project. A kernel is the core component of an operating system. Linux itself is an operating system kernel, while distributions, such as Ubuntu, package this kernel along with essential utilities and thereby qualify to be called a complete operating system. This project mainly aims to port the kernel to the ρ-VEX architecture, while adding essential utility support to test the performance of standard applications.

The fact that the Linux kernel code is open source is not the only reason it for choosing it as an operating system kernel for the ρ-VEX. Linux has been ported to many standard architectures like ARM, x86, MIPS, etc., and its kernel code is available with such architecture-specific code included. This was sought to be an important reference while writing the architecture specific code for the ρ-VEX. The other reason for choosing Linux is its modularity. The architecture-specific code in the kernel is well separated from the architecture-independent code[2][36]. This makes it easier in the sense only the architecture-specific code needs to be written for the new architecture. The code concerning features such as file system management, scheduling, networking, device drivers, etc. need not be re-written for a new port (although these require some functionality in the architectural specific code to be implemented).

The toolchain of the ρ-VEX system has matured over the years. [23] also implements a trap controller in the ρ-VEX processor to support interrupts and exceptions. These features enable further explorations of experimenting with operating system ports. The ρ-VEX compiler, `rvex-gcc`, is a modified version of the gcc compiler, as discussed in Chapter 2. The entire Linux kernel code is written in C and assembly, which makes it a favourable choice.

[1] Provided, caches are disabled.
[2] Actually, little overlap

Now that *simrvex* has been chosen as the platform over which the Linux kernel is to be ported, it is important to choose the version of the Linux kernel to be modified to implement the port. This is because Linux is being developed by many companies and individuals all over the world, and has many releases till date, and the existing ρ-VEX toolchain may not be able to support all Linux kernel versions. The version of the Linux kernel modified for the ρ-VEX is `2.6.32`. Versions 2.6 onwards have matured documentation availability, but more importantly, the same version of Linux has been ported to the `st231` processor, which belongs to the `st200` family of processors. Again, as discussed in Chapter 2, `st200` has a similar architecture to that of ρ-VEX, and it is interesting to further recall that simrvex is based on the architectural simulator for the `st200` series of processors. The idea of modifying this very Linux kernel port is therefore considered for the work to fall under the scope of a master's thesis. Also, the efforts thus saved are deemed to be utilized in exploring the dynamic properties of the ρ-VEX.

3.3 Porting Linux on simrvex

According to [37], there can be three different types of porting, when it comes to porting Linux to a processor:

1. Porting Linux to a new processor, but with an architecture already supported by Linux.
2. Porting Linux to a new processor with an architecture not directly supported through standard Linux ports, but falling in the same family as a supported architecture.
3. Porting Linux to a new processor with an entirely new architecture.

The implementation under this project can be said to belong to the third category above, as we are dealing with a completely new architecture here[3]. Although, as mentioned in Section 3.2, the task comprises of modifying the `st200` port, the implementation belongs to category 3, as the ρ-VEX architecture does not lie under the `st200` family of architectures, even though it is similar.

As mentioned in Section 3.2, Linux kernel has a modular structure, with the architecture-specific code separated from the architecture-independent code. This is what makes Linux *portable*. Therefore, all the steps required to modify the Linux kernel version 2.6.32, modified for the st200 architecture, can be broadly divided into the following two parts:

1. Making the architecture-independent part of the Linux kernel code compilable for the ρ-VEX.

[3] Meaning an architecture entirely different from the standard ones to which Linux has already been ported.

2. Writing the architecture-specific part of the Linux kernel code and making it compilable and working for the ρ-VEX.

Before discussing the above two steps in detail, a discussion about the initial setup is necessary. With the ρ-VEX toolchain and the source code of the Linux kernel available, the first step is to set the kernel configuration. This configuration decides which modules of the kernel will be compiled, while taking care of the dependencies. From the `linux/` directory, the following command needs to be run, in order to start configuring the kernel.

```
make CROSS_COMPILE=rvex- menuconfig
```

An interactive screen appears, allowing to selectively enable the required kernel components. The Linux kernel source code is huge, and apart from the essential operating system components, includes code to support, for example, different file systems, USB devices, display devices, networking, etc. Rather than going for such *extra* features, the focus here is to have a minimalistic kernel comprising of the essential operating system functions and support for running standard C programs. This would ease achieving the main objectives of this project, saving (potential) debugging time. Keeping this in mind, only the very basic required components are enabled for compilation.

3.3.1 Modifications to the architecture-independent part of the Linux kernel

Most of the architecture-specific code in the Linux kernel lies in a specific directory under the `linux/arch` directory[4]. Most of the remaining architecture-specific code lies under a directory called `linux/include` meant to contain all the header files. Almost all of the other code forms the architecture-independent part of the Linux kernel. The strategy applied here, in order to make the architecture-independent part compatible with the ρ-VEX, was to write minimal architecture-specific code as required for compilation, and then try to compile the entire kernel code to eliminate/fix non-compilable components. The process is important as it exposes non-compatibilities with the compiler early, and it also makes debugging the architecture-specific code related compilation and linking errors manageable.

To begin with the minimal architecture-specific code, a directory named `rvex` was created under path `linux/arch/`. This is where most of the architecture-specific code resides. Inside this folder, subdirectories were created following a standard layout. Some of these are listed below, along with the functionality of the code contained in them.

- `include/asm/` - for header files which are required by the kernel source code
- `kernel/` - functions for management of core kernel operations
- `mm/` - code for memory management
- `lib/` - some important utility routines

[4]Here, linux is the name of the directory which contains the entire kernel source code.

The necessary files, with bare minimum code were included within these directories, including the Makefiles. A Makefile is a file which defines certain compilation and linking rules on UNIX-based systems. Linux follows a recursive build structure, meaning, there is a top-level Makefile in the `linux/` directory, which looks for Makefiles in appropriate subdirectories and thus carries out the build process in a branching fashion. The whole kernel can be built by running the following command from inside the `linux/` directory:

```
make CROSS_COMPILE=rvex- ARCH=rvex
```

An important part of the architecture-specific code that needs to be added to proceed with the compilation is the `vmlinux.lds` script. This script, along with specifying the format of the build result, specifies which piece of code goes in which segment of the memory. The script from the `st200` port was directly used with minor modifications related to the output binary format, the start address of physical memory, the entry point function name, the base address of control registers, etc.

It is important to note here that since many of the needed architecture-specific modules would be missing, many linking errors would be encountered. In order to tackle only the compilation incompatibilities first, relocatable linking was adopted. The linker can be told beforehand that relocatable linking is to be performed by passing a `-r` flag to it. With the relocatable linking option enabled, the final executable, or rather object file, does not include the object files generated from the source code files of the submodules, and is therefore not executable. Basically, references to the symbols in the object files of the submodules are not yet provided correct memory addresses. Therefore, linking does not happen and naturally, no linking errors were reported.

With relocatable links creation enabled, the compilation for the entire kernel, along with a small amount of architecture-specific code, was proceeded with. The following techniques were employed in order to resolve the errors encountered, depending on the nature of the errors:

1. Changes were made in the kernel source code wherever possible, in order to make the code compilable for the ρ-VEX.
2. When 1. was not possible, different compiler versions were tried.
3. When 2. also did not work, the kernel configuration was adjusted to remove the error causing components which were not crucial for core kernel functionality.

A noteworthy modification at this stage was removing the optimization option and the debug symbol generation option from the top Makefile. These are currently not supported by any of the ρ-VEX toolchain versions. Also, dynamic loading of libraries is not possible with the current toolchain, therefore, the `vDSO` feature from the st200 port had to be disabled. "vDSO" stands for *Virtual Dynamic Shared Object* and is a shared library maintained by the kernel (from versions 2.6 onwards[38]) to speed up certain system calls.

With the compilation errors now resolved, completing the architecture-specific code was pursued, and also, the option to create relocatable links was removed to resolve further linker incompatibilities.

3.3.2 The architecture-specific kernel code

As the porting is based on the `st200` Linux port, the entire architecture-specific code from the `st200` was taken and a register translation was performed on it by running a script. Most of the architecture-specific code in Linux is written in assembly, which makes use of registers specific to the processor architecture. Although the layout of the registers in the st200 architecture is broadly similar to that in the ρ-VEX architecture, it is not exactly the same. Equivalent registers exist in the ρ-VEX in most cases, and also, in some cases, there is no one-to-one mapping for a particular group of registers. This was taken care of by the translating Python script and was possible due to the availability of the register translation table between st200 and ρ-VEX registers. This table, which can also be found in the `README` file of the `open64-rvex` repository, which is used by TU Delft to maintain the source code of the ρ-VEX compiler, is presented below.

ST200	ρ-VEX	Register type
r0	r0	general-purpose
r1 - r7	r57 - r62	callee-saved
r7	r63	stack pointer, but not always used
r8 - r11	r11 - r56	scratch
r12	r1	stack pointer
r13	r55	thread pointer
r14	r56	global pointer
r15	r2	struct/union return pointer
r16 - r23	r3 - r10	arguments
r24 - r62	r11 - r56	scratch
r63	l0	link register

Table 3.1: Register translation scheme for st200 to ρ-VEX[4]

After the the register translation, further modifications to the st200 code were pursued in steps. The relocatable links creation option was also disabled, so as to deal with linker errors on build attempt.

The next step was to identify the interfaces between the architecture-specific and the architecture-independent code. The header files to be included under `linux/arch/rvex/include/` would provide such interfaces. As Linux has evolved a lot over time, with porting being done on many architectures, developers have aggregated most commonly used header files under the directory `linux/include/asm-generic/`. The task here was to examine these files, along with the custom header files for st200, and come up with custom header files for the ρ-VEX, wherever required. Fortunately, most of the declarations specific to the ρ-VEX were present under the header file `rvex.h` written during previous ρ-VEX-based projects, and this header file was appropriately included in the architecture-specific code.

