UNIVERSITY OF OSLO
Department of Informatics

Modeling large populations of full-sized virtual machines using minimal virtual instances

Håvard Ostnes
haavako@ifi.uio.no

Network and System Administration

Oslo University College

May 23, 2012

Modeling large populations of full-sized virtual machines using minimal virtual instances

Håvard Ostnes
haavako@ifi.uio.no

Network and System Administration
Oslo University College

May 23, 2012

Abstract

This thesis proposes the use of minimal virtual machines to model larger populations of instances in different environments, and investigates if a cloud environment is able to take the populations even further. Finally the thesis wants to investigate if minimal virtual machines are suitable to host custom application stacks and are able to compete with full-sized virtual machines. As virtualization technology has achieved increased popularity the recent years virtual machines are now used by many businesses, institutions and consumers for different purposes. Full-sized virtual machines are large, and demand considerable amounts of computing resources from the Cloud Resource Pool. This project was able to significantly reduce the size of virtual machines and the amount of computing resources required to host them. The smallest virtual machine accomplished in this project had a size of merely 1.5MB allowing a population of almost 500 times, or at least two orders of magnitude, larger than one standard-sized Ubuntu Server instance. Custom written software was also created for each type of virtual machine for the purpose of simulating real-world CPU usage patterns. Several population sizes of minimal virtual machines were deployed and tested in Hypervisor-on-Hardware and Hypervisor-in-Cloud labs to compare their behavior and performance in different environments.

Acknowledgements

I would like to express my gratitude to my supervisor, Alfred Bratterud, for his guidance, encouragement and support throughout this project. Secondly I express my regards to Ian Seyler, the Founder and Lead Programmer at Return Infinity and the maker and sponsor of BareMetal OS, for taking his time to assist me with problems regarding Bare Metal OS. I am also grateful to Dr. Jan Stoess in the KIT System Architecture Group for providing me with access to important documentation on the L4 Pistachio kernel and for taking his time to answer my many e-mails. I express my appreciation to family and friends for the mental support. Finally a heartfelt thanks goes to my girlfriend Marie for her love, support and patience during this project.

May, 2012
Håvard Ostnes

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Virtual machines (VMs) have become an important part of the IT industry as they offer significant benefits to IT companies by allowing multiple operating systems and applications to run in parallel on a single physical computing node. Virtualization technology has introduced a new concept where infrastructure, applications and storage have been made available to the public over the Internet which is called *cloud computing*.

1.1.1 Virtual Machines

VMs may be used for different purposes besides being used in cloud computing environments. Institutions use them for educational purposes, programmers use them to test their applications on different platforms, businesses use them as a replacement for physical workstations in order to reduce power usage and maintenance costs. A common use for system administrators is to use them for testing different tools and to host different services in a network.

VMs may be used for many purposes, and within academia they are used to provide students with the means to deploy full-sized machines in a virtual network environment and to solve problems in numerous types of assignments ranging from basic firewall configuration to deploying different services such as DNS servers and load balancers. However, as most educational institutions have limited hardware resources and funding this impose a restriction on the number of VMs which may be distributed to each student, hence narrowing the possibilities of their research.

In 2009 Google began work on a storage and computation system called *Spanner* with the purpose of spanning all their data centers. In a keynote held by Jeff Dean, a *Google Fellow* in the Systems Infrastructure Group, Google estimated their future data centers to scale from 1,000,000 to 10,000,000 machines and 1,000,000,000 client machines[1]. Google's focus on scalability, cloud computing environments and large networks emphasizes the importance of pro-

viding students with the opportunity of working with large-scale networks and to provide them with practical assignments dealing with scalability issues.

As full-sized VMs demand a substantial amount of system resources this limits their full potential as a tool to simulate large-scale environments. This suggests looking at less resource demanding alternatives such as minimal VMs which are built around a stripped-down version of an existing kernel, a microkernel or a minimalistic operating system which requires significantly less system resources.

1.1.2 Cloud Computing

The Amazon Elastic Compute Cloud (Amazon EC2)[2] and Windows Azure[3] are examples of public cloud computing platforms using virtualization to allow consumers to host and run their own applications in the cloud by renting VMs. These cloud computing platforms deliver services in a subscription-based model allowing companies to invest less capital on expensive network infrastructure, and eliminates the need of setting up basic software infrastructures. Minimal VMs would be beneficial in cloud environments for research purposes, and as a platform to provide centralized hosting of software to its users in a software-as-a-service (SaaS) model. Next to the public clouds there are private clouds which often serve research and development purposes for products/services such as Dropbox[4], Google Docs[5] and Microsoft Office365[6]. As more applications are moved to the cloud the infrastructure must also be able to provide sufficient computing resources to host both these applications in addition to client machines.

1.1.3 The Benefits of using minimal VMs

Academic institutions such as Oslo University College who wish to perform research on cloud computing, while at the same time support a large number of VMs on existing hardware for their students, the use of full-sized VMs is considered to be a bottleneck due to their resource requirements. As institutions do not have access to the same level of funding as businesses they are not able to afford the expenses of expanding existing infrastructure and buying the necessary hardware to support all their needs. This exemplifies the benefits of less resource demanding VMs as they would be able to open up additional venues of research while keeping expenses at a minimum.

The need to support a large number of clients, and providing students and researchers with the possibility of simulating large-scale networks on the same infrastructure suggests creating two types of VMs. The first type is to be used for research purposes only and would at least need to support the creation and simulation of different usage patterns of real-world applications. This type of VM would be suitable for organization and administration of a large number

of instances on a network as features such as a command line interface (CLI) or a graphical user interface (GUI) are not required. By removing most of the system services and features found in full-sized VMs, the amount of system resources required by each VM of this type would be reduced substantially, allowing a significantly larger number of VMs to be deployed on the existing infrastructure.

The other type of VM is suggested as a multi-purpose instance offering the same level of usability, extensibility and support of popular application stacks while demanding less system resources compared with full-sized VMs. These VMs may also be used as a replacement for existing client machines, to perform real tasks and to perform communication between instances across the network. This second type of minimal VM is recognized by its broad scope and extensibility which allows researchers and students to model a population of full-sized VMs using minimal VMs while keeping system resource usage at a minimum.

1.1.4 Business Opportunities

Minimal VMs would also be able to be deployed as ready to run environments equipped with custom application stacks for the purpose of running a web server or specific types of services. An example of such a service is *BitNami*[7] which deliver self-contained environments using minimal installations of Ubuntu and openSUSE which are ready to be deployed on Amazon EC2, Windows, Linux and Mac OS X. As minimal VMs would be greatly reduced in size this would reduce the storage space needed on the cloud and require less computing resources as a large number of services have been removed.

1.1.5 Related Work

As data centers and networks continue to grow in size they become more complex, and designers are in need of simulation tools which are able to deliver predictable results about the system requirements of the data center. *CloudSim* is ”a framework for modeling and simulation of cloud computing infrastructures and services”[8] and provides a simulation framework for system designers and developers to enable *”seamless modeling, simulation and experimentation with cloud computing infrastructures and management services.”*[9, p.1]. By instantiating each VM as a small java object the tool is able to create a large number of ”machines” to allow researchers and developers to focus more on system design issues without having to be concerned about low level details related to Cloud-based infrastructures and services. The framework gives researchers the possibility of deploying a large number of VMs in a framework to simulate a variety of different application configurations and to perform extensive testing in multiple scenarios. However, CloudSim does not provide

the ability to deploy a large number of real VMs to create small-scale models of networks and to use them in a real environment as client machines.

The motivational pointers in this chapter show the possibilities of using minimal VMs for modeling, simulation and cloud services. One feasible approach to test the hypothesis of this project is to perform research into the use of μ-kernel technology and minimal operating systems to examine if it is feasible to use minimal VMs as a viable alternative to full-sized VMs by reducing the amount of system resources required by each VM, hence allowing a larger number of VMs to be deployed on existing system infrastructure and hardware.

1.2 Problem Statement

The following problem statements were chosen for this project as outlined by the motivational section.

1. ***To which extent is it possible to model large populations of full-sized virtual machines, using minimal virtual machines on fewer hosts?***

   1.1 *To which extent is a public cloud computing environment able to provide the resources to increase the model-populations even further?*

2. ***Would minimal VMs be able to host custom application stacks as ready to run environments and be able to compete with full-sized VMs used for the same purpose?***

The main focus of this project is to research to which extent minimal VMs may be used to model large populations of full-sized VMs.

1.2.1 Modeling large populations of full-sized virtual machines

The keywords for the given problem statement needs to be examined to express the purpose of the thesis.

To which extent means to examine the feasibility of using a small-scale model and compare it to the behavior of a real-sized population. To which degree would it be possible to achieve the behavior of a large population of full-sized VMs using a small-scale model of minimal virtual machines.

Possible to model means to examine if it would be realistic to create a small population of VMs as a small-scale model that may be used for the design and implementation of a full-scale system. The idea is to see whether or not the results from the small population is able to show the behavior of larger populations.

Large populations defines the number of VMs to be significant. Its size can be defined as being the largest possible number of VMs which may be populated on a system with respect to the limitations imposed by system resources such as CPU, memory and storage. A

Full-sized virtual machine is a virtual machine of standard size. These are VMs built on traditional monolithic kernel designs and *full-size* suggests the sum of its virtual disk size and memory footprint to be of a significance.

Virtual machine, or VM, deviate from a physical machines as it does not provide an hardware abstraction layer like other operating systems.[11, p. 4] The lack of such a layer requires a VM to use partitions of existing hardware resources from the host in order to simulate its own hardware. Multiple VM instances are able to run simultaneously next to each other as they are fully isolated from other VMs as separate processes in user space.

Minimal virtual machine refers to a VM demanding the least amount of computing resources while being smaller than a standard sized virtual machine. The definition of minimal in this context is also used for a type of kernel or operating system which has been stripped down to its core elements.

Fewer hosts describes more than one host, and defines the number of *hosts* as being an indefinitely smaller number. In context with the problem statement it describes a population of VMs to be able to exist on fewer physical machines than full-sized VMs would demand.

1.2.2 Using a cloud to further increase the model-populations

Cloud is the concept of making infrastructure, applications and storage available to consumers through a service provider such as Microsoft or Amazon. These service providers own and maintain the cloud infrastructure freeing companies from the task of low-level hardware configurations and creating their own software infrastructures.

Able to provide the resources means if the public cloud is able to deliver the computing resources to host, increase or move the populations from a physical environment over to the cloud.

Increase the populations of small-scale models even further is the idea of moving a small-scale population of VMs from a physical host over to the cloud and at the same time scale up the number of VMs in the population. It also means to look at which degree a public cloud computing environment is able to provide the resources to increase the populations of small populations to a greater extent, and if scalability and behavior of a such a population is an issue when moving to the cloud.

1.2.3 Using minimal VMs as ready to run environments

The next problem statement wish to research the possibility of using minimal VMs to host custom application stacks as ready to run environments similar to the BitNami[12] model and if they would be able to compete with full-sized VMs with respects to storage and performance.

Custom application stacks is a term used for describing a suite or group of software which is typically required to serve a specific purpose. An example of such a stack would be the LAMP-stack(**L**inux, **A**pache, **m**ySQL and **P**HP) to provide basic web-service functionality.

Ready to run environment in this context is used to describe a virtual machine containing application software which comes pre-configured and ready for deployment on a physical machine or on the cloud. An example is BitNami[12] which supply consumers with a great number of pre-installed application stacks on ready to run images which are compatible with the Amazon EC2 cloud computing environment and are also provided as stand-alone installers supporting the most popular operating systems, such as Linux, Mac OS X and Windows.

Competing with full-sized VMs is a term used to describe if minimal VMs would be able to host custom application stacks while at the same time demand less hardware resources than full-sized VMs. It also wish to examine if minimal VMs are able to offer the same level of performance, extensibility and usability for the consumers in the same way as full-sized VMs are able to.

Institutions and companies who wish to develop SaaS products similar to the *BitNami* model would want to reduce the system requirements of their VMs as much a possible. The BitNami LAMP stack is installed on a minimal installation of Ubuntu and requires at least 256MB of memory and a minimum of 150MB of storage space[13] which suggest a minimal VM would be able to achieve a much smaller memory footprint and storage requirements as system services and device drivers have been moved out of kernel space. Instead of choosing a minimal installation of the Ubuntu distribution there might be even smaller kernels and operating systems offering a better solution. As a μ-kernel or a minimalistic OS only provide the user with system critical elements such as a tiny kernel, a file system, network drivers and a CLI this would support the idea of being able to create a minimal VM which is able to compete with full-sized VMs such as the ones provided by companies such as *BitNami*. The use of minimal VMs would open up business opportunities for institutions and companies seeking to increase their profits by being able to deploy a much larger number of VMs on less hardware than existing alternatives are able to offer.

1.3 Summary of the results

Creating a small-scale model with a population of minimal VMs required to do research into the field of μ-kernel and monolithic kernels. This was necessary in order achieve an understanding of their design- and behavioral differences and to decide if they would be able to result in a minimal VM.

Tiny Core Linux, BareMetal OS and the L4 Pistachio was found to be the most suitable candidates for the purpose of creating a population of minimal VMs as they are significantly smaller when compared with traditional kernels. Researching these kernels was the first step towards creating a minimal VM, and lots of effort was done into learning how to successfully compile the kernels and how to create custom application software for each one.

The minimal VMs created in this project were significantly reduced in size when compared with a full-sized Ubuntu server image which is currently 684MB. The smallest VM image was L4 Pistachio with a size of 1.5MB, a reduction in size of at least two orders of magnitude when compared with standard Ubuntu. BareMetal OS achieved an image size of 32MB and the Tiny Core Linux image achieved 20MB. In theory, this allows to deploy a population of at least 500 minimal L4 Pistachio VMs, or at least two orders of magnitude larger than Ubuntu, and on the same hardware.

Two different types of application software was created for the purpose of benchmarking each of the kernels. The first type of applications had the task of simulating three different CPU usage patterns. The second type of applications generated 100% CPU load by calculating the fibonacci sequence for a given number of iterations without calling the *sleep* function. This caused the CPU to become 100% utilized for the duration of the tests, resulting in a lot of context switches. The results from these tests were used to examine which kernel had the least overhead and better performance.

For this project two different labs were used to conduct the tests. The first lab was a Hypervisor on Hardware lab with KVM virtualization, while the other was a Hypervisor in Cloud environment offering virtualization of VMs inside VMs (nested virtualization). Nested virtualization does not offer the same performance which can be achieved by using a bare metal hypervisor which is why these two environments were chosen for comparison in this project.

Custom tools were created to assist with the deployment of a large number of virtual machine simultaneously, to collect system information and to compile the kernels and build their custom application software. A number of tests were done and the data was analyzed by looking at important metrics such as CPU- and memory usage to examine the behavior of the populations of VMs when deployed on the hardware- and cloud labs.

1.4 Thesis Outline

The structure of this paper is as follows. **Chapter one** is the *introduction* chapter stating the motivational pointers of this thesis, describes the approach and the problem statements.

Chapter two is the *background and literature* chapter which mainly goes into details about the principles and differences of μ-kernels and monolithic kernels and continues on virtualization, clouds, literature and related work.

Chapter three is the *approach* chapter describing the system design of the two labs and the different development environments used for this project. It also elaborates about the approach used to compile, assemble and install the three different kernels and how to create applications for each one. The end of the approach chapter talks about how to create different usage patterns as the approach used to achieve a fair test for all three.

Chapter four is the *results* chapter which begins by going into detail about the operationalization of the different tools created for this project. It continues by presenting the results from the different tests.

Chapter five is the *discussion* chapter which examine the findings of this project and discuss what has been achieved and if there is future work to be done within this research topic.

Chapter six, the final chapter, is the *conclusion* chapter stating the final conclusions for this project.

Chapter 2

Background and literature

2.1 Data Center Modeling

Data centers keep growing in size and increasing their complexity levels as they are used for a range of different network services and for hosting client machines. The design and deployment process of a data center has a significant lead time as designing the infrastructure involve many different stages of development and often require the use of simulation tools to assist in the process. These tools are becoming increasingly more popular among designers and are used to model and display the data center environment by setting a set of parameters for the infrastructure, such as equipment positioning and heat distribution. [14, p. 97]

The increased use of simulation tools used for data center modeling suggests the next step would be simulation tools for modeling the client capacity of the data center. No research could be found where small-scale models of minimal VMs had been used, and while data centers continue to grow this is why a simple small-scale model of minimal VMs would be a useful tool in predicting their client capacity.

A model of a population of minimal VMs may be created by using a variety of different kernels and operating systems of different architecture and size. A truly minimal VM may be built on top of μ-kernels which are the smallest kernels available as they have moved all drivers and system services out of kernel space and into user space where they can be attached to the kernel as optional modules in an un-layered structure. This makes the μ-kernel much smaller than the monolithic kernels which include most of their system services in kernel mode in a layered structure and all system services and drivers are included in the same source code. The difference is illustrated on figure 2.1.

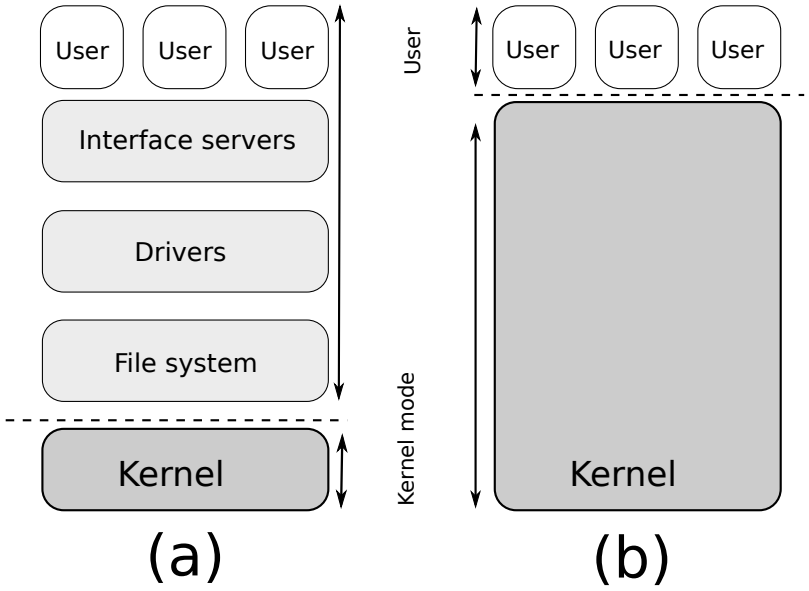

Figure 2.1: Illustrating the difference between μ-kernels (a) and monolithic kernels (b). Notice the difference in kernel size.

2.1.1 Microkernels and monolithic kernels

The microkernel

Microkernels are the result of ongoing changes in the computer world during the 1980s. Especially the development of new device drivers, protocol stacks and file systems spawned the idea of creating a smaller and more extensible type of kernel as an alternative to the existing monolithic kernels. Monolithic kernels are based on a design where all the system services are placed inside kernel space[15] and executed in the same address space resulting in a large kernel. This design offer poor extensibility and becomes difficult to maintain as more functionality is added to the kernel. As a result a simple operation such as bug fixing require the entire kernel to be recompiled. The limitations imposed by the monolithic kernels gave birth to the new microkernel where the idea was to move existing system services out of kernel space into user space and to make each part of the operating system run as separate servers which could be worked on monolithically making it easier to customize the kernel and adding or removal of specific services without working directly with the kernel itself. The new μ-kernel was to be responsible for performing only basic tasks such as process communication and I/O control by implementing alternative inter process communication (IPC) mechanisms.

The new generation of kernels contained less than 20,000 lines of source code resulting in a considerable size reduction and lower fault density when compared with their monolithic counterpart. Some studies have shown that source code in general generate between 16 and 75bugs pr 1,000 lines of code. [16] [17] Smaller kernels generate fewer bugs which makes them more secure

and more reliable as they do not include device driver code in the kernel which is shown to have an even higher fault density with three to seven times the error rate of ordinary binary code[18]. This said, moving device drivers and services out of kernel space drastically improves reliability, increases performance and strengthens the security of the kernel.

As μ-kernels in their earlier years suffered from poor performance as a consequence of their developers trying to implement as many system services as possible, the next generation became far more efficient. Monolithic kernels required on an average of 100μs for a short message transfer on a system with 50Mhz clock rate[10, p. 1] and with the development of the L3 μ-kernel[19] a significantly higher IPC performance was achieved. In 1987 the L3 kernel was built from scratch on the generalization of the Eumel principles[20] which was an operating system developed by Jocken Liedtke and was built upon the principles of persistent processes and data spaces.

The L3 μ-kernel was able to increase IPC performance by an order of magnitude by lowering the message transfer time by a twentyfold optimizing the IPC time from 100μs to 5μs, which the developers claimed to be the result of a *"synergetic approach in design and implementation on all levels"*[10, p. 1]. The successor of the L3 μ-kernel, *L4Ka::Pistachio*, delivered even higher performance than its predecessor as it introduced better support for multi-processor systems, looser ties between threads and address spaces, user-level thread control blocks, virtual registers and had a fast local IPC mechanism.

The Monolithic Kernel

Monolithic kernels are the traditional counterpart next to the μ-kernel. The kernels are designed to include system services inside kernel space as a part of the same address space. Examples of such kernels are *OpenVMS*, *Linux* and *BSD* all of which are able to dynamically load and unload executable modules at runtime. This modularity does not happen at the architectural level but at the binary level which mean unloaded modules are not loaded and stored in memory before they are needed. This flexibility means the operating system image does not need to reboot in order to load additional modules, but instead loads them as they are needed by the kernel. Such an ability is useful for embedded devices or in systems running on limited hardware resources. However, when the code is loaded into memory a small overhead incur which hurt overall performance but also adds flexibility. This feature has made the Linux kernel the most popular choice for embedded devices as a core in the Android operating system[21] which has a high focus on reducing its memory footprint.

A monolithic kernel, such as Linux, contain support for a large number of devices by embedding support for a large number of device drivers inside kernel space which is loaded by the kernel when needed. The extensive num-

ber of device drivers embedded inside the operating system introduce a large amount of bugs into the kernel and each bug has the potential to bring down the entire system. Bug fixing these large kernels requires the entire kernel to be recompiled which is a time consuming operation in the case of Linux, which now has reached in the order of 15 million lines of source code[22].

2.1.2 Importance of Context Switching

Processes can be isolated or may cooperate in accomplishing a common objective as part of a cluster of processes. Sometimes processes also need to exchange data or synchronize their activities and inter process communication(IPC) is the mechanism which provide the ability to communicate between these processes. [23, p. 1-2]

The IPC mechanism implemented by the kernel aims at reducing the CPU overhead generated by context switching as a result of the communication between processes, and reducing the overhead is essential in order to increase kernel performance. A context switch means *"changing currently active memory mappings and CPU registers to the last saved state of a process"*[24, p. 13], in other words switching from one process to another. When a context switch occurs a process saves its current process information, which is available in the CPU registers, and goes from running state into a ready state. While in this waiting state the process waits to be restarted and to receive interrupt instructions. The time for this communication process to finish requires a certain amount of time which is called CPU overhead.

Measuring the CPU overhead generated by each kernel is possible by executing two identical CPU-intensive tasks and measuring the time required to complete each task. The time difference needed to complete the task suggests which kernel generates the most overhead.

2.2 Selecting the right kernel

A number of μ-kernels and tiny operating systems are available and currently in development which are considered to be mature, small, sufficiently documented and fast enough to be compete with monolithic kernels . L4Ka::Pistachio[25], BareMetal OS[26] and Tiny Core Linux [27] have been chosen for this thesis as they are built on the idea of three different designs such as a μ-kernel (L4 Pistachio), minimal operating system (BareMetal OS) and a monolithic operating system (TinyCore Linux).

2.2.1 L4 Pistachio

A cooperation between The System Architecture Group[28] at the University of Karlsruhe and the DiSy group[29] at the University of New South Wales, Australia resulted in the L4 Pistachio μ-kernel[25] and introduced several new concepts to lower IPC costs in order to greatly improve the kernel performance. Direct process switching[10, p. 8], lazy scheduling[10, p. 7], synchronous IPC and using registers when passing parts of, or the entire message[10, p. 8] are some of the mechanisms implemented in contrast to the first generation of microkernels which supported both synchronous and asynchronous IPC resulting in poor performance.

The high performance and small size of the L4 Pistachio kernel makes it suitable as a minimal VM for modeling purposes. The L4 Pistachio kernel is in the order of 10,000 lines of code which makes it tiny when compared to the current Linux kernel which as of kernel 3.2 is in the order of 14,998,651 lines.[22] The L4 kernel exists in both 32 and 64 bit versions, provides multiprocessor support and features a local IPC. L4 Pistachio was built from ground up and is the first available kernel implementation of the L4 Version 4 kernel API and is the product of seven years of research. Its kernel is written entirely in C++ with emphasis on performance and portability. It is currently in development and is maintained by the System Architecture Group in cooperation with the DiSy group.

Because the L4 kernel features an ***"ultra fast local IPC"*** mechanism [30] , is written in C++ and contain less than 10,000 lines of source code it would suggest it to be a good candidate for a large scale population of VMs as context switching between the VMs and the CPU happens much faster when compared to traditional monolithic kernels. An important distinction must be made between L4 Pistachio and a traditional operating system which is that L4 is regarded as a pure μ-kernel and lacks the basic functions of an OS such as a file system, command line interface and device drivers. All of these services must be added to the kernel as modules to classify as an OS. Using a μ-kernel in a large scale scenario when deploying a large number of virtual machines would help keeping memory footprint and system resource usage to an absolute minimum allowing a large number of minimal VMs to be deployed. Though this kernel is not an OS, running the kernel with only a selected number of custom written user space applications simulating real usage patterns would make it possible to simulate the real behavior of an OS.

2.2.2 BareMetal

BareMetal OS is a 64-bit operating system written entirely in Assembly. Although the kernel itself is written entirely in assembly code it support applications written in both Assembly or CC++. It offers a native file system with readwrite support for FAT16 as well as networking capabilities and monotask-

ing capabilities with support for up to 128 64-bit processors. To use BareMetal OS an IntelAMD-based 64-bit CPU is required and with at least 32 MiB available on the hard drive. In addition the OS itself needs 2MiB of memory as well as reserving 2MiB of memory per CPU core. BareMetal is an open source project and also offer online documentation and forums which makes the OS a good candidate for this thesis.

2.2.3 Tiny Core Linux

Tiny Core Linux[27] is one of the smallest available distribution of the Linux kernel and is between 1100 and 1400 the size of the most common operating systems used worldwide. It includes a set of tools such as busybox, offer extensibility by supporting a large number of extensions and features a command line interface. The current version of TCL at the time of writing support version 3.03 of the Linux kernel, is less than 10 megabytes and does not require a harddrive installation as the operating system is able to run in its entirety with 48MB of RAM. TCL has an active open source community and is led by a team of developers and also offer a lot of documentation.

2.3 Virtualization

Virtualization had its origins in the 1960's and was developed by IBM Corporation as they had one single physical hardware mainframe host and wished to partition it into several logical instances. Since the 60's the virtualization technology has become increasingly more popular and is now recognized as being an essential part of the IT industry as we know it.

Virtualization is achieved by installing a piece of software which imitates a selection of hardware components or even the whole computer itself. The software is installed on a computer and acts as a virtualization layer by using either a hypervisor or a hosted architecture solution. A hypervisor implements a virtual operating platform for the guest operating systems by installing the virtualization layer on a clean x86 system which is dedicated to running guest operating systems. In contrast, the hosted architecture is installed as an application on top of an existing OS and supports the widest variety of hardware configurations.

2.3.1 Kernel-based Virtual Machines

Kernel-based Virtual Machines (KVM) is a full virtualization solution for Linux on x86 hardware supporting extensions such as Intel VT or AMD-V.[31] Virtual machine monitors require the use of such extensions and they enable running fully isolated virtual machines at native hardware speeds, with the exception

for some workloads. KVM allows unmodified Linux or Windows images to be used as virtual machines as each machine recognizes virtualized hardware such as network cards, disks and graphics adapters and supports 64 bit processors. A slightly modified QEMU program is used by KVM to execute the virtual machine as a regular process which can be managed by *top*, *kill*, *taskset* and other tools. Using KVM in a production environment recommends the use of KVM modules shipped together with the Linux distribution to avoid critical errors and instability. As previously mentioned, the KVM VM instance is seen as a process by the operating system and a command such as *kill -9* would kill the process and reclaim all the resources it used to have.
KVM is open source and included as a kernel component in the Linux kernel from version 2.6.20 and higher which makes implementation easy. Their website also include an extensive amount of documentation combined with the use of multimedia which makes usability and documentation one of KVMs strong sides.

2.3.2 Cloud Computing

Cloud computing environments such as Amazon EC2[2] and Windows Azure[3] are examples of public clouds which provide consumers with virtual resources available over the Internet. Clouds use the following service models:

Software as a Service(SaaS): The cloud service provider deliver software as a service to the consumer according to their requirements.

Platform as a Service(PaaS): Consumers are given platform access on the cloud enabling them to move custom software and different types of applications onto the cloud.

Infrastructure as a Service(IaaS): Basic computing resources such as storage and network capacity is granted to the consumer. This provides the consumer with the possibility of managing operating systems and network connectivity on the cloud.

The Cloud makes it possible for anyone to gain access to their own personal VM on the Internet by using web service APIs. Having root access to a VM on the Cloud offers the same level of control as with any other physical machine. Public clouds such as the Amazon EC2 is flexible as it offers the choice of deploying multiple instance types, operating systems and software packages as a subscription based service where you pay as you use the service. In addition it is also possible to choose from a variety of different memory, CPU and storage configurations for each VM.

Using the Amazon EC2 cloud is relatively inexpensive when compared to using physical systems since customers pay a low rate for the processing power. There are three types of instances to choose from, and their economic

models range from payment **by the hour**, a **one time payment** for each instance and **spot instances** where you bid on unused Amazon capacity and let the instances run until the bid meets or exceeds the current spot price.

In addition to the different instance types it is also possible to deploy VMs on multiple geographically dispersed locations sorted into different regions and availability zones. The advantage of locating multiple instances in different geographical locations across the globe is protection from a single point of failure of one single region or location. Regions are dispersed availability zones are placed in separate geographic areas; *US East, US West, EU, Asia Pacific (Singapore), Asia Pacific(Tokyo) South America and AWS GovCloud*, all of which adhere to the Amazon Ec2 Service Level Agreement of 99,95% availability. [32]

Consumers are able to choose from a variety of different instance types to meet their needs. The *standard family* of instances in the Amazon EC2 Cloud will be suitable for most uses and deliver from 1.7-7.5GB of memory and 1-8 EC2 Compute Units. Other instance types are high-memory instances which offer a large amount of memory to support high throughput applications such as databases. High-CPU instances are designed to meet the demands of consumers requiring the support for compute-intensive applications.

Preconfigured Amazon Machine Images (AMIs) are provided to the consumers by Amazon but it is also possible to upload custom operating systems and both Linux and Windows operating systems are supported.

2.4 Virtual machines as ready to run environments

With the increasing popularity of cloud computing environments such as Amazon EC2, new products have started emerging such as VMs with ready to run environments delivering pre-installed bundles of the most popular open source web applications, frameworks and their dependencies.

BitNami[12] is one provider of such services where a minimal installation of Ubuntu and openSUSE is used to host the application stack free of charge to consumers. The goal of BitNami is to make open source software more available which is why their stacks are available as native installers, VM images and Amazon EC2 cloud images. BitNami also has plans to release cloud images supporting additional clouds in the near future. Current stacks are available for all the major operating systems such as Linux, Windows and Mac OS X and are compatible with virtualization software packages such as VMWare and VirtualBox.