The architecture-specific code in Linux can be broadly divided into two categories[37]. The first part deals with the booting process, i.e, from the moment the kernel comes into picture till *init* is executed. The second part deals with operations post booting,

like thread management, interrupt handling, handling of other system calls, etc. The implementation pertaining to the first part will be covered in this chapter, as running the *init* typically signifies the end of porting[39]. The implementation pertaining to the second part comes under experimentation to evaluate the timing performance of benchmarks. This will be briefly discussed in Chapter 4.

The boot code

Before starting with the boot code implementation, it is important to identify the boot sequence for the ρ-VEX processor. The general booting process was discussed in Chapter 2, however, since the porting is done on a simulator, a bootloader is not required. The starting memory address can therefore be kept as 0, specified by the option `CONFIG_MEMORY_START` in the kernel configuration file. Also, the discussion here is of a more microscopic nature as compared to the general booting process discussed in Chapter 2, which means the functions called during the booting phase will be discussed here.

Many functions are executed during the booting process, with some of them being architecture-specific. The task was to identify these architecture specific functions and modify/implement them accordingly.

The boot sequence, once the kernel takes over, starts with a function written in assembly called `_start` in the file `head.S` located under `linux/arch/rvex/kernel/`. The `head.S` for the ρ-VEX was formed by referring to the same file for the st200 architecture and the `mmu_start.S` file, developed under another project to test the MMU on the simrvex. The documentation in these files in particular was helpful in identifying the required sequence of activities to be performed. The sequence is as noted below:

1. Clear and initialize the TLBs
2. Clear the BSS section of the kernel
3. Initialize the stack pointer
4. Jump to the `start_kernel` function

The caches are not activated throughout the project so as to maintain the cycle-accurate property of the simulator and justify the timing performance of benchmarks. Therefore, initialization of the caches is not required in the startup code. The MMU is activated just before initializing the BSS section. Before that, the accesses to the kernel symbols, all of which lie in the kernel's virtual address space[40], are done by means of a macro, `PHYS_ADDR`. This macro subtracts the start of the kernel address space appropriately to obtain the physical address. For the ρ-VEX, the kernel's virtual address space spans the higher two GB of the 4GB address space. This essentially results in the kernel's virtual address space starting from address `0x80000000`.

It is particularly important to clear the BSS section of the kernel, as it contains uninitialized data, and the state of the memory is unknown during startup. The function *start_kernel* represents the first architecture-independent function in the boot process,

and performs most of the subsystem initialization. However, many of the functions called from *start_kernel* require architecture-specific implementation. The important ones are described below[40].

1. setup_arch() - This function starts with an attempt to register the available console. This is done by the console driver. The st200 port already has the code for this driver. The driver for the ρ-VEX is almost entirely based on this, along with minor modifications. The console initialization, performed by the console driver, is important the obtain `early printks` - the first step towards a debug infrastructure. A `printk` is a function used exclusively by the kernel to print messages on an available console, thereby serving as an important kernel-level debugging mechanism. To enable these early printks, however, the configuration option `CONFIG_SIMULATOR_CONSOLE` was enabled.

The *setup_arch()* function also sets up the page table and performs the initialization of the trap handler[5]. The code for the page table set up was taken as it is from the st200 port, as the ρ-VEX uses almost the same TLB mechanism. The page size used is also the same, and equal to `8192 bytes`. The code for the trap handler, written in assembly, required modifications mainly taking into account the register layout difference between st200 and ρ-VEX, and also some modifications to eliminate incompatibilities with the assembler.

2. mem_init() - This function zeros the *zero page*[6] It also initializes certain global variables, representing boundaries of the different segments of the memory, to appropriate memory address values. The code from the st200 port, with certain changes in the macros used, does the job for the ρ-VEX. However, simrvex does not have the Speculative Control Unit (SCU) which the st200 series of processors possess. Therefore, code related to this unit had to be deactivated.

3. init_st200_irq() - This function, specific to the st200 family of processors, was analyzed, and it was found that it could be used without major modifications due to large similarities in interrupt handling techniques by the two architectures. Basically, this function tries to map the interrupt controllers to the kernel's address space.

4. time_init() - The main purpose of this function is to initialize timer parameters as well as timer control registers. It also performs memory mapping of the clock source driver. Again, the code from the st200 port was usable due to similar timekeeping infrastructure followed by st200 and the ρ-VEX. However, there a few important modifications were identified. Fortunately, the code for simrvex revealed a lot about the timekeeping infrastructure followed. The information about the macros to be used to write to the registers, as well as the appropriate offsets of the register addresses in the memory map of the ρ-VEX's register layout, were gathered from here and [25]. For example, it was noted that there are four timer control registers in simrvex, the

[5] This means providing address of the ρ-VEX's trap handler routine to its trap handler control register

[6] A zero page is the first page mapped to kernel memory which exists to trap null pointer references in the kernel [41].

corresponding timer IRQ descriptor[7] is mapped to IRQ line 8, and so on. Changes to the interrupt infrastructure (e.g., enabling and disabling the interrupts) also follow along a similar line - writing to and reading from registers with appropriate memory address offsets, and following ρ-VEX's convention for macros, as observed from the simrvex code.

The last function called by *start_kernel()* is *rest_init()*. A successful call to this function indicates that it is possible to execute in virtual memory[39]. It is here that two kernel threads are created:

1. `kernel_thread`
2. `kthreadd`

The `kernel_thread` is spawned first, after which it waits for the creation of `kthreadd`. The `kthreadd` is actually a kernel thread daemon[8], which gives rise to other kernel threads when required. After the `kthreadd` is created, `kernel_thread` proceeds with executing the remaining part of the boot process[39]. This includes certain device driver initializations, unpacking and mounting of the provided initial root filesystem [39], etc. Here, the involvement of the architecture-specific code is only for the device drivers. The kernel needs to be provided with an initial root filesystem, which it mounts in memory. This initial filesystem should contain the *init*. A successful execution of the *init* marks the end of the porting process. Setting up the *init* is a part of experimentation to (ultimately) execute and obtain timing performance of benchmarks. Details regarding this are discussed in Chapter 4.

Ported kernel components

Figure 2.4 provides a block diagram representation of the components of the Linux kernel. This figure can be adapted to show the components of the Linux kernel actually ported as a result of the implementation discussed so far. The corresponding representation is shown in Figure 3.1. This is actually a very early stage for considering input-output handling capabilities. The ported kernel lacks essential driver support. Besides, at such an initial stage, before going for I/O testing, important features like process handling, memory management, etc., need to work correctly, in order to proceed with the desired benchmark evaluation. Therefore, the entire I/O block was removed from the kernel configuration (as can be seen by dashed-line boxes and lightened text in Figure 3.1) while modifying the st200 port for the ρ-VEX, and the process and memory handling related blocks were retained and modified. Even support for file systems was not considered as it was not required yet. The entire implementation of the boot code, as discussed previously, has effectively equipped the kernel[9] with the ability to execute in virtual memory (through memory management support implementation),

[7]An IRQ is an external interrupt request to the processor and interrupt descriptors are basically structures maintained by the kernel internally to store the description of an interrupt.

[8]A daemon is basically a background process.

[9]theoretically, yet

and handle tasks as independent execution units. The architecture-specific code port to support the C functions, especially the ones called from the *start_kernel()* function, has resulted in this. Some other noteworthy features of the kernel resulting from the port are availability of system calls and timer and interrupt handling. One important point to note here is that the kernel was not yet built, therefore, a minimalistic kernel, with bare-essential working features was very much desired to efficiently handle unforeseen compiling/linking errors. Due to this, even important components like the implementations of different scheduling policies were not included. The kernel, as of now, is only capable of performing time-sharing of the CPU among the different runnable tasks. Of course, *scheduling* is an important area for exploration, but at this stage, the time-sharing feature was deemed sufficient for benchmark evaluation, which will be discussed in Chapter 4.

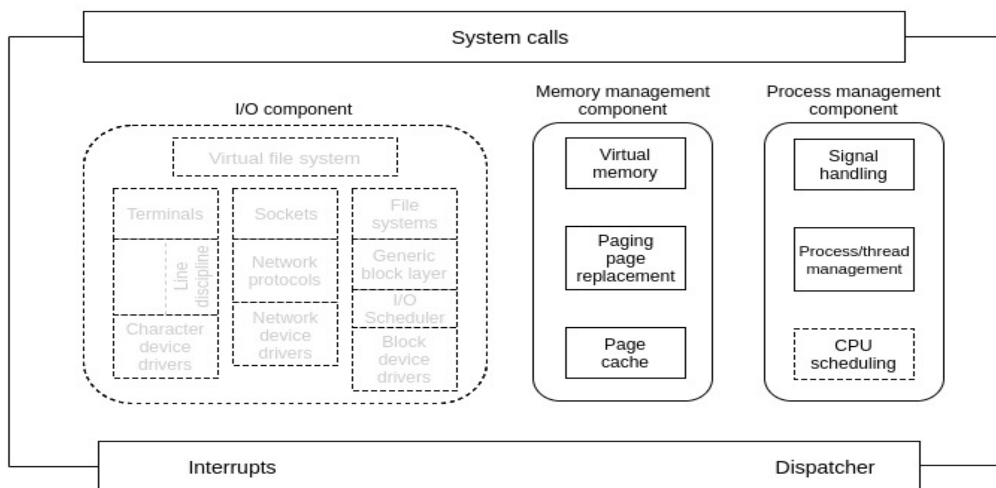


Figure 3.1: Components of the Linux kernel structure that were ported[3]

Overcoming linker limitations

So far, the implementation of the boot code was discussed, something which is expected to lead to the execution of the *init*. However, as can be recalled, overcoming other toolchain (especially linker) limitations, after disabling the relocatable link creation option from the top-level Makefile, was remaining. This was done, and the entire kernel was compiled. Many linker errors were encountered, which can be placed into two categories. These categories, along with the solution approaches are described below:

1. Relocation errors - An example of such an error is presented below:

```
kernel/built-in.o: In function ‘hrtimer_nanosleep’:
```

```
(.text+0x4772c): relocation truncated to fit: R_RVEX_BRANCH against
'.sched.text'
```

To debug such an error, e.g., the one presented above, the object file `linux/kernel/builtin.o` was dissembled using the tool `rvex-objdump` from the ρ-VEX toolchain. This resulted in the following log:

```
4772c: 22 00 00 02 call l0.0 = 47730 <hrtimer_nanosleep+0x160>;;
```

The call was made from inside the function 'hrtimer_nanosleep' to a function 'do_nanosleep' which has the signature:

```
static int __sched do_nanosleep(struct hrtimer_sleeper *t, enum
hrtimer_mode mode)
```

The '__sched' indicates that the function needs to be placed in the *sched* region in the final object file (this is specified in the custom linker script `vmlinux.lds.S`). This error is usually caused when the code becomes too large for call/br/goto instructions, in this case, the (large) value 47730 being stored in the link register. To overcome this error, ideally the link register should be made capable of handling larger offsets, by making appropriate modifications in the assembler. However, for this project, a preference is given to making changes in the kernel code, due to the high impact of changes in the toolchain. Therefore, the direct function calls were made indirect through the use of function pointers. For example, in the above case, inside the function 'hrtimer_nanosleep', for invoking the function 'do_nanosleep', a function pointer was created using appropriate signature, and the function was called using this function pointer. This essentially causes the compiler to generate `icall` instructions instead of `call` instructions. The `icall` instruction is used for dynamic function calls or calls to functions that cannot be reached using the branch offset immediate method[25]. The same approach was adopted for all the errors, and 746 such section mismatches were resolved for the entire code.

2. Undefined reference to function errors, in spite of the presence of function bodies - An example of this type of error is given below:

```
kernel/built-in.o:(.text+0x1168): undefined reference to
'__xchg_called_with_bad_pointer'
```

The above error was specific to the file `pgtable.c` located under `linux/arch/rvex/mm/`. This error occurred due to a bug present in gcc, upon which the ρ-VEX compiler is based. This bug causes the compiler not to generate code for functions which it thinks can never be called. The workaround for this bug was discussed in [42] and involves adding the following function declaration in the same file:

```
void __attribute__ ((weak)) __xchg_called_with_bad_pointer(void)
{ panic(__func__); }
```

> In this manner, dummy functions are made available, in case compiler tends to generate code to call these functions[42]. The other linker errors belonging to this category were fixed using the same solution.

With these approaches, the entire kernel compiled successfully to generate the expected `vmlinux` executable.

3.4 Conclusion

This chapter dealt with the implementation details for this project. The choice of the ρ-VEX simulator, simrvex, as the platform, and modifying the st200 Linux port for the ρ-VEX, were justified. Thereafter, the porting of this chosen Linux kernel on simrvex was discussed in detail. Although, there can be many approaches to porting, and the approach presented may or may not have been employed before, but the end result (porting) was achieved. The approach followed under this project for porting took advantage of the modular nature of Linux and its separation of the architecture-specific code from the architecture-independent code. The architecture-independent part was dealt with first by writing the bare minimum code for the architecture-specific part for compilation's sake. Also, at this stage the relocatable linking option was enabled ensuring objects were not linked into the final kernel object upon successful build, so as to approach the toolchain incompatibilities in a systematic manner. The compiler errors were resolved through a combination of kernel source code changes and compiler version changes. After the unlinked final kernel image was successfully built, which essentially meant that compiler-related incompatibilities were resolved, the architecture specific code was produced through heavy modification of the st200 Linux port. Thereafter, linker errors were tackled after disabling the relocatable linking option, which allowed exposing the linker's limitations while attempting to link the entire kernel code. All these modifications resulted in the successful generation of the Linux kernel image `vmlinux` built using the ρ-VEX toolchain, and meant to be executed on simrvex. This chapter only dealt with implementing the kernel modifications in order to boot Linux on simrvex, the experimentation to test this boot, by executing some useful user space programs will be discussed in the next chapter.

4 Experimentation and Results

In this chapter, the results obtained after experimentation with the implementation presented in Chapter 3 will be discussed. The implementation done to produce the results for evaluation will also be discussed. So far, the implementation discussed pertains to making the kernel compilable and bootable on simrvex. Of course, after the kernel was compiled successfully, debugging efforts were involved in making the kernel bootable on simrvex. The fixes were mainly concerned with the architecture-specific initialization part of the Linux kernel, and were presented together with the implementation/modifications required to make the kernel compilable, in Chapter 3. Before starting with execution-based results, it is important to describe the dissection of the Linux kernel image, to review which part of the code went where in the kernel image after a successful build.

4.1 The Linux kernel image

The end result of a successful compilation of the Linux kernel is the generation of the binary called `vmlinux`. The size of this binary is `1957.3 kiB`. In order to display more information about object files compiled for the ρ-VEX, the `rvex-objdump` and `rvex-elf32-readelf` tools are available with the ρ-VEX toolchain. Vmlinux is an ELF (Executable and Linking format) executable. Compilers or linkers produce binary files in this format for UNIX-based systems. Studying the details of binary files provides a better insight into the internal working of an operating system. The `rvex-elf32-readelf` tool was used to extract information from the generated `vmlinux`. The information obtained can be divided into three parts:

1. ELF header
2. Section headers
3. Program headers

The ELF header information for the vmlinux can be found in Figure 4.1. Most of the information under the ELF header is specific to the ELF format, such that the file is identifiable as an ELF executable. A notable point is that ρ-VEX is a 32-bit big endian architecture, and the executable complies with this.

```
ELF Header:
  Magic:   7f 45 4c 46 01 02 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, big endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           <unknown>: 0xffab
  Version:                           0x1
  Entry point address:               0x2000
  Start of program headers:          52 (bytes into file)
  Start of section headers:          2003280 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         2
  Size of section headers:           40 (bytes)
  Number of section headers:         25
  Section header string table index: 24
```

Figure 4.1: The ELF header information for the vmlinux

The next category is section headers, which provide information regarding different sections present in the object file, like BSS, stack, heap, text, etc. Figure 4.2 shows this information specific to the vmlinux generated.

```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .empty_zero_page  PROGBITS        80000000 002000 002000 00  WA  0   0  1
  [ 2] .text             PROGBITS        80002000 004000 173c18 00  AX  0   0 32
  [ 3] .cpuinit.text     PROGBITS        80175c20 177c20 000d40 00  AX  0   0 16
  [ 4] __ex_table        PROGBITS        80176960 178960 000140 00   A  0   0  4
  [ 5] .rodata           PROGBITS        80177000 179000 01095c 00   A  0   0  4
  [ 6] .init.rodata      PROGBITS        8018795c 18995c 000181 00   A  0   0  1
  [ 7] __param           PROGBITS        80187ae0 189ae0 000520 00   A  0   0  4
  [ 8] .data             PROGBITS        80188000 18a000 021710 00  WA  0   0 8192
  [ 9] .cpuinit.data     PROGBITS        801a9710 1ab710 000040 00  WA  0   0  4
  [10] .ref.data         PROGBITS        801a9750 1ab750 000638 00  WA  0   0  4
  [11] .data.init_task   PROGBITS        801ac000 1ac000 004000 00  WA  0   0  8
  [12] .init.data        PROGBITS        801b0000 1b0000 001288 00  WA  0   0  8
  [13] .init.setup       PROGBITS        801b1290 1b1290 0001bc 00  WA  0   0  4
  [14] .taglist          PROGBITS        801b144c 1b144c 000020 00  WA  0   0  4
  [15] .initcall.init    PROGBITS        801b146c 1b146c 000104 00  WA  0   0  4
  [16] .con_initcall.ini PROGBITS        801b1570 1b1570 000008 00  WA  0   0  4
  [17] .init.ramfs       PROGBITS        801b2000 1b2000 000200 00   A  0   0  1
  [18] .bss              NOBITS          801b4000 1b2200 046e70 00  WA  0   0 32
  [19] .stack            NOBITS          801fae70 1b2200 0787b8 00  WA  0   0  1
  [20] .heap             NOBITS          80273628 1b2200 078010 00  WA  0   0  1
  [21] text              PROGBITS        00000000 1b2200 000008 00      0   0  1
  [22] .symtab           SYMTAB          00000000 1b2208 01c6b0 10     23 3979  4
  [23] .strtab           STRTAB          00000000 1ce8b8 01a7b3 00      0   0  1
  [24] .shstrtab         STRTAB          00000000 1e906b 0000e2 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  p (processor specific)
```

Figure 4.2: The section headers inside vmlinux

The script `vmlinux.lds.S` aids in providing specifics for the generation of these sections for the executable, conforming to the ELF format. If some function needs to be

assigned to a specific section, it can be specified in its signature. For example, functions with `__sched` in their declaration are inserted into the .sched.text section in the object file. The placement of data and functions into such sections ensures that they are accessed in intended ways only. The Linux kernel code is huge and complex, and hence difficult to debug if an unwanted access causes an undesired behaviour. For example, functions declared as `__init` are placed in the `.init.text` section, and can only be invoked during the initialization phase of the Linux kernel. Such separation also helps Linux decide the release of allocated page frames when needed[43]. As can also be observed from Figure 4.2, the kernel symbols start above memory address `0x80000000`, which is the region assigned to the kernel. It can be observed that the first section, after the start of kernel memory address space, starts with an offset of `0x2000` or 8192 bytes. This is precisely the zero page.