2.4.1 BitNami Amazon Cloud Images

BitNami Cloud Images are a collection of applications and its dependencies. These images are pre-configured and ready to be deployed on the Amazon Elastic Compute Cloud (EC2). Consumers choosing to use these images on the cloud does not need to invest in hardware or to install the applications on a physical machine. The minimum requirement on the consumer's side is signing up for an account at Amazon EC2 after which launching the cloud image is an automatic process without the need of uploading the image to the cloud. Once started the instance is operated as any other instance hosted on the cloud by using the web interface of EC2 to start and stop the instance when needed.

2.4.2 BitNami Virtual Machine Images

Minimal Linux operating systems are used as operating systems for these images which come pre-installed and configured with a BitNami application stack. These images are delivered as ready to run VMs compatible with VMware and Virtualbox and enables consumers to start and stop VMs as any other application without having to install anything besides the virtualization software on their machine.

2.5 Related Work

2.5.1 CloudSim

CloudSim[8] is a tool used by researchers to design and manage large data centers and is also used to simulate large populations of VMs. The tool is used for several purposes such as evaluation of resource allocation algorithms for HP's Cloud data centers, energy-efficient management of Data Centers, evaluating design and application scheduling in Clouds, SLA oriented management and optimization of Cloud computing environments and investigation on workflow scheduling in Clouds. [33, p.22]

The tool is able to simulate millions of VMs by instantiating them as tiny java objects. Benchmarks show deploying a population of 1,000,000 hosts required approximately 12 seconds and memory usage never grew beyond 320MB.[33, p.18]. CloudSim is able to offer researchers with a framework requiring only a fraction of the hardware resources needed to simulate a large scale environment in environments such as a Cloud. However, it is not able to create real VMs for simulation purposes.

However, as java objects are not real VMs, the use of minimal alternatives for simulation purposes is an important research topic as this approach may prove as a viable alternative for simulating real environments. Minimal VMs

would provide students and researchers with a tool to deploy large populations of VMs and to simulate scalability, network connectivity and real-world problems.

Chapter 3

Approach

This chapter states the different system designs used in this projects. It explains how to create the minimal VMs for L4 Pistachio, BareMetal OS and TCL and continues on how to develop custom applications for each kernel. It begins with suggesting a preliminary specification of the different tools required to fulfill the requirements of the project. A description of the operationalization of the final tools are explained later in the Results chapter.

As L4 Pistachio and BareMetal OS are active research projects their documentation was not able to provide all the details necessary about how to create build environments, Qemu images, applications or how to install applications onto the images to make them available for the kernel. The purpose of this chapter is to show the steps required to successfully compile/assemble the kernels, how to create applications for each of them, how to create Qemu images and how to install the applications onto these images. As the task of installing files onto the images proved to be a complex and time consuming task custom tools had to be created to speed up the process.

3.1 Required Tools

Two different tools have been suggested as a minimum for this project; one should have the responsibility of deploying virtual machines while the other has the responsibility of collecting system information from the host. The programming language Perl is a good candidate for the task as it is suitable for gathering system information, performing file operations, handling user input and is able to execute system commands.

3.1.1 Automatically Booting Virtual Machines

The script must accept input from the user such as the name of the Qemu-image, the number of virtual machines to be deployed and the type of VMs to

boot. Before each VM is deployed a mechanism in the script might be implemented to check CPU and memory usage to prevent more VMs to be deployed if the system is running out of available resources.

A loop could be used to boot one VM until a stop condition terminates the initialization of additional VMs. The stop condition is suggested as being the same as the maximum number of VMs decided by the user upon execution of the tool.

To initialize multiple VMs each VM should be booted with a universally unique identifier (UUID). The script should generate as many UUIDs as the requested number of deployed VMs and store each ID in an array or a file. There is a Linux tool available which is used generate valid UUIDs called *uuidgen* and generates a string similar to *84da4f5f-b884-441d-b9e5-5ff37e866973*. This tool may be executed inside its own loop at the beginning of the script, before the actual deployment of VMs, where the loop must iterate an equal amount of times as the desired number of VMs in order to create a unique identifier for each instance. To be able to boot multiple instances using the same VM image it is necessary to give each VM instance unique process IDs and process names which also helps to identify each VM. The following naming scheme is suggested and use only a string and a counter to achieve the following format; *vm_process_1*, (...) *vm_process_12*. These names may be achieved by inserting the VM- and process-names in the *Qemu* command used to execute each VM by using the following Qemu-options: ***-name*** *l4_12,****process****=l4_process_12*.

After generating the UUIDs the script is suggested to collect system information such as CPU and memory and calculate system critical levels to decide if there are enough available system resources for deploying additional VM on the system. This is a useful feature if one wishes to avoid saturation of memory or CPU by not allowing additional VMs to boot if the CPU load is too high or the amount of available memory is too low. Standard Linux system tools such as *top* and *free*, or the CPAN Perl module *Sys::Statistics::Linux* may be used to gather CPU and memory information to be used for this purpose.

As different VM instance types are decided by user input the script should check the input to decide which Qemu image to load. This decision can be achieved by comparing the user input in an if-test to decide the appropriate command to execute.

To keep the loop from deploying all of the VMs immediately a delay function is suggested. This could be achieved by using a *sleep* timer to allow enough time for the current VM to finish its boot process and to let the CPU and memory finish its operations and to avoid queuing and slowdown of the system during the boot phase.

Using the *system(), exec()* or backticks ‘‘ functions of *Perl* may be used to issue the appropriate Qemu-commands to boot the VMs. The difference be-

tween the three methods are that *system()* will execute a command specified in $command by calling "/bin/sh -c $command", and returns after the command has been completed, waits for the command to finish and only returns the exit status of the executable ignoring any output. Using *exec()* will return an error code only if it cannot find the executable and ignores any output or return values, while using the *backticks ` ` * method should be used only if it requires to collect the output generated by running the command. Using the system() function is suggested as the most suitable function to use as booting a VM requires only the execution of a command "/bin/sh", wait for the command to have been executed and then ignore any output. It also returns the exit status of the command making it possible to exit the script if something unexpectedly occur such as a VM crashing during boot. The loop should keep iterating an equal amount of times as there are UUIDs in the array or in the output file.

3.1.2 Statistics Script

A script is suggested to be developed to collect statistics about system resource usage on the host. By comparing these data one should be able to decide which of the kernels would be suitable for creating the largest population of VMs. Analyzing CPU and memory usage while the VMs are running suggest the script to accept user input such as the number of data samples to collect, the delay in seconds between each collected sample, name of the log file and name of the process to search for to allow counting the number of running VMs. The results in the collected samples should be printed into a comma separated log file for readability and would be useful for later analysis and for generating graphs.

A variety of metrics are suggested to be collected such as CPU usage and averages, paging, memory, processes and swap statistics to be able to measure how the different VMs behave on different environments. As the VMs are suggested to execute a selection of custom made applications with the purpose of generating real usage patterns, increasing the amount of VMs and comparing the data from running a smaller population should be able to tell if the VMs are behaving in a predictable way.

Linux offer multiple tools which may provide information about system resource usage such as *free, uptime, top, pidstat*. These are tools which may be executed inside the script in order to gather the output, however, they require a lot of additional CPU and memory resources when they are executed. Counting the number of VMs is important to verify how many VMs that were active while collecting the data samples and counting these processes may be achieved by searching through active processes using the VM process name using parsing of output from running the "top" command. Executing these tools using the and capturing its output using *backticks ` ` * is rather resource demanding and is not recommended. However, a faster and more

appropriate method of collecting these data is to use the CPAN Perl module, *Sys::Statistics*[34], which collects data from the virtual proc filesystem instead of using the aforementioned Linux tools. The CPAN module is also able to search through running processes on the system, which is a better solution than collecting output from other more resource demanding system utilities.

The documentation of *Sys::Statistics::Linux* CPAN module recommends setting a sleep timer of minimum 1 second to be implemented to force a delay between the data collecting in order to let the module complete its collection process.

The user should provide the number of data samples to collect when calling the script which suggest to use a loop which uses the number of data samples to collect as its stop condition. Calling a subroutine in each loop iteration is suggested as being a convenient method as it moves a lot of code outside of the loop and results in better readability of the source code. Each time the subroutine is called it should collect the requested type of system information from the *procfs* filesystem and store the data in the output file specified by the user.

Relevant Data To Collect From The Host

This section suggests, and explains, five categories of relevant system information to be collected by the script using the *Sys::Statistics::Linux* module.

1. CPU Usage
2. CPU Load
3. Memory Usage
4. Swap Usage
5. Paging

CPU Usage statistics includes the following subcategories:

- user
- system
- idle
- total

The ”user” category collects a summary of the percentage of CPU utilization happening at user level and these values are related to processes running at user level, which are processes belonging to applications. Each guest VM

on the host is regarded as an application by the system and will show up in this category. For all the subcategories, with the exception for ”idle”, a value of 100 means the CPU is constantly busy, and a value of 0(zero) means there are no CPU activity.

CPU utilization occurring at the system(kernel) level show up in the category ”system” and the values show a summary of the percentage of CPU utilization. The summary does not include time spent servicing interrupts or softirqs.

Time spent by the CPU in idle mode show up in the ”idle” category which show percentage of time the CPU spends in idle state. A value of 0 means the CPU is constantly busy, while 100 means the CPU is idling.
The final subcategory, ”total”, adds up the total percentage of CPU utilization at both user and system levels.

As the CPU usage may be significant when collected by the script the burst of activity might occur at just that specific moment which is why **CPU load** must also be collected. CPU load average is the average of the actual load on the CPU, and there are three load-average values possible to collect from Linux, the 1-minute, 5-minute and 15-minute averages. Measuring the CPU load is the same as measuring the trend in CPU utilization instead of a single snapshot. Additionally it includes a complete measurement of the entire CPU demand instead of a what the demand was at one given time. For each CPU core the number should stay below 1.00, which mean that for a dual core system the maximum number is 2.00, while on a quad core CPU the number is 4.00.

Collecting information about **memory** is important in order to examine how the different VMs use memory resources, and the following four subcategories have been suggested:

- Total
- Used
- Free
- Cached

Memory total is the total amount of memory available on the system and does not change during the test. The most important data is collected from the *Used* category which tell how much memory is being used in the order of kilobytes.
The ”cached” values tell how much data is stored in a temporary reserved area in RAM in order to increase the processing speed, while ”free” show how much memory is available.

The **swap** category is suggested to collected the same categories as for memory:

- Total
- Used
- Free
- Cached

The first subcategory, ”Total”, show the total size of swap space available on the host while ”Used” and ”Free” show the size of swap space which is either used or available and all values are collected in kilobytes. Swapping occur when applications use all available memory and stores data on the disk or another location during transfer. This decrease system reliability and is useful data to collect to examine if some of the kernels suddenly show a decrease in performance.

Sharing memory among multiple user space processes simultaneously is achieved by **paging**. Processes are allocated fixed sized memory pages each having its own logical memory space used to process data, which is key to effective memory utilization. A a high number of page faults indicates degraded performance of either a program or an operating system and a low number indicates the opposite. Collecting information about the number of page faults occurring on a system is important in order to compare the different kernels with respects to optimization and performance.

Collecting information about ”minor” and ”major” page faults are suggested. Minor page faults require the page fault handler in the operating system to tell the memory management unit to point to the page, indicate it as being loaded in memory but does not require the contents to be read into memory. The major page are more expensive than minor faults as the the page fault handler needs to find a free page, read the data to that page, mark it as not being loaded into memory and read the data for that page into the page itself. THe memory management unit is then told to make an entry for that page pointing to the page in memory and finally indicate that the page is loaded in memory. Major faults are used by an operating system when the amount of memory available on demand needs to be increased.

3.2 System Design

This section states the design and system requirements for the build-, and development environments and the virtualization labs. It also explains the approach on how to build and configure L4 Pistachio, BareMetal OS, how to install TCL and how to develop basic applications for each of the three.

Type	Description
PC	Dell Dimension 920 Workstation
CPU	Intel(R) Core(TM)2 Quad CPU Q6600 @ 2.40GHz. 1 CPU, 4 cores, 4 threads, L1 Cache=32KiB,L2 Cache=8MiB
RAM	2x2GiB DIMM DDR. Synchronous 800 MHz (1.2 ns).

Table 3.1: Hypervisor on Hardware system specifications

3.2.1 Hypervisor on Hardware

A Dell Dimension 9200 workstation with an Intel Core2 Quad-CPU Q6600 @ 2.4Ghz.and 4GiB of DIMM DDR 800Mhz memory is used as a *Hypervisor on Hardware* (HoH) lab and is also used as to hold the development environments for the three kernels. It is configured as a dual-boot system in order to be used as both a 64-bit virtualization lab and build environment for BareMetal OS and as a 32-bit development environment for L4 Pistachio and Tiny Core Linux. Two separate hard disks are installed for storage each with its separate OS to allow dual booting Ubuntu Server 11.10 in 32-bit, and 64-bit distributions.

Table 3.1 lists the system specifications:

3.2.2 Build and Development Environments

Build- and development environments for *L4 Pistachio*, *BareMetalOS* and *TinyCoreLinux* require a dual-boot system to be created for this project. L4 Pistachio needs to use a x86-x32 Gnu/Linux system for this project in order to successfully compile a 32-bit version of its kernel and its binaries. BareMetal OS needs a 64-bit environment to assemble its kernel and its custom applications. Using a 64-bit system is to avoid creating a 32-bit cross compiler toolchain which is a complex approach. TinyCoreLinux is able to be used on both 32- and 64-bit environments as does not require its core to be compiled, but requires 32-bit build options to be enabled when compiling its applications.

The HoH lab uses KVM virtualization and to make sure the CPU supports hardware virtualization two commands can be used to verify that KVM is supported:

```
1: egrep -c '(vmx|svm)' /proc/cpuinfo
2: kvm-ok
```

If the CPU supports hardware virtualization command A should show **1** it KVM is enabled and **0** if not. The second command should provide the following output:

```
INFO: Your CPU supports KVM extensions
```

Type	Description
EC2 VM	High-CPU Extra Large Instance. Suitable for compute intensive applications.
CPU	26 EC2 Compute Units with 8 virtual cores. (2.5 EC2 Compute Units pr. core)
RAM	7GiB Total
Disk	298GiB
OS	64-bit Ubuntu 11.10 Server

Table 3.2: Hypervisor in Cloud system specifications

HoH is used both as a 64-bit and 32-bit virtualization lab using the Ubuntu 11.10 distribution. The KVM documentation website[35] recommends using a 64 bit kernel as a hypervisor of two reasons. First, using a 64-bit OS will allow to serve more than 2GB of RAM for the VMs, and second a 64 bit kernel may host both 32 bit and 64 VMs.

3.2.3 Hypervisor in Cloud

Hypervisor In Cloud (HiC) is a 64-bit Ubuntu 11.10 Amazon EC2 instance and has the purpose of serving as a virtualization lab and offer 20 EC2 Compute Units for a total of 8 CPU cores and 7 GB of memory. System specifications for the HiC lab is listed in table 3.2.

3.3 Getting started with L4 Pistachio

This show the approach used to successfully build a 32-bit L4 Pistachio kernel and how to develop a simple *Hello World* root-task application. The procedure involves configuring and compiling the kernel, configuring and installing user-level files, creating the *Hello World* root-task binary and explains how to make the kernel bootable. Compiling the kernel involves four steps where the first is to configure and compile the kernel3.3.2, next the *Hello World* application is created3.3.3, the third step is to configure and install user-level code3.3.4 and the final step is to create a bootable floppy disk image by installing GRUB legacy onto the image file.3.3.5.

Table 3.3 lists all the system requirements for creating the L4 build environment.

Ubuntu 11.10 comes with GRUB 2 as its default boot loader which is a replacement for the previous GRUB version 0.9x, now known as GRUB Legacy. Creating a bootable floppy disk image for Qemu requires GRUB Legacy to be installed on the host. This is necessary in order to install GRUB files onto the image. GRUB2 (v1.99) is the default boot loader and manager used in

Software	Version
OS A: Ubuntu Server 32bit	11.10
qemu-kvm	0.14.1
build-essential	11.5ubuntu1
Grub	0.97
autoconf	2.68

Table 3.3: L4 Pistachio build environment Specifications

Ubuntu since version 9.10 and as no documentation exists on how to use GRUB2 with the L4 Pistachio kernel reverting to GRUB Legacy was necessary for this project. The procedure on how to revert to GRUB legacy may be found on the GRUB2 documentation webpage[36].

The package *Autoconf* is necessary to create the file *configure* during the build procedure and may be installed using apt-get:

```
apt-get install autoconf
```

L4 Pistachio source files may be downloaded from its Github repository[37] and unzipped into your source directory on the build machine. Unzip the contents of the zip file in your user's home directory and rename it to *l4ka-pistachio* and the source files should now be located in */home/$USER/l4ka-pistachio/*.

Qemu is not required for building L4 Pistachio but is required to test the VMs to verify if the build was successful and the VM works. The documentation for L4 Pistachio recommends the use of GCC version 3.2 or later, and version 4.6.1 was found to successfully compile the kernel. GCC and other build tools are delivered through the *build-essential* package and these software packages does not require additional configuration before compiling the kernel.

Before compiling L4 Pistachio the dependencies in table 3.3 is assumed to have been installed on an x86-x32 Gnu/Linux system.

3.3.1 The Several Components of L4 Pistachio

The Kernel

The L4 kernel source code is located in the "kernel" directory of its source files and is configured by executing the command *make menuconfig* which opens up a configuration program where parameters for Hardware, Kernel and Debugger can be set.

Kickstart

The kickstart module is a generic and extensible boot-strapper for the L4 Pistachio kernel. It supports the IA32 and AMD64 systems and is responsible for loading and configuring the kernel with architectural configuration parameters including available and reserved memory areas. Other responsibilities include loading and registering the initial server's memory location, and it also starts the kernel.

Sigma0

Sigma0 is the memory manager for L4 Pistachio which owns the entire address space during startup. It is responsible for resolving page faults for the root task which is the first task run by the kernel. It runs in unprivileged mode, but can also be seen as part of the kernel.

The Roottask

A root task is the first address space created at boot time and can perform privileged system calls and can control system resources such as threads, address spaces and physical memory.

3.3.2 Building the L4 Kernel

This states the approach used to successfully compile the L4 kernel source code located in the "l4-source/kernel" directory resulting in the binary kernel file *x86-kernel* and explains the configuration parameters which have to be set before the build process is initiated. The steps will be explained thoroughly to allow replicating the same settings used for this project.

Throughout this approach it is assumed the source files of L4 Pistachio have been extracted to the "/home/$USER/l4ka-pistachio/" directory. Enter the kernel directory and while inside the directory create the build directory where the binary kernel will be built:

```
cd /home/$USER/l4ka-pistachio/kernel
make BUILDDIR=/home/$USER/l4ka-pistachio/x86-x32-kernel-build
```

The next step is to configure the kernel according to the same settings used for this project. Enter the kernel build directory created by the "BUILDDIR" command and execute the following commands which will configure and make the kernel:

```
cd /home/$USER/l4ka-pistachio/x86-x32-kernel-build
make menuconfig
make
```

The following options were used when running the *make menuconfig* command inside the ”kernel” directory.

- Hardware

  X86

  32 bit CPU

  Pentium 1

  PC99 compatible

- Kernel

  Disable debugging mode

Leave all other options disabled
After running *make* the kernel binary has been successfully compiled and placed in the path of the *BUILDDIR* directory set earlier.

3.3.3 L4 Pistachio ”Hello World” application

The following explain the approach used to create a simple roottask application whose single purpose is to print a single line of text in a certain time interval. This is the same approach which is used to create custom applications later in this project and is explained thoroughly as the online documentation does not mention how to create programs for L4 Pistachio. Creating an L4 root-task requires basic knowledge about programming in CC++. For conveniency the complete code and the structure of its Makefile is provided in appendix A and B. Place both the source code and its Makefile inside a directory in */home/$USER/l4ka-pistachio/user/apps/hello*.

The following files will have to be edited to set the location of the application source files and its Makefile required for the compiler to compile the application:

```
/home/$USER/l4ka-pistachio/user/apps/Makefile.in
/home/$USER/l4ka-pistachio/user/apps/user/configure
/home/$USER/l4ka-pistachio/user/apps/user/configure.in
```

Find the following line in *Makefile.in*, add the *hello* directory but **keep** the *l4test* directory as it contain some applications which may be useful to verify that L4 has been successfully compiled. This step has to be performed so the compiler can find the hello world application files during the make process:

```
1 SUBDIRS= l4test hello
```

Next, go to line **3105** in the file *configure* which should look like the line below and add the string ”apps/hello/Makefile” which should point to the location where the compiler may expect to find the *hello world* application *Makefile*:

```
1 ac_config_files="$ac_config_files config.mk Makefile lib/Makefile lib/l4/Makefile
      lib/io/Makefile serv/Makefile serv/sigma0/Makefile apps/Makefile
      apps/bench/Makefile apps/bench/pingpong/Makefile apps/grabmem/Makefile
      apps/l4test/Makefile apps/hello/Makefile util/Makefile util/kickstart/Makefile
      util/grubdisk/Makefile util/piggybacker/Makefile util/piggybacker/ofppc/Makefile
      util/piggybacker/ofppc64/Makefile contrib/Makefile contrib/elf-loader/Makefile"
```

In the same file go to line **3823** and add the following line:

```
1 apps/hello/Makefile") CONFIG_FILES="$CONFIG_FILES apps/hello/Makefile" ;;
```

Finally open the file *configure.in*, go to line 353 and add *apps/hello/Makefile* on a new line just below the other applications which are already listed.

```
1 apps/l4test/Makefile
2 apps/hello/Makefile
3 util/Makefile
```

3.3.4 Configuring User-Level

This explains the approach used to configure and install user-level code. The first step is to change to the "user"" directory where the commands "autoheader" and "autoconf"" must be executed which will create the files *config.h.in* and the *configure* script.

```
cd /home/$USER/l4ka-pistachio/user
autoheader
autoconf
```

Enter the L4 source file top directory and create a new sub directory which will contain the temporary user-level build files:

```
mkdir /home/$USER/l4ka-pistachio/x86-x32-user-build
cd /home/$USER/l4ka-pistachio/x86-x32-user-build
```

While inside the above mentioned directory execute the configure command below to execute the "configure" script which will configure the software and check for dependencies:

```
../user/configure --without-comport --with-kickstart-linkbase=0x148030
    --with-s0-linkbase=0x20000 --with-roottask-linkbase=0xEA60
    --prefix=/home/$USER/L4/l4ka-pistachio/x86-x32-user-install
    --with-kerneldir=/home/$USER/L4/l4ka-pistachio/x86-x32-kernel-build/
```

Linkbase commands are necessary to link kickstart, s0 and roottask (hello world) to a different base in order to reduce memory footprint. Without the linkbase parameters the kernel will require at least 16MB of RAM to boot successfully. The linkbase values are hexadecimal and reflect the memory allocation size of each module which require these values to be changed according to the size of each binary.

Setting the kickstart linkbase to a value lower than 1MB (1 048 576 bytes) results in a warning saying setting the value lower than 1MB is not allowed. Setting the kickstart value to lower than 0x148030 also caused the kernel to crash on some occations. Setting the sigma0 linkbase to lower than 0x20000 (131 072 bytes) also makes the kernel un-bootable, and 0x2000 was found to be the minimum value. Setting the roottask linkbase looks to be a matter of converting the size of the binary from size in bytes converted into hexadecimal values. Since the binary size of the hello world code in appendix A is 38966 bytes it was decided to use a linkbase of 0xEA60 which is 60 000 bytes to be sure the binary would have enough memory allocated to execute.

After the configuration of the user-level code has completed, run the following commands inside the "/home/$USER/l4ka-pistachio/x86-x32-user-build" directory to begin the software installation.

```
make
make install
```

The final command will place all the user-level binaries in the directory passed in the "–prefix" option:

```
/home/$USER/l4ka-pistachio/x86-x32-user-install/libexec/l4
```

3.3.5 Grub Legacy

Make sure the build environment is using GRUB legacy as there are issues making Grub2 work with L4Ka-pistachio. As ubuntu versions later than v9.10 use GRUB2 as boot loader it is recommended to read the following guide[36] on how to revert to the previous GRUB legacy.

The first step in making L4 bootable using GRUB Legacy is to create a temporary directory structure which will contain the files needed for creating a bootable floppy disk image for qemu-kvm. This step involves creating a directory structure where the kernel, kickstart, sigma0, roottask and the GRUB Legacy files will be placed.

```
mkdir /home/$USER/l4ka-pistachio/fdsource
mkdir /home/$USER/l4ka-pistachio/fdsource/boot
mkdir /home/$USER/l4ka-pistachio/fdsource/boot/grub
touch /home/$USER/l4ka-pistachio/fdsource/boot/grub/menu.lst
```

All the binaries and Grub files must then be copied to the *fdsource* directory.

```
cp /home/$USER/l4ka-pistachio/x86-x32-kernel-build/x86-kernel
    /home/$USER/l4ka-pistachio/fdsource/
cp /home/$USER/l4ka-pistachio/x86-x32-user-install/libexec/l4/sigma0
    /home/$USER/l4ka-pistachio/fdsource/
cp /home/$USER/l4ka-pistachio/x86-x32-user-install/libexec/l4/kickstart
    /home/$USER/l4ka-pistachio/fdsource/
cp /home/$USER/l4ka-pistachio/x86-x32-user-install/libexec/l4/hello
    /home/$USER/l4ka-pistachio/fdsource/
```

```
cp /boot/grub/stage1 /home/$USER/l4ka-pistachio/fdsource/boot/grub
cp /boot/grub/stage2 /home/$USER/l4ka-pistachio/fdsource/boot/grub
touch /home/$USER/l4ka-pistachio/fdsource/boot/grub/menu.lst
```

Edit "menu.lst" add the following lines and be sure to enter a new line after the last entry.

```
1 root (fd0)
2 default=0
3 timeout=3
4
5 title L4Ka::Pistachio
6         kernel /kickstart
7         module /x86-kernel
8         module /sigma0
9         module /hello
```

3.3.6 Creating an L4 Bootable Qemu Image

This part explains how to create a bootable floppy disk image with enough space to hold the kernel, user-level, root task and GRUB Legacy files.

Create the floppy disk image inside the L4 Pistachio source code directory by following the steps below.

```
dd if=/dev/zero of=fdimage.img bs=512 count=2880

/sbin/losetup /dev/loop0 fdimage.img
/sbin/mke2fs /dev/loop0

mkdir /mnt/fda
mount /dev/loop0 -o loop /mnt/fda
chmod 777 /mnt/fda

cp -aR /home/$USER/l4ka-pistachio/fdsource/* /mnt/fda/

umount /mnt/fda

cat <<EOF | /usr/sbin/grub --batch --device-map=/dev/null
> device (fd0) /dev/loop0
> root (fd0)
> setup (fd0)
> quit
> EOF

# Unmount loop device
/sbin/losetup -d /dev/loop0
```

The L4 image is now a compatible Qemu virtual machine image can be booted by using the kvm command below:

```
qemu-system-x86_64 -cpu pentium -m 2 -fda fdimage.img -net none
```

Software	Version
Ubuntu Server 64bit	11.10
qemu-kvm	0.14.1
build-essential	11.5ubuntu1
Pure64 Image	v0.5.0
BareMetal OS Source	0.5.3
Newlib C Library	1.20.0
kpartx	0.4.9-2
NASM	2.09.08

Table 3.4: BareMetal OS build environment

3.4 Getting Started with BareMetal OS

3.4.1 BareMetal Environment

BareMetal OS build environment is recommended to be created on a 64-bit x86_64 GNULinux system in order to avoid building a cross compiler. BareMetal OS does not accept the standard C headers but comes with support of its own library and also support the *Newlib* C library. Building applications for BareMetal OS require 64-bit compiler tools and also requires linking GCC to the BareMetal OS library or Newlib headers.

Table 3.4 lists the required software needed for creating a working 64-bit build environment.

3.4.2 Assembling the Kernel

When executing an application made for BareMetal OS and implementing its native sleep function ”b_delay” encounters a bug where the CPU utilization stays at 100% when the function is called. Fixing this behavior is possible using a simple hack in the BareMetal OS source code in the two files ”misc.asm”, ”interrupt.asm” and consists of a 2-line fix.

The fix is to add a call to ”os_smp_wakeup_all” to broadcast whenever the RTC interrupt fires. This will wake up all CPU cores instead of the one that is waiting so it is to be regarded as a simple ”hack” until the developer implements something more ideal.

To implement these changes it is necessary to re-assemble the BareMetal OS kernel using **NASM** and place the new binary (”kernel64.sys”) on a working Pure64 Qemu image. There is a Qemu compatible image available on the website of Pure64[38] and is located inside the zip file as an ”.img” file.

Download and extract the BareMetal OS source files to your harddrive which is available at the BareMetal OS Github project website[39].

After extracting the source files edit ”misc.asm” in the ”os_delay” function

and make the following changes to modify the loop to 'hlt' until an interrupt is received:

```
os_delay_loop:
        hlt
        cmp qword [os_ClockCounter], rax ; Compare it against our end time
        jle os_delay_loop  ; Loop if RAX is still lower
```

Finally, in "interrupt.asm" in the "rtc function" edit the following:

```
rtc_end:
        mov al, 0x0C   ; Select RTC register C
        out 0x70, al   ; Port 0x70 is the RTC index, and 0x71 is the RTC data
        in al, 0x71   ; Read the value in register C
        mov rsi, [os_LocalAPICAddress] ; Acknowledge the IRQ on APIC
        xor eax, eax
        mov dword [rsi+0xB0], eax
        call os_smp_wakeup_all  ; A simple but "terrible" hack
```

After having made the changes in the two files the new BareMetal OS kernel must be assembled:

```
nasm kernel64.asm -o kernel64.sys
```

After having assembled the kernel it must be copied to the Pure64 disk image:

```
mkdir /mnt/pure64
kpartx -av Pure64.img
mount /dev/mapper/loop0p1 /mnt/pure64
cp kernel64.sys /mnt/pure64
umount /mnt/pure64
kpartx -d Pure64.img
```

The image now contains the new BareMetal OS kernel without the bug causing the CPU to stay at 100% when "b_delay" is called.

3.4.3 How to compile Newlib for use with BareMetal OS

BareMetal OS may use the functions provided by the *Newlib* C library when compiling applications. It requires additional packages in order to successfully compile and running the following command will fetch and install the required dependencies:

```
apt−get install autoconf libtool sed gawk bison flex m4 texinfo texi2html unzip make
```

After these packages have been installed a directory must be created to hold the Newlib source files which must be downloaded from the Redhat website. Finally a fresh BaremetalOS zipball must be downloaded from Github and extracted on your harddrive.

```
mkdir newlib
cd newlib
```

```
wget ftp://sources.redhat.com/pub/newlib/newlib-1.20.0.tar.gz
tar xf newlib-1.20.0.tar.gz
wget https://github.com/ReturnInfinity/BareMetal-OS/zipball/master
unzip master
mkdir build
```

After having downloaded *Newlib* and *BareMetalOS* some files must be edited to specify the location of the BareMetalOS source files.