The program headers, or simply, segments, basically exist to add information to the sections so as to create an executable memory image[44]. The program headers for the vmlinux can be observed in Figure 4.3.

```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x002000 0x80000000 0x00000000 0x1a9d88 0x1a9d88 RWE 0x2000
  LOAD           0x1ac000 0x801ac000 0x001ac000 0x06200 0x13f638 RW  0x2000

 Section to Segment mapping:
  Segment Sections...
   00     .empty_zero_page .text .cpuinit.text __ex_table .rodata .init.rodata __param .data .cpuinit.data .ref.data
   01     .data.init_task .init.data .init.setup .taglist .initcall.init .con_initcall.init .init.ramfs .bss .stack .heap
```

Figure 4.3: The program headers inside vmlinux

4.1.1 Compressed kernel image

To generate a compressed image of the Linux kernel, the following command was used:

```
make CROSS_COMPILE=rvex- ARCH=rvex compressed
```

With everything else compilable, a compressed image of the Linux kernel is generated called `zImage`. While booting with this image, however, an additional decompression needs to be performed. The code to perform this decompression was available in the st200 port. A different `head.S` file was created, which, along with supporting piece of code, would be invoked only when the *compressed* option was passed while building the kernel. The size of `zImage` is `794.14 kiB`. A compressed image is desired under Goal 1, discussed in Section 1.3.

4.2 Booting

The implementation towards booting, which involved writing/modifying code performing a series of initializations from the entry point function (inside `head.S`) till before running the *init* executable, was discussed in Chapter 3. Basically, with such implementation, the stage till the formation of the first kernel thread can be reached in booting. The

kernel needs to be provided with the *init* application, in an appropriate manner, so that it can be executed after the creation of `kernel_thread`. The *init* is actually the first user space process and is created from `kernel_thread`, thereby inheriting its PID of 1[39]. In order to be able to run a fully functional init, the following should be ensured:

1. The function in the architecture-specific code to handle page faults must be implemented.

   The `do_page_fault` function in the file `fault.c`, located under `linux/arch/rvex/mm/` is responsible for handling page faults[1]. Basically, the Linux kernel loads only the required kernel structures into the memory for user space applications. The data and text segments are loaded when these applications *fault* when they find during the execution of their first instruction that the corresponding pages are not loaded in the memory[39]. Again, the code for this function was present in the st200 port and was used as it is, except for the change in the address space identifier field as per the ρ-VEX. This field basically provides the process identifier.

2. The required system calls must be implemented.

   The system calls had already been ported to the ρ-VEX through register translation and omission/appropriate modification of unsupported instructions and functions. Some important modifications were for the `sim_write` function, which is involved in writing to the UART. From inside this function, a trap was generated which printed on the terminal using the simrvex's `write` function. Since a simulator is used as a platform, writing to the UART is not needed.

3. There should a C library for enabling C programs (in this case, the *init*) to be compiled for running on Linux, which in turn runs on the ρ-VEX (or in this case, simrvex).

   Under another project at TU Delft, the `uclibc` library was ported for the ρ-VEX. The `uclibc` is a standard C library intended for application development for Linux-based systems. This library was primarily developed for the uCLinux, intended for MMU-less systems. The port of this library, the `uclibc-rvex`, was deemed helpful for compiling standard C programs to behave as inits.

4. A (known) file system should be provided to the kernel which would act as the initial root file system.

   For this purpose, the *initramfs* is the simplest file system, with a lot of documentation available online related to its creation. The intended *init* binary must be placed inside an appropriate location in the initramfs, so that it can be located by the kernel and executed.

[1]Basically, it just identifies the fault and calls appropriate routines.

With the above conditions being satisfied, a simple "Hello World" printing program was written in C, and compiled using the ρ-VEX toolchain, but also including `uclibc-rvex`. The next task was to produce an initramfs with the binary so formed placed at an appropriate location inside the file system. Steps to create a simple initramfs are available in standard online references.

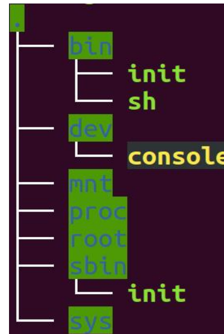


Figure 4.4: The directory structure of the simple initramfs created to test the boot

A directory hierarchy was formed as shown in Figure 4.4. The inits that can be seen are actually C programs compiled using methods mentioned above and renamed as `init`. Actually, inside file `main.c`, located under `linux/init/`, the sequence of execution of user space programs can be known, and also the location inside the file system where the kernel searches for these programs. The binaries were placed in the directory structure accordingly and a compressed file system, `initramfs_data.cpio.gz`, was created[2]. This is a `gzip` compression, and the option to enable compression and decompression in the gzip format was enabled in the kernel configuration file (`.config`) for the kernel to identify the compressed file system. The size of this compressed file, as well as the address in memory where it would be available, were required to be made known to the kernel before the build. This was done via the list of command line parameters present in the file `setup.c` located under `linux/arch/rvex/kernel/`. Then the simrvex binary was run, instructing it to place the compressed root file system at a desired memory address, and also, the kernel image, vmlinux, was passed to it. The boot logs obtained can be seen in Figure 4.5.

[2]This file needs to be placed under `linux/usr/` .

```
[    0.000000] Linux version 2.6.32.9 (anurag@ubuntu) (Modified for rVEX reconfigurable VLIW processor 2016 TU Delft) #767 Fri Oct 26 19:48:24 CEST 2018
[    0.000000] console [ttyFW00] enabled
[    0.000000] _end = 801fae70
[    0.000000] Bootmemory allocator at 254 describing 0x00000000 to 0x04000000
[    0.000000] freeing pages 254:8192
[    0.000000] reserving pages 254:255
[    0.000000] Initial ramdisk at: 0x82000000 (87087 bytes)
[    0.000000] paging_init()
[    0.000000] pagetable_init()paging_init 2
[    0.000000] paging_init 3
[    0.000000] pgdat 801a9750
[    0.000000] calculate_node_totalpages(801a9750, 801affc0, (null))
[    0.000000] calculate_node_totalpages 2
[    0.000000] after alloc_node_mem_map
[    0.000000] after free_area_init_core
[    0.000000] paging_init 4
[    0.000000] Built 1 zonelists in Zone order, mobility grouping on.  Total pages: 8160
[    0.000000] Kernel command line: mem=64m console=ttyAS0 initrd=0x2000000,87087
[    0.000000] PID hash table entries: 256 (order: -3, 1024 bytes)
[    0.000000] Dentry cache hash table entries: 8192 (order: 2, 32768 bytes)
[    0.000000] Inode-cache hash table entries: 4096 (order: 1, 16384 bytes)
[    0.000000] mem_init()Memory: 63104k/65536k available (2432k reserved including: 1495k kernel code, 208k data, 283k BSS, 16k init, 256k page map)
[    0.000000] Hierarchical RCU implementation.
[    0.000000] NR_IRQS:128
[    0.003755] Mount-cache hash table entries: 1024
[    0.012629] CPU: TUDelft rVEX
[    0.148177] vdso_kbase: 80190000, 0x1 pages
[    0.148499] vDSO setup failure, not enabled !
[    0.168075] Switching to clocksource rVEX timer
[    0.192024] Unpacking initramfs...
[    0.660449] Freeing initrd memory: 85k freed
[    0.666516] IN DTLB FAULT HANDLER!! address 0x00000000, TLB_EXCAUSE 0x00000000, EXCAUSE 0x00000011
[    0.667053] Saved pc 8005e148, ccr 000005a5
[    0.842597] free_initmem: 801b0000 to 801b4000
[    0.843019] Freeing unused kernel memory: 16k freed
[    0.860222] IN DTLB FAULT HANDLER!! address 0x00005e00, TLB_EXCAUSE 0x00080000, EXCAUSE 0x00000011
[    0.860759] Saved pc 8000a8a8, ccr 000005a5
[    0.870284] IN ITLB FAULT HANDLER!! address 0x00000000, TLB_EXCAUSE 0x00000000, EXCAUSE 0x00000010
[    0.870813] Saved pc 00000000, ccr 000006a5
[    0.871382] IN ITLB FAULT HANDLER!! address 0x000023d0, TLB_EXCAUSE 0x00000000, EXCAUSE 0x00000010
[    0.871915] Saved pc 000023d0, ccr 000006a5
[    0.872746] IN DTLB FAULT HANDLER!! address 0x00006e18, TLB_EXCAUSE 0x00080000, EXCAUSE 0x00000011
[    0.873279] Saved pc 00001f20, ccr 000006a5
total cycles = 0
Hello world
[    0.879605] Kernel panic - not syncing: Attempted to kill init!
[    0.880130] IN ITLB FAULT HANDLER!! address 0x00000000, TLB_EXCAUSE 0x00000000, EXCAUSE 0x00000010
[    0.880660] Saved pc 00000000, ccr 000005a5
[    0.881008] Unable to handle kernel NULL pointer dereference at virtual address 00000000
[    0.881560]  pc 00000000
[    0.882112]  cause 00000010
[    0.882380]  tlb_cause 00000000
[    0.882808]  pc=00000000  ccr=000005a5  lr=8004d860          Not tainted
[    0.883212]  br=ff (br0=1 br1=1 br2=1 br3=1 br4=1 br5=1 br6=1 br7=1)
[    0.884537]  r55 =********  r56 =********  r57 =********  r58 =********  r59 =********
[    0.885502]  r60 =********  r61 =********  r62 =********  r63 =********  r1=83813e2c  r2=80195724  r3=00000001  r4=00000000
[    0.887063]  r5=801b43b4  r6=ffffffff  r7=8004d860  r8=00000006
[    0.887659]  r9=00000006  r10=801b47dc  r11=8018cb50  r12=00000000
[    0.888261]  r13=00000001  r14=00000c02  r15=00000c02  r16=0003ffff
```

Figure 4.5: The Linux kernel boot logs observed on passing the kernel image to simrvex

As can be noted from Figure 4.5, "Hello World" is printed, which indicates the successful run of *init*. The clock ticks can be seen at the start of each line. These kernel logs are obtained due to the printk statements. To differentiate kernel logs from user space logs, the traces printing option needs to be enabled while running simrvex, which results in memory addresses being printed corresponding to each instruction. Any trace with address value more than `0x80000000` can be classified as a kernel trace. The initialization operations and also, the unpacking and mounting of the initramfs can be noted additionally from Figure 4.5. The kernel enters panic mode after the init is executed, which is expected, as the init is expected to sustain execution and give rise to or execute other processes. A shell, for example, can do this, and it can be made to run as the first user space. For testing under this project, the `sash`[3] shell was compiled using uclibc-rvex for the ρ-VEX. The boot logs, indicating that sash has started execution, are provided in Figure 4.6.

[3]Sash stands for Stand-alone Shell, and is a UNIX shell, which basically means it is a UNIX-specific program which provides an interactive environment through interpreting user commands and thereby enabling user-kernel interaction.

```
[    0.895827] Freeing initrd memory: 133k freed
[    0.902229] IN DTLB FAULT HANDLER!! address 0x00000000, TLB_EXCAUSE 0x00000000, EXCAUSE 0x00000011
[    0.902767] Saved pc 8005e148, ccr 000005a5
[    1.078069] free_initmem: 801b0000 to 801b4000
[    1.078492] Freeing unused kernel memory: 16k freed
[    1.095809] IN DTLB FAULT HANDLER!! address 0x00025240, TLB_EXCAUSE 0x00080000, EXCAUSE 0x00000011
[    1.096348] Saved pc 8000a8a8, ccr 000005a5
[    1.107368] IN ITLB FAULT HANDLER!! address 0x00000000, TLB_EXCAUSE 0x00000000, EXCAUSE 0x00000010
[    1.107902] Saved pc 00000000, ccr 000006a5
[    1.108475] IN ITLB FAULT HANDLER!! address 0x00010d30, TLB_EXCAUSE 0x00000000, EXCAUSE 0x00000010
[    1.109015] Saved pc 00010d30, ccr 000006a5
[    1.109586] IN ITLB FAULT HANDLER!! address 0x00013320, TLB_EXCAUSE 0x00000000, EXCAUSE 0x00000010
[    1.110125] Saved pc 00013320, ccr 000006a5
[    1.110692] IN ITLB FAULT HANDLER!! address 0x00016c10, TLB_EXCAUSE 0x00000000, EXCAUSE 0x00000010
[    1.111378] Saved pc 00016c10, ccr 000006a5
[    1.111945] IN ITLB FAULT HANDLER!! address 0x0001b200, TLB_EXCAUSE 0x00000000, EXCAUSE 0x00000010
[    1.112485] Saved pc 0001b200, ccr 000006a5
[    1.113094] IN ITLB FAULT HANDLER!! address 0x00020a10, TLB_EXCAUSE 0x00000000, EXCAUSE 0x00000010
[    1.113634] Saved pc 00020a10, ccr 000006a5
[    1.114200] IN ITLB FAULT HANDLER!! address 0x00015ef0, TLB_EXCAUSE 0x00000000, EXCAUSE 0x00000010
[    1.114740] Saved pc 00015ef0, ccr 000006a5
[    1.115599] IN ITLB FAULT HANDLER!! address 0x00002690, TLB_EXCAUSE 0x00000000, EXCAUSE 0x00000010
[    1.116137] Saved pc 00002690, ccr 000006a5
[    1.116707] IN ITLB FAULT HANDLER!! address 0x0001eb50, TLB_EXCAUSE 0x00000000, EXCAUSE 0x00000010
[    1.117246] Saved pc 0001eb50, ccr 000006a5
[    1.117826] IN DTLB FAULT HANDLER!! address 0x00022920, TLB_EXCAUSE 0x00000000, EXCAUSE 0x00000011
[    1.118365] Saved pc 000149c8, ccr 000006a5
[    1.119005] IN DTLB FAULT HANDLER!! address 0x00027090, TLB_EXCAUSE 0x00080000, EXCAUSE 0x00000011
[    1.119691] Saved pc 00015f80, ccr 000006a5
[    1.123959] IN ITLB FAULT HANDLER!! address 0x0001c3d0, TLB_EXCAUSE 0x00000000, EXCAUSE 0x00000010
[    1.124498] Saved pc 0001c3d0, ccr 000006a5

Sash command shell (version 1.1.1)
[    1.125352] IN DTLB FAULT HANDLER!! address 0x00028554, TLB_EXCAUSE 0x00000000, EXCAUSE 0x00000011
[    1.125891] Saved pc 00016ff0, ccr 000006a5
[    1.126628] IN ITLB FAULT HANDLER!! address 0x00018000, TLB_EXCAUSE 0x00000000, EXCAUSE 0x00000010
[    1.127167] Saved pc 00018000, ccr 000006a5
[    1.127898] IN DTLB FAULT HANDLER!! address 0x00028614, TLB_EXCAUSE 0x00190023, EXCAUSE 0x00000011
[    1.128437] Saved pc 00019fd8, ccr 000006a5
[    1.132545] IN DTLB FAULT HANDLER!! address 0x7fba3ee0, TLB_EXCAUSE 0x00000000, EXCAUSE 0x00000011
[    1.133084] Saved pc 00010d98, ccr 000006a5
[    1.133743] pid 1 do_notify_resume(regs 83813f04, ss 83813ed4, syscall 0, r3 0, flags 4, oldset 0
```

Figure 4.6: Some of the kernel boot logs to show a shell can be started

Appropriate `char` driver code needs to be written, and keyboard input should be enabled in the kernel configuration, so as to make sash accept user input, like a normal shell. In fact, this has already been done, not under this project, however, and sash can sustain its execution, while executing instructions entered by user through key strokes, such as copy, move, list, etc., in the standard UNIX format.

4.3 Evaluating timing performance of benchmarks from the Powerstone benchmark suite

Now that it is established that user space programs can indeed be run on Linux running on simrvex, certain standard C applications could be tested for timing performance. As can be noted from Section 2.3, simrvex is a cycle accurate simulator. Therefore, the number of clock cycles required for execution of an application running on simrvex would be the same as those obtained by executing the application on the hardware ρ-VEX platform[4]. This essentially means that timing results obtained under this project hold true for Linux ported to the ρ-VEX architecture with any implementation.

[4]When caches are not activated

Infrastructure to read execution cycles

To obtain the execution cycle count, however, some groundwork needed to be performed. The basic idea was to read the *cycle counter register* of the ρ-VEX (`CR_CNT`) before the start of execution of the application and after the end of the same, and the difference between these values would give the execution cycles consumed by the application. This figure is a worthy metric for evaluating the timing performance of the applications because of the following two reasons:

1. Caches are disabled, therefore, performance obtained would be the same, irrespective of the repetition count of the application execution.

2. Only a single user space process is running during the evaluation, therefore, there is no other (user space) process sharing the processor's resources[5].

The cycle counter register can only be read by the kernel, not by any user space process. Therefore, the easiest way to enable a user space application to do this is to make it generate a trap, and inside the trap, make the processor print out the value of the register on the console. This is because by calling a trap essentially kernel mode is entered. This obviously required changes in the simrvex source code, in the file `instr.c`. This trap is assigned a code `0x94`[6], and when a check for this code is satisfied inside `instr.c`, the value of the cycle count register is simply read and printed on the console using `printf`. To generate such a trap, inside the user space application source code, an assembly instruction needs to be provided in the following manner:

```
asm volatile("trap 0x94\n");
```

Changing the simrvex configuration

The concept of *generic binaries* was discussed in Section 2.2.5. It is to be noted that the compiled Linux kernel image, `vmlinux`, which was obtained after building the kernel using the ρ-VEX toolchain, is actually a generic binary. Therefore, it can be run on any configuration of the ρ-VEX. The configuration word encoding (hex value) can be passed to simrvex as an argument using the option `-i`. Although there can be multiple ρ-VEX configurations, the following configurations were used for evaluation:

1. 1x8-issue configuration

2. 1x4-issue configuration

3. 1x2-issue configuration

[5]It is interesting to note that even if other user space processes are scheduled in between by the scheduler, this metric would still be valid, and would mean response time.

[6]Just a random, unused code.

Choice of benchmarks

Benchmarks from the `Powerstone benchmark suite` were chosen as applications for timing performance evaluation. The reason behind this choice was that these benchmarks are standard C programs, having been used for evaluation in many projects, and are of diverse nature. From this suite, the following three benchmarks were chosen for performance evaluation:

1. CRC[45] - CRC (Cyclic Redundancy Check) is one of the most powerful error-detecting schemes available and is commonly used in sender-receiver communication. One such implementation uses a polynomial generator at the sending end which encodes messages by padding redundant bits. The receiver uses the same polynomial generator on the received message to check for errors. The CRC scheme finds its applications in:
   1) The Data Link Layer of the Bluetooth protocol stack.
   2) Digital networks and storage devices like hard disks.

2. Ucbqsort - This is a benchmark developed at UC Berkeley. It uses the quick sort algorithm to sort an array of 1000 numbers. This array is provided through hard coding within the program.

3. JPEG - JPEG (Joint Pictures Motion Group) is a well known (digital) image compression algorithm. JPEG performs a lossy compression on an input image and uses a technique based on the discrete cosine transform (DCT)[46]. The available benchmark uses hardcoded matrices to hold image pixel values, the DCT coefficients, and other required data. It involves considerable number of mathematical operations. This benchmark can be classified as computationally intensive and can therefore prove to be a good candidate for evaluation under this project.

All the above benchmarks have their own verification techniques within the code. Before the application exits, a log, indicating whether the operation ran successfully, is printed out. This was verified for each benchmark during evaluation.

Evaluation of the benchmarks

The procedure to run these benchmarks is exactly the same as followed for the "Hello World" application, as discussed in the previous section. However, for an evaluation of the results, the same benchmarks were run on simrvex bare-metal[23][7]. For running the applications bare-metal, they need to be compiled by including the `newlib` library, available for the ρ-VEX, instead of `uclibc-rvex`. The number of execution cycles obtained for the chosen benchmarks when run on Linux and bare-metal on simrvex are provided in Tables 4.2 and 4.2, respectively. The results have been obtained for different simrvex configurations.