Modify the following files:

```
newlib-1.20.0/config.sub
@ Line 1334
              | -sym* | -kopensolaris* \
              | -amigaos* | -amigados* | -msdos* | -newsos* | -unicos* | -aof* \
              | -aos* | -aros* \
+ | -baremetal* \
              | -nindy* | -vxsim* | -vxworks* | -ebmon* | -hms* | -mvs* \
              | -clix* | -riscos* | -uniplus* | -iris* | -rtu* | -xenix* \
              | -hiux* | -386bsd* | -knetbsd* | -mirbsd* | -netbsd* \

newlib-1.20.0/newlib/configure.host
@ Line 506
      z8k-*-coff)
      sys_dir=z8ksim
      ;;
+ x86_64-*-baremetal*)
+ sys_dir=baremetal
+ ;;
   esac

newlib-1.20.0/newlib/libc/sys/configure.in
@ Line 46
      tic80) AC_CONFIG_SUBDIRS(tic80) ;;
      w65) AC_CONFIG_SUBDIRS(w65) ;;
      z8ksim) AC_CONFIG_SUBDIRS(z8ksim) ;;
+ baremetal) AC_CONFIG_SUBDIRS(baremetal) ;;
      esac;
   fi
```

In newlib-1.20.0/newlib/libc/sys create a directory called ”baremetal” and copy the contents of the ”newlib/baremetal” directory from the BareMetal OS code into the ”newlib/libc/sys/baremetal” directory.

```
mkdir newlib-1.20.0/newlib/libc/sys/baremetal
cd newlib-1.20.0/newlib/libc/sys
autoconf
cd baremetal
autoreconf
cd ../../../../..
cd build/
../newlib-1.20.0/configure --target=x86_64-pc-baremetal --disable-multilib
```

When inside the ”build” directory edit the Makefile and remove all instances of ”x86_64-pc-baremetal-” in the FOR_TARGET section but keep the rightmost part of the line containing the name of the tools. Changing these parameters will tell the compiler to use the default applications instead of look-

ing for specific cross-compiler tools for the x86_64-pc-baremetal tools which does not exist.

```
AR_FOR_TARGET=x86_64-pc-baremetal-ar
AS_FOR_TARGET=x86_64-pc-baremetal-as
CC_FOR_TARGET=$(STAGE_CC_WRAPPER) x86_64-pc-baremetal-cc
etc...
```

Change to:

```
AR_FOR_TARGET=ar
AS_FOR_TARGET=as
CC_FOR_TARGET=$(STAGE_CC_WRAPPER) cc
etc...
```

Also remember to add *-fno-stack-protector* to the ″CFLAGS_FOR_TARGET″ and ″CXXFLAGS_FOR_TARGET″ variables and compile Newlib using the following command:

```
make
```

When the build process has completed there should be two new folders inside the ″build″ directory called ″etc″ and ″x86_64-pc-baremetal″. The compiled Newlib C library which is now ready for linking with BareMetal OS programs can be found in ″build/x86_64-pc-baremetal/newlib/libc.a″. The size of libc.a is 5 MiB but can be shrunk to about 1.2MiB by running the following command:

```
strip --strip-debug libc.a
```

3.4.4 Creating a BareMetal ″Hello World″ Application

Compiling BareMetal applications using the Newblib C library gives the applications access to the standard set of C library calls like ″printf()″, ″scanf()″ etc. and the *gcc -I* (capital i) argument can be used to point to the ″newlib-1.20.0/newlib/libc/include″ directory which is the directory containing the Newlib headers.

The unzipped zipball from Github containing the BareMetal OS code contain a directory called ″programs″, which will be the working directory for developing your applications, and a directory called ″newlib″ which contain the linker script ″app.ld″. The linker script, ″libc.a″ and the Pure64 image must be copied to the ″programs″ directory in order to use it during the compilation of your applications.

When inside the ″programs″ directory, create a bash script ″build.sh″ to compile the applications:

```
1 #!/bin/bash
2 # Name: build.sh
3
4 # Compiles the BareMetal application using Newlib and BareMetal headers
```

```
5  gcc −I /newlib/newlib−1.20.0/newlib/libc/include −c newlib_hello.c −o newlib_hello.o
6  gcc −L . −l libBareMetal.h −c libBareMetal.c −o libBareMetal.o
7  ld −T app.ld −o newlib_hello.app newlib_hello.o libc.a libBareMetal.o
8
9  # Compiles the BareMetal application using only libBareMetal.h headers
10 gcc −−verbose −m64 −nostdblib −nostartfiles −nodefaultlibs −mno−red−zone −o
       bare_hello.o bare_hello.c
11 libBareMetal.c −DBAREMETAL −Ttext=0x200000
12 objcopy −O binary bare_hello.o bare_hello.app
```

The ”programs” directory should contain the following files:

- bare_hello.c
- newlib_hello.c
- app.ld
- libc.a
- libBareMetal.h
- libBareMetal.c
- build.sh
- BareMetal.img

The above script (build.sh) compiles two different *Hello world* applications and their source code is provided as appendix E and F. For the first program the compiler is told to point to the directory containing the Newlib C headers by using the -I parameter while the *ld* command links to the linker script ”app.ld” which divides the code into different memory regions and finally includes the C library file ”libc.a”.The end result is a BareMetal Newlib application with the name ”newlib_hello.app” which must be added to the BareMetal file system.

The procedure for the native BareMetal application is different and use the tool objcopy with the option ”-O binary” which tells objcopy to write the output file in the binary object format. The final result is an application with the name ”bare_hello.app”.

3.4.5 Moving Applications To the Image

Having built the BareMetal OS applications they also need to be transferred to the BareMetal file system which is achieved by mounting the Qemu image using the tool ”kpartx” and mounting the new device on the system using the ”mount” command.

Make sure the Pure64 image is placed inside the same directory as your BareMetal applications directory and create the following script which will copy the applications to the image and start the VM. Place this script in the same directory as the Pure64 image.

```
#!/bin/sh
kpartx −av BareMetal.img
mkdir /mnt/baremetal
mount /dev/mapper/loop0p1 /mnt/baremetal
cp newlib_hello.app /mnt/baremetal
cp bare_hello.app /mnt/baremetal
ls −la /mnt/baremetal
umount /mnt/baremetal
kpartx −d BareMetal.img
qemu−system−x86_64 −m 16 −hda BareMetal.img −net none −M pc
```

When BareMetal boot has completed, list the contents of the directory by using the ”dir” command and execute the application by typing in the name of the application on the CLI: ”NEWLIB_HELLO”. This should output ”I’m Newlib” and wait 5 seconds before printing ”Goodbye”, and the same behavior will apply for the ”BARE_HELLO” application. Notice that Newlib does not include the *sleep* function, so the headers of BareMetal will be used to offer this functionality.

To start an application automatically after the boot process has finished follow the same procedure but rename the application to ”startup.app”.

3.5 Getting Started with Tiny Core Linux

This section explains how to install Tiny Core Linux (TCL) and how to create a working Qemu-image and how to create a ”Hello World” application. TCL can be used on the same environment as L4 Pistachio as it is a 32-bit kernel, but the process of creating a Qemu image and transferring files to the file system is different and will be explained in this section of the report.

TCL use a pre-compiled Linux kernel and will be installed onto a Qemu-image by mounting the image file as a hard disk device where TCL will be installed onto. A script was developed in the programming language *Perl* to help with this procedure, see Appendix H. This script performs the mounting and copying of the applications over to the image. Table 3.5 lists the required software used for creating a development environment for TCL for this project.

Software	Version
Ubuntu Server 32bit	11.10
qemu-kvm	0.14.1
build-essential	11.5ubuntu1
CorePlus-current.iso	4.4
kpartx	0.4.9-2

Table 3.5: Tiny Core Linux build environment specifications

3.5.1 Installing Tiny Core Linux

As Tiny Core Linux comes pre-compiled the system needs to be installed onto a qemu disk image that can be mounted on your harddisk and will hold Tiny Core Linux and your applications.

The first step is to create a Qemu image by using the tool *qemu-img* to create a 20MB image file with the name ”tinycore_default.img” by running the following command:

```
qemu−img create tinycore_default.img 20M
```

Download CorePlus-current.iso which contains the installation program for Tiny Core Linux and issue the command below to start the installation procedure. [1] The command will mount the *iso* file as a cd-rom device and mount the *img* file as a harddisk which will be detected as device ”sda” by the installation program. This disk will hold the core and the applications when the installation has completed and is Qemu compliant.

```
qemu −hda tinycore_default.img −m 256 −cdrom CorePlus−current.iso
```

When prompted, choose to boot Core Plus with the ”default FLWM topside” option which will start the graphical desktop environment and click on the ”TC_Install” icon to start the installation process.

Step 1 Select the qemu image disk by the checking the ”Frugal” and ”Whole Disk” options which will make the image appear as ”sda” in the ”Select disk for core” window. Select it and check ”Install boot loader”.

Step 2 Select ”ext4” in the ”Formatting options” screen.

Step 3 The next step configures the boot options found in the ”Boot Options Reference List”. Enter the following:

```
vga=769 nodhcp noswap norestore text waitusb=0 base superuser
```

This configuration will speed up the boot process by disabling certain services during the boot.

Step 4 Choose to install ”Core Only (Text Based Interface)” and check the ”Non-US keyboard layout support”.

Step 5 Initialize the installation by clicking ”Proceed” and wait for the installation to finish.

[1] "http://distro.ibiblio.org/tinycorelinux/4.x/x86/release/CorePlus-current.iso"

3.5.2 TCL Hello World

Compiling and building an application which is able to execute on Tiny Core Linux requires the Linux kernel versions to be fairly close on both machines and that the GNU Standard C Library is at the same version level. After having transferred the application to the TCL machine the permissions of the binary must be set to executable in order to make it run.

The ″Hello World″ application for Tiny Core Linux can be found in appendix G and the program must be compiled using the following command:

```
gcc −o hello hello.cc
```

This compiles into a binary file called ″hello″ and must be transferred to the TCL file system.

3.5.3 Mounting the Filesystem

Transferring an application to the TCL file system and making it execute automatically at boot is a lengthy procedure. It involves mounting the qemu image, copying the TCL core file ″core.gz″ to the Ubuntu machine and mounting it as a file system. The application is then copied onto the file system and the TCL boot file ″bootlocal.sh″ must be edited to include the command to automatically execute the application at boot. Finally the filesystem must be repacked into a new TCL core file before copying it back onto the image file replacing the existing core and finally unmounting the image using the ″umount″ and ″kpartx -d″ commands.

To simplify and speed up this procedure a tool was made using the programming language *Perl* and will require the user to enter the name of the application to be transferred as well as the name of the qemu image. The script must be called with the command:

```
./tcl_add_file.pl −p hello −i tinycore_default.img
```

The ″-p″ and ″-i″ flags are mandatory and the script will exit if the script is initialized without them.

To start an application automatically after boot the script edits the file ″/opt/bootlocal.sh″ adding the command to start the start the executable file.

```
#!/bin/sh
# /opt/bootlocal.sh
/etc/init.d/hello
```

3.6 Building application software for simulation purposes

To make L4, Tiny Core Linux and BareMetal OS simulate real system activity a suggestion is to create applications for each system programmed to replicate

real-world CPU usage patterns. Since all systems depend to some degree on different headers and libraries it is essential keeping the code and libraries as simple and similar as possible in order to collect comparable metrics.

The suggested approach is to build custom application software designed to generate varying bursts of CPU activity for the purpose of simulating real-world usage patterns. These applications are suggested to calculate the Fibonacci sequence in order to generate CPU activity allowing to compare the performance between the different types of VMs. The software is suggested to calculate a given Fibonacci sequence number, n, in a combination with different sleep values to generate periods of CPU- and background activity.

3.6.1 Real-World Usage Patterns

Pattern A on figure 3.1 illustrates low background activity which brief periods of CPU activity which could be due to short bursts of user activity. The ratio between background activity and usage activity in this profile should be low in order to keep the CPU load minimal. This is an example of a pattern seen by web sites requiring additional compute cycles periodically as well as lower throughput applications.

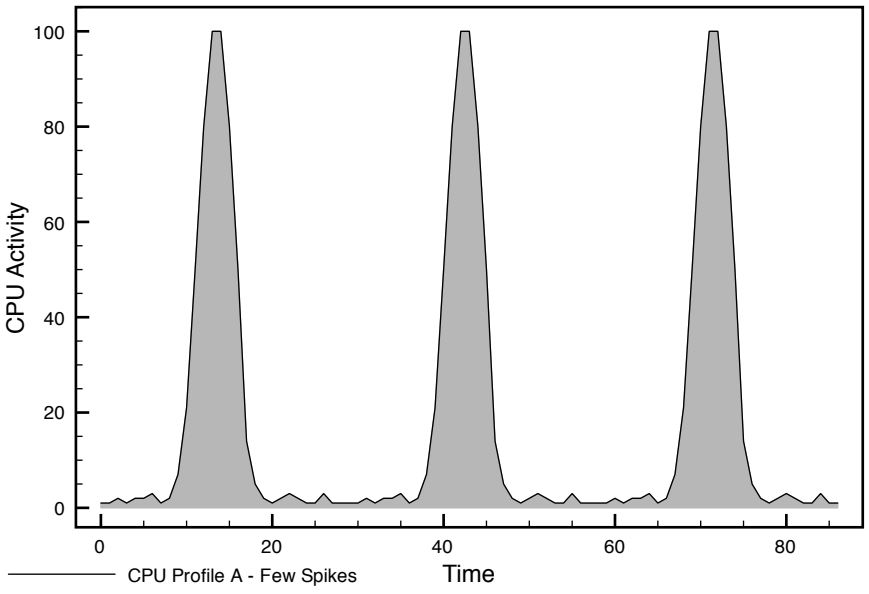

Figure 3.1: Pattern A - Short and intense periods of CPU activity

Pattern B, as illustrated on figure 3.2 intends to simulate the behavior of a typical data-crunching application requiring continuous CPU resources but has longer periods of no background activity. This might be an example of an compute-intensive application collecting information from a database periodically such as for generating business reports and to generate live statistics.

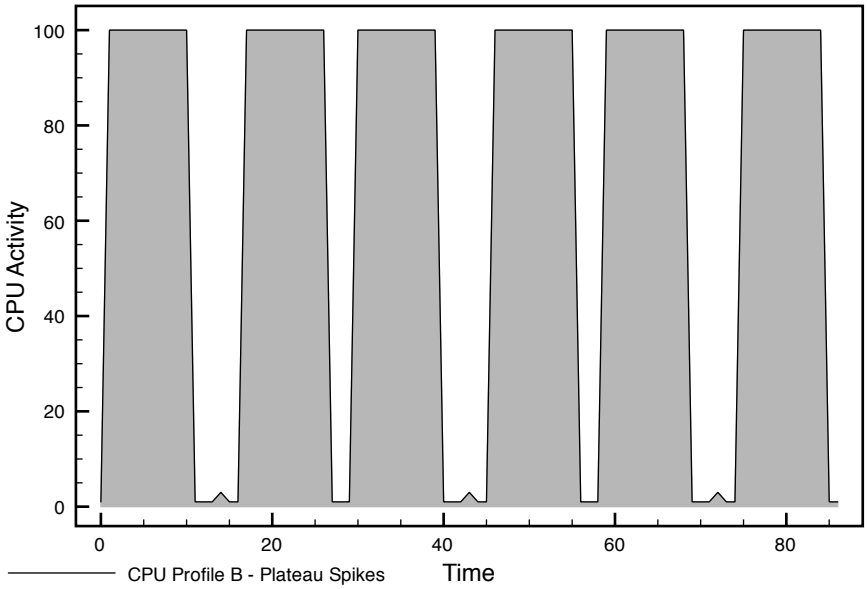

Figure 3.2: Pattern B - Longer periods of higher CPU activity

Pattern C illustrated on figure 3.3 simulate the behavior of a system with frequent application activity. There is a higher frequency of CPU activity when compared with patterns A and B, and the ratio between background- and CPU-activity for this profile is higher in favor of CPU utilization. This is typical behavior of a busy web server or an application collecting data from a database regularly.

3.6.2 Designing a Fair Test for all Kernels

Creating fair tests for L4, BareMetal OS and Tiny Core Linux based on different CPU patterns suggests developing applications performing the same tasks. The approach is to use the same source code for all kernels and using as few additional libraries as possible to keep each OS from implementing features unique for that specific system. Any additional libraries may cause additional function calls which do not occur in the other systems which is why the code must be as similar as possible to achieve comparable results.

Using the same code on all systems would suggest creating identical patterns is an easy task. However, as Tiny Core Linux and BareMetal OS are defined as operating systems while L4 Pistachio is a pure μ-kernel with a different architectural design this suggests slight changes have to be made in the source code for each kernel.

The first task would be to create applications to achieve the patterns illustrated in figure 3.1, 3.2 and 3.3. The approach is to experiment with different sleep- and fibonacci sequence values until these patterns have been achieved.

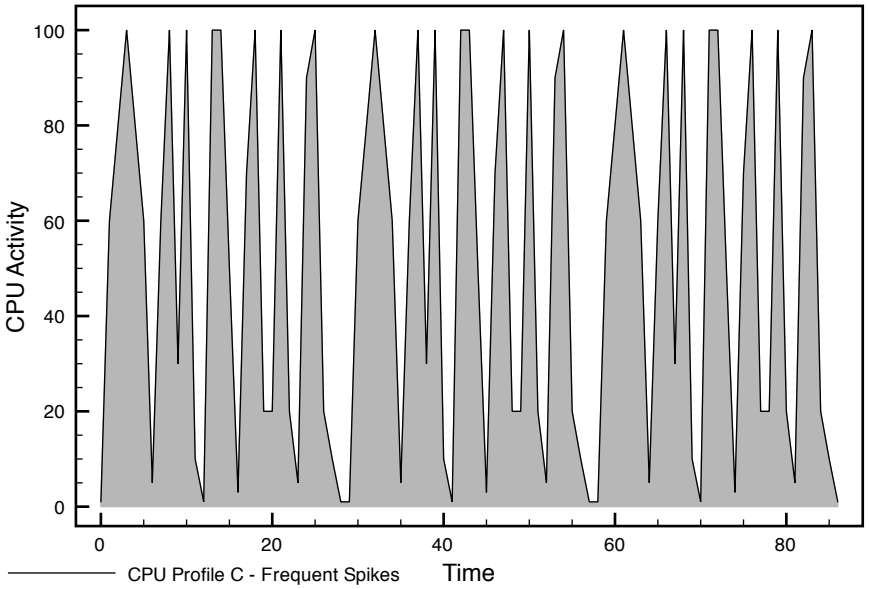

Figure 3.3: Pattern C - Short and frequent bursts of CPU activity

Fibonacci Sequence

The Fibonacci sequence in mathematics are the numbers in the following integer sequence:
$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...$ Based on looking at the sequence above it is clear that by definition, the first two numbers are 0 and 1, while the next number is the sum of the previous two. This is formally expressed by using the formula below:

$$F_n = F_{n-1} + F_{n-2}$$

The recursive implementation in the programming language C below is an example of a poor approach to calculate the sequence as it makes two function calls to itself which in turn leads to an exponential time complexity during calculation of n.

```
1  int fib(int n)
   {
3      if (n==0 || n==1)
           return 1;
5      else
           return fib(n-1)+fib(n-2);
7  }
```

Choosing a recursive implementation creates many function calls and may lead to stack overflow problems caused by running out of memory if the value

to be calculated is too large.

The following implementation is the iterative version of calculating the Fibonacci sequence:

```
1  int Fib_I (int i) {
     int Fib, Fib_1, Fib_2,j;
3
     if (i == 0) {
5        Fib = 0;
     } else if (i < 3) {
7     Fib = 1;
     } else {
9        Fib_1 = 1;
         Fib_2 = 1;
11       j = 3;
         while (j <= i) {
13           Fib = Fib_1 + Fib_2;
             Fib_2 = Fib_1;
15           Fib_1 = Fib;
             ++j;
17       }
     }
19   return Fib;
}
```

Looking at the implementations above it is to be noticed that the iterative function is more complex, and also faster as it does not call itself, but instead remembers the current result and moves on.

Iteration and Recursion

Both iterative and recursive implementations were considered for this project to generate the different usage patterns.

When using recursion the variables loaded by the 1st and n-th function call cannot affect each other directly. One should be careful using this approach as a substantial number of function calls require lots of memory eventually leading to stack overflow errors. In the case of the recursive Fibonacci implementation a stop condition such as ”if(n==0 —— n==1) return 1;” must be used to stop the calculation process from iterating forever.

Showing the limits of using recursion is best express by the following example; If the fib(n) function is called with $n = 20$ it returns the sum of calling the function fib(19) and fib(18), these two function calls will call a set of new fib functions with fib(18) fib(17), and fib(17) fib(16) and continues to generate a tree of function calls. Each time a function is called a small amount of memory is set aside, which in this case will happen 13,529 times. Even a small number such as 20 demands 13,529 function calls in 10945 steps which shows why recursion is not the optimal approach for calculating the Fibonacci sequence. Calculating a large sequence number increases the depth in function calls may eventually lead to stack overflow errors when running out of available mem-

ory.

Iteration works much faster than recursion in the case of the Fibonacci sequence as it calculates a sum, remembers it and moves on to the next calculation. Iteration removes the need of recalculations and in comparison require 19 steps instead of the 10,945 required by recursion.

Deciding on which implementation to choose depends on the problem at hand as some problems are best solved using recursion while others are best solved using iteration. In the case of the Fibonacci sequence the use of iteration is the better approach as using recursion to calculate a high sequence number would demand a substantial number of function calls, hence increased time complexity and eventually leading to buffer overflow errors if the depth is too great.

Despite of iteration being the better approach for calculating the Fibonacci sequence, recursion was chosen for this project as the purpose was to generate CPU activity. As low sequence numbers were used for this project the risk of buffer overflow errors was regarded as improbable as the number of function calls would never achieve a substantial depth.

Chapter 4

Results

This is the results chapter in which all the different tools, applications and tests are presented.

4.1 Result 1: Tools

Different tools were developed in the programming language Perl to help with the process of booting multiple virtual machines, compiling applications, gathering system statistics, creating images and transferring files onto them. Some of the tools have been developed specifically for each type of VM while others are common for all. Especially tools related to compilation, mounting images and transferring files are examples of tools which had to be custom made due to the different layout of the images and their file systems.

4.1.1 Mass Deployment of Virtual Machines

A script was developed to automatically boot VM instances on the labs by using user input to decide the number of VMs to boot, the type of operating system or kernel and the number of VMs to boot. The final script is provided in appendix C.

The script uses the following CPAN library modules to enable user input, and to inspect contents of scalars or reference variables.

- Getopt::Std
- Data::Dumper

To pass arguments to the boot-script the Perl module *Getopt::Std* implements the *getopts()* function which accepts switches defined by the user in a predefined list. The user is warned if an unrecognized option was passed to it and returns true if an invalid option was not passed. The boot script accepts the input options *-i, -t, -n and -h* which are short for ***i****mage,* ***t****ype and* ***n****umber* of

VMs to boot and ***h**elp*.

When the script is called the input is parsed through the *getopts()* function which will filter the input up against the list of allowed options. If one of the options are missing, or an option other than the ones specified in the list have been passed to the script it will exit. To allow multiple instances of virtual machines running on the same host using the same image a set of *universally unique identifiers* (UUIDs) must be generated for each guest. The script will use the value passed using the *-n* parameter to generate an equal amount of UUIDs as there are guests and store these in a text file on the hard drive which will be opened and read by the script when booting the virtual machines.

After the UUIDs have been stored in the text file on the hard drive the file is opened for reading and a variable ”$vm_id” is then initialized to the integer number 1 and used as a part of the process name and id when the VMs are booted. This variable is finally incremented by 1 at the end of every loop in the next part of the script.

Booting the virtual machines begins with a *foreach* loop which reads through the UUIDs file, one line at a time, passing one ID into a variable named ”$id” which is later used in the command to boot the VMs using the command line option *-uuid $id*.

Two additional variables are initialized at the beginning of the loop named ”$vm_name” and ”$vm_process” where the former will set the name of the guest and the latter will set the top visible process name in Linux. Setting these variables makes it possible for the statistics tool script to count the number of virtual machines running on the system.

Before the actual boot process is initialized a statistics script4.1.2 is executed to collect a number of data samples during the tests. The script is discussed in section 4.1.2.

The foreach loop has three *if* tests which checks the input given by the user in the *-t* option and checks if the input is one of either ”l4”, ”bm” or ”tcl”. If a match to one of these strings occur a custom Qemu command is executed using the Linux *system()* call to execute a command by using /bin/sh -c **command**. The options specified in the command is mostly the same for both L4 Pistachio, BareMetal OS and TinyCore Linux with only a few exceptions such as RAM and disk type options. Notice that L4 requires the option *-fda* to be used instead of *-hda* as the image is made in floppy disk format and also that the memory option *-m* is set to 5MB for L4, 16MB for BareMetalOS and 50MB for Tiny Core Linux. Additionally the *-cpu **pentium*** is used with L4 to ensure that Qemu defaults to simulating an intel CPU as this is the CPU architecture chosen during the build process of the kernel. BareMetal OS and TinyCore Linux does not require the use of the *-cpu* option.

To disable additional devices to be emulated by Qemu, such as a network

interface card (NIC), multiple CPU cores and graphical display unit (GPU) the options *-net **none*** and *-nographic* will disable the NIC and GPU. Disabling additional devices keeps the memory usage of the virtual machine to a minimum as well as it helps keeping the kernel busy with as few services as possible. To ensure that Qemu does not automatically compute and assign more than one single CPU core for each VM the option *-smp 1[,**cores**=1][,**threads**=1][,**sockets**=1][,**maxcpus**=1]* is used to constrict simulation to 1 CPU. The last option, *-M pc*, tells Qemu to emulate a standard PC machine.

The script executes VMs with three different version of the command below:

```
        if($TYPE eq "<TYPE>"){
2           print "Executing <TYPE> instance..\n";
            system( "qemu-system-x86_64 -m [5,16 or 50] [-cpu pentium] [-fda or
                -hda] $IMAGE -uuid $id -name $vm_name,process=$vm_process
                -net none -M pc -smp 1,sockets=1,cores=1,threads=1,maxcpus=1
                -nographic &" );
4           print "Sleeping 10 seconds\n";
            sleep (10);
6           print "Done sleeping..\n\n";
        }
```

After each VM has booted the script waits for 10 seconds to enable guests to be executed at different times. This makes the VMs execute their applications at different times and is more similar to real system behavior from real users rather than if the VMs were executed simultaneously.

4.1.2 Data Collection

A tool was developed to collect system statistics for the duration of the tests. The script *perf.pl*, found in appendix I, was developed using the programming language *Perl* and implements a number of useful CPAN Perl modules which have been especially developed for the purpose of collecting system information and to allow a user to initialize the script with input options.

The following Perl modules are used by *perf.pl*:

- Sys::Statistics::Linux;
- Time::HiRes qw/gettimeofday/;
- Data::Dumper;
- Getopt::Std;

Sys::Statistics::Linux reads the virtual /proc filesystem (procfs) and should work with the major Linux distribution on kernel version 2.4 and/or 2.6 and later and requires the desired procfs features to be enabled in the kernel. The user may also enter input options when initializing the script where the module *Getopt::Std* handles the user input. The module *Data::Dumper* was used to

inspect contents of scalars or reference variables available by using the *Sys::Statistics::Linux* module, and produced a human readable print of the different reference variables available and is not responsible for collecting any system information.

The script is initialized with user input with the options, where "[]" specify optional parameters, *[-**h** elp]*, *-**d** elay*, *[-**D** ebug]*, *-[**o**] utput file*, *-**s** amples* and *[-**t**] ype of VM*.

Some of the options are mandatory, such as the output file and type of VM and must be passed to the script by the user, and if one of the options are not provided the script will exit. The script has been configured to set the number of samples to collect to 30 and a delay of 1 second as default values to simplify the initialization of the script. As 30 samples and a 1 second delay has a limited use in most scenarios it is possible to suppress the default values by providing the options *-d* and *-s* with custom values. The smallest delay time is recommended to be no less than 1 second to allow the module *Sys::Statistics::Linux* to read from *procfs*, and is also the minimum recommended value found in the documentation.

Once user input has been accepted the script creates an output file in the current directory with the first line in the file being a header row containing 23 names corresponding to the type of system information to be gathered by the script.

After having created the output file a *for* loop will loop a number of times which correspond to the value provided by the user with the *-s* option when initializing the script. The first line inside the loop executes a subroutine, *getStats($TYPE, $DELAY)*, which accepts the values provided with the *-t* and *-d* options.

Collecting system information from *Sys::Statistics::Linux* is done inside the subroutine *getStats()* and is achieved by requesting a number of statistics by using a *Sys::Statistics::Linux* compilation object. The method *get()* will prepare and return the requested statistics and update the initial statistics. The subroutine accepts "$TYPE" and "$DELAY" as input which the user must provide with the *-t* and *-d* options. The former variable will be used to search the system for process names matching the given VM type, while the latter controls the frequency of the statistics collection by controlling how long the subroutine sleeps between each sample. As some tests will utilize 100% of the CPU the script might be affected so that the script is not able to collect statistics every second. To ensure that each data set collects comparable metrics the module "Time::HiRes" is used to convert the current unix timestamp into milliseconds which is printed in the log file and stored along with each data sample. To decide the duration of a test the number of milliseconds may be used to draw an exact timeline making the script more reliable.

```
1   my $lxs = Sys::Statistics::Linux->new(
        cpustats => 1,
3       pgswstats => 1,
```

```
        memstats => 1,
        processes => 1,
        loadavg => 1
    );

    sleep($delay);

    my $stat = $lxs->get;
```

The script requests CPU, Paging, Memory, processes and CPU Load average using the *Sys::Statistics::Linux* compilation object as seen in the code above, sleeps for at least 1 second between each data sample and executes the get() method which extracts the data from the system. All the statistics are collected and stored inside the $stat variable:

```
my $cpu = $stat->cpustats->{cpu};
$cpu_usr = $cpu->{user};
$cpuavg_one = $stat->loadavg->{avg_1};
$page_fault = $stat->pgswstats->{pgfault};
(...)
```

Collecting the number of virtual machine processes running on the system is done by using the *$stat->search()* method which searches through all the process names running on the host machine and using a regular expression to match the process names. All the process names are stored in a hash and counting the number of keys in the hash is done to get the total number of processes.

Finally the values are written into the output file and the loop continues to collect another sample of data from the system until the stop condition, "i ¡ *-s* samples", is reached.

4.1.3 Automated Build Tool for L4 Pistachio

A tool was developed in Perl to automatically perform compilation of L4 and all its binaries as well as adding files into to the L4 image. The manual procedure is a lengthy as it involves many steps in order to complete the configuration and installation of the kernel and its applications which is why the tool was created. The tool performs all of the required steps automatically without the need of user interaction other than setting the initial directory pathsand linkbase values in the beginning of the script. The script may be found in appendix D and will be explained in detail.

The tool has the following operationalization, and performs a clean installation of L4 and its binaries:

1. Clean the system of the previous build
2. Configuring the new software package
3. Building the software

4. Installing the software
5. Copying compiled binaries and Grub 0.97 stage1, stage2 and menu.lst files to a temporary directory
6. Mount the L4 Qemu image as a loop device
7. Mount the loop device in a mount directory
8. Copy binaries and grub files from the temporary directory to the mount directory
9. Install Grub on the loop device
10. Unmount the looop device
11. Remove the mountdir
12. Lists the files inside the temporary directory

The first part of the script is a configuration section where variables are used to set the locations of the L4 directories which is required by the installation process. Setting the directory locations in variables is done to make it easy to install the software in other locations if necessary. These variables may be freely customized to reflect a different directory structure.

After the configuration variables have been configured by the user to reflect the directory structure of L4 step 1 performs the initial clean-up operation to remove previous executable files and all the object files by executing the ”make clean” command.

Step 2 executes a script named *”configure”* located in the ”<l4-sourcedir >/user/” directory which is a script packaged with L4 that checks details about the host machine on which the software is being installed onto. It also checks the system for dependencies to make sure the software works properly and the script will exit immediately if some dependencies are missing on the system. Installing the missing dependencies are required in order to continue the installation process.

The code below shows the structure of the script of how it uses the variables together with the ”configure” script to perform step 2:

```
1  my $srcdir = ”/project/L4/l4ka−pistachio”;
   my $installdir = ”$srcdir/x86−user−install”;
3  my $kerneldir = ”$srcdir/x86−kernel”;
   my $kickbase = ”0x148030”;
5  my $sigbase = ”0x20000”;
   my $rootbase = ”0xEA60”;
7  (...omitted lines...)
   system(”../user/configure −−without−comport −−with−kickstart−linkbase=$kickbase
        −−with−s0−linkbase=$sigbase −−with−roottask−linkbase=$rootbase
        −−prefix=$installdir −−with−kerneldir=$kerneldir”);
```

The script passes options to the configure script to configure the software with different settings which will be further explained. To disable console output from being redirected to a serial port the option *–without-comport* is used. This option will force the L4 virtual machine to show its output inside the virtual machine console instead of being redirected to a console on your own system.

The *linkbase* options are used to suppress the default linkbase values of sigma0, roottask and kickstart. These values are architecture-dependent and the settings used for an ”ia32” system must be changed if configuring the software for a 64-bit system, so a bit of experimentation might be necessary as the documentation on how to set the minimum allowed linkbase values are not available.

The default linkbase values are located in the file ”l4-sourcedir/user/configure.in” and are expressed in hexadecimal format:

```
ia32)
        default_kickstart_linkbase=00800000 // 8 388 608 bytes
        default_sigma0_linkbase=00020000 // 131 072 bytes
        default_roottask_linkbase=00400000 // 4 194 304 bytes
        ;;
```

As the default values require L4 to receive at least 12 713 984 bytes of memory from the host system in order to boot these values were changed in the ”makescript.pl” to pass smaller values reducing the amount of memory needed to a much lower 1 534 608 bytes, a reduction of 11 179 376 bytes(!). As Grub also demands a small amount of additional memory the L4 VM was able to boot with only 1.68MB of RAM, or 1 761 607.68 bytes. Using these settings the VM is smaller than any other virtual machine found on the Internet, and understanding how the linkbase values work will be essential in keeping the VM as small as possible.

```
Kickstart linkbase: 0x800000 (8 388 608 bytes) changed to 0x148030 (1 343 536 bytes)
Sigma0 linkbase: 0x20000 (131 072 bytes) changed to 0x20000 (131 072 bytes)
Roottask linkbase: 0x400000 changed to 0xEA60 (60 000 bytes)
```

The ”–prefix=*installdir*”*and*”$--with-kerneldir$=kerneldir” options provide the locations of the installation directory where the compiled binaries will be installed, and the directory containing the L4 kernel binary. When the ”configure” script has completed it has created a file named Makefile which is going to be used in the next step when calling ”make”.

Step 3 and 4 in the script calls ”make” and ”make install” to perform the building and installation of L4.

Running ”make” reads the Makefile in the current directory in which you run make. Make reads the directions found inside the Makefile and then proceeds with the installation. The recipe tells Linux the sequence to build various

components of L4 and includes checking dependencies which have to be in place on the system. If some dependencies are not met when running ”make” the installation will exit and the dependencies have to be installed before continuing. After step 3 has completed the program code and the executables have been compiled. If ”make” is successful the script calls ”make install” which will place the executables and other files into the final directories on the machine.

After having successfully compiled and installed L4 the script moves on to step 5 which perform the copying of the executables to a temporary directory be used to hold the files which will be copied to the L4 Qemu image.

Step 6 and step 7 in the script is responsible for mounting the image as a loop device and creating a mount directory where the loop device will be mounted. All files which are located in the temporary directory are then copied to the mount directory which is the same as placing the files onto the image, which is performed in step 8.

When all the files have been copied to the mount directory GRUB must be installed on the loop device to make it bootable, and a tool, ”grub.sh” is called by the script to install GRUB Legacy using GRUB batch mode. The script is located in appendix J and completes step 9.

Step 10 unmounts the mount directory and step 11 deletes the directory and all files located in it.

The final step is number 12 which list all the files located inside the temporary directory to visually present the files which have been copied by the script. This is an optional step and can be safely removed if needed as it has nothing to do with compiling L4.

4.1.4 BareMetal OS Tools

Compiling BareMetal OS applications and transferring them onto the VM images became a time consuming process which is why two different tools were created to automate the procedure. These tools are presented in this section and are provided as appendix K and L.

Compiling BareMetal Applications

A bash script was developed to perform the compilation of applications for BareMetal OS. The script is called ”build.sh” and is located in appendix K. The script does not accept any user input and is only responsible for compiling applications and a typical set of compilation instructions for each applications look like the following:

```
#!/bin/bash


# CPUA − Iterative Fibonacci application, CPU profile A
gcc −−verbose −c cpua.c −o cpua.o
gcc −−verbose −L . −l libBareMetal.h −c libBareMetal.c −o libBareMetal.o
ld −−verbose −T app.ld −o cpua.app cpua.o libBareMetal.o

(...)
rm *.o
```

When the script has completed the binaries are called "*.app" and are placed in the current directory.

Transferring Files To the BareMetal VM Image

After compiling the executable files they are transferred to the BareMetal OS Qemu VM-images which is achieved by another bash script. The tool "mount.sh", located in appendix L copies a default image and mounts it as a loop device using the tool "kpartx". The loop device is then mounted on a mount directory on the local file system before the executable file is copied to the mount directory and unmounts the mount directory.

The script has the following structure with changes made in the names of the images and applications for each image:

```
cp BareMetal_new.img BareMetal_sprint_iter_20.img
kpartx −av BareMetal_sprint_iter_20.img
mount /dev/mapper/loop0p1 /mnt/baremetal
cp /sdc/project/BM/programs/CPU/sprint_iter_20.app /mnt/baremetal/startup.app
echo "Listing contents SPRINT 20:"
ls −la /mnt/baremetal
umount /mnt/baremetal
kpartx −d BareMetal_sprint_iter_20.img
```

4.1.5 Tiny Core Linux Tools

This section describes the tool used for this project used to add applications to TCL Qemu VM-images. The script is provided as appendix H

Transferring Files to Tiny Core Linux

A tool was used to transfer executable files to the file system of Tiny Core Linux. The script was programmed in Perl and is located in appendix H.

The Perl module "Getopt::Std" is used by the script to handle user input and the script requires the user to execute the script using the options "**-p**" and "**-i**". The former will be the name of the executable file to transfer to the file system, while the latter is the name of the image where the executable file

will be transferred to. Both options are mandatory upon executing the script. Failing to provide one of the options causes the script to exit with an error.

The name of the executable file and the name of the image are provided as input from the user when the script is executed and the values are stored in variables inside the script which are used to mount the image and to copy the executable files onto it. The script then checks if the mount directory exists and deletes the directory if it is present. This is done to ensure that no old files are present to achieve a clean modification of the Tiny Core Linux file system.

When the mount directory has been deleted the script then creates an empty mount directory which will be used by the script to mount the image. A temporary directory containing a subdirectory ”extract” is then created by the script to hold the Tiny Core Linux file system and once created the script mounts the image as a loop device using the tool ”kpartx” inside the subdirectory. The command ”kpartx -av $IMAGE” is executed in backticks to capture the output generated. The output is stored inside an array and the script then searches through the output searching for the name of the loop device the image was mounted on. Having the name of the correct loop device enables the script to automatically mount the Tiny Core Linux image on the correct loop device.

The script then mounts the loop device in the empty mount directory and copies ”core.gz” containing the file system from the ”mountdir/tce/boot/” directory into the temporary directory. Next, the script enters the subdirectory ”extract” and unpacks the Tiny Core Linux file system by executing the command ”zcat ../core.gz — sudo cpio -i -H newc -d”.

The entire Tiny Core Linux filesystem with the familial Linux directory structure is now visible inside the ”extract” directory and the executable file is then copied into the ”/etc/init.d/” directory. To make the executable start automatically after the boot process has completed the script edits the file ”bootlocal.sh” located in the ”extract/opt” directory with the appropriate command to start the application:

```
1 chdir(”$tempdir/extract/opt/”);
2 open FILE, ”>$tcl_boot_file” or die $!;
3 print FILE ”#!/bin/sh\n”;
4 print FILE ”# Add startup commands in this file\n”;
5 print FILE ”/etc/init.d/$PROGRAM\n”;
6 close FILE;
```

Once the executable file and the startup command has been added the old ”core.gz” file is backed up in case something goes wrong and then re-packs the file system in a new ”core.gz” file. The command ”find — sudo cpio -o -H newc — gzip -2 ¿ ../core.gz” will create the file in the temporary directory before it copies it into the mount directory in ”mountdir/tce/boot” replacing the original.

Finalizing the operation the script unmounts the mount directory and issues the command ”kpartx -d $IMAGE” to unmount the loop device itself.

4.2 Result 2: Fair Test

The purpose of creating real-world usage patterns was to perform fair tests on L4 Pistachio, BareMetal OS and TCL. The results are used to analyze their behavior when scaling from a smaller to a larger population of VMs and also if they behave differently when used for different tasks. The latter is especially valuable when used as minimal VMs equipped with custom application stacks, such as a LAMP stack. This chapter describe the results from creating a fair test for each CPU profile.

As calculating the Fibonacci sequence could be achieved using both iterative and recursive implementations both were initially tested to see if the CPU patterns were easier to resemble using one of the approaches. Recursion proved to be the easiest implementation for this project as lower fibonacci numbers could be used. The iterative approach also required the source code for each kernel to be modified more than when using the recursive approach. Recursion involve more of the OS itself which is part of the decision to why this approach was chosen instead of the iterative approach. When using the iterative code, the compiler options used by L4 Pistachio during compilation of the applications optimized the code in such a way that the function call to the iterative subroutine were lost resulting in inconclusive results. To prevent the compiler from optimizing the L4 application code the keyword ”volatile” had to be used which would result in a different source code when compared with the other kernels. Even though iterative implementations is the preferred approach when calculating the Fibonacci sequence in general it was not chosen as the purpose of the tests were not to benchmark each kernel with respects to their speed.

The decision to abandon the iterative approach in favor of recursion enabled the possibility to implement nearly the same source code for all applications with the only difference being L4 Pistachio and BareMetal using their own native libraries while Tiny Core Linux used the standard GNU C library.

The details for each test is presented in table 4.1 and lists the name of each system, the type of pattern, the Fibonacci number to be calculated, how often the application sleeps and finally the number of virtual machines which For the duration of the tests a statistics tool4.1.2 collected a total of 200 data samples. Table 4.1 lists the fair tests performed on each system. The table shows the name of the kernel, the CPU pattern (A,B or C), the fibonacci sequence number to be calculated, the number of seconds to sleep between each calculation and the number of VMs used in the test.

Kernel	Pattern Type	Fibonacci of n	Sleep (seconds)	Num. VMs
TCL	A	40	20	1
TCL	B	43	40	1
TCL	C	40	5	1
BareMetal	A	40	20	1
BareMetal	B	43	40	1
BareMetal	C	40	5	1
L4 Pist.	A	41	20	1
L4 Pist.	B	44	40	1
L4 Pist.	C	41	5	1

Table 4.1: CPU Pattern test chart

The results show Tiny Core Linux and BareMetal OS to achieve similar CPU profiles as illustrated on figure 4.1 and 4.2. However, there were some abnormalities when looking at the L4 Pistachio results as seen on figure 4.3 as the bursts of CPU activity occur more frequently. To achieve a fair test the frequency of CPU activity bursts must be similar for all systems in order to assume identical CPU patterns have been created. The source code for the fair tests for Tiny Core Linux is attached as appendix R and BareMetal OS as appendix Q.

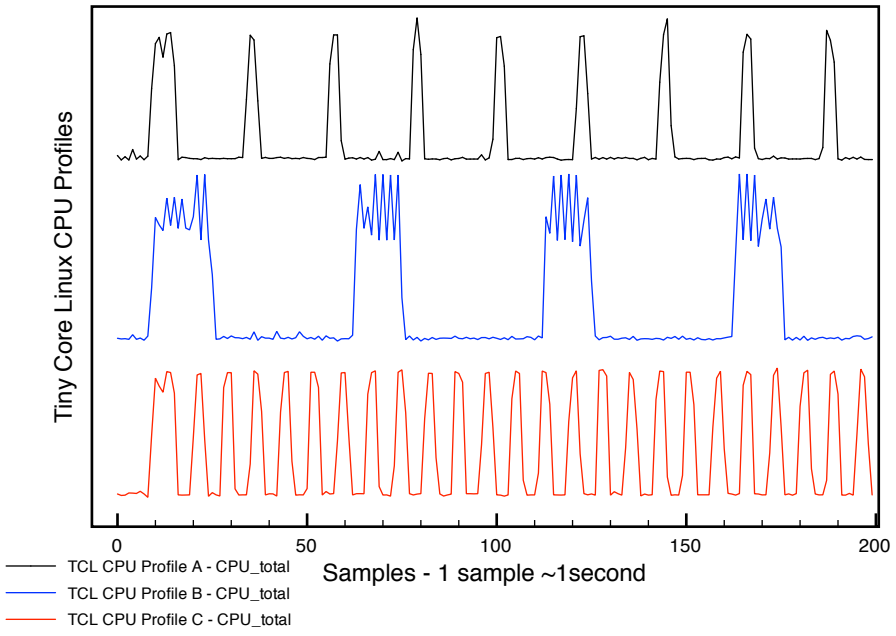

Figure 4.1: CPU pattern A, B and C on Tiny Core Linux

Version 1 of the source code for the L4 Pistachio CPU patterns calculated the fibonacci sequence number 40 for pattern A and C, 43 for pattern B. These values were changed to 41 and 44 before a second test was initiated to compare

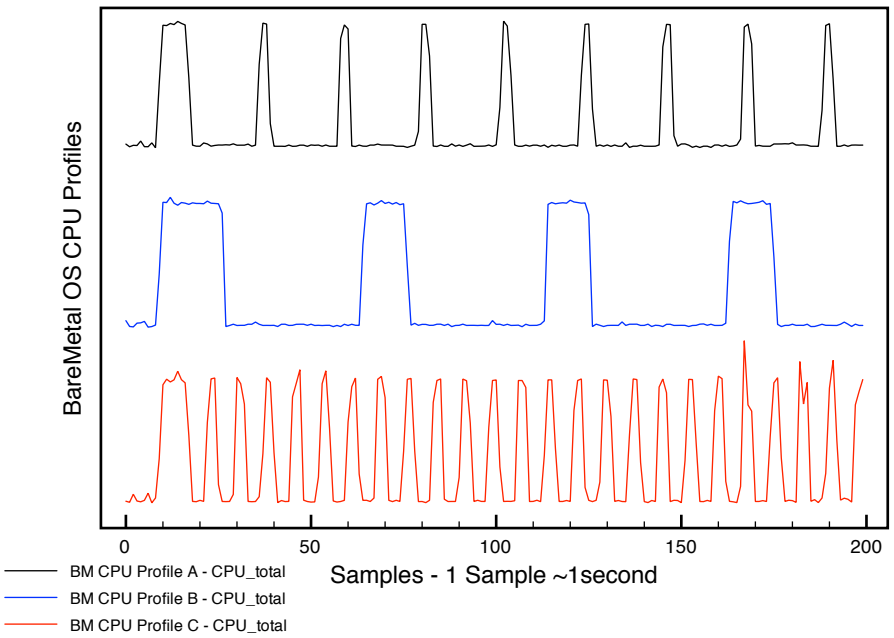

Figure 4.2: CPU pattern A, B and C on BareMetal OS

if the patterns would be more similar to the other two systems. Version 1 of the source code is attached in appendix O and the updated code is attached as appendix P.

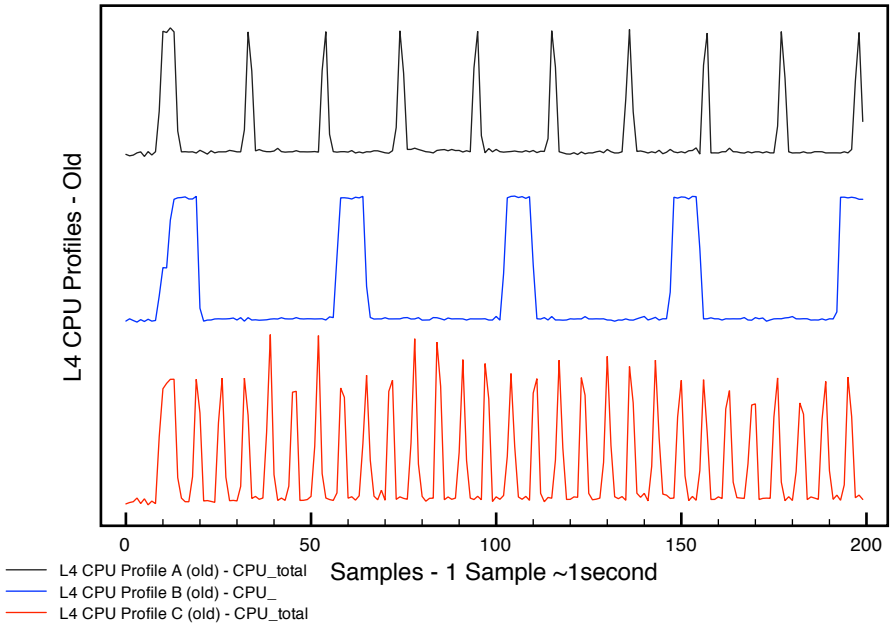

Figure 4.3: Test results from version 1 of CPU pattern A, B and C on L4 Pistachio

The initial results showed a fair test had not been achieved. However, version 2 of the L4 source code resulted in a CPU pattern matching TinyCore Linux and BareMetal OS. The results from the updated test on L4 are illustrated on figure 4.4.

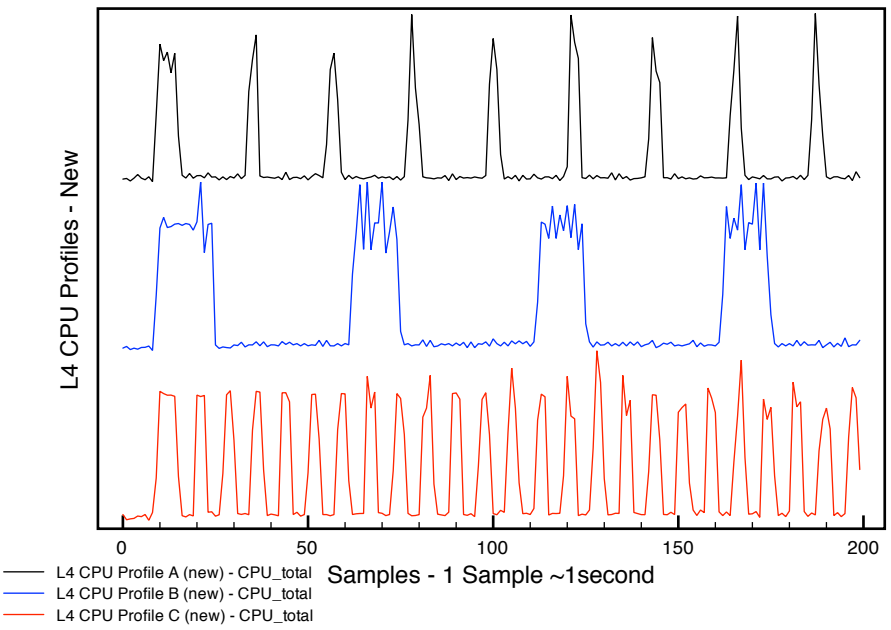

Figure 4.4: Test using version 2 of CPU pattern A, B and C on L4 Pistachio

Comparing the results show CPU pattern A to have an occurrence of 9 bursts of CPU activity for all three systems, 4 bursts for profile B and 25 bursts for pattern C which suggested a fair test had been achieved.

4.3 Result 3: Scalability of minimal VMs

This section show how the minimal VMs behaved in different population sizes on the HoH and HiC labs. The tests were performed by utilizing 100% CPU load on different sized models of L4 Pistachio, BareMetal OS and TCL. Fully utilizing the CPU required all CPU cores being used requiring a minimum of 4 VMs on the HoH3.1 and 8 VMs on the HiC3.2 lab to be deployed. As the HoH lab used a quad-core CPU is was decided to deploy population sizes of 4,8,16 and 32 virtual machines while the HiC lab had twice the number of (virtual)CPU cores allowing 8,16,32 and 64 VMs in order to utilize all CPU cores on both labs. All VMs were booted simultaneously and the applications had a 60 seconds sleep delay before they began the calculations in order to let the boot process complete.

The purpose of these tests were to create many context switches, hence generating additional CPU overhead. Measuring CPU overhead was achieved by

measuring the total time spent by L4 Pistachio, BareMetal OS and TinyCore Linux in milliseconds necessary to complete an equal amount of work. The results of these tests were compared between HoH and HiC to examine if the kernels performed differently when using KVM or not.

To achieve fair tests each population of VMs executed different versions of the same applications to perform the same calculations for a total of 1536 times. A loop was used to perform the calculations and a stop condition was used to limit the maximum number of calculations to end the loop. The five applications were configured to calculate the fibonacci sequence 384(4 VMs), 192(8 VMs), 96(16 VMs), 48(32 VMs) and 24 times (64VMs) adding up to a total of 1536 calculations for each of the population sizes. However, the L4 Pistachio applications were slightly changed to calculate a different fibonacci sequence than the other kernels as mentioned in *Chapter 2: Fair Test*. In short, the change had to be done as L4 spent less time on calculating the fibonacci sequence when compared to the other systems resulting in the CPU being utilized in a shorter amount of time resulting in an unfair test. Changing the number from 40 to 41 made L4 spend an equal amount of time when compared with Tiny Core Linux and BareMetal OS suggesting a fair test had been achieved.

The statistics tool4.1.2 collected system information for 1500 samples on the hardware lab, and between 3000 and 4500 data samples on the cloud lab as the tests took longer time in order to complete.

4.3.1 Explaining the data

The following sections use graphs to present the results from these tests. The lines in the graph has been skewed to the left eliminating boot times, and a grey vertical zero line indicates where the calculations began. The vertical lines to the right of the zero line indicates the calculations have completed. The CPU activity to the left of the zero line express boot times and is regarded as irrelevant data as the calculations have not yet been initiated. Each graph use the same color scheme where red is used for identifying Tiny Core Linux, black for BareMetal OS and blue for the L4 Pistachio Kernel. Dotted and solid lines are used to distinguish between the HoH and HiC labs where the former is dotted and the latter is presented by solid lines.

As the number of data samples vary, the lines in each graph also differ and is why some of the lines end abruptly in the middle of the graphs. The reason for the different lengths was that the first round of tests were performed on the HoH lab which finished these tests in less time than the HiC lab.

4.3.2 Hypervisor on Hardware vs. Hypervisor in Cloud

This section present the results from scaling up the number of virtual machines from the HoH to the HiC lab. The results are compared between each of the labs and presented in graphs to see if it is possible to recognize predictable behavior when the kernels are scaled up to twice the original population size.

The results from the tests performed on a population size of 4 VMs on the HoW lab and 8 VMs on the HiC lab are illustrated in graph 4.5 and shows many differences with regards to elapsed time and CPU utilization between the kernels on the two different labs.

On the hardware lab Tiny Core Linux complete the tests in less time than the other kernels and also use the least amount of CPU when idling. L4 Pistachio completes the test next after Tiny Core Linux, while BareMetal OS is the slowest of the three. However, L4 Pistachio requires significantly higher CPU load than both BareMetal OS and Tiny Core Linux during idling. The graph show the CPU utilization of L4 Pistachio to differ significantly when compared with the other kernels.

The results on the cloud lab show a significant change in the time required to complete tests when compared with the hardware lab. Especially Tiny Core Linux show a significantly higher performance hit on the cloud lab than the other kernels. However, L4 Pistachio improved its performance compared with the other kernels by completing the tests within the shortest amount of time, and also looks to achieve similar CPU utilization levels as the two other kernels when idling. By looking at the cloud results Tiny Core Linux show erratic CPU utilization levels before dropping to a slightly lower usage when compared with the others.

Figure 4.6 compares the results between the HoH lab having a population size of 8 VMs and the HiC lab with a population of 16 VMs. Looking at the results on the hardware lab it shows Tiny Core Linux to continue to complete the tests before the other kernels, which is coherent with previous results on graph 4.5. Also notice the increase in L4 Pistachio CPU levels when compared with the previous population size as illustrated by the blue dotted lines in the graphs. Comparing the CPU utilization of L4 Pistachio between these two populations shows the CPU utilization increase to almost be twice as high on the largest population.

The cloud results also show all the kernels to significantly increase the time needed to complete the tests when increasing the population size. However, Tiny Core Linux increase the most and BareMetal OS the least. Looking at the CPU levels for L4 Pistachio and Tiny Core Linux they suddenly show significantly larger levels after completing the tests when compared with the previous results, as seen on figure 4.5. BareMetal OS, however, show a slight increase but does not have the varying CPU levels as seen on the other kernels.

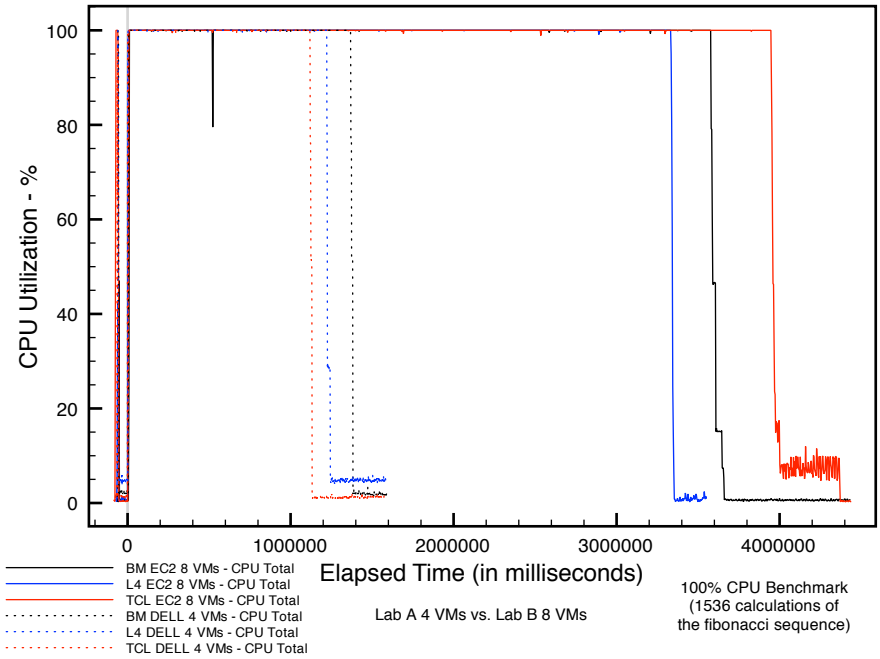

Figure 4.5: Time in milliseconds spent for a population of 4 and 8 virtual machines.

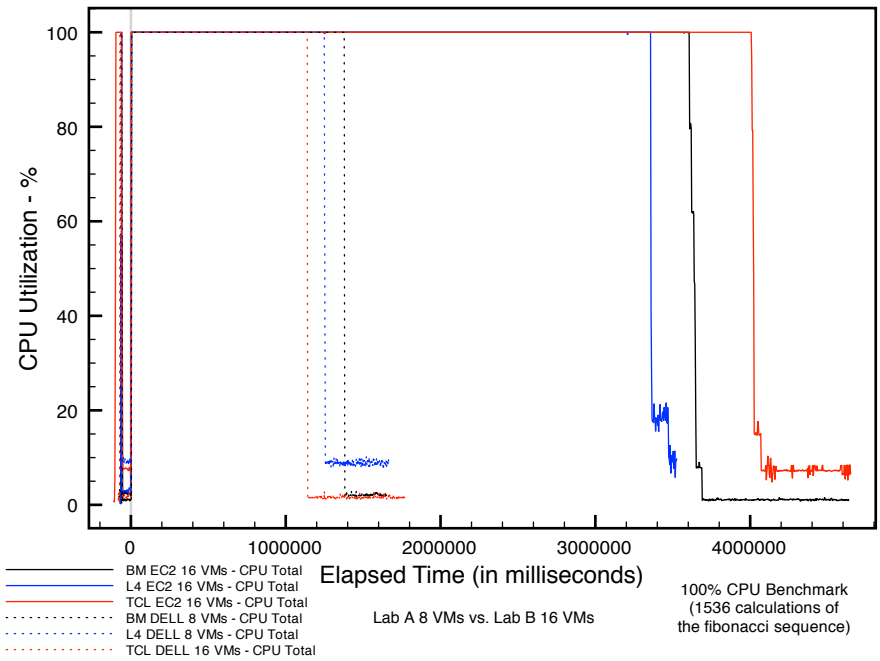

Figure 4.6: Time in milliseconds spent for a population of 8 and 16 virtual machines on the HoH and HiC lab respectively.

The results for a population size of 16 virtual machines on the HoH lab versus 32 VMs on the HiC lab are illustrated on figure 4.7. The graph shows a coherence with previous results where Tiny Core Linux continue to complete the calculations first on the hardware lab while last on the cloud lab.

Notice on the results for the hardware lab that L4 Pistachio begins to spend significantly more time on the tests and almost spend the same time as BareMetal OS. During idling the CPU utilization for BareMetal OS and TinyCore Linux is still at similar levels while L4 Pistachio continue to increase with each population size. Comparing the hardware lab results for L4 Pistachio with previous results on figure 4.5 and 4.6, L4 Pistachio continues to require a significantly longer time in order to complete the tests when increasing the population sizes. BareMetal OS and Tiny Core Linux, however, does not show the same significant changes as L4 Pistachio.

On the cloud lab results on figure 4.7 Tiny Core Linux continue to increase the time to complete the tests with a significantly larger increase when compared with the other kernels. Looking at the CPU utilization for all the kernels L4 Pistachio show a significant change while BareMetal OS show a slight rise in the CPU levels as seen on figure 4.5 and 4.6. Notice how the varying CPU levels for L4 Pistachio and Tiny Core Linux, as seen on 4.6, seems to have stabilized. These results may indicate that the changing behavior of the cloud affected the tests and shows the difficulty of performing repeatable tests on a cloud.

The maximum population size deployed on the HoH lab was 32 VMs and 64 VMs on the HiC lab. The results are illustrated in figure 4.8.

Looking at the dotted lines on the graph representing the hardware lab, L4 Pistachio now spend slightlymore time than BareMetal OS suggesting the latter to perform better than the former when increasing the population of VMs. BareMetal OS does not show the same increase as L4 Pistachio and Tiny Core Linux continues to spend the least amount of time as it did on the previous tests. The results also show L4 Pistachio to continue increasing its CPU utilization levels when the tests were done. L4 Pistachio might therefore be unsuitable for modeling large populations of VMs as it would require substantial amount of CPU resources by just idling. Tiny Core Linux still spends the least amount of time on completing the tests and also show the least amount of CPU utilization, suggesting it to handle scalability well. BareMetal OS show a slight increase in CPU levels but does not increase significantly when compared with Tiny Core Linux and has a low overall CPU utilization level suggesting it also to be suitable for large scale populations on a hardware lab environment.