[7] Bare-metal means running the programs directly on the processor, without an operating system running on top of it.

Benchmark	1x2-issue	1x4-issue	1x8-issue
CRC	76557	73009	72429
Ucbqsort	747363	705107	704349
JPEG	9429951	8723667	8713776

Table 4.1: Execution cycles consumed by the benchmarks when run on Linux ported on simrvex, for different configurations of simrvex

Benchmark	1x2-issue	1x4-issue	1x8-issue
CRC	72975	68882	68154
Ucbqsort	701870	646199	630414
JPEG	8525856	7998047	7997406

Table 4.2: Execution cycles consumed by the benchmarks when run bare-metal on simrvex, for different configurations of simrvex

With the available data, the latency due to page faults, timer interrupts, etc. can be calculated for the applications running on Linux. This would simply be the difference between the execution cycle counts for the bare-metal and Linux cases. These figures for the chosen benchmarks for different ρ-VEX configurations are shown in Figure 4.7.

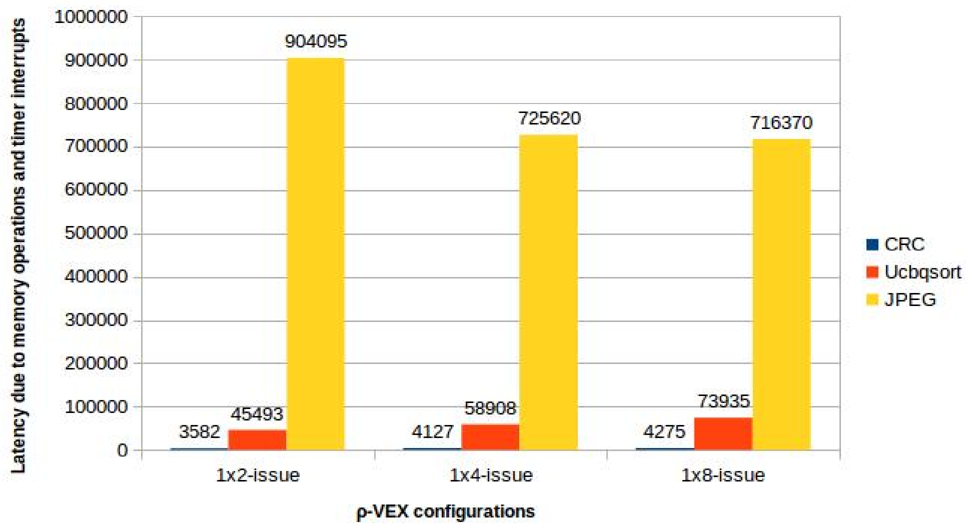


Figure 4.7: Latency due to the use of Linux kernel code and virtual memory for the three benchmarks

Applications running on Linux consuming more execution cycles is expected. For example, as can be observed from Table 4.1 and Table 4.2, the timing performance of even the 1x2-issue bare-metal case appears to be better than the 1x8-issue Linux case. The increase in the number of execution cycles when using the kernel is actually the

price paid for using the kernel code and virtual memory. The importance of operating system support and virtual memory, however, have already been discussed under Section 1.2. Using virtual memory (and therefore, memory management) is an important step towards multitasking, while the operating system (or rather, the kernel) can provide the necessary abstraction of the reconfiguration mechanism to the applications. The kernel also provides a scope for the ρ-VEX to perform its reconfigurations independently, and also schedule meaningful reconfigurations. It is important to note here that the intention is not to compare the performance of the benchmarks running on Linux and bare-metal. The difference in the execution cycle count, or the latency, can however serve as a figure to provide a scope for optimization. A higher latency in terms of number of execution cycles for JPEG than CRC, for example, does not necessarily mean a largely worsening performance with increasing complexity. It is logical that JPEG being a larger and more complex code than CRC necessitates a higher number of page replacements and witnesses the occurrence of a higher number of timer interrupts during its execution. Therefore, latency in terms of percentage of increase in the number execution cycles when using Linux is sometimes a better means of expression. For example, for the 1x8-issue configuration, when using Linux, the latency for CRC is 6.27%, and for JPEG it is 8.96%.

Benchmark	1x2-issue	1x4-issue	1x8-issue
CRC	4.90%	5.99%	6.27%
Ucbqsort	6.48%	9.11%	11.73%
JPEG	10.60%	9.07%	8.96%

Table 4.3: Latency in terms of percentage of increase in the execution cycle count for the three benchmarks when run on Linux ported on simrvex

The values now do not seem to vary as much as observed in Figure 4.7. The highest latency among those obtained for all the test scenarios was observed to be 11.73% for the *ucbqsort* benchmark for 1x8-issue configuration of simrvex. A possible reason for this can be the fact that the bare-metal 1x8-issue situation is highly suited for the *ucbqsort* benchmark, but with the involvement of the Linux kernel, the wider issue width of the 1x8-issue configuration cannot be leveraged due to increased number of page faults.

4.4 Thread switch latency

Thread switching is the *context switching* between threads. It involves saving processor register values specific to the thread being switched, switching the control to the new thread, and restoring the state of the registers. The ρ-VEX provides a multithreaded environment, and an operating system providing multithreading support would be beneficial to it[8]. Linux provides a matured multithreading support, and could help channel independent tasks on the ρ-VEX pipelanes by means of kernel threads. However, a multithreading environment often involves thread switching, which may turn out to be a time consuming operation. An accurate measure of the thread switch latency is a step

towards measuring the process context switch, which is a very important consideration while going for parallelizing the execution of multiple programs for performance gains.

Two kernel threads were created in the Linux kernel ported on simrvex. The thread functions corresponding to these threads contained infinite loops printing out logs and calling the *schedule()* kernel function to schedule the next kernel thread. This way, the two kernel threads were made to constantly switch between each other, as can be seen from the logs in Figure 4.8.

```
context 0 starting at PC = 0x00002000
context 1 starting at PC = 0x00002000
context 2 starting at PC = 0x00002000
context 3 starting at PC = 0x00002000
[    0.000000] Linux version 2.6.32.9 (anurag@ubuntu) (Modified for rVEX reconfigurable VLIW processor 2016 TU Delft) #791 Tue Nov 6 00:31:38
CET 2018
[    0.000000] console [ttyFW00] enabled
[    0.000000] _end = 801fae70
[    0.000000] Bootmemory allocator at 254 describing 0x00000000 to 0x04000000
[    0.000000] freeing pages 254:8192
[    0.000000] reserving pages 254:255
[    0.000000] Initial ramdisk at: 0x82000000 (88343 bytes)
[    0.000000] paging_init()
[    0.000000] pagetable_init()paging_init 2
[    0.000000] paging_init 3
[    0.000000] pgdat 801a9750
[    0.000000] calculate_node_totalpages(801a9750, 801affc0, (null))
[    0.000000] calculate_node_totalpages 2
[    0.000000] after alloc_node_mem_map
[    0.000000] after free_area_init_core
[    0.000000] paging_init 4
[    0.000000] Built 1 zonelists in Zone order, mobility grouping on.  Total pages: 8160
[    0.000000] Kernel command line: mem=64m console=ttyAS0 initrd=0x2000000,88343
[    0.000000] PID hash table entries: 256 (order: -3, 1024 bytes)
[    0.000000] Dentry cache hash table entries: 8192 (order: 2, 32768 bytes)
[    0.000000] Inode-cache hash table entries: 4096 (order: 1, 16384 bytes)
[    0.000000] mem_init()Memory: 63104k/65536k available (2432k reserved including: 1495k kernel code, 208k data, 283k BSS, 16k init, 256k pag
e map)
[    0.000000] Hierarchical RCU implementation.
[    0.000000] NR_IRQS:128
[    0.003755] Mount-cache hash table entries: 1024
[    0.012629] CPU: TUDelft rVEX
[    0.018589] Switching from kernel thread 1: counter val = 2651162
Switched to kernel thread 2: counter val = 2653508
Switched to kernel thread 2: counter val = 2674243
Switching from kernel thread 1: counter val = 2676695
Switched to kernel thread 2: counter val = 2679018
Switching from kernel thread 1: counter val = 2681470
Switched to kernel thread 2: counter val = 2683793
Switching from kernel thread 1: counter val = 2686245
```

Figure 4.8: Logs indicating constant thread switching between the two kernel threads created, along with the cycle counter register values

The latency between such switches was calculated, this time by directly reading out the cycle counter register values from the kernel code[8]. The entire operation was obviously placed before the *init* comes into picture. As can be observed from Figure 4.8, the kernel thread switch consumes execution cycles of the order of `2400 cycles`. This value seems to be very low compared to a similar quantity derived in [23]. However, it is to be noted here that the thread switch latency also depends on factors such as the number of local variables being handled by the thread before the switch. For this project, the kernel thread functions used are extremely simple - they just print out logs and schedule the other kernel thread inside an infinite loop, which is not the exact test scenario used in [23]. Nevertheless, the value obtained in the current project's experimentation still qualifies as an indicative of the kernel thread switch latency, albeit for simple (kernel) threads.

[8]This latency also includes the cycles consumed due to invoking of the scheduler.

4.5 Conclusion

In this chapter, the results, and the infrastructure that was required to be established to obtain the results, were discussed. The kernel image, obtained after a successful build of the kernel code, was dissected and the information contained in it was discussed. Knowledge of the memory organization followed by the kernel image object file is important for understanding some crucial operating system internals. This also helps in debugging, for instance, knowing what would result in a stack overflow.