The cloud results on figure 4.8 show the same trend as seen on the previous populations as illustrated on figures 4.5, 4.6 and 4.7; L4 continues to increase the CPU utilization levels but still completes the tests before the other kernels. BareMetal OS require slightly more CPU resources after completing

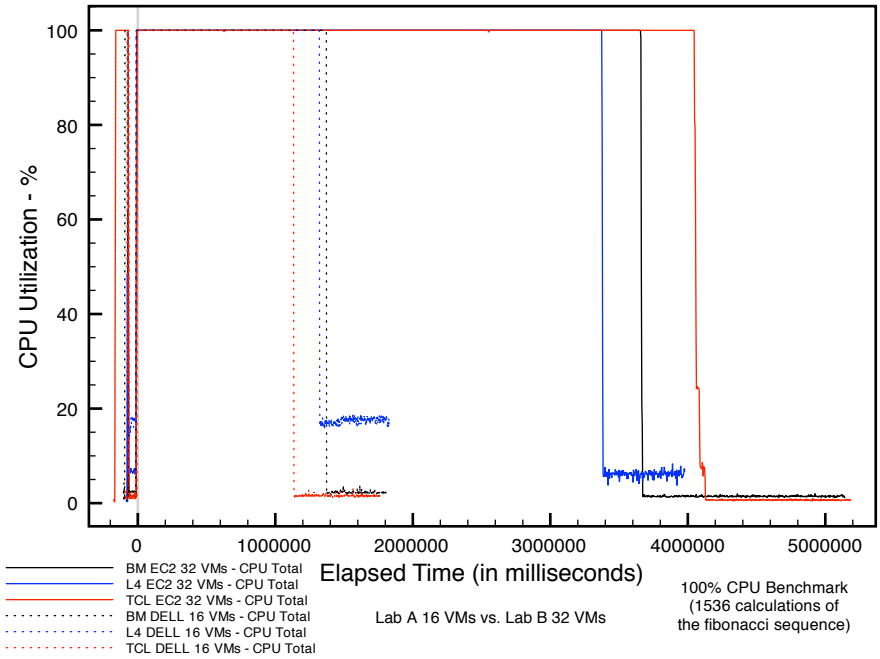

Figure 4.7: Time in milliseconds spent for a population of 16 and 32 virtual machines on the HoH and HiC lab respectively.

the tests. However, the CPU increase is not significant and suggests the kernel to be suitable for large populations on a cloud environment. Tiny Core Linux show the least CPU utilization after completing the tests but continues to be the slowest kernel, suggesting a full-sized VM does not offer the same performance as minimal VMs. The results show L4 Pistachio to spend less time than BareMetal OS to complete the tests suggesting the use of minimal VMs on a cloud environment has performance benefits when compared with full-sized VMs.

The source code for the BareMetal tests are found in appendix T, L4 Pistachio as appendix S and Tiny Core Linux as appendix U.

Summary of the results

Summarizing the results for the hardware lab, Tiny Core Linux and BareMetal OS VMs complete each test with similar results for every population increase, suggesting minimal BareMetal OS VMs to be able to compete with full-sized VMs and would be suitable for small-scale modeling in an hypervisor on hardware environment.

Tiny Core Linux VMs show to complete all the tests faster than BareMetal OS VMs, however, it might be possible that one of the VMs have a slightly larger increase in the time needed to complete the tests. If BareMetal OS in-

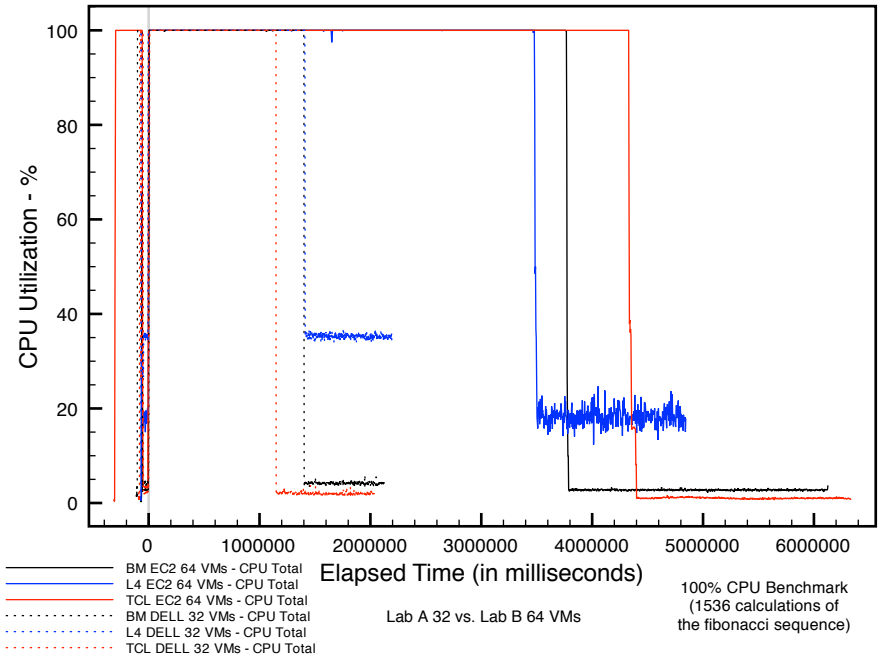

Figure 4.8: Time in milliseconds spent for a population of 32 and 64 virtual machines on the HoH and HiC lab respectively.

crease less than Tiny Core Linux over time this will suggest minimal VMs are able to compete with full-sized VMs as the latter would require more computing resources if using a large population of VMs. The results also show L4 Pistachio requires a vast amount of CPU resources on a hypervisor on hardware environment when performing basic idling. This suggests L4 Pistachio is an unlikely candidate for large-scale populations of VMs. BareMetal OS finishes last in the tests performed on the smallest population sizes, but for populations of more than 32 VMs L4 Pistachio spends slightly more time to complete the tests. All kernels show an increase in amount of time needed to complete the tests at each population increase which show the impact of context switching when deploying larger populations of VMs.

The current results asks for another test to be done for a population of 64 VMs on the hardware lab to confirm if L4 Pistachio will continue to slow down when compared with the other kernels. The new test will also show if L4 Pistachio will pass BareMetal OS by a wide margin on the next population increase and should provide a an answer to if L4 Pistachio is unsuitable to be used for large-scale population modeling purposes.

Results from the hypervisor in cloud lab show that the L4 Pistachio VMs completes all the tests faster than Tiny Core Linux and BareMetal OS VMs. The graphs also show the CPU utilization level for L4 Pistachio to keep increasing with every population increase. However, it does not increase as significantly as seen on the hardware lab.. Tiny Core Linux VMs look to have the largest

increase in time needed to complete the tests while BareMetal OS looks to increase the least with L4 Pistachio showing a slightly larger increase than the latter.

4.4 Result 4: Usage Patterns

This section examines the CPU and memory usage when testing CPU patterns A, B and C for a population of 10 virtual machines on the HoH and HiC labs. Each VM was booted with a 5 second boot delay to start each calculation at different times to simulate real-world behavior. The statistics script collected a total of 1500 data samples on each lab and the results were compared between the different types of kernels analyzing CPU and memory usage for each VM.

It was important to compare differences in behavior between the minimal VMs when deployed on different environments to see if it would be possible to show how a small-scale model on a hardware lab would behave when moved onto the cloud. The results would also be able to tell if minimal VMs would be able to compete with full-sized VMs by comparing their behavior and system resource usage. Especially memory footprints and CPU usage were examined to see how the minimal VMs compared with full-sized VMs.

4.4.1 CPU Usage

The graphs illustrating CPU *total*, *system* and *user* levels are included in appendix M and M. The grey lines show the CPU activity occurring at user level while the green line displays the average of total CPU usage which includes both system and user level activity. To reduce the large number of data points to a more practical number they were reduced from 1500 to 15 points where each point contained the average of 100 data samples.

The results from the HoH lab show a high level of CPU activity occurring at system level during boot of the VMs which was expected as KVM is enabled on this lab. By comparing the graphs for L4 Pistachio, included in appendix M.7, M.8 and M.9, these graphs show system level activity to decrease as the CPU workload increase. Comparing the system level activity for TinyCore Linux in appendix M.4, M.5, M.6 and BareMetal OS in appendix M.1, M.2, M.3 show system level activity to remain close to zero, suggesting these kernel to demand less communication with the kernel. Looking at the user level activity for all the kernels, L4 Pistachio and TCL show a larger variation in CPU activity especially at the end of the tests when using pattern A and B, when compared with BareMetal OS.

The CPU averages for the HoH lab are illustrated in graph 4.9 and show the sum of both system and user level activity for all three kernels for all the

CPU patterns. Notice the similarity in CPU usage between TinyCore Linux and BareMetal OS on pattern A and B and how L4 Pistachio deviate from the two other kernels by utilizing a significantly larger amount of CPU. This difference seems to level out when looking at the lines representing pattern C.

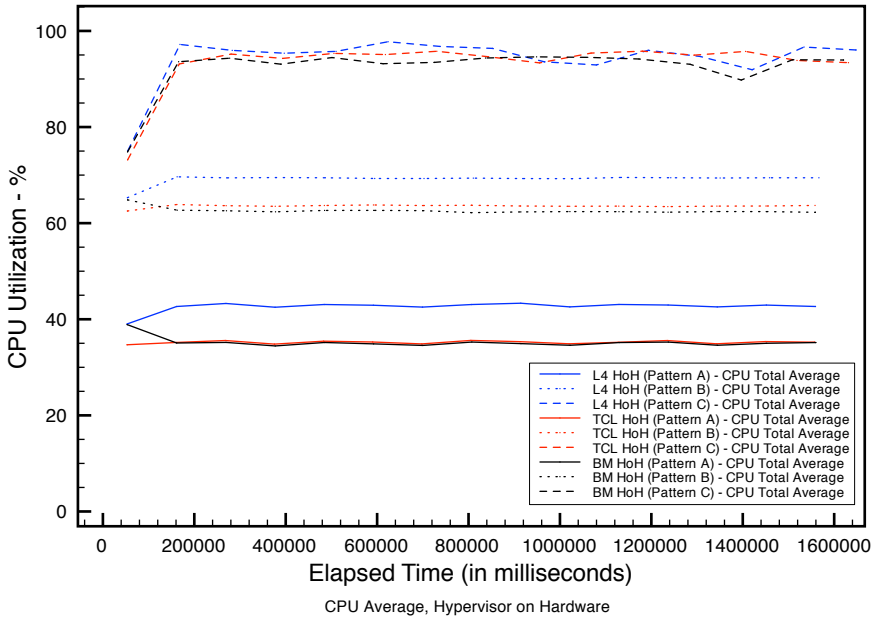

Figure 4.9: Graph showing the total CPU usage average for a population of 10 virtual machines on the hardware lab. Each of the 15 points contain the average of 100 data samples.

The results from the HiC lab are illustrated in figure 4.10 and show TinyCore Linux to have a higher CPU usage when compared with BareMetal OS and L4 Pistachio. L4 Pistachio has a lower CPU utilization than the other kernels for all three tests and also show the average to be more even than the other kernels.

4.4.2 Memory Usage

Graph 4.11 illustrates the memory footprints created by each kernel during the tests on the HoH lab while figure 4.12 shows the memory footprints on the cloud.

Graph 4.11 illustrates the memory usage of the hardware lab, and shows L4 Pistachio and BareMetal OS to have a significantly lower memory footprint when compared with TCL.

The average memory footprint of the host operating system was 321249kB for the L4 Pistachio tests, and the memory usage peaked to a maximum aver-

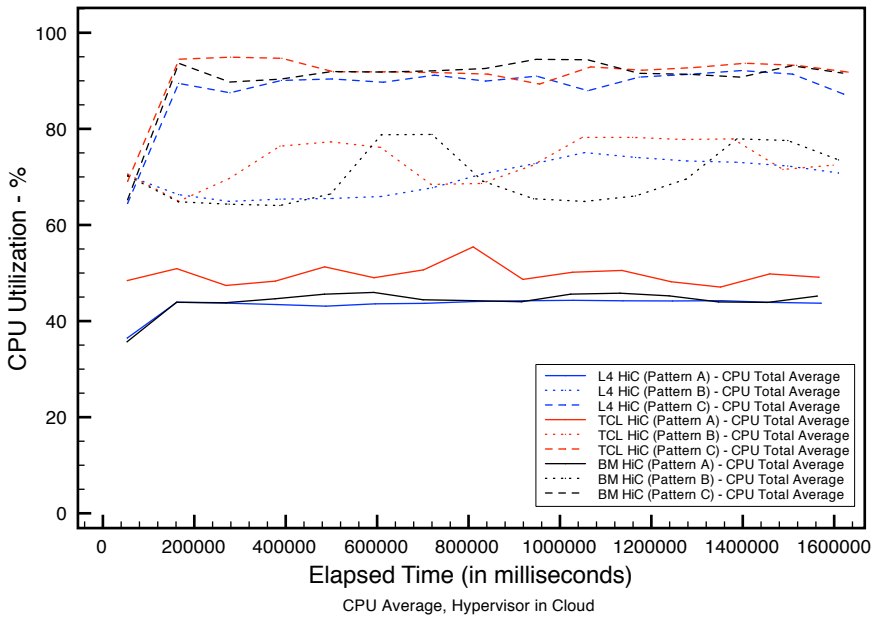

Figure 4.10: Graph showing the total CPU usage average for a population of 10 virtual machines on the cloud lab. Each of the 15 points contain the average of 100 data samples.

age of 380546kB during boot. However, for the remainder of the test the combined memory footprint of these 10 VM instances was on average 35338kB, or an average memory footprint for each VM of 3533.8kB, or 3.45MB. The average memory footprint of the host operating system was 321165kB for the BareMetal OS tests, and the memory usage peaked to an average maximum of 394740kB during boot. Average memory footprint of the 10 VM instances was 36249kB, or an average memory footprint for each VM of 3624.9kB, or 3.53MB. For Tiny Core Linux the host had an average memory footprint of 322088kB, and reached its average maximum at 872649kB. When the boot process had completed the memory footprint dropped to an average of 576306kB for the remainder of the tests. Subtracting the host memory footprint these results show 10 TCL VMs to have a total memory footprint of 254218kB, an average footprint for each VM of 25421.8kB, or 24.82MB.

The memory usage on the HiC lab is illustrated on figure 4.12 and show all of the kernels have a higher memory footprint than on the hardware lab. Notice on the graph that BareMetal OS and L4 Pistachio VMs has the same memory footprints which coheres with the behavior seen on the hardware lab tests.

The average memory footprint of the operating system for the L4 Pistachio tests was 174413kB. When booting the L4 Pistachio VMs the memory usage reached an average maximum of 255028kB but dropped to and average footprint of 232029kB for the duration of the tests. The total memory footprint

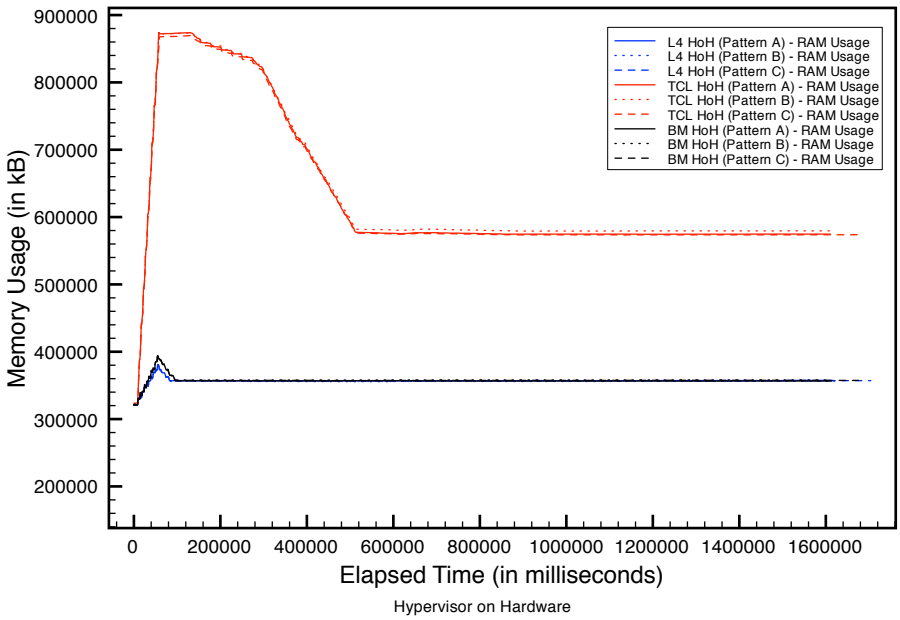

Figure 4.11: Graph showing the memory usage in kilobytes on the hardware lab for all tests with a population of 10 virtual machines.

created by a population of 10 VMs of L4 Pistachio was 57616kB, an average of 5761.6kB, or 5.62MB for each VM. Average memory footprint of the host operating system for the BareMetal OS tests was 174574kB. At boot time of these VMs the memory footprint reached an average maximum of 265137kB, but dropped to an average memory footprint of 231429kB. The total memory footprint of 10 BareMetal OS VMs was 56855kB, an average of 5685.5kBs, or 5.55MB for each VM. Results for the Tiny Core Linux VMs showed the host operating system to have an average memory footprint of 174018kB. Booting the TCL VMs reached an average maximum of 888668kB during boot of the VMs, but dropped to an average of 598110kB for the duration of the tests. The total memory footprint of 10 Tiny Core Linux VMs was 424092kB, an average of 42409.2kBs, or 41.41MB for each VM.

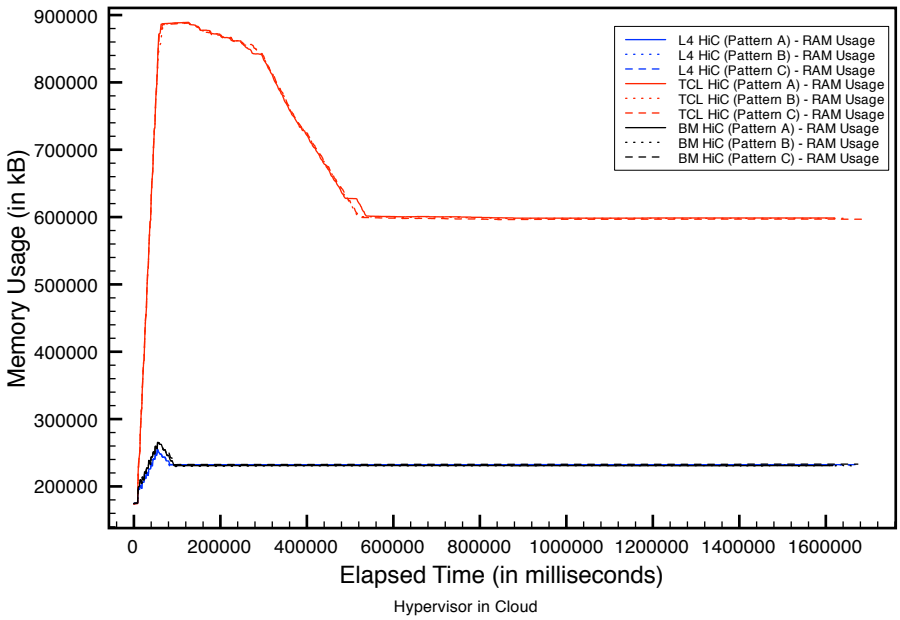

Figure 4.12: Graph showing the memory usage in kilobytes on the cloud lab for all tests with a population of 10 virtual machines.

Chapter 5

Discussion and Analysis

This chapter discuss different parts of the project and aims to provide the reader with a retrospective view of the entire process from beginning to end, and with an understanding of what has been achieved. The final sections talk about the performance of the minimal VMs and suggests future work.

5.1 Evaluating the choices made in the project

5.1.1 Minimal Virtual Machines

The idea behind this project was to see if it would be possible to model large populations of full-sized VMs by using a population of minimal VMs on fewer hosts. As full-sized VMs require a lot of computing resources they are not suitable candidates for modeling large populations of VMs as computing resources are usually limited on most systems. Especially within academia funding does not allow large investments to be made in expensive hardware to support a large number of VMs. The motivation behind this project was to be able to provide researchers and academia in general with the possibility of using real VMs in their work while at the same time using existing infrastructure. To be able to deploy large populations of VMs while being confined by the limitations of existing infrastructure this project wanted to take a closer look at the possibility of using μ-kernel technology and minimal operating systems as VMs to reduce the overall size and system requirements of full-sized VMs.

Virtual machines as a product

With cloud computing we have started to see new and original products emerge offering custom VMs for consumers packaged as ready-to-run environments with the most popular web-applications provided as application stacks. This project wanted to investigate if minimal VMs could be used for the same purpose, and if they would be able to compete with full-sized VMs. The obvious advantages of using minimal VMs when compared with full-sized instances

is their reduced size and impact on computing resources such as CPU and memory. For a cloud service provider such as Amazon EC2, minimal VMs would result in using less computing resources while increasing the number of client machines supported on the same infrastructure. Minimal VMs could also open new and exciting business opportunities such as simulating large-scale networks on the Cloud without further increasing the capacity of the data centers.

Cloud computing services such as Amazon EC2 and Windows Azure provide customers with different types of cloud services ranging from *SaaS*, *IaaS* and *PaaS*. As computing resources are delivered by the cloud service providers based the needs of each customer this project wanted to research into which extent a public cloud computing environment would be able to provide the resources needed to increase a small-scale population model even further.

The minimal VMs created for this project were able to achieve a significant size reduction, and application software was built and installed onto the VM images. However, using these VMs to host different application stacks proved difficult as they were not able to offer the same system calls and services as full-sized VMs, thus limiting the possibilities of their use for hosting many of the most common applications.

The L4 Pistachio VMs were found to be unsuitable for such purposes as the kernel would require its own file system, supporting a network interface and a command line interface to allow configuration of the services while the VM is running. Using L4 Pistachio for this purpose does not seem feasible at this point as it is a pure μ-kernel and would require lots of additional work in order to increase its usability and to achieve support for mainstream application software. Currently, it is only regarded as suitable for modeling purposes to simulate real-world usage patterns using custom made applications and to model populations of VMs in a cloud environment.

The results from this project show BareMetal OS VMs to be suitable for modeling purposes, however, its file system has a flat file structure and limited system calls which makes it unable to support application stacks. BareMetal OS is designed around a mono-tasking kernel which makes it unsuitable for multitasking purposes. As there looks to be no future plans to add multitasking capabilities to the kernel, this suggest BareMetal VMs are suitable for executing a single application at a time and would therefore be unable to compete with full-sized VMs.

Tiny Core Linux has a file system with much the same structure as mainstream Linux distributions. It supports running multitasking and is able to host the most popular applications including the popular LAMP stack. Although it was found to have poor performance on the cloud, it was able to outperform L4 Pistachio and BareMetal OS on the hardware lab. This shows Tiny Core Linux to be able to compete with full-sized VMs for the purpose of

hosting custom application stacks while having a significantly smaller memory footprint.

The experiences from this project show it is feasible to use minimal VMs for the purpose of hosting custom application stacks.

5.1.2 System Design

To provide answers to the problem statements, a hardware lab and a cloud lab was regarded as a requirement for this project. Deploying a hardware lab would offer the benefits of a bare metal hypervisor on the host and the use of Kernel-based Virtual Machine (KVM) as a virtualization infrastructure. Using a cloud lab would enable the possibility to deploy VMs inside other VMs (Nested Virtualization) and to compare the performance and behavior of VMs with and without the use of a bare metal hypervisor.

The computing resources on the hardware lab (Hypervisor on Hardware) for this project was approximately half of that of the cloud lab (Hypervisor in Cloud) in order for the latter. In theory this would allow doubling the population sizes and to compare scalability and performance of the populations when comparing the hardware and cloud environments.

The main benefit of choosing a lab environment with support for KVM was the use of a bare metal hypervisor, and as KVM is already part of the Linux kernel this makes KVM a commodity in the IT world as there are a large number of Linux installs in todays data centers. The results from this project would therefore be regarded as useful for the research community. KVM is also currently the only Linux kernel-integrated hypervisor, hence offering support of the same hardware which is offered by Linux device driver support.

A drawback of this system design is its modest computing resources when compared with the hardware used by well-funded researchers and businesses. However, it is able to show it is possible to create small-scale populations of VMs and repeating the same tests. The chosen design shows a single physical workstation may be used for the purpose of developing minimal VMs and to deploy small-scale populations without major investments in expensive hardware.

5.1.3 Choosing the right kernels

The first step towards creating minimal VMs for this project was to select suitable kernels. At the beginning of the project lots of effort was made into researching the most suitable candidates, and the approach was to select kernels of different designs in order to study their performance and differences. By comparing different types of kernels and studying their behavior the idea was

to be able to tell if they could be used for different purposes, such as hosting application stacks, modeling purposes and deployment on a cloud.

The choice were three different kernels; L4 Pistachio, BareMetal OS and Tiny Core Linux, as they represented three types of designs. The former is a pure μ-kernel written in C/C++, BareMetal OS is a mono-tasking OS with a kernel written entirely in Assembly and the latter use the traditional monolithic Linux kernel. These kernels were thought to be able to offer a significant reduction of memory footprints, system resource usage and overall size of VMs.

Three kernels was decided as a suitable number for this project to be able to stay within the time limitations for the project.

Why choose The L4 Pistachio μ-kernel?

The smallest kernel selected for this project was the L4 Pistachio μ-kernel. It was chosen for numerous reasons, but the obvious benefits was the availability of its source code, an active research community and online documentation. The kernel was considered to be mature enough for this project as it is the result of the last seven years of research within the field of microkernel technology and is actively maintained by the System Architecture Group at the University of Karlsruhe.

The strengths of L4 Pistachio is its focus on performance and portability and its small size which is in the order of 10,000 lines of source code. It supports C/C++ applications by allows implementing its own library. As support for C applications was required for this project this played an important part of the decision process. A 32-bit version of the kernel was chosen for this project as the strengths of choosing a 32-bit kernel would allow deployment of VMs on both 64- and 32-bit virtualization environments. Overall, its tiny size, elaborate documentation, Qemu compatibility and support for C applications made it an excellent choice of kernel as it met all the requirements for this project.

The weak points of L4 Pistachio μ-kernel is its lack of extensibility and usability as it does not deliver the same features as is expected of a complete operating system. It does not provide a CLI, file system or device drivers which suggests the kernel to be a good candidate to be used for modeling purposes by simulating real-world usage patterns. Using L4 Pistachio for simulating real networks with network connectivity would also require custom made device drivers to be developed which is a tedious task.

Choosing this kernel for this project proved to be useful as it was able to significantly reduce the size of the virtual machine. Looking back on the experiences gained from using L4 Pistachio it has shown a great deal of work is involved when choosing a μ-kernel for such a project. However, the work paid off with the smallest virtual machine in the project. The results also show that

it performs better than traditional kernels on a cloud environment, and has acceptable performance in populations of up to 32 VMs on a hardware lab. It is suggested adding a file system and network drivers to L4 Pistachio and test its performance as a operating system in even larger populations. As the kernel allows to be configured for several CPU architectures it suggests testing the performance using different CPUs and on several cloud environments.

Why choose BareMetal OS?

BareMetal OS, a minimal 64-bit operating system was chosen as its kernel is written completely in assembly code and offer more functionality and extensibility than L4 Pistachio. Noticeable features of BareMetal OS is the implementation of a CLI, focus on mono-tasking, and its own file system (BMFS) optimized for large files by dividing the disk into 2 MiB blocks, making it suitable for being used for CPU intensive tasks such as working with large data sets.

The strengths behind BareMetal OS is the philosophy behind the OS. It is designed as a mono-tasking OS to offer reduced complexity when compared with multitasking systems. It also allows sub-tasks to be submitted to multiple CPUs, supporting multiprocessor systems with up to 128 x86_64 processors. Comparing its performance with a μ-kernel and a Linux kernel would provide valuable knowledge about its performance and if it would be able to compete with full-sized VMs.

The weak sides of BareMetal OS is the flat file structure of its file system which makes it difficult to organize files into separate directories, hence reducing its usability. As the entire kernel is written in Assembly code this also increase the difficulty for programmers in modifying the kernel as it requires knowledge about Assembly programming. Converting the kernel code into C/C++ would make the kernel more suitable for this type of research as it would allow a larger community to be a part of the development process, further increasing the performance, security, stability and extensibility of the kernel.

This kernel was a good choice for this project when looking back on the results. It showed to perform better than the Linux kernel on the cloud environment and had the least increase in CPU overhead between the different population sizes. Together with L4 Pistachio it achieved the smallest memory footprint during the tests, but performed better on population sizes larger than 32 VMs. The experiences and results from this project show BareMetal OS to be a good candidate for future research on a larger scale on both hardware- and cloud environments as its performance does not vary significantly suggesting it to be a highly optimized kernel.

Why choose Tiny Core Linux?

A full-sized kernel had to be chosen next to L4 Pistachio and BareMetal OS for comparison reasons. Tiny Core Linux is a stripped down version of the Linux core and supports more than 3200 extensions to provide the support of additional features. It is also available for download in 3 variants of "cores"; *TinyCore* and *CorePlus* and *Core* offering both new and experienced users with the options to choose from using just the core system or a version offering a range of extensions.

The strengths of choosing Tiny Core Linux *CorePlus* was the implementation of a CLI, an installer as well as a range of extensions to support window managers and wireless support. It also uses the Linux kernel which is able to show how the performance and behavior of one of the most commonly used kernels when used compared with minimal VMs for modeling purposes.
.

The use of Tiny Core Linux for this project showed it to be difficult to modify the files on its filesystem as it is contained within a single file. Additionally, there are a large number of minimal Linux distributions available which proved it difficult to know if the most extensible and minimal distribution had been chosen for the project. The results from the hardware and cloud lab also indicate KVM to be able to recognize the use of a Linux kernel and was able to perform some sort of optimization for these types of VMs. As the lack of KVM resulted in a significant performance hit when compared with the other kernels this suggests future work to focus on testing the same populations on the hardware lab with and without the use of KVM.

5.1.4 The process of building the kernels

After selecting the kernels for this project the next task was to compile and assemble the kernels and compile custom applications for each one. Before beginning work on this project the author had no previous knowledge about C programming and the use of compiler tools which proved to be the most significant challenge. It was therefore especially rewarding, and regarded as a personal achievement, to acquire knowledge about the operationalization of compiler tools and how to create applications in the C language.

It was especially challenging during this project to create a working environment for the L4 Pistachio kernel as the 32-bit Linux Debian distribution which was initially used for this purpose did not seem to successfully compile a working kernel. The kernel was unable to boot on both VMware, Virtualbox and Qemu despite trying different versions of the GCC compiler and it was thought to be the result of Debian not using the most recent version of GCC. As lots of time and effort was put into compiling the L4 Pistachio kernel these problems almost made the author abandon the kernel.

The same problems were not encountered with BareMetal OS as the developer provided a working Qemu image for download on its project website. The biggest problem encountered with this kernel was a bug which resulted in 100% CPU load when using the sleep function ”d_delay()” when building the application software. This bug was solved by contacting the developer of BareMetal OS for suggestions on how to resolve it. The response from the developer was a 2-line fix which solved the problems and required the source code to be modified and the kernel had to be re-assembled. After applying the fix an image file containing Pure64, the software loader for BareMetal OS, had to be downloaded and the new kernel had to be transferred onto it. As this bug was encountered in the middle of the project this could have easily eliminated the kernel from the project as it would have been impossible to create the same applications for this kernel as for the others.

Tiny Core Linux was able to be installed using an installer and did not require any specific development environment as its kernel was already compiled. However, a 32-bit compiler had to be used to build the custom applications. This kernel proved to require the least amount of time in getting to work and also was the only kernel supporting the GNU C library making it easier to develop custom applications.