Thereafter, the execution of the *init* process was presented, which signals the end of porting. The kernel that has been ported on simrvex can start a shell, and can also run some standard benchmarks. Any intended user space application needs to be fed to the kernel through a filesystem, *initramfs* in this case. With the current level of implementation, the user space applications need to be compiled using the C library for the ρ-VEX, called uclibc-rvex. This library is actually a port of the standard uclibc for the ρ-VEX, and provides important supporting code for compilation, for example, for the `printf` function. As of now, the kernel enters a panic mode after execution of the first user space application. This is because it is essential that the first user space process should sustain execution and give rise to other (user space) processes. The benchmarks used here are not capable of doing that, but that was also not their purpose. Along with the uclibc-rvex components, only the `sash` binary, which is the standard SASH (Stand Alone Shell) compiled for the ρ-VEX, was available, and therefore, this was the only shell that could be tested. An equivalent of the standard `pthread` library needs to be available for the ρ-VEX, so that new user-level threads can be created, which can lead to multithreading at the user-level. As of now, kernel threads can be created and the scheduler can schedule different kernel-level threads on a time-sharing basis only. This is because currently, the kernel configuration has been kept to a minimum, doing away with many components like those concerned with networking, file system, drivers for inputs and outputs, and even scheduling policies. Therefore, enabling different scheduling policies can yield some further interesting results.

The booting was also tested successfully on 2-issue, 4-issue and 8-issue configurations of simrvex[9]. This was made possible due to the fact that the Linux kernel image obtained after compilation by the ρ-VEX toolchain is a generic binary. Even the execution cycle count for the chosen benchmarks could be obtained by reading the cycle counter control register. There is actually a restriction imposed by the kernel regarding the resources which user space programs can access. CPU registers fall under the kernel's scope. Therefore, to enable the user space programs to read the cycle counter register values in order to get the execution cycle count, a new trap was incorporated into the simrvex code. Switching to traps means switching to kernel code, and therefore, making register access possible.

The overheads in the execution cycle counts obtained when benchmarks were run on Linux ported on simrvex were calculated for the 2-issue, 4-issue and 8-issue configurations of simrvex. Memory management operations are clearly expensive, with the maximum overhead being 11.73% for the *ucbqsort* benchmark for 1x8-issue configuration of simrvex. With the involvement of the Linux kernel, the advantage of the wider

[9]These configurations were changed statically

issue width of the 1x8-issue configuration was observed to be diminished due to the increased number of page faults. However, virtual memory, which can be aided by memory management, is a crucial step towards multitasking, which is more important than the execution cycle overhead. In order to aid in future calculations of context switch latency, the kernel thread switch latency was calculated for the ported kernel. This value was found out to be of the order of 2400 execution cycles. Context switching is bound to happen in largely multitasking systems, and is an important consideration when opting for parallelization.

5 Conclusion

This chapter concludes the thesis, starting with a summary of discussed topics, moving on to summarizing the contributions under this graduation project, and then finally the future work desired by this project.

5.1 Summary

In Chapter 1, some important characteristics of the ρ-VEX, which enable it to extract a high level of parallelism from applications, was discussed. Along with it, the motivation behind providing an operating system support, especially in context of processor reconfiguration management and usage of virtual memory, was discussed. Thereafter, the research question was stated and the goals of this project were visited.

In Chapter 2, the background needed to understand the thesis was discussed. It was noted that porting an operating system to a processor requires a thorough knowledge of the processor architecture, the toolchain capabilities, and of course, the operating system code itself (at least the part interacting with the hardware). This is the reason a study for these three components was presented in detail. The discussion related to the toolchain part was kept short, and was emphasized more on a study of the visible (and known) capabilities, rather than internal working. Implementing toolchain modifications was not a desired component of this project. The emphasis is more on modifying the operating system code to make it suitable for the current ρ-VEX system as much as possible, before considering toolchain modifications. However, knowledge of the toolchain limitations is important to consider modifying the operating system for the port, and were presented along with the implementation for a better reference. The functionality/characteristics of the ρ-VEX visible to the programmer, including some details of its internal working to gauge the impact of changing such functionality through program, were considered. As far as discussion over the Linux kernel is considered, efforts were put into discussing the required details, keeping in mind audience having some basic operating system background (particularly UNIX-based operating systems). This is not to say, however, that the discussion about the Linux kernel presented in this chapter completely covers the background for the implementation, or completely lies within the scope of the implementation, but the overlap is high. Much more detailed and extensive documentation about the Linux kernel can be found online, and in standard references. The same goes for the discussion about the ρ-VEX characteristics and the TLB unit presented in this chapter - the ρ-VEX user manual[25] and the ST231 Core and Instrucion Set Architecture manual[30], respectively, are more detailed sources of reference, and relevant topics out of these manuals were chosen for discussion. Finally,

a discussion about projects related to this one was presented, while highlighting the key differences.

Chapter 3 dealt with the implementation details for this project. The choice of the ρ-VEX simulator, simrvex, as the platform, and modifying the st200 Linux port for the ρ-VEX, were justified. Thereafter, the porting of this chosen Linux kernel on simrvex was discussed in detail. Although, there can be many approaches to porting, and the approach presented may or may not have been employed before, but the end result (porting) was achieved. The approach followed under this project for porting took advantage of the modular nature of Linux and its separation of the architecture-specific code from the architecture-independent code. The architecture-independent part was dealt with first by writing the bare minimum code for the architecture-specific part for compilation sake. Also, at this stage the relocatable linking option was enabled ensuring objects were not linked into the final kernel object upon successful build, so as to approach the toolchain incompatibilities in a systematic manner. The compiler errors were resolved through a combination of kernel source code changes and compiler version changes. After the unlinked final kernel image was successfully built, which essentially meant that compiler-related incompatibilities were resolved, the architecture specific code was produced through heavy modification of the st200 Linux port. Thereafter, linker errors were tackled after disabling the relocatable linking option, which allowed exposing the linker's limitations while attempting to link the entire kernel code. All these modifications resulted in the successful generation of the Linux kernel image `vmlinux` built using the ρ-VEX toolchain, and meant to be executed on simrvex.

In Chapter 4, the results, and the infrastructure that was required to be established to obtain the results, were discussed. The kernel image, obtained after a successful build of the kernel code, was dissected and the information contained in it was discussed. Knowledge of the memory organization followed by the kernel image object file is important for understanding some crucial operating system internals. This also helps in debugging, for instance, knowing what would result in a stack overflow.Thereafter, the execution of the *init* process was presented, which signals the end of porting. The kernel that has been ported on simrvex can start a shell, and can also run some standard benchmarks. Any intended user space application needs to be fed to the kernel through a filesystem, *initramfs* in this case. With the current level of implementation, the user space applications need to be compiled using the C library for the ρ-VEX, called uclibc-rvex. This library is actually a port of the standard uclibc for the ρ-VEX, and provides important supporting code for compilation, for example, for the `printf` function. As of now, the kernel enters a panic mode after execution of the first user space application. This is because it is essential that the first user space process should sustain execution and give rise to other (user space) processes. The benchmarks used here are not capable of doing that, but that was also not their purpose. Along with the uclibc-rvex components, only the `sash` binary, which is the standard SASH (Stand Alone Shell) compiled for the ρ-VEX, was available, and therefore, this was the only shell that could be tested. An equivalent of the standard `pthread` library needs to be available for the ρ-VEX, so that new user-level threads can be created, which can lead to multithreading at the

user-level. As of now, kernel threads can be created and the scheduler can schedule different kernel-level threads on a time-sharing basis only. This is because currently, the kernel configuration has been kept to a minimum, doing away with many components like those concerned with networking, file system, drivers for inputs and outputs, and even scheduling policies. Therefore, enabling different scheduling policies can yield some further interesting results. The booting was also tested successfully on 2-issue, 4-issue and 8-issue configurations of simrvex (These configurations were changed statically). This was made possible due to the fact that the Linux kernel image obtained after compilation by the ρ-VEX toolchain is a generic binary. Even the execution cycle count for the chosen benchmarks could be obtained by reading the cycle counter control register. There is actually a restriction imposed by the kernel regarding the resources which user space programs can access. CPU registers fall under the kernel's scope. Therefore, to enable the user space programs to read the cycle counter register values in order to get the execution cycle count, a new trap was incorporated into the simrvex code. Switching to traps means switching to kernel code, and therefore, making register access possible.

The overheads in the execution cycle counts obtained when benchmarks were run on Linux ported on simrvex were calculated for the 2-issue, 4-issue and 8-issue configurations of simrvex. Memory management operations are clearly expensive, with the maximum overhead being 11.73% for the *ucbqsort* benchmark for 1x8-issue configuration of simrvex. With the involvement of the Linux kernel, the advantage of the wider issue width of the 1x8-issue configuration was observed to be diminished due to the increased number of page faults. However, virtual memory, which can be aided by memory management, is a crucial step towards multitasking, which is more important than the execution cycle overhead. In order to aid in future calculations of context switch latency, the kernel thread switch latency was calculated for the ported kernel. This value was found out to be of the order of 2400 execution cycles. Context switching is bound to happen in largely multitasking systems, and is an important consideration when opting for parallelization.

5.2 Main contributions

In Chapter 1, the research question for this project was presented. It is presented again below:

How to provide operating system support to the ρ-VEX?

The answer to the above-stated research question involves many considerations ranging from the choice of the OS, to ensuring whether the ρ-VEX processor and the associated toolchain support it, and also the modifications in the chosen OS to make it "fit" for the ρ-VEX. Testing the implementation for the port is implicit. Therefore, the following two goals were identified for this project

1. Porting an operating system to the ρ-VEX

The tasks to be done under this project in order to achieve this goal were identified and are stated below, along with the steps taken to achieve them:

1. Choose an appropriate microarchitecture of the -VEX which would act as the platform upon which the operating system will be ported.

   Simrvex was chosen as the platform, mainly because it had an MMU integrated. The desirability of virtual memory and the other factors in favour of simrvex have been discussed in Section 1.2 and Section 3.1 respectively.

2. Ensure that the toolchain is capable of supporting an operating system port.

   This was mainly done by compiling and running simple programs on simrvex. At this stage, it was not possible to completely determine whether the toolchain was mature enough to support an operating system. Therefore, it was deemed to attempt tackling the potential incompatibilities through code changes and toolchain changes.