The task of compiling the kernels proved to be a time consuming and complex task as little, or no documentation existed to explain the exact details of the approach. The experiences suggests previous knowledge about the operationalization of compiler tools is necessary in order to increase the efficiency in this part of the project. The approach documented in this report also provide others with valuable knowledge about how to compile and configure these kernels which is beneficial for others wishing to use the same kernels for their own research. The tool developed for L4 Pistachio was especially useful as it automates the entire process of building the kernel and its applications and only requires a few variables to be set at the beginning of the script. As no other tools like this exists for L4 Pistachio this is regarded as an important contribution for future research projects.

5.1.5 The process of creating minimal virtual machines

As the kernels used for this project had to be used for simulation purposes in the form of minimal VMs they had to be transferred onto bootable Qemu images to make them into minimal VMs. There were several challenges when creating these images and making them bootable, as each kernel required a different approach. The image files had to be manually created for L4 Pistachio and Tiny Core Linux while the image for BareMetal OS was already available for download from the project website.

As previously mentioned there was a bug in the BareMetal kernel which caused the CPU usage to rise to 100% and required to download a Pure64 im-

age containing a 64-bit software loader which had the purpose of loading the BareMetal kernel when booting the VM. There were no additional problems in making a BareMetal VM and the final result was a VM of 32MBs VM image size able to boot on 16MBs of virtual memory.

The instructions found on the website of L4 Pistachio mentioned the approach to create the image and how to make it bootable. As the instructions found on the L4 website explained how to make the image bootable using GRUB Legacy the author tried to find a way to install GRUB2 on the image as this is the default version of GRUB on recent Linux distributions. This attempt proved to be unsuccessful. The problems of making GRUB2 work on the image resulted in following the existing documentation and reverting to GRUB Legacy. In addition to installing GRUB Legacy onto the image the image had to be mounted as a loop device and the kernel and binaries had to be copied to the image. The final result was a minimal VM of 1.5MBs being able to boot using a minimum of 1.68Mbs of virtual memory.

Creating a VM image for Tiny Core Linux was easier as it cam with an installer. The approach required to tell Qemu to boot from the ISO file and use the image as a hard-drive. The problems which were encountered were mostly regarding with experimenting with the different image sizes to allow enough disk space for the file system while achieving the smallest size. The final result was a 20MB image file and a VM able to boot with a minimum of 48MBs of memory.

What was learned from this? Even though only three kernels were used they show how only a small number of kernels require different approaches and that it looks feasible to create minimal VMs for most types of kernels given enough time and effort. It also shows it is possible to create minimal VMs which are able to compete with mainstream Linux distributions.

5.1.6 Process of creating a fair test

The main focus of the project had been on creating minimal VMs for the purpose of competing with full-sized VMs which proved feasible. But in order to make them simulate real-world behavior they had to perform some type of task to be able to tell if they would react or perform differently when performing real work. The idea was to create applications which were able to simulate real-world application patterns and automatically boot these applications after the boot of each VM.

In order to create a fair test all usage patterns had to be identical for each kernel. Because of time restrictions for this project it was decided to create three different CPU patterns. Effort was made into using code which required as few additional headers as possible and would allow to use the same source code for all three kernels. The decision was made to calculate the fibonacci

sequence recursively as it did not require the use of any additional headers which made the code identical. The only difference was the inclusion of kernel-specific headers providing the sleep functions. Using the iterative approach when calculating the fibonacci sequence was also experimented with, but when compiling the source code for L4 Pistachio the compiler optimized the code in such a way that the test never initialized. Effort was made into making iterative calculations work with L4 Pistachio and the community was also asked for suggestions without any answers leading to a final solution to the problem.

Three patterns were tested for each VM instance and the results showed BareMetal OS and Tiny Core Linux to achieve identical behavior. However, L4 Pistachio deviated slightly from the other two suggesting the source code had to be modified. Inspecting the results it showed the problem was related to L4 Pistachio spending slightly less time on calculating the fibonacci sequence number which indicated the fibonacci sequence number had to be slightly increased to match the time of the other VMs.

The process of creating fair tests showed that the difference between the kernels makes it difficult to create fair tests using the same source code. The reason for these subtle changes in results might be one of several. The most probable reason is that the configure script for L4 Pistachio performs optimization on the code during build time of the application software, hence causing a slight difference in the compiled code.

5.2 Performance analysis and tools

The performance of the VMs are discussed in this section and shows the total time spent by the same population sizes for each of the three types of VMs. These results are illustrated in figure 5.1 and 5.2. As suggested in section 4.3.2 in the Results chapter, an additional population of 64 VMs was deployed on the hardware lab to see if the results were able to show an overall trend. It continues discussing the trend when increasing the population sizes and talks about the results from the usage pattern tests.

5.2.1 Time required to complete the tests

Figure 5.1 illustrates the total time spent in milliseconds for each kernel to complete the tests on the hardware lab. The hardware clock on the host was used to collect timestamps for each data sample and was used to give an accurate estimation of the time required in completing these tests.

All the VMs show an increase in the total time needed to complete the tests with each population increase , however, L4 Pistachio show a significantly larger increase when compared to TinyCore Linux and BareMetal OS. L4 Pis-

tachio show the highest increase of 389876ms, TinyCore Linux 108480ms and BareMetal OS to have the lowest increase of 54375ms. The results suggest BareMetal VMs to be surpassed by TinyCore Linux at some given population size. The results also show L4 Pistachio to have a significantly higher performance hit when compared to the other kernels on populations greater than 32 VMs suggesting it to be a poor candidate for large-scale models while BareMetal OS is the most stable kernel as it increases the least in total time as the population size increases.

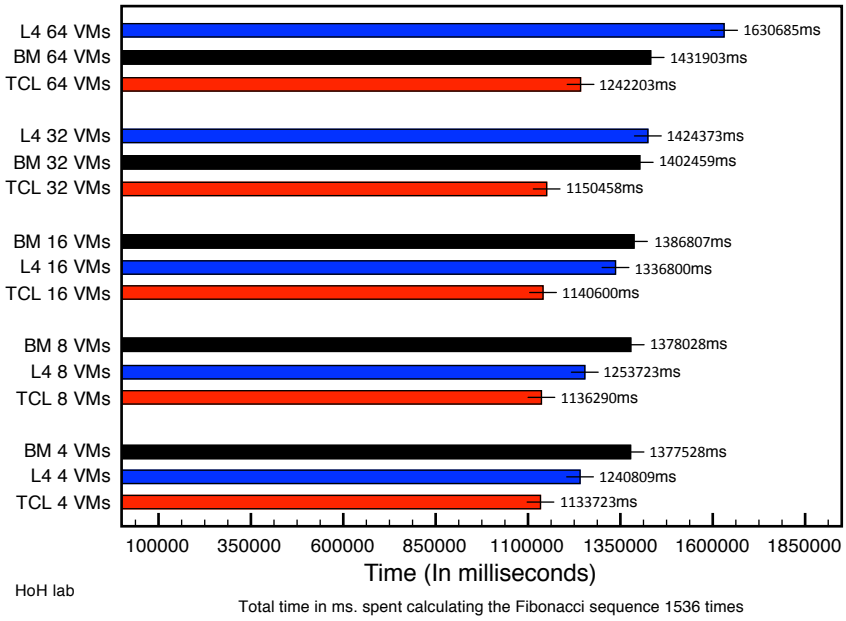

Figure 5.1: Comparison of the total time spent in completing the tests for populations of 4,8,16, 32 and 64 virtual machines on the hardware lab. The time is shown in milliseconds.

The results for the cloud lab is illustrated in figure 5.2. Examining the results show all VMs to spend more time as the population size increases. However, there were some unexpected data when looking at the results from the L4 population of 16 VMs. This result showed the total time needed to complete the tests was higher than for a population size of 32 VMs, suggesting the L4 16VMs result might be the result of varying conditions on the cloud at the time when the test was performed. As the bar chart illustrates, TinyCore Linux increase the most with a total time of 402341ms, L4 Pistachio increase by 155689ms and BareMetal OS show the least increase of 139158ms.

Summary of the results

The results were expected to show an increase in the time needed to complete the tests when the population sizes increased as they generate more context

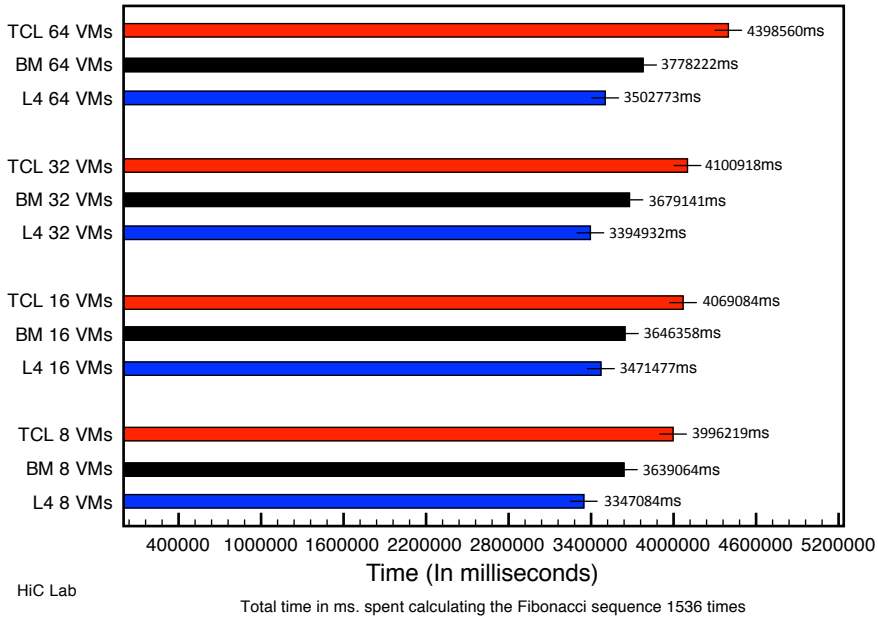

Figure 5.2: Comparison of the total time spent to complete the tests for populations of 4,8,16, 32 and 64 virtual machines on the cloud lab. The time is shown in milliseconds.

switches than small populations.

Tiny Core Linux VMs were able to complete all the tests on the hardware lab within the shortest amount of time when compared with the other VMs. However, comparing these with the cloud results showed Tiny Core Linux to finish last on all the tests. L4 Pistachio VMs completed the tests before the BareMetal VMs on the hardware lab up until a population of 16 VMs, but when reaching a population of 32 VMs the former spent more time than the latter. Analysis of the results show L4 Pistachio to complete all the tests in a shorter amount of time than BareMetal VMs for all the tests. Worth noticing is the difference of the Tiny Core Linux VMs as they go from requiring the least amount of time on the hardware lab to spend the longest time on the cloud tests. There might be a possibility that KVM is able to recognize that these VMs are using the Linux kernel and therefore is able to perform some sort of optimization to increase the performance.

These results are useful as they are able to determine the population limits of each VM and show minimal BareMetal VMs to perform better than L4 Pistachio VMs when the model is larger than 32 VMs. As the L4 Pistachio VMs performs better than BareMetal OS in populations smaller than 32 VMs this suggests the former VM to be used for simulation purposes in populations below 32 VMs on the hardware lab, and allow the cloud to take the population even further.

The results also tell us that the performance of the three types of minimal

VMs vary between population sizes, and additional tests have to be performed on different hardware and with more kernels in order to find the most optimal environment for modeling purposes. L4 Pistachio is able to exemplify the potential of future minimal VMs as its image size was only 1.44MBs. This is a significant reduction of 2.67 orders of magnitude when compared with the standard Ubuntu image which is 684MBs. As this project selected only three out of the many available kernels and was able to achieve a working minimal VM within a short amount of time, this suggests the possibilities are endless as to creating even smaller VMs.

5.2.2 The overall trend when increasing population sizes

The bar charts in section 5.2 illustrated the total time spent by each minimal VM population size to complete the same amount of work. However, these bar graphs were unable to accurately depict the time difference for each population and type of VM. The purpose of this section is to illustrate this difference and to discuss the results.

Figure 5.3 shows the difference in time required to complete the tests on the hardware lab for different population sizes. Each VM has its corresponding trend line illustrating this difference, and by looking at the line representing BareMetal OS it is clear that these VMs have predictable performance with each population increase. Tiny Core Linux show a slight linear growth until a population size of 32 after which the line shows a sudden and significant increase when scaling to 64 VMs. L4 Pistachio show a small, but sudden, increase when scaling from 8 to 16 VMs and then continues on a much steeper linear increase for the other populations.

The trend lines for the cloud lab is illustrated on figure 5.4. By looking at the BareMetal OS trend line it is clear that the increase has predictable results for all the population increases. There are no sudden or major increases in the time needed to complete the tests when increasing the population sizes, and the line shows a slight curve suggesting BareMetal OS handles scalability on the cloud nicely. Also, notice the trend line for L4 Pistachio which shows a sudden increase when scaling to a population of 16VMs before dropping and continuing on almost the same path as BareMetal OS when reaching a population of 32 VMs. At this point the L4 pistachio trend line continues on a similar path as the BareMetal OS trend line.

As a result of the abnormal test result as seen in the results for L4 Pistachio for a population of 16 VMs the test was repeated in order to see if the new results would drop to a lower level. However, as all the tests illustrated in the graph were performed two days before the data was analyzed additional tests proved inconclusive. The new results showed an even larger increase which was most likely as a result of varying cloud conditions. The Tiny Core Linux trend line shows a significant difference with every increase in the population

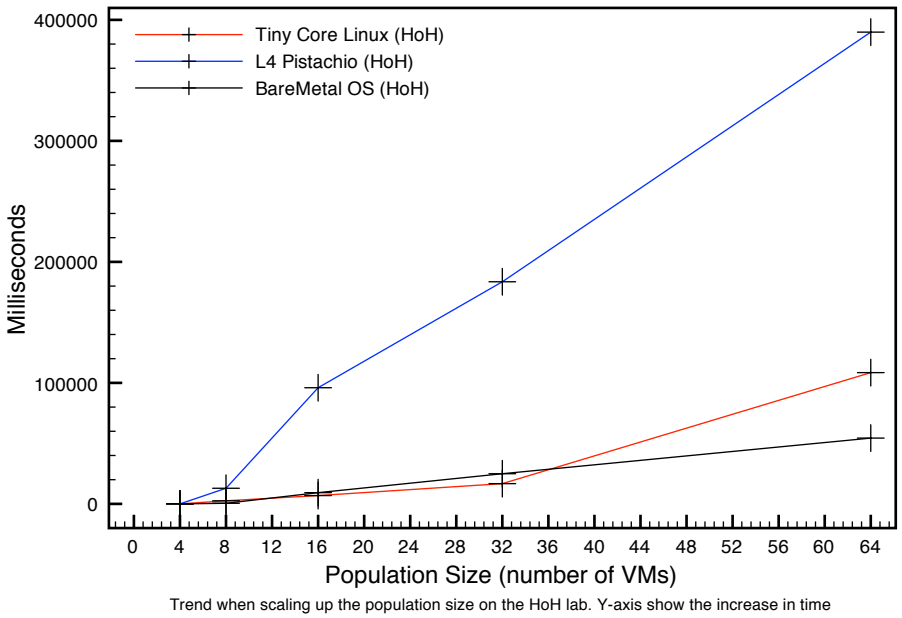

Figure 5.3: Graph shows how much longer each population size required in order to complete the same work and shows the difference for population sizes of 4 to 8, 16, 32 and 64 virtual machines on the hardware lab. Y-axis is the difference in milliseconds.

size when compared with the other kernels, and additional tests with larger population sizes would be required to tell if this behavior will continue to increase at the same rate or if it will start showing signs of slowing down.

These results shows how minimal VMs built on less traditional kernel designs are capable of competing with the Linux kernel when increasing the population sizes. Especially comparing the results from BareMetal OS with Tiny Core Linux shows the former to require less additional time to complete each test when increasing the population size. What we have learned from these results are that minimal VMs are well suited to be used for modeling larger populations of VMs and they should also be well capable of competing with full-sized VMs.

5.2.3 Usage Patterns

The source code of all the three patterns for L4 Pistachio was slightly modified to calculate a fibonacci number higher than the two other kernels and the results were analyzed to see if the patterns had changed. The results showed the modifications had achieved its purpose, and all three kernels had the same behavior for all three patterns. The results from the usage pattern tests show it is possible to create software applications for minimal VMs with much the same code for the purpose of simulating real-world usage patterns, and that

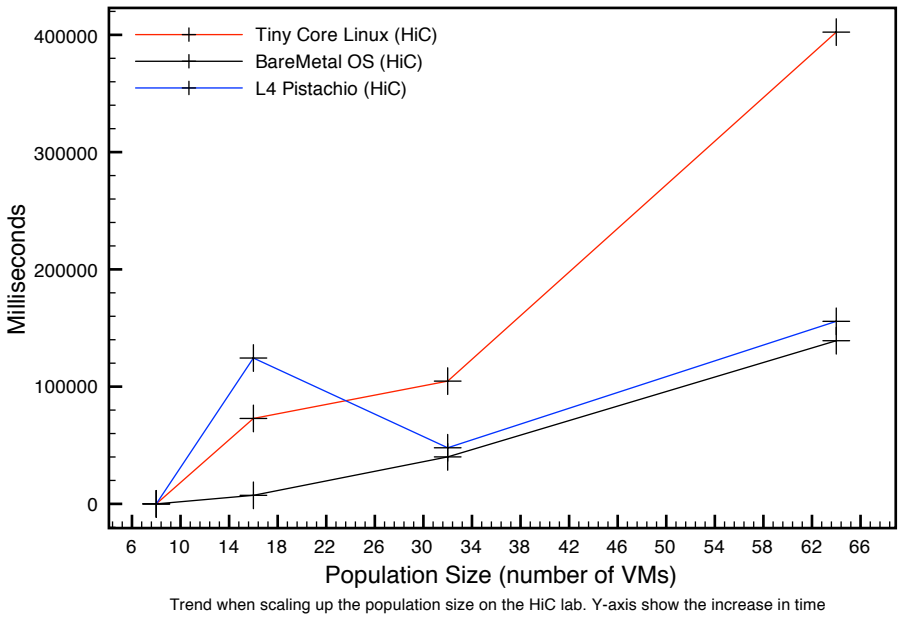

Figure 5.4: Graph shows how much longer each population size required in order to complete the same work and shows the difference for population sizes of 8 to 16, 32 and 64 virtual machines on the cloud lab. Y-axis is the difference in milliseconds.

each VM is able to execute these programs successfully. This feature would be especially useful for researchers and students wishing to create their own usage patterns to simulate different behavior on minimal VMs. The results also indicated fair tests had been achieved for all three kernels.

Looking at the results from the usage pattern tests the cpu usage average is more predictable on the hardware lab than on the cloud. The results seen on figure 4.9 and 4.10 in section 4.4.1 show how the CPU averages on the hardware lab remain more or less at the same level for the duration of the tests, while the results on the cloud show the load averages to have variations throughout the tests. Such behavior was expected and shows the behavior of cloud environments where the performance vary with the current number of users on the cloud, which directly affects the computing resources available for each VM. However, the variations were not significant and suggests a population of minimal VMs can be deployed on a hardware lab and behave similarly on the cloud. This is useful knowledge when deciding to increase the model populations from a hardware lab to a cloud environment.

Memory usage was also analyzed in section 4.4.2, and results show there are significant memory footprint differences between Tiny Core Linux and the two other kernels on both labs. On the hardware lab the results from figure 4.11 showed the average memory footprint for L4 Pistachio VMs to be 3.45MB, BareMetal VMs 3.53MB while each Tiny Core Linux had a footprint

of 24.82MB. The results from the cloud lab, as seen on figure 4.12, showed all the kernels to have significantly increased their memory footprints when compared with the hardware lab to a total of 5.60 MBs for L4 Pistachio, BareMetal 5.54MB, and 41.41MBs for each TCL VM. Tables 5.1 and 5.2 lists the memory footprints for each pattern and the averages for each VM.

Memory footprints - Hypervisor on Hardware			
Pattern	L4 Pistachio	BareMetal OS	Tiny Core Linux
A	3.48MB	3.54MB	24.55MB
B	3.41MB	3.52MB	25.26MB
C	3.46MB	3.55MB	24.66MB
Average	**3.45MB**	**3.53MB**	**24.82MB**

Table 5.1: Footprints for pattern A, B and C on the hardware lab. The bottom row shows the average memory footprint for each VM.

Memory footprints - Hypervisor in Cloud			
Pattern	L4 Pistachio	BareMetal OS	Tiny Core Linux
A	5.62MB	5.52MB	41.48MB
B	5.57MB	5.48MB	41.48MB
C	5.68MB	5.65MB	41.27MB
Average	**5.62MB**	**5.55MB**	**41.41MB**

Table 5.2: Footprints for pattern A, B and C on the cloud lab. The bottom row shows the average memory footprint for each VM.

The disadvantage of changing the fibonacci sequence number was that the source code was no longer identical. To be sure the problem was not related to the specific source code used for the tests, several other types of calculations were tested on all the kernels. Factorials, Iterative fibonacci implementation and counting every integer produced the same difference between L4 Pistachio and the other kernels. As the purpose of these patterns was to give the CPU an equal amount of work to be able to produce identical patterns it was decided to stay with the recursive implementation of the fibonacci sequence and slightly modifying the code to produce the same CPU load.

The benefits of testing only the CPU patterns instead of both CPU and memory combinations was that the VMs would only utilize the CPU which kept the memory footprint of the VMs at a minimum throughout the tests. The data was able to show the average memory footprint for all VMs, which is useful knowledge when wanting to increase the model population further. This project has proven it is feasible to create such patterns, hence researchers would be able to use even more advanced patterns future projects.

5.2.4 Tools created for the project

Many different tools were used during the project. Most of the tools perform simple tasks, such as compiling applications and adding them onto an image. Other tools performed more advanced tasks such as booting multiple VMs and collecting system information. All of the tools have proved valuable for the project by automating many of the manual procedures allowing more focus on the task at hand. Table 5.3 lists all the different tools which were developed during this project, and they are also found in the Appendix.

Name	Description
Bootscript.pl	A script made for the purpose of booting multiple VMs.
Build.sh	Shell-script which compiles all the applications created for BareMetal OS.
Mount.sh	A script made for the purpose of mounting an empty BareMetal OS image and transfer applications onto it.
Perf.pl	A script for collecting system information.
tcl_add_files.pl	A script to add an application to a Tiny Core Linux image.
Makescript.pl	A script to automatically configure and compile the L4 Pistachio kernel and its software applications. It also adds the binaries onto the image and calls the script ”grub.sh” to make the image bootable.
grub.sh	A script for installing GRUB on the L4 image

Table 5.3: List of the different tools used in this project.

Especially the development of the ”Makescript” tool is regarded as the most useful tool for this project as it greatly simplify the process of building the L4 Pistachio VMs. By using the tool the entire VM is automatically created which is valuable for researchers wishing to spend their time on research rather than on entering build commands and manually transferring application software to the images.

The project also provides a tool to automatically boot the VMs, simplifying the deployment of large models. The tool ”Bootscript” takes care of this project and gives the researcher control of the number of VMs to deploy and also automatically executes the script ”Perf” which collects statistics from the host system while the model is deployed.

The other tools perform simpler tasks such as building the software and transferring it to the VMs.

Developing these tools allowed more focus on analyzing the behavior of the VMs and the results instead of spending the time on building them. The weak sides of some of the tools are that they would have to be customized in order to work with other kernels, but this should not require too many changes to be made.

5.3 Future Work

5.3.1 Even smaller minimal VMs

The size reduction which was achieved for this project within the short amount of time available suggests future research would be able to provide even smaller virtual machines, further increasing the population sizes. There are still a large number of kernels available for this type of research, such as Minix3[40], μ-velOSity[41] and [42]. Future research into additional kernels such as the aforementioned is suggested.

5.3.2 Tiny Core Linux and application stacks

There is still lots of work having to be done where the Tiny Core Linux VMs must be equipped with the stacks and tested under different scenarios to find the most optimal performance. This is suggested as future work as this project focused on the feasibility of using minimal VMs with application stacks, and not the actual implementation of application stacks.

5.3.3 Improve the fair tests

Creating fair tests for the different kernels require choosing code which is as simple and similar as possible requiring the least amount of changes. L4 Pistachio had a slight performance difference when compared to BareMetal OS and Tiny Core Linux when the source code for all three kernels was identical. This difference might be the result of the compiler optimizing the L4 Pistachio source code during compile time, or some unique features of the kernel itself. This exemplifies the difficulties of creating an identical cross-kernel fair test without making slight changes in the source code. However, it should be possible to create fair tests which behave equally on all kernels when using the same code, but the time limitations for this project suggests this to be future work.

5.3.4 Additional lab environments

The results are able to show it is feasible to deploy minimal VMs on different environments and being able to compare their performance by using the same tests. On the other hand, these results do not provide answers to the behavior on other types of hardware and how the varying cloud conditions affected the tests performed in this project. Repeating these tests on private clouds and additional hardware is suggested as future research.

5.3.5 More Complex usage patterns

Only three usage patterns were created for this project, and it is suggested for future work to increase the number of patterns. One suggestion would be to monitor real usage on a web server and simulate the same patterns on the VMs. Memory patterns should also be created, either as isolated tests, or in combination with CPU usage in order to measure the overall impact on system resources. This is suggested as future work as it would be valuable knowledge on how minimal VMs would perform and compete with full-sized VMs when doing ”real” work.

Chapter 6

Conclusion

The main goal of this project was to investigate *to which extent it is possible to model large populations of full-sized virtual machines, using minimal virtual machines on fewer hosts.*. Closely related to this problem statement the project also investigated *to which extent a public cloud computing environment is able to provide the resources to increase the model-populations even further.*. Finally the project wanted to see if *minimal VMs would be able to host custom application stacks as ready to run environments and be able to compete with full-sized VMs used for the same purpose*.

During this project the following has been achieved:

- Minimal VMs were created using kernels still in the early research stages
- Custom application software was built specifically for each kernel
- New tools have been developed
- Small-scale populations were created
- Both hardware- and cloud environments were tested
- Results have been produced and analyzed

The project was able to create minimal VMs significantly smaller than full-sized VMs, and also showed it was possible to build custom application software able to generate different CPU usage patterns. Minimal VMs were deployed in small populations and scaled up to 16 times the original population size on both a hardware- and a cloud environment, and were able to outperform the Linux kernel on the cloud.

Using minimal VMs for custom application stacks, such as LAMP, does not seem feasible for L4 Pistachio and BareMetal OS at this moment. These kernels are currently active research projects and it would be cumbersome work to achieve support for such stacks. Currently these VMs may only be used for modeling purposes as they lack many of the system calls necessary to host such stacks. However, Tiny Core Linux can be used for such purposes as it use the Linux kernel which supports many of the same applications as mainstream

Linux distributions.

The following are some notable achievements accomplished during this project:

- The smallest VM created was for L4 Pistachio with a image size of 1.5MB and required a minimum of 1.68MBs of memory to boot
- Tiny Core Linux VMs were found to be the most extensible minimal VM, and able to support custom application stacks
- BareMetal OS VMs had the most predictable performance on both the hardware and the lab environment when increasing populations sizes (Table 5.3 and 5.4)
- A population of almost 500 minimal L4 Pistachio VMs equals a population of one Ubuntu Server (currently 684MBs).

The following achievements have also been made:

- The smallest average physical memory footprint achieved on the hardware lab was L4 Pistachio VMs with 3.45MB (Table 5.1)
- The smallest average physical memory footprint on the cloud lab was Bare Metal OS with 5.54MB (Table 5.2)
- Tiny Core Linux had the best performance on the hardware lab (Table 5.1)
- L4 Pistachio had the best performance on the cloud lab (Table 5.2)
- A public cloud has shown to deliver varying performance compared with a hardware environment, which may cause noise in measurements (Figure 5.4)

Minimal VMs have proven they are a viable approach to simulating real behavior of full-sized VMs. They are able to give researchers and students the tool needed to deploy large populations of VMs on limited computing resources. The results show it is possible to use minimal VMs for the purpose of hosting popular application stacks as they are able to offer the same level of extensibility as full-sized VMs. Minimal VMs can deliver better performance and achieve significantly lower memory footprints than Linux-kernel VMs on a cloud environment, which shows that cloud environments are able to provide the resources needed to take the model populations even further. This project has shown the many possibilities which exists within this research topic and that there is still lots of research needed to be done. Most importantly, the project has proven large populations of virtual machines can be modeled by using minimal virtual machines as they are significantly smaller and are able to outperform full-sized VMs .

Bibliography

[1] Jeff dean, keynote @ONLINE, May 2012. http://www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis2009.pdf, [Online; accessed 10-May-2012].

[2] Amazon ec2 homepage @ONLINE, March 2012. http://aws.amazon.com/ec2, [Online; accessed 12-March-2012].

[3] Windows azure website @ONLINE, March 2012. http://www.windowsazure.com/en-us/, [Online; accessed 10-March-2012].

[4] Dropbox website @ONLINE, May 2012. https://www.dropbox.com/, [Online; accessed 11-May-2012].

[5] Google docs website @ONLINE, May 2012. https://docs.google.com/, [Online; accessed 11-May-2012].

[6] Microsoft office365 website @ONLINE, May 2012. http://www.microsoft.com/en-us/office365/, [Online; accessed 11-May-2012].

[7] Bitnami website @ONLINE, May 2012. http://www.bitnami.org, [Online; accessed 11-May-2012].

[8] Cloudsim website @ONLINE, May 2012. http://www.cloudbus.org/cloudsim/, [Online; accessed 11-May-2012].

[9] Rodrigo N. Calheiros, Rajiv Ranjan, César A. F. De Rose, and Rajkumar Buyya. Cloudsim: A novel framework for modeling and simulation of cloud computing infrastructures and services. *CoRR*, (GRIDS-TR-2009-1), March 2009. http://arxiv.org/abs/0903.2525.

[10] Jochen Liedtkde. Improving ipc by kernel design. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, volume 27, pages 175–188, New York, NY, USA, December 1993. ACM.

[11] Jorrit N. Herder. Towards a true microkernel operating system. Master's thesis, Vrije Universiteit Amsterdam, February 2005.

[12] Bitnami website @ONLINE, February 2012. http://www.bitnami.org, [Online; accessed 06-February-2012]].

[13] Bitnami readme @ONLINE, May 2012. http://bitnami.org/files/stacks/lampstack/5.3.12-0/README.txt, [Online; accessed 16-May-2012].

[14] T. Mikjaniec, A. Manning, D. Small, and J. VanGilder. Data center design using improved cfd modeling and cost reduction analysis. In *Semiconductor Thermal Measurement and Management Symposium (SEMI-THERM), 2011 27th Annual IEEE*, pages 97–103, march 2011.

[15] Wikipedia. Monolithic kernel — wikipedia, the free encyclopedia @ONLINE, 2012. [Online; accessed 2-20-2012].

[16] Victor R. Basili and Barry T. Perricone. Software errors and complexity: An empirical investigation. *Commun. ACM*, 27(1):42–52, jan 1984. http://doi.acm.org/10.1145/69605.2085.

[17] Thomas J. Ostrand and Elaine J. Weyuker. The distribution of faults in a large industrial software system. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 55–64, New York, NY, USA, 2002. ACM. http://doi.acm.org/10.1145/566172.566181.

[18] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 73–88, New York, NY, USA, 2001. ACM.

[19] L4 kernel @ONLINE, March 2012. http://os.inf.tu-dresden.de/L4/l3.htm, [Online; accessed 1-March-2012].

[20] Eumel principles @ONLINE, April 2012. http://tunes.org/wiki/eumel.html, [Online; accessed 2-April-2012].

[21] Android website @ONLINE, May 2012. http://www.android.com/, [Online; accessed 5-May-2012].

[22] Thorsten Leemhuis. Linux code size @ONLINE, February. http://www.h-online.com/open/features/What-s-new-in-Linux-3-2-1400680.html?page=3, [Online; accessed 17-February-2012].

[23] Larry Hughes, Hosein Marzi, and Yanting Lin. A new approach in designing interprocess communication for real-time systems. *International Journal of Software Engineering and Knowledge Engineering IJSEKE*, 15(2):259–264, 2005.

[24] Niek Linnenbank. Implementing minix on the single chip cloud computer. Master's thesis, Vrije Universiteit Amsterdam, August 2011.

[25] L4 pistachio website @ONLINE, January 2012. http://www.l4ka.org/.

[26] Baremetal os website @ONLINE, January 2012. http://www.returninfinity.com/baremetal.html,[Online: accessed 4-March-2012].

[27] Tiny core linux website, http://distro.ibiblio.org/tinycorelinux/, February 2012.

[28] System architecture group website @ONLINE, March 2012. http://os.ibds.kit.edu/, [Online; accessed 8-March-2012].

[29] Disy group website @ONLINE, March 2012. http://www.disy.cse.unsw.edu.au/, [Online; accessed 9-March-2012].

[30] L4 project website @ONLINE, February 2012. http://www.l4ka.org/65.php,[Online accessed: 5-march-2012].

[31] Kvm homepage @ONLINE, January 2012. http://www.linux-kvm.org/page/Main_Page,[Online accessed: 18-january-2012].

[32] Amazon ec2 availability zones @ONLINE, March 2012. http://aws.amazon.com/ec2/features, [Online; accessed 12-March-2012].

[33] Rodrigo N. Calheiros, Rajiv Ranjan, Anton Beloglazov, César A. F. De Rose, and Rajkumar Buyya. Cloudsim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. Paper, School of Computer Science and Engineering The University of New South Wales, Sydney, Australia and Department of Computer Science Pontifical Catholic University of Rio Grande do Sul Porto Alegre, Brazil, 2010.

[34] Sys::statistics::linux cpan module website, April 2012. http://search.cpan.org/ bloonix/Sys-Statistics-Linux-0.66/lib/Sys/Statistics/Linux.pm, , [Online; accessed 5-April-2012].

[35] Kvm documentation website @ONLINE, March 2012. https://help.ubuntu.com/community/KVM/Installation, [Online; accessed 7-March-2012].

[36] Grub documentation website @ONLINE, March 2012. https://help.ubuntu.com/community/Grub2#Reverting_to_GRUB_Legacy.

[37] L4 pistachio github website @ONLINE, February 2012.

[38] Pure64 website @ONLINE, May 2012. http://www.returninfinity.com/pure64.html,[Online accessed 17-april-2012].

[39] Baremetal os github project site @ONLINE, April 2012. https://github.com/ReturnInfinity/BareMetal-OS, [Online; accessed 12-April-2012].

[40] Minix3 website @ONLINE, February 2012. http://www.minix3.org/,[Online; accessed 1-March-2012].

[41] velosity website @ONLINE, May 2012. http://ghs.com/products,[Online; accessed 18-May-2012].

[42] Kolibri os @ONLINE, May 2012. http://kolibrios.org/en/,[Online; accessed 18-May-2012].

Appendices

Appendix A

L4 Hello World, hello.cc

```
1   #include <l4io.h>
2   #include <l4/ipc.h>
3
4   int main (void)
5   {
6     int i=0;
7       //sleeping for 60 seconds
8       //timeout = L4_TimePeriod(1140000000);
9       //L4_Sleep (timeout);
10      L4_Time_t timeout;
11      timeout = L4_TimePeriod(20000000);
12
13      printf("Starting infinite loop\n");
14      while(true){
15          printf ("Sleeping. Count %i \n",i++);
16          //sleeping for 10 seconds
17          L4_Sleep (timeout);
18      };
19
20      return 0;
21  }
```

Appendix B

L4 Hello World Makefile, Makefile.in

```
1  srcdir= @srcdir@
2  top_srcdir= @top_srcdir@
3  top_builddir= @top_builddir@
4
5  include $(top_srcdir)/Mk/l4.base.mk
6
7
8  PROGRAM= hello
9  PROGRAM_DEPS= $(top_builddir)/lib/l4/libl4.a \
10                 $(top_builddir)/lib/io/libio.a
11
12 SRCS= crt0-$(ARCH).S hello.cc
13
14 LIBS+= -ll4 -lio
15 LDFLAGS+= -Ttext=$(ROOTTASK_LINKBASE)
16
17 CFLAGS_powerpc+= -fno-builtin -msoft-float
18 CXXFLAGS_powerpc+= -fno-rtti
19
20 include $(top_srcdir)/Mk/l4.prog.mk
```

Appendix C

Deploy Multiple VMs, bootscript.pl

```
1  #! /usr/bin/perl
2
3  # Needed Packages
4  use Getopt::Std;
5  use strict "vars";
6  use warnings;
7  use Sys::Statistics::Linux;
8  use Data::Dumper;
9
10
11
12 # Letters followed by ":" are mandatory
13 my $opt_string = "i:t:n:";
14
15 getopts("$opt_string", \my %opt ) or usage() and exit 1;
16
17 # Print help message if -h is invoked
18 if( $opt{'h'} ){
19         usage();
20         exit 0; # Zero means everything is OK.
21     }
22
23 my $NUMID;
24 my $TYPE;
25 my $IMAGE;
26
27
28 my ($mem_total, $mem_used, $mem_cached, $mem_free, $cpu_usr, $cpu_total, $cpu_idle,
       $cpu_sys);
29
30 # Handle user input
31 # $NUMID = 1 if $opt{'d'};
32 $NUMID = $opt{'n'};
33 $TYPE = $opt{'t'};
34 $IMAGE = $opt{'i'};
35 die "Image name is mandatory." unless $IMAGE;
36 die "Number of machiness to create is mandatory" unless $NUMID;
37 die "Type is mandatory (bm or l4)" unless $TYPE;
```

```
38
39  # my $uuid;
40  my $uuid;
41
42  my $idfile = "uuid.txt";
43  open FILE, ">$idfile" or die $!;
44
45  for(my $i = 0; $i < $NUMID; $i++){
46      $uuid = `uuidgen >> uuid.txt`;
47      # $uuid = `uuidgen`;
48  }
49  close FILE;
50
51  my $command;
52
53  open FILE, "$idfile" or die $!;
54
55  my $vm_id = 1;
56
57  # Number of data samples to collect
58  my $samples = 4000;
59
60  print "Starting statistics script. Collecting $samples samples...\n";
61  system("/usr/bin/perl ./perf.pl -t $TYPE -o $TYPE\_baseline_perf.log -s $samples -d 1
        &");
62  print "Waiting 10 seconds before booting first VM. Allows system to "breathe"...\n";
63  sleep(10);
64
65  foreach my $id (<FILE>){
66
67      chomp($id);
68
69      my $vm_name = "$TYPE\_$vm_id";
70      my $vm_process = "$TYPE\_process_$vm_id";
71      print "ID: $id Name: $vm_name Process: $vm_process\n";
72
73      if($TYPE eq "l4"){
74          print "Executing L4 Pistachio instance..\n";
75          system( "qemu-system-x86_64 -m 5 -cpu pentium -fda $IMAGE -uuid $id
                -name $vm_name,process=$vm_process -net none -M pc -smp
                1,sockets=1,cores=1,threads=1,maxcpus=1 -nographic &" );
76          print "Sleeping 10 seconds\n";
77          sleep (10);
78          print "Done sleeping..\n\n";
79      }
80
81      if($TYPE eq "bm"){
82          print "Executing BareMetal OS instance..\n";
83          system( "qemu-system-x86_64 -m 16 -hda $IMAGE -uuid $id -name
                $vm_name,process=$vm_process -net none -M pc -smp
                1,sockets=1,cores=1,threads=1,maxcpus=1 -nographic &");
84          print "Sleeping 10 seconds\n";
85          sleep (10);
86          print "Done sleeping..\n\n";
87      }
88
89      if($TYPE eq "tcl"){
```

```
90              print "Executing Tiny Core Linux instance..\n";
91              system( "qemu-system-x86_64 -m 50 -hda $IMAGE -uuid $id -name
                    $vm_name,process=$vm_process -net none -M pc -smp
                    1,sockets=1,cores=1,threads=1,maxcpus=1 -nographic &");
92              print "Sleeping 10 seconds\n";
93              sleep (10);
94              print "Done sleeping..\n\n";
95          }
96
97          $vm_id++;
98
99      }
100     close FILE;
101
102     sub getstats{
103
104         my $type = $_[0];
105         ## print "INPUT IS: $type\n";
106
107         my $lxs = Sys::Statistics::Linux->new(
108             cpustats => 1,
109             pgswstats => 1,
110             memstats=> 1,
111             processes=> 1
112         );
113
114         sleep(1);
115         my $stat = $lxs->get;
116
117         ## print Dumper($stat);
118
119         ## Read CPU stats
120         my $cpu = $stat->cpustats->{cpu};
121         $cpu_usr = $cpu->{user};
122         $cpu_sys = $cpu->{system};
123         $cpu_idle = $cpu->{idle};
124         $cpu_total = $cpu->{total};
125
126         ## Read Memory stats
127         ### Memory
128         $mem_total = $stat->memstats->{memtotal};
129         $mem_used = $stat->memstats->{memused};
130         $mem_free = $stat->memstats->{memfree};
131         $mem_cached = $stat->memstats->{cached};
132
133         my $time = $lxs->gettime;
134
135     }
136
137
138     exit 0;
```

Appendix D

L4 Compiler tool, makescript.pl

```
1  #!/usr/bin/perl
2  use File::Find;
3
4  # Paths must be kept absolute
5  my $srcdir = "/project/L4/l4ka-pistachio";
6
7  my $kerneldir = "$srcdir/x86-kernel";
8  my $builddir = "$srcdir/x86-user-build";
9  my $installdir = "$srcdir/x86-user-install";
10 my $libexecdir = "$srcdir/x86-user-install/libexec/l4";
11 my $fdsourcedir = "$srcdir/fdsource";
12 my $mountdir = "/mnt/fda";
13 my $scriptdir = "/project/L4/scripts";
14 # Image and kernel name
15 my $imagedir = "/project/L4/images";
16 my $default_imagedir = "/project/L4/images/default";
17 my $default_image = "fdimage.img";
18 my $new_image = "new_fdimage.img";
19 my $kernel = "x86-kernel";
20 my $cmd;
21 my $kerneldir = "$srcdir/x86-kernel";
22 my $kickbase = "0x148030";
23 my $sigbase = "0x20000";
24 my $rootbase = "0xEA60"; ## 0xEA60 == 50 000, 0x7A120 == 500 000
25 my $programdir = "$srcdir/user/apps";
26 ##### Contents of menu.lst for L4 must be:
27 # root (fd0)
28 # default=0
29 # timeout=3
30 #
31 # title L4Ka::Pistachio
32 # kernel /kickstart
33 # module /x86-kernel
34 # module /sigma0
35 # module /hello
36 ####
37
38
39 ## Compile L4
40 print "Entering directory ===> $builddir\n";
41 chdir("$builddir");
```

```
42 system("make clean");
43 system("../user/configure --without-comport --with-kickstart-linkbase=$kickbase
       --with-s0-linkbase=$sigbase --with-roottask-linkbase=$rootbase
       --prefix=$installdir --with-kerneldir=$kerneldir");
44 system("make");
45 system("make install");
46
47 # Checking if all files were compiled sucessfully
48 print "===> Checking if all files were compiled...\n";
49 my @files = ("kickstart", "sigma0", "hello", "$kernel", "math", "cpua", "cpub", "cpuc");
50
51 foreach my $file (@files){
52     print "Checking file: \"$libexecdir/$file\"\n";
53
54     if( "$file" ne "$kernel"){
55         unless ( -e "$libexecdir/$file"){
56             die "Error: File \"$libexecdir/$file\" was not found... Re-compile and try
                   again...\n";
57         }
58     }
59
60     if("$file" eq "$kernel"){
61         unless ( -e "$kerneldir/$kernel"){
62             die "Error: File: \"$kerneldir/$kernel\" was not found... Re-compile and try
                   again...\n";
63         }
64     }
65 }
66
67
68 #### Done compiling L4
69 print "\n===> Checking if mount directory \"$mountdir\" exists...\n";
70
71 if (-d "$mountdir"){
72     print "\tMountdir \"$mountdir\" exists...\n";
73     system("umount /mnt/fda");
74     print "\tSuccessfully unmounted: \"$mountdir\"\n";
75     system("rm -r $mountdir");
76     print "\tSuccessfully removed: \"$mountdir\"\n";
77 }else{
78     print "\t\"$mountdir\" does not exist...\n";
79     }
80 print "\tDone...\n";
81
82 print "\n===> Creating \"$mountdir\"...\n";
83 unless ( -d "$mountdir" ){
84     system("mkdir $mountdir");
85     print "\tSetting permissions to \"777\" on \"$mountdir\"...\n";
86     system("chmod 777 $mountdir");
87 }
88
89 if ( -d "$mountdir" ){
90     print "\tMountdir: \"$mountdir\" was successfully created ...\n";
91 }
92 print "\tDone...\n";
93
94 # Set up $fdsourcedir
```

```
95  #
96  # If fdsourcedir exists, remove it (cleanup)
97  print "\n===> Checking if \"$fdsourcedir\" exists and creating if necessary...\n";
98  if ( -d "$fdsourcedir" ){
99      print "\tRemoving: $fdsourcedir\n";
100     system("rm -r $fdsourcedir");
101 }
102
103 # Creates directory structure for all files
104 unless ( -d "$fdsourcedir" ){
105     print "\tCreating: $fdsourcedir/boot/grub/\n";
106     system("mkdir -p $fdsourcedir/boot/grub");
107 }
108 print "\tDone...\n";
109 sleep(1);
110 # Copy default image to work with
111 print "\n===> Copying default image\n";
112 system("cp $default_imagedir/$default_image $imagedir/$new_image");
113 print "\t\"$default_imagedir/$default_image\" \"$imagedir/$new_image\"\n";
114 print "\tDone...\n";
115 sleep(1);
116
117 # Set up loop device to use fdimage.img
118 print "\n===> Setting up loopdevice to use $new_image...\n";
119 system("/sbin/losetup /dev/loop0 $imagedir/$new_image");
120 print "\tDone...\n";
121 sleep(1);
122 # Mount loop device on /mnt/fda
123 print "\n===> Mounting loopdevice on $mountdir...\n";
124 system("mount /dev/loop0 -o loop $mountdir");
125 print "\tDone...\n";
126 #### Copy binaries to $fdsourcedir
127 #
128 sleep(1);
129 print "\n===> Copying binaries...\n";
130 system("cp $libexecdir/kickstart $fdsourcedir/");
131 print "\t\"kickstart\" \"$fdsourcedir\"\n";
132
133 system("cp $libexecdir/sigma0 $fdsourcedir/");
134 print "\t\"sigma0\" \"$fdsourcedir\"\n";
135
136 system("cp $libexecdir/hello $fdsourcedir/");
137 print "\t\"hello\" \"$fdsourcedir\"\n";
138
139 system("cp $libexecdir/cpua $fdsourcedir/");
140 print "\t\"cpua\" \"$fdsourcedir\"\n";
141
142 system("cp $libexecdir/cpub $fdsourcedir/");
143 print "\t\"cpub\" \"$fdsourcedir\"\n";
144
145 system("cp $libexecdir/cpuc $fdsourcedir/");
146 print "\t\"cpuc\" \"$fdsourcedir\"\n";
147
148 system("cp $kerneldir/$kernel $fdsourcedir/");
149 print "\t\"$kernel\" \"$fdsourcedir\"\n";
150
151 system("cp $libexecdir/cpu_iterative_a $fdsourcedir/");
```

```
152 print "\t\"cpu_iterative_a\" \"$fdsourcedir\"\n";
153
154 system("cp $libexecdir/cpu_iterative_b $fdsourcedir/");
155 print "\t\"cpu_iterative_b\" \"$fdsourcedir\"\n";
156
157 system("cp $libexecdir/cpu_iterative_c $fdsourcedir/");
158 print "\t\"cpu_iterative_c\" \"$fdsourcedir\"\n";
159
160 system("cp $libexecdir/sprint_iter_200 $fdsourcedir/");
161 print "\t\"sprint_iter_200\" \"$fdsourcedir\"\n";
162
163 system("cp $libexecdir/sprint_iter_20 $fdsourcedir/");
164 print "\t\"sprint_iter_20\" \"$fdsourcedir\"\n";
165
166 system("cp $libexecdir/math $fdsourcedir/");
167 print "\t\"math\" \"$fdsourcedir\"\n";
168
169 ##system("cp $libexecdir/ram_a $fdsourcedir/");
170 ##print "\t\"ram_a\" \"$fdsourcedir\"\n";
171
172
173 print "\tDone...\n";
174 ###
175
176 #### Installing Grub legacy files
177 #
178 print "\n===> Installing Grub files...\n";
179 system("cp $scriptdir/menu.lst $fdsourcedir/boot/grub");
180 print "\t\"$scriptdir/menu.lst\" \"$fdsourcedir/boot/grub/\"\n";
181
182 system("cp /boot/grub/stage1 $fdsourcedir/boot/grub");
183 print "\t\"/boot/grub/stage1\" \"$fdsourcedir/boot/grub/\"\n";
184
185 system("cp /boot/grub/stage2 $fdsourcedir/boot/grub");
186 print "\t\"/boot/grub/stage2\" \"$fdsourcedir/boot/grub/\"\n";
187 print "\tDone...\n";
188 ###
189
190 #### Copy all files to mountdir /mnt/fda
191 print "\n===> Copy all files to \"$mountdir\"...\n";
192 system("cp -aR $fdsourcedir/* $mountdir");
193 print "Done...\n";
194 ####
195 sleep(2);
196 # Unmount /mnt/fda
197 print "\n===> Unmounting \"$mountdir\"...\n";
198 system("umount $mountdir");
199 print "\tDone...\n";
200 sleep(1);
201 print "\n===> Setting permissions to \"777\" on \"$new_image\"\n";
202 system("chmod 777 $imagedir/$new_image");
203 print "\tDone...\n";
204
205 print "\n===> Setting up grub...\n";
206 system("$scriptdir/grub.sh");
207 print "\tDone...\n";
208
```

```
209 sleep(3);
210
211 # Unmount loop device
212 print "\n===> Unmounting loop device...\n";
213 system("/sbin/losetup -d /dev/loop0");
214 print "\tDone...\n";
215 # Cleaning up
216 print "\n===> Removing mountdir...\n";
217 system("rm -r $mountdir");
218 print "\tDone...\n";
219
220 print "\n===>Printing contents of \"$fdsourcedir\":\n";
221 $" = "\n";
222 my @FDToRead = "$fdsourcedir";
223
224 if (!@FDToRead)
225 {@FDToRead = ".";}
226
227 my @allFiles = readdirR (@FDToRead);
228 @allFiles = sort @allFiles;
229
230 foreach my $file (@allFiles){
231     print "\t$file\n";
232     }
233
234 ##print "\t@allFiles\n";
235 print "\tDone...\n";
236 ############### SUBROUTINES #####
237 # Accepts one argument: the full path to a directory.
238 # Returns: A list of files that end in '.html' and have been
239 # modified in less than one day.
240
241 sub readdirR{
242     my @FDList = @_;
243
244     my $first = shift @FDList;
245
246     if (!$first){
247         return ();
248     }
249     elsif (-f $first){
250         return ($first, readdirR (@FDList));
251     }
252     elsif (-d $first){
253         opendir DIR, $first || warn "Cannot open directory $first: $!";
254         my @files = readdir DIR ;
255         closedir DIR;
256         @files = grep {$_ !~ /^[.]{1,2}$/} @files;
257         @files = map {"$first/$_"} @files;
258         return ($first, readdirR (@files), readdirR(@FDList));
259     }
260 }
261
262 ####main program###########
```

Appendix E

BareMetal native Hello World, bare_hello.c

```
1  // Hello world application for BareMetal using native headers
2  // found in libBareMetal.h
3
4  #include "libBareMetal.h"
5
6  int main (void)
7  {
8      b_print_string("I'm BareMetal\n");
9      // Sleep 1 second (8 = 1 second, 16 = 2 seconds)
10     b_delay(8)
11     b_print_string("Goodbye\n")
12     return 0;
13 }
```

Appendix F

BareMetal Hello World Newlib C library, newlib_hello.c

```
1  // Hello world application for BareMetal using Newlib C library
2  // Newlib does not include sleep or nanosleep functions
3  // so bareMetal headers will be used as well
4  #include <stdio.h>
5  #include "libBareMetal.h"
6
7  int main (void)
8  {
9      printf("I'm Newlib\n");
10     b_delay(8)
11     printf("Goodbye\n")
12     return 0;
13 }
```

Appendix G

TCL Hello World, hello.cc

```
1  #include <stdio.h>
2  #include <unistd.h>
3
4  int main (void)
5  {
6          printf("Hello World\n");
7          sleep(5);
8          printf("Goodbye\n");
9          return 0;
10 }
```

Appendix H

TCL Add file to filesystem, tcl_add_files.pl

```
1  #! /usr/bin/perl
2
3  use Getopt::Std;
4  use strict "vars";
5
6
7  #### Input handling #########
8  #
9  # Letters followed by ":" are mandatory
10 my $opt_string = "hp:i:";
11
12 getopts("$opt_string", \my %opt ) or usage() and exit 1;
13
14 my $IMAGE;
15 my $PROGRAM;
16
17 # Print help message if -h is invoked
18 if( $opt{'h'} ){
19       usage();
20       exit 0; # Zero means everything is OK.
21 }
22
23 $IMAGE = $opt{'i'};
24 $PROGRAM = $opt{'p'};
25
26 die "Image name is mandatory." unless $IMAGE;
27 die "Program name to include on TCL is mandatory" unless $PROGRAM;
28
29 ##############################
30
31 # Print path of current directory
32 my $tcldir = `pwd`;
33 chomp($tcldir);
34
35 my $cmd;
36 my $tempdir = "$tcldir/temp";
37 my $mountdir = "/mnt/tcl";
38 my $programdir = "$tcldir";
```

```
39 my $tcl_boot_file = "bootlocal.sh";
40 #### Cleaning up before mounting TCL
41
42 # Unmount mountdir
43 system("umount $mountdir");
44 print "Unmounted: $mountdir\n";
45 system("kpartx -d $IMAGE");
46
47 if ( -e $mountdir){
48     print "Mountdir exists.. Deleting..\n";
49     system("rm -r $mountdir");
50 }
51
52 # Create mount directory if it doesn't exist
53 unless ( -d $mountdir ){
54     print "Creating directory: $mountdir\n";
55     system("mkdir $mountdir");
56 }
57
58 # If tempdir exists, delete it.
59 if ( -e "$tempdir"){
60     print "$tempdir exists. Deleting\n";
61     system("rm -r $tempdir");
62 }
63
64 # Create directories
65 unless( -d $tempdir){
66     print "Creating: $tempdir\n";
67     system("mkdir -p $tempdir/extract");
68 }
69
70 # Mount image and read name of mounted loop device
71 my @cmd = `kpartx -av $IMAGE`;
72 my $match;
73
74 foreach my $line (@cmd){
75     if($line =~ m/\w+\s+\w+\s+(.+)\s+\(/){
76         $match = $1;
77         chomp($match);
78     }
79 }
80
81 sleep(2);
82
83 # Mounting mount directory
84 system("mount /dev/mapper/$match $mountdir");
85 print "Mounted $match on $mountdir\n";
86 sleep(2);
87 # Copying TCL core (core.gz) from tiny core linux mount directory
88 system("cp $mountdir/tce/boot/core.gz $tempdir");
89 print "Copied core.gz to $tempdir\n";
90
91 # Extracting core and mounting filesystem in <tempdir>/extract
92 chdir("$tempdir/extract") or die $!;
93
94 system("zcat ../core.gz | sudo cpio -i -H newc -d");
95 sleep(3);
```

```
96  # Copying program to /etc/init.d
97  chdir("$programdir");
98  system("cp $PROGRAM $tempdir/extract/etc/init.d");
99
100 chdir("$tempdir/extract/opt/");
101
102 # Adding startup command for program to make TCL automatically boot it
103 open FILE, ">$tcl_boot_file" or die $!;
104 print FILE "#!/bin/sh\n";
105 print FILE "# Add startup commands in this file\n";
106 print FILE "/etc/init.d/$PROGRAM\n";
107 close FILE;
108
109 # Copying old core to backup file
110 system("mv $tempdir/core.gz $tempdir/1core.gz");
111 chdir("$tempdir/extract");
112
113 # Re-packing TCL filesystem which now include the program
114 system("find | sudo cpio -o -H newc | gzip -2 > ../core.gz");
115 chdir("$tempdir");
116
117 # Copy the new TCL core to the mountdir
118 system("cp core.gz $mountdir/tce/boot/");
119 print "Done copying core.gz to $mountdir/tce/boot/\n";
120 sleep(2);
121 # Unmount mountdir then the image
122 system("umount $mountdir");
123 print "Unmounted: $mountdir\n";
124 sleep(2);
125 chdir("..");
126 system("kpartx -d $IMAGE");
127
128 #### Subroutine
129
130 sub usage{
131
132     print "Usage:\n";
133     print "-h Usage information. (Optional)\n";
134     print "-i Image name. (Required)\n";
135     print "-p Program name. (Required)\n";
136
137
138 }
```

Appendix I

Statistics Script, perf.pl

```
1  #! /usr/bin/perl
2  ## PERFSCRIPT
3  use strict;
4  use warnings;
5  use Sys::Statistics::Linux;
6  use Data::Dumper;
7  use Time::HiRes;
8  use Getopt::Std;
9
10
11 ##### Global Variables
12 #
13 my $TYPE;
14 my $FILE;
15 my $SAMPLES = 30;
16 my $DEBUG = 0;
17 my $VERBOSE = 0;
18 my $DELAY = 1;
19
20 ##### Initializing variables
21 #
22 # General
23 my ($vm_num,$time_get,$time_start,$time);
24 ## CPU
25 my ($cpu_usr,$cpu_sys,$cpu_idle,$cpu_total);
26 # LoadAvg
27 my ($cpuavg_one,$cpuavg_five,$cpuavg_fifteen);
28 ## RAM
29 my ($mem_total,$mem_used,$mem_free,$mem_cached);
30 # Swap
31 my ($swap_total,$swap_used,$swap_free,$swap_cached);
32 # Pages
33 my ($page_fault,$page_majfault, $page_pgin,$page_pgout,$page_pswpin,$page_pswpout);
34 ##### Handle User Input
35 #
36 my $opt_string = 'hd:Dvo:s:t:';
37 getopts( "$opt_string", \my %opt ) or usage () and exit 1;
38
39 if ( $opt{'h'} ){
40     usage();
41     exit 0;
```

```
42 }
43
44
45 $VERBOSE = 1 if $opt{'v'};
46 debug("Verbose: $VERBOSE");
47 $DEBUG = 1 if $opt{'D'};
48 debug("Debug: $DEBUG");
49 $DELAY = $opt{'d'} if $opt{'d'};
50 debug("Delay: $DELAY");
51 $SAMPLES = $opt{'s'} if $opt{'s'};
52 debug("Samples: $SAMPLES");
53 $FILE = $opt{'o'} if $opt{'o'};
54 debug("File: $FILE");
55 $TYPE = $opt{'t'};
56 debug("VM Type: $TYPE");
57
58 die "Error: Missing parameter: '-t'. See -h for usage info." unless $TYPE;
59 die "Error: Missing parameter: '-o'. See -h for usage info." unless $FILE;
60
61
62 ##### File handling
63 #
64 my $file = "$FILE";
65 open LOG, ">$file" or die $!;
66
67 # Making filehandle HOT to disable buffering
68 ##{ my $ofh = select OUTPUT;
69 # $| = 1;
70 # select OUTPUT;
71 ##}
72 print LOG "Samples,VM_num,CPU_usr,CPU_sys,CPU_total,CPU_idle,CPUavg_one,
73 CPUavg_five,CPUavg_fifteen,MEM_total,MEM_used,MEM_free,MEM_cached,
74 SWAP_total,SWAP_used,SWAP_free,SWAP_cached,PAGE_fault,PAGE_majfault,
75 PAGE_pgin,PAGE_pgout,PAGE_pswpin,PAGE_pswpout\n";
76
77 ##### Gather statistics
78
79 for (my $sample = 0; $sample < $SAMPLES; $sample++){
80     &getcpu($TYPE, $DELAY);
81     debug("$sample,$vm_num,$cpu_usr,$cpu_sys,$cpu_total,$cpu_idle,
82 $cpuavg_one,$cpuavg_five,$cpuavg_fifteen,$mem_total,$mem_used,
83 $mem_free,$mem_cached,$swap_total, $swap_used,$swap_free,$swap_cached,
84 $page_fault,$page_majfault,$page_pgin,$page_pgout,$page_pswpin,
85 $page_pswpout");
86     print LOG "$sample,$vm_num,$cpu_usr,$cpu_sys,$cpu_total,$cpu_idle,
87 $cpuavg_one,$cpuavg_five,$cpuavg_fifteen,$mem_total,$mem_used,
88 $mem_free,$mem_cached,$swap_total,$swap_used,$swap_free,$swap_cached,
89 $page_fault,$page_majfault,$page_pgin,$page_pgout,$page_pswpin,
90 $page_pswpout\n";
91 }
92
93 close LOG;
94
95 ############ SUBROUTINES ###########################################
96 ####################################################################
97
98 sub getcpu{
```

```
99
100     my $type = $_[0];
101     my $delay = $_[1];
102     chomp($type, $delay);
103
104     ##print "INPUT IS: $type\n";
105
106     my $lxs = Sys::Statistics::Linux->new(
107         cpustats => 1,
108         pgswstats => 1,
109         memstats => 1,
110         processes => 1,
111         loadavg => 1
112     );
113
114     sleep($delay);
115
116  # $time_start = Time::HiRes::time();
117    # $time = sprintf ("%.6f", $time_start);
118
119
120     ##$time = $time_get;
121 # print $time;
122 ##my $time;
123 ## print "$time_get\n";
124     ##return($time_get);
125
126     my $stat = $lxs->get;
127
128    # print Dumper($stat);
129 ## exit 0;
130 #####
131     #
132     ### CPU stats
133 # print "CPU\n";
134     my $cpu = $stat->cpustats->{cpu};
135     $cpu_usr = $cpu->{user};
136     $cpu_sys = $cpu->{system};
137     $cpu_idle = $cpu->{idle};
138     $cpu_total = $cpu->{total};
139     debug("CPU_usr: $cpu_usr");
140     debug("CPU_sys: $cpu_sys");
141     debug("CPU_idle: $cpu_idle");
142     debug("CPU_total: $cpu_total");
143
144 ##### Load Average
145     #
146 # print "AVG\n";
147     $cpuavg_one = $stat->loadavg->{avg_1};
148     $cpuavg_five = $stat->loadavg->{avg_5};
149     $cpuavg_fifteen = $stat->loadavg->{avg_15};
150     debug("CPUavg_one: $cpuavg_one");
151     debug("CPUavg_five: $cpuavg_five");
152     debug("CPUavg_fifteen: $cpuavg_fifteen");
153 ##print "$cpuavg_one and $cpuavg_five and $cpuavg_fifteen\n";
154 ##### Read Memory stats
155     #
```

```
156     ### Swap
157 # print "SWAP\n";
158     $swap_total= $stat->memstats->{swaptotal};
159     $swap_used= $stat->memstats->{swapused};
160     $swap_free= $stat->memstats->{swapfree};
161     $swap_cached= $stat->memstats->{swapcached};
162     debug("SWP_total: $swap_total");
163     debug("SWP_used: $swap_used");
164     debug("SWP_free: $swap_free");
165     debug("SWP_cached: $swap_cached");
166 #####
167     #
168     ### Memory
169 # print "RAM\n";
170     $mem_total = $stat->memstats->{memtotal};
171     $mem_used = $stat->memstats->{memused};
172     $mem_free = $stat->memstats->{memfree};
173     $mem_cached = $stat->memstats->{cached};
174     debug("RAM_total: $mem_total");
175     debug("RAM_used: $mem_used");
176     debug("RAM_free: $mem_free");
177     debug("RAM_cached: $mem_cached");
178 #####
179     #
180     ### Paging
181     # Avail. from kernel 2.6 and above
182     $page_fault = $stat->pgswstats->{pgfault};
183     debug("Number of page faults pr. sec: $page_fault");
184     $page_majfault = $stat->pgswstats->{pgmajfault};
185     debug("Number of major page faults pr. sec: $page_majfault");
186     #
187     $page_pgin = $stat->pgswstats->{pgpgin};
188     debug("Number of pages paged in from disk pr.sec: $page_pgin");
189     $page_pgout = $stat->pgswstats->{pgpgout};
190     debug("Number of pages paged out to disk pr. sec: $page_pgout");
191     $page_pswpin = $stat->pgswstats->{pswpin};
192     debug("Number of pages swapped in from disk pr. sec: $page_pswpin");
193     $page_pswpout = $stat->pgswstats->{pswpout};
194     debug("Number of pages swapped out to disk pr. sec: $page_pswpout");
195
196
197
198 ## Processes
199     # Search for this process name
200 # print "PROCS\n";
201     $type = "$type\_process";
202
203     my %hash = $stat->search({
204                                 processes => {
205                                     cmd => qr/($type)/
206                                 }
207                             });
208
209     $vm_num = 0;
210     foreach my $cmd ( keys %hash){
211
212         while (my ($key, $value) = each %{ $hash{$cmd}} ) {
```

```
213                 $vm_num++;
214                 ##print "Key: $key Value: $value\n";
215             }
216
217         }
218         my $time = $lxs->gettime;
219     }
220
221     sub usage{
222
223         print "Usage:\n";
224         print "-h Usage information. (Optional)\n";
225         print "-v Output is verbose. (Optional)\n";
226         print "-D Enable debugging. (Optional)\n";
227         print "-t <l4|bm> VM Type. (Required)\n";
228         print "-s <num> Number of samples. Default: 30\n";
229         print "-d <seconds> Sample delay. Default: 1\n";
230
231     }
232
233     sub verbose{
234         print "VERBOSE: $_[0]\n" if ( $VERBOSE or $DEBUG );
235     }
236
237     sub debug{
238         print "DEBUG: $_[0]\n" if ( $DEBUG );
239
240     }
```