3. Choose the right operating system for porting.

   The operating system kernel chosen for the port was the one ported to the st200 architecture. The version of this kernel is 2.6.32.

4. Implement the changes in the chosen operating systems kernel code in order to accomplish the port.

   These have been discussed in Chapter 3.

What was also desirable to achieve under this goal is the compression of the end product of the operating system kernel compilation. This was deemed help achieve faster startup time, apart from the reduction in the requirement of external memory for storing the kernel. This was also achieved, as described in Section 4.1.1.

To test whether an operating system port is successful is done by running applications on it. Validation of the end result achieved by following the steps described above thus forms another implicit goal, the second goal:

2. Evaluating performance of applications run by the operating system ported to the ρ-VEX

The setup required to achieve the results, as well as the achieved results were discussed in Chapter 4.

Finally, coming to the long term goal of this project:

Implementing modifications in the kernel as mentioned in [12] to support fast

runtime reconfiguration in the ρ-VEX, thereby creating a truly dynamic environment

This goal was beyond the scope of a graduation project, and could not be achieved due to time constraints. However, an approach that was identified in order to achieve this goal will be discussed in Section 5.3.

Summary of the contributions

Considering the discussion above, the main contributions made under this project can be summarized as follows:

- Exploring and choosing the ρ-VEX implementation for the port.
- Exploring and choosing the Linux kernel version for the port.
- Implementing all the required modifications in the st200's Linux kernel port in order to achieve the operating system port on simrvex.
- Implementing the infrastructure to execute applications in user space and also to evaluate their timing performance.

5.3 Future work

The implementation under this project is actually a step towards achieving the long term goal of this project. If the techniques mentioned in [12] can be adapted, the ρ-VEX can result in a truly dynamic environment. The Linux kernel version modified in the implementation of [12] is 2.6, which is the same version used for porting on simrvex. Modifications to this extension to suit the ρ-VEX can thus be the easiest method to add dynamic properties to the kernel ported on simrvex.

As of now, simrvex needs to be statically configured to different ρ-VEX configurations before executing the Linux kernel image. However, in order to achieve the long term goal, it should be possible to do this dynamically. The means presented in [12] would be helpful in understanding the operations to be undertaken in order to retain software contexts during core reconfigurations. The most heavily impacted components of the kernel would be the `scheduler` and the `irq`, for which, appropriate kernel configurations also need to be activated. In fact, to support additional modules/features, further kernel configuration options need to be enabled and the corresponding code must be made compatible with the ρ-VEX. Like the structure `task_struct` carries information about a thread, a similar new kernel structure needs to be maintained for representing information about the structures abstracting the cores. Information about core reconfigurations must be communicated to concerned modules, in a manner similar to the hotplug mechanism in Linux. If caches are enabled, their access synchronization needs to be taken care of.

But before proceeding with the above implementation, the SMP (symmetric multiprocessing) module should be implemented in the kernel code specific to the ρ-VEX

architecture. In fact, due to the lack of SMP support in the kernel, it can currently only execute in one of the ρ-VEX contexts at a time. Multithreading support should also be made available at the user level by porting appropriate thread libraries. Besides this, the linker should be modified to accept dynamic loading of objects, thereby making creation of shared object files possible. This is very important for the `vDSO` feature, which is required to speed up certain system calls.

The kernel needs to be ported to the hardware ρ-VEX eventually. Therefore, another interesting exploration would be to integrate a hardware-based MMU (free from OS management) in the hardware ρ-VEX implementation, and port the kernel on it.

Bibliography

[1] M. U. Saleem, “Dynamically reconfigurable fault-tolerant design of ρvex softcore processor,” Master’s thesis, TU Delft, The Netherlands, 2018.

[2] J. Hoozemans, “Targeting static and dynamic workloads with a reconfigurable vliw processor,” Ph.D. dissertation, TU Delft, The Netherlands, 2018.

[3] A. S. Tanenbaum, *Modern Operating Systems*, 3rd ed. Pearson Education, 2009.

[4] “The ρ-vex compiler repository.”

[5] “ET4370 Reconfigurable Computing Design 2017, Lecture 2,” TU Delft, The Netherlands.

[6] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed. Burlington, MA: Morgan Kaufmann, 2012.

[7] “Illiac — Wikipedia, the free encyclopedia,” 2018. [Online]. Available: https://en.wikipedia.org/w/index.php?title=ILLIAC&oldid=851409840

[8] J. van Straten, “A dynamically reconfigurable vliw processor and cache design with precise trap and debug support,” Master’s thesis, TU Delft, The Netherlands, 2016.

[9] “Power4 — Wikipedia, the free encyclopedia,” 2018. [Online]. Available: https://en.wikipedia.org/w/index.php?title=POWER4&oldid=834358968

[10] S. W. K. Doug Burger and S. Sethumadhavan, *Multicore Processors and Systems*. US: Springler, 2009.

[11] E. Ipek, M. Kirman, N. Kirman, and J. F. Martínez, “Core fusion: accommodating software diversity in chip multiprocessors,” in *ISCA*, 2007.

[12] S. Panneerselvam and M. M. Swift, “Chameleon: Operating system support for dynamic processors,” *SIGARCH Comput. Archit. News*, vol. 40, no. 1, pp. 99–110, Mar. 2012. [Online]. Available: http://doi.acm.org/10.1145/2189750.2150988

[13] J. Charles, P. Jassi, N. S. Ananth, A. Sadat, and A. Fedorova, “Evaluation of the intel®core™ i7 turbo boost feature,” in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, ser. IISWC ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 188–197. [Online]. Available: https://doi.org/10.1109/IISWC.2009.5306782

[14] A. Waza, R. N. Mir, and H. N. ud din, “Reconfigurable architectures,” *Journal of Advanced Computer Science & Technology*, vol. 1, no. 4, pp. 337–346, 2012. [Online]. Available: http://www.sciencepubco.com/index.php/JACST/article/view/518

[15] “The ρ-vex official webpage.” [Online]. Available: https://rvex.ewi.tudelft.nl/

[16] K. Vipin and S. A. Fahmy, "Dyract: A partial reconfiguration enabled accelerator and test platform," in *24th International Conference on Field Programmable Logic and Applications (FPL)*, Sept 2014, pp. 1–7.

[17] "Hyper-threading — Wikipedia, the free encyclopedia," 2018. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Hyper-threading&oldid=850117761

[18] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded computing: a VLIW approach to architecture, compilers and tools.* Amsterdam, London: Elsevier, 2005.

[19] [Online]. Available: http://linux-hotplug.sourceforge.net/

[20] "Barrelfish — Wikipedia, the free encyclopedia," 2018. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Barrelfish&oldid=838200632

[21] [Online]. Available: http://www.barrelfish.org/

[22] [Online]. Available: https://www.usenix.org/sites/default/files/conference/protected-files/osdi14_slides_zellweger.pdf

[23] J. Hoozemans, "Porting linux to the ρ-vex reconfigurable vliw softcore," Master's thesis, TU Delft, The Netherlands, 2014.

[24] [Online]. Available: https://stackoverflow.com/questions/19055276/why-compressed-kernel-image-is-used-in-linux

[25] J. van Straten, "The ρ-vex user manual," 2017. [Online]. Available: http://rvex.ewi.tudelft.nl/wp/wp-content/uploads/2017/02/rvex-user-manual.pdf

[26] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoll, and F. M. O. Homewood, "Lx: a technology platform for customizable vliw embedded processing," in *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)*, June 2000, pp. 203–213.

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

[28] "Computer architecture simulator — Wikipedia, the free encyclopedia," 2017. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Computer_architecture_simulator&oldid=799049324

[29] "Translation lookaside buffer — Wikipedia, the free encyclopedia," 2018. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Translation_lookaside_buffer&oldid=856161176

[30] STMicroelectronics, "St231 core and instruction set architecture manual," March 2014.

[31] "Multithreading tutorial." [Online]. Available: https://www.cs.rutgers.edu/~pxk/416/notes/05-threads.html

[32] R. Love, *Linux Kernel Development*, 2nd ed. Novell Press, 2005.

[33] M. M. Yousaf, "Exploiting the reconfigurability of ρ-vex processor for real-time robotic applications," Master's thesis, TU Delft, The Netherlands, 2016.

[34] "Freertos — Wikipedia, the free encyclopedia," 2018.

[35] J. Johansen, "Implementing virtual address hardware support on the ρ-vex platform," Master's thesis, TU Delft, The Netherlands, 2016.

[36] D. Bovet and M. Cesati, *Understanding the Linux Kernel*, 3rd ed. O'Reilly Media, 2008.

[37] "Porting linux to a new processor architecture, part 1: The basics." [Online]. Available: https://lwn.net/Articles/654783/

[38] [Online]. Available: http://man7.org/linux/man-pages/man7/vdso.7.html

[39] "Porting linux to a new processor architecture, part 3: To the finish line." [Online]. Available: https://lwn.net/Articles/657939/

[40] "Porting linux to a new processor architecture, part 2: The early code." [Online]. Available: https://lwn.net/Articles/656286/

[41] "Linux memory management overview." [Online]. Available: https://www.tldp.org/LDP/khg/HyperNews/get/memory/linuxmm.html

[42] [Online]. Available: http://lists.linux-xtensa.org/pipermail/linux-xtensa/Week-of-Mon-20110321/000333.html

[43] [Online]. Available: https://lwn.net/Articles/531148/

[44] [Online]. Available: https://linux-audit.com/elf-binaries-on-linux-understanding-and-analysis/

[45] "Cyclic redundancy check — Wikipedia, the free encyclopedia," 2018. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Cyclic_redundancy_check&oldid=864279455

[46] Wikipedia contributors, "JPEG — Wikipedia, the free encyclopedia," 2018. [Online]. Available: https://en.wikipedia.org/w/index.php?title=JPEG&oldid=865526554