Appendix J

L4 GRUB Installation Script, grub.sh

```
1 #! /bin/bash
2
3 cat <<EOF | /usr/sbin/grub --batch --device-map=/dev/null
4 device (fd0) /dev/loop0
5 root (fd0)
6 setup (fd0)
7 quit
8 EOF
```

Appendix K

BareMetal Application Build script, build.sh

```
1  #!/bin/bash
2
3  # Backup lines if something goes haywire
4  #gcc --verbose -I/newlib/newlib-1.20.0/newlib/libc/include/ -c cpua.c -o cpua.o
5  #gcc --verbose -L . -l libBareMetal.h -c libBareMetal.c -o libBareMetal.o
6  #ld --verbose -T app.ld -o cpua.app cpua.o libBareMetal.o
7
8  # rm *.o
9
10 # CPUA - Recursive fibonacci, profile A
11
12 gcc --verbose -c cpua.c -o cpua.o
13 gcc --verbose -L . -l libBareMetal.h -c libBareMetal.c -o libBareMetal.o
14 ld --verbose -T app.ld -o cpua.app cpua.o libBareMetal.o
15
16 rm *.o
17
18 # CPUB - Recursive fibonacci, profile B
19 gcc --verbose -c cpub.c -o cpub.o
20 gcc --verbose -L . -l libBareMetal.h -c libBareMetal.c -o libBareMetal.o
21 ld --verbose -T app.ld -o cpub.app cpub.o libBareMetal.o
22
23 rm *.o
24
25 # CPUC - Recursive fibonacci, profile C
26 gcc --verbose -c cpuc.c -o cpuc.o
27 gcc --verbose -L . -l libBareMetal.h -c libBareMetal.c -o libBareMetal.o
28 ld --verbose -T app.ld -o cpuc.app cpuc.o libBareMetal.o
29
30 rm *.o
31
32 # Fib iterative A - Iterative fibonacci, profile A
33 gcc --verbose -c iterative_a.c -o iterative_a.o
34 gcc --verbose -L . -l libBareMetal.h -c libBareMetal.c -o libBareMetal.o
35 ld --verbose -T app.ld -o iterative_a.app iterative_a.o libBareMetal.o
36
37 rm *.o
38
```

```
39 # Fib iterative B - Iterative fibonacci, profile B
40 gcc --verbose -c iterative_b.c -o iterative_b.o
41 gcc --verbose -L . -l libBareMetal.h -c libBareMetal.c -o libBareMetal.o
42 ld --verbose -T app.ld -o iterative_b.app iterative_b.o libBareMetal.o
43
44 rm *.o
45
46 # Fib iterative C - Iterative fibonacci, profile C
47 gcc --verbose -c iterative_c.c -o iterative_c.o
48 gcc --verbose -L . -l libBareMetal.h -c libBareMetal.c -o libBareMetal.o
49 ld --verbose -T app.ld -o iterative_c.app iterative_c.o libBareMetal.o
50
51 rm *.o
52
53 # Sprint Iter 200 - Iterative fibonacci, 1 VM calculating fib(n) 200 times
54 # with n = 2147483646
55 gcc --verbose -c sprint_iter_200.c -o sprint_iter_200.o
56 gcc --verbose -L . -l libBareMetal.h -c libBareMetal.c -o libBareMetal.o
57 ld --verbose -T app.ld -o sprint_iter_200.app sprint_iter_200.o libBareMetal.o
58
59 rm *.o
60
61 # Sprint Iter 20 - Iterative fibonacci, 10 VMs calculating fib(n) 20 times
62 # simultaneously with n = 2147483646
63 gcc --verbose -c sprint_iter_20.c -o sprint_iter_20.o
64 gcc --verbose -L . -l libBareMetal.h -c libBareMetal.c -o libBareMetal.o
65 ld --verbose -T app.ld -o sprint_iter_20.app sprint_iter_20.o libBareMetal.o
66
67 rm *.o
```

Appendix L

BareMetal OS, Mounting images and transferring files, mount.sh

```
#!/bin/sh

# Cleaning image
kpartx -av BareMetal_new.img
mount /dev/mapper/loop0p1 /mnt/baremetal
echo "Listing contents DEFAULT IMAGE:"
ls -la /mnt/baremetal
umount /mnt/baremetal
kpartx -d BareMetal_new.img
echo "\n\n\n"
#qemu-system-x86_64 -m 16 -hda BareMetal_cpua.img -net none -M pc
sleep 1


# Creating Default Images
cp BareMetal_new.img BareMetal_cpua.img
cp BareMetal_new.img BareMetal_cpub.img
cp BareMetal_new.img BareMetal_cpuc.img
cp BareMetal_new.img BareMetal_hello.img
cp BareMetal_new.img BareMetal_iterative_a.img
cp BareMetal_new.img BareMetal_iterative_b.img
cp BareMetal_new.img BareMetal_iterative_c.img
cp BareMetal_new.img BareMetal_sprint_iter_200.img
cp BareMetal_new.img BareMetal_sprint_iter_20.img



kpartx -av BareMetal_cpua.img
mount /dev/mapper/loop0p1 /mnt/baremetal
cp /sdc/project/BM/programs/CPU/cpua.app /mnt/baremetal/startup.app
echo "Listing contents CPUA:"
ls -la /mnt/baremetal
umount /mnt/baremetal
kpartx -d BareMetal_cpua.img
echo "\n\n\n"
#qemu-system-x86_64 -m 16 -hda BareMetal_cpua.img -net none -M pc
sleep 1
```

```
kpartx -av BareMetal_cpub.img
mount /dev/mapper/loop0p1 /mnt/baremetal
cp /sdc/project/BM/programs/CPU/cpub.app /mnt/baremetal/startup.app
echo "Listing contents CPUB:"
ls -la /mnt/baremetal
umount /mnt/baremetal
kpartx -d BareMetal_cpub.img
#qemu-system-x86_64 -m 16 -hda BareMetal_new.img -net none -M pc
echo "\n\n\n"
sleep 1

kpartx -av BareMetal_cpuc.img
mount /dev/mapper/loop0p1 /mnt/baremetal
cp /sdc/project/BM/programs/CPU/cpuc.app /mnt/baremetal/startup.app
echo "Listing contents CPUC:"
ls -la /mnt/baremetal
umount /mnt/baremetal
kpartx -d BareMetal_cpuc.img
#qemu-system-x86_64 -m 16 -hda BareMetal_new.img -net none -M pc
echo "\n\n\n"
sleep 1

kpartx -av BareMetal_hello.img
mount /dev/mapper/loop0p1 /mnt/baremetal
cp /sdc/project/BM/programs/CPU/hello.app /mnt/baremetal/startup.app
echo "Listing contents HELLO:"
ls -la /mnt/baremetal
umount /mnt/baremetal
kpartx -d BareMetal_hello.img
#qemu-system-x86_64 -m 16 -hda BareMetal_new.img -net none -M pc
echo "\n\n\n"
sleep 1

kpartx -av BareMetal_iterative_a.img
mount /dev/mapper/loop0p1 /mnt/baremetal
cp /sdc/project/BM/programs/CPU/iterative_a.app /mnt/baremetal/startup.app
echo "Listing contents ITERATIVE A:"
ls -la /mnt/baremetal
umount /mnt/baremetal
kpartx -d BareMetal_iterative_a.img
#qemu-system-x86_64 -m 16 -hda BareMetal_new.img -net none -M pc
echo "\n\n\n"
sleep 1

kpartx -av BareMetal_iterative_b.img
mount /dev/mapper/loop0p1 /mnt/baremetal
cp /sdc/project/BM/programs/CPU/iterative_b.app /mnt/baremetal/startup.app
echo "Listing contents ITERATIVE B:"
ls -la /mnt/baremetal
umount /mnt/baremetal
kpartx -d BareMetal_iterative_b.img
#qemu-system-x86_64 -m 16 -hda BareMetal_new.img -net none -M pc
echo "\n\n\n"
sleep 1

kpartx -av BareMetal_iterative_c.img
mount /dev/mapper/loop0p1 /mnt/baremetal
```

```
    cp /sdc/project/BM/programs/CPU/iterative_c.app /mnt/baremetal/startup.app
 97 echo "Listing contents ITERATIVE C:"
    ls -la /mnt/baremetal
 99 umount /mnt/baremetal
    kpartx -d BareMetal_iterative_c.img
101 #qemu-system-x86_64 -m 16 -hda BareMetal_new.img -net none -M pc
    echo "\n\n\n"
103 sleep 1

105 kpartx -av BareMetal_sprint_iter_200.img
    mount /dev/mapper/loop0p1 /mnt/baremetal
107 cp /sdc/project/BM/programs/CPU/sprint_iter_200.app /mnt/baremetal/startup.app
    echo "Listing contents SPRINT 200:"
109 ls -la /mnt/baremetal
    umount /mnt/baremetal
111 kpartx -d BareMetal_sprint_iter_200.img
    #qemu-system-x86_64 -m 16 -hda BareMetal_new.img -net none -M pc
113 echo "\n\n\n"
    sleep 1
115
    kpartx -av BareMetal_sprint_iter_20.img
117 mount /dev/mapper/loop0p1 /mnt/baremetal
    cp /sdc/project/BM/programs/CPU/sprint_iter_20.app /mnt/baremetal/startup.app
119 echo "Listing contents SPRINT 20:"
    ls -la /mnt/baremetal
121 umount /mnt/baremetal
    kpartx -d BareMetal_sprint_iter_20.img
123 #qemu-system-x86_64 -m 16 -hda BareMetal_new.img -net none -M pc
    echo "\n\n\n"
125 sleep 1
```

Appendix M

CPU Profile Tests - Hardware Lab

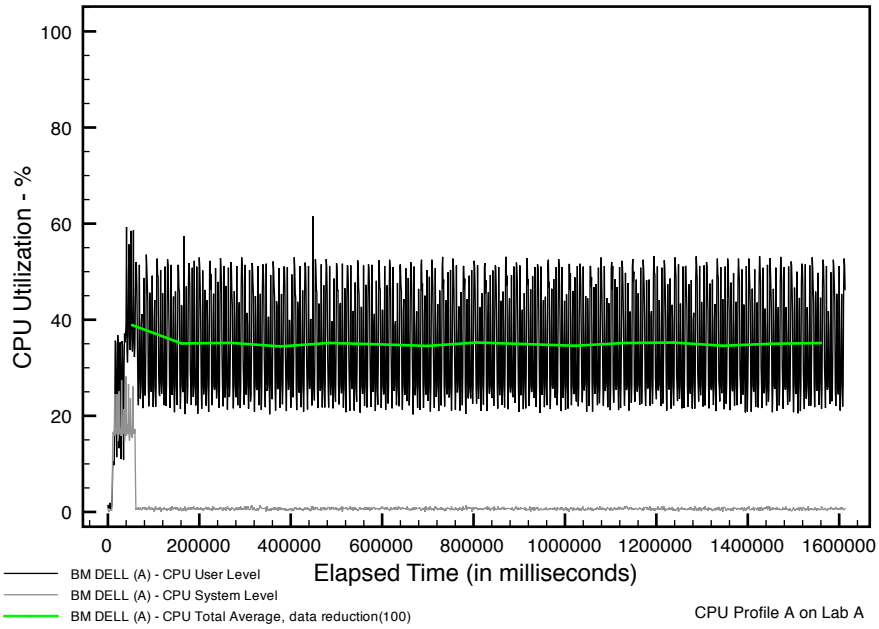

Figure M.1: Graph showing the CPU utilization of BareMetal OS at user and system level with 10 VMs testing pattern A on the hardware lab.

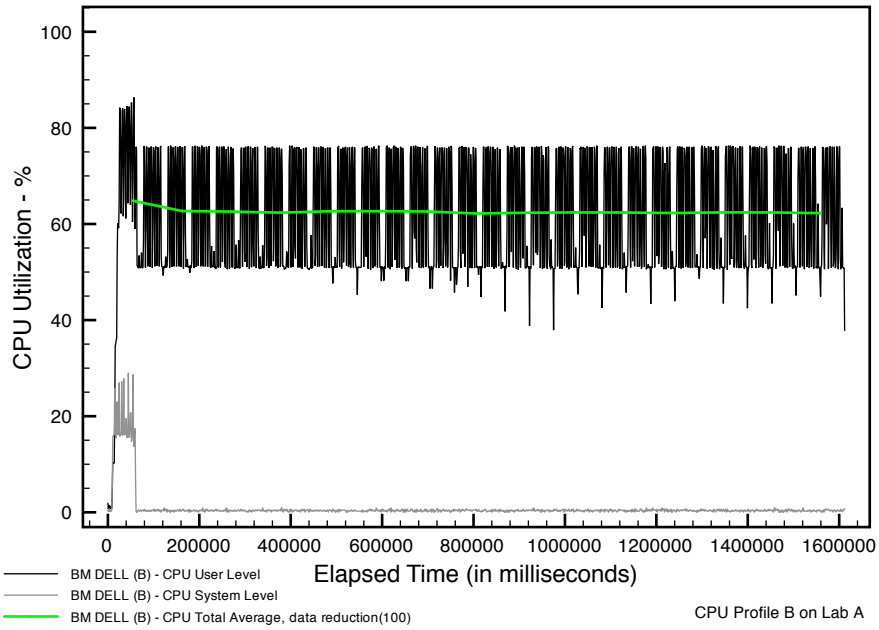

Figure M.2: Graph showing the CPU utilization of BareMetal OS at user and system level with 10 VMs testing pattern B on the hardware lab.

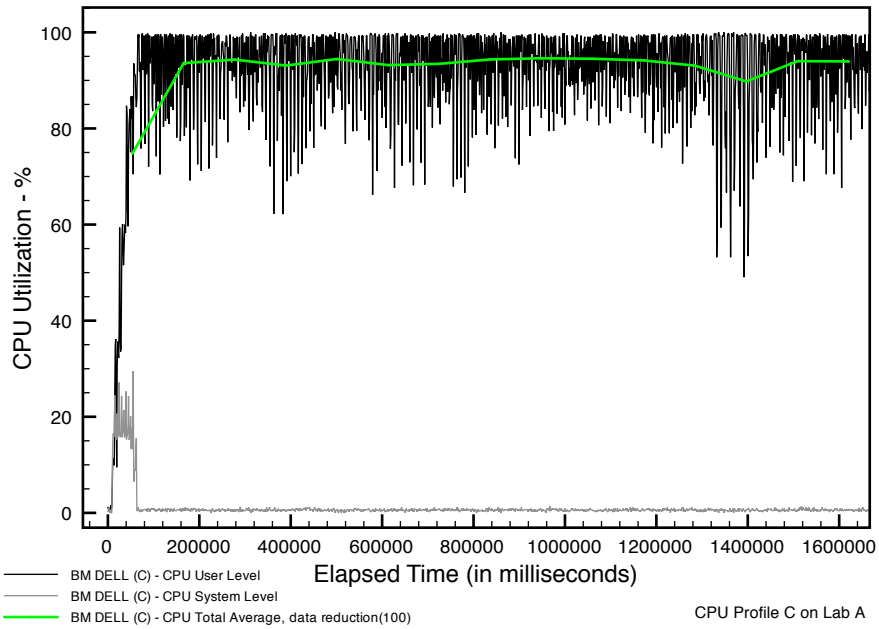

Figure M.3: Graph showing the CPU utilization of BareMetal OS at user and system level with 10 VMs testing pattern C on the hardware lab.

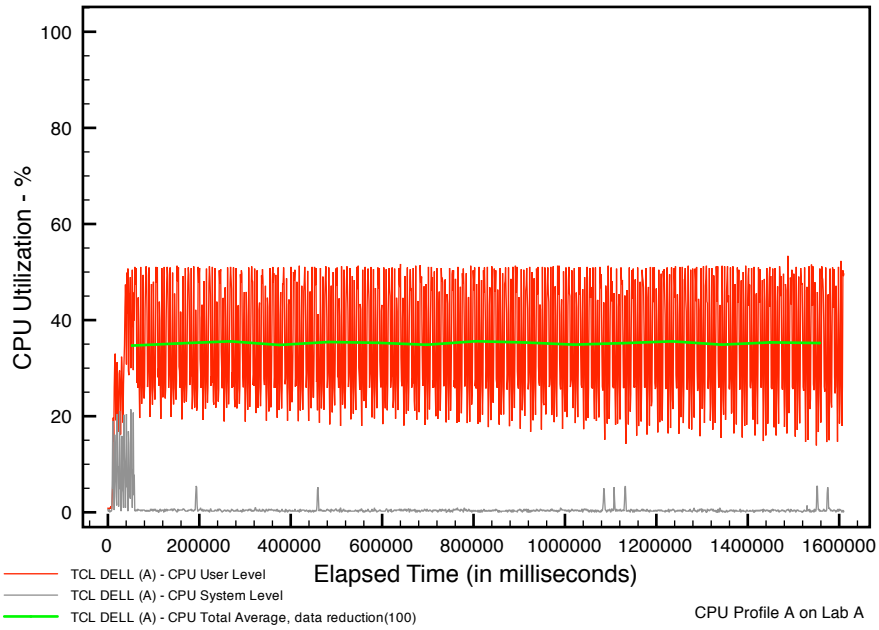

Figure M.4: Graph showing the CPU utilization of TinyCore Linux at user and system level with 10 VMs testing pattern A on the hardware lab.

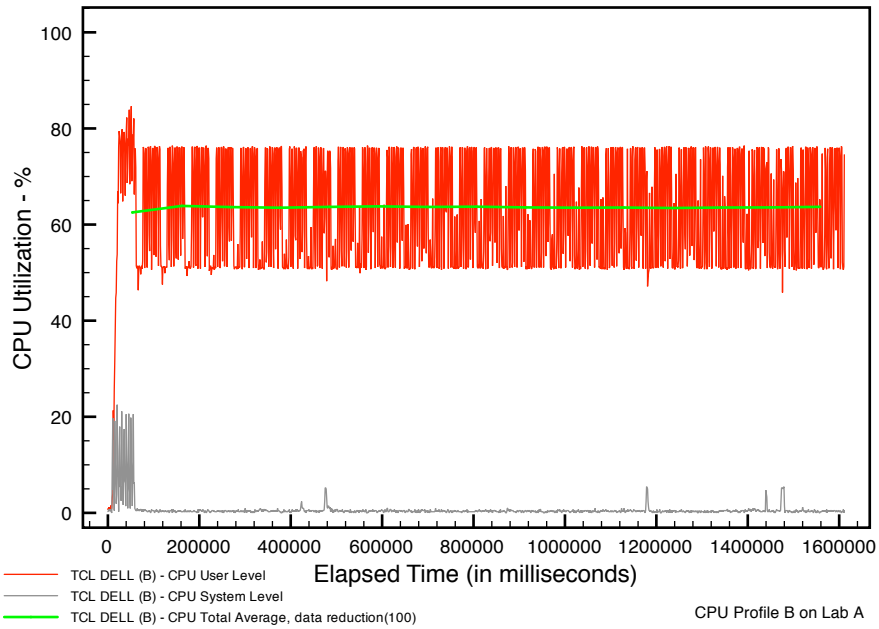

Figure M.5: Graph showing the CPU utilization of TinyCore Linux at user and system level with 10 VMs testing pattern B on the hardware lab.

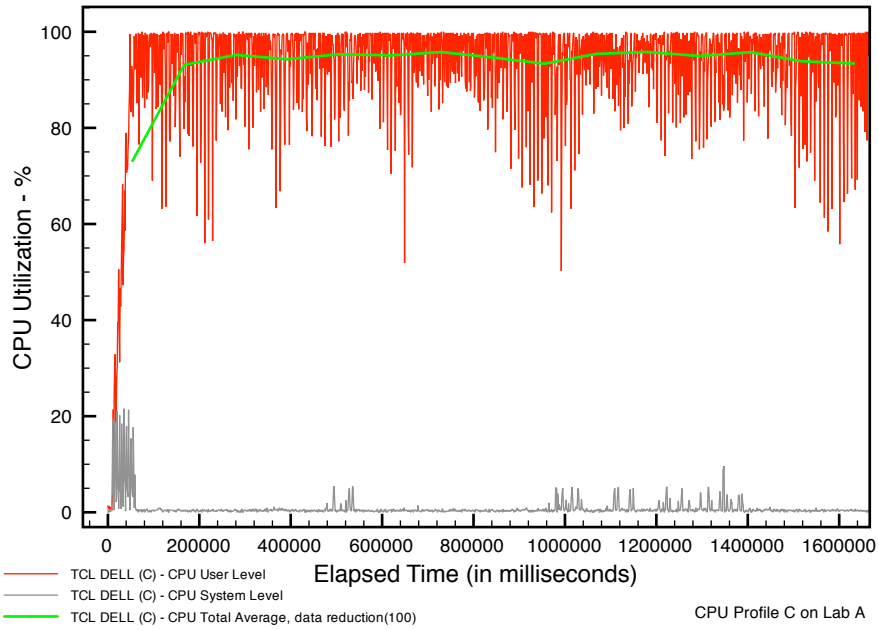

Figure M.6: Graph showing the CPU utilization of TinyCore Linux at user and system level with 10 VMs testing pattern C on the hardware lab.

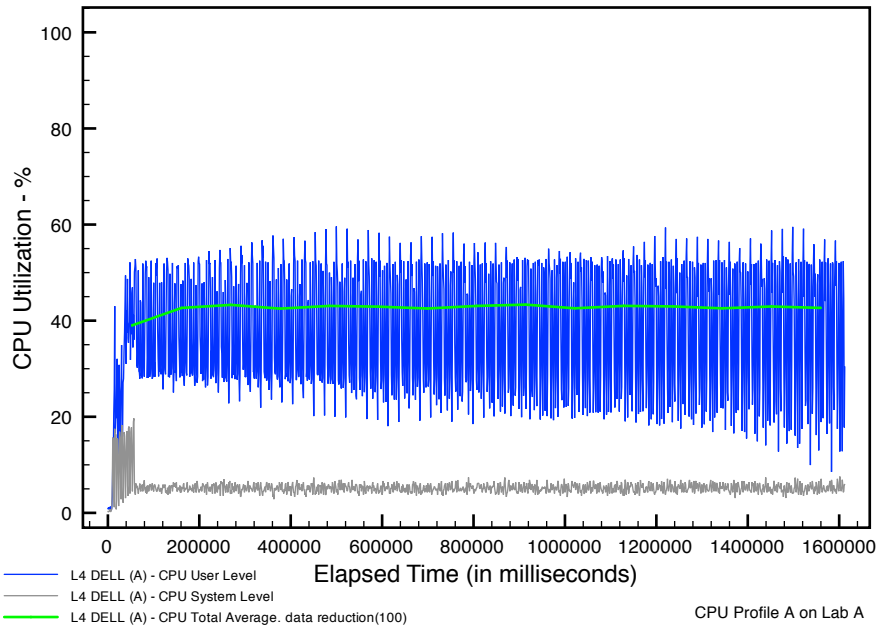

Figure M.7: Graph showing the CPU utilization of L4 Pistachio at user and system level with 10 VMs testing pattern A on the hardware lab.

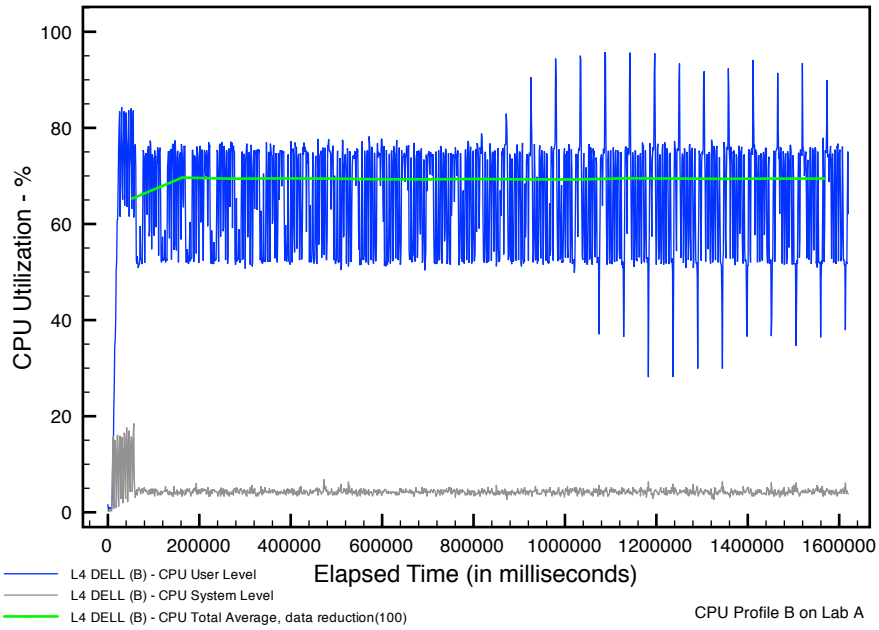

Figure M.8: Graph showing the CPU utilization of L4 Pistachio at user and system level with 10 VMs testing pattern B on the hardware lab.

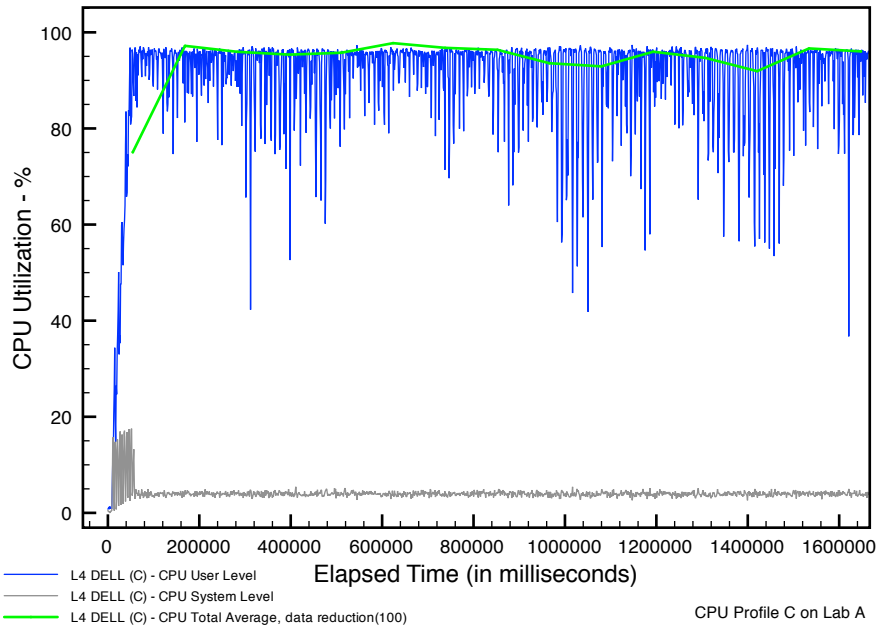

Figure M.9: Graph showing the CPU utilization of L4 Pistachio at user and system level with 10 VMs testing pattern C on the hardware lab.

Appendix N

CPU Profile Tests Cloud lab

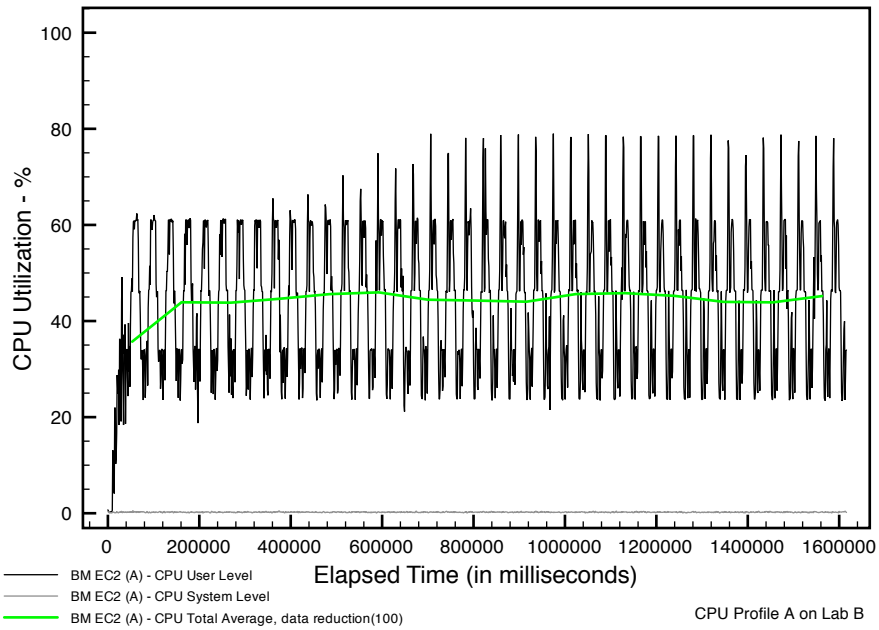

Figure N.1: Graph showing the CPU utilization of BareMetal OS at user and system level with 10 VMs testing pattern A on the cloud lab.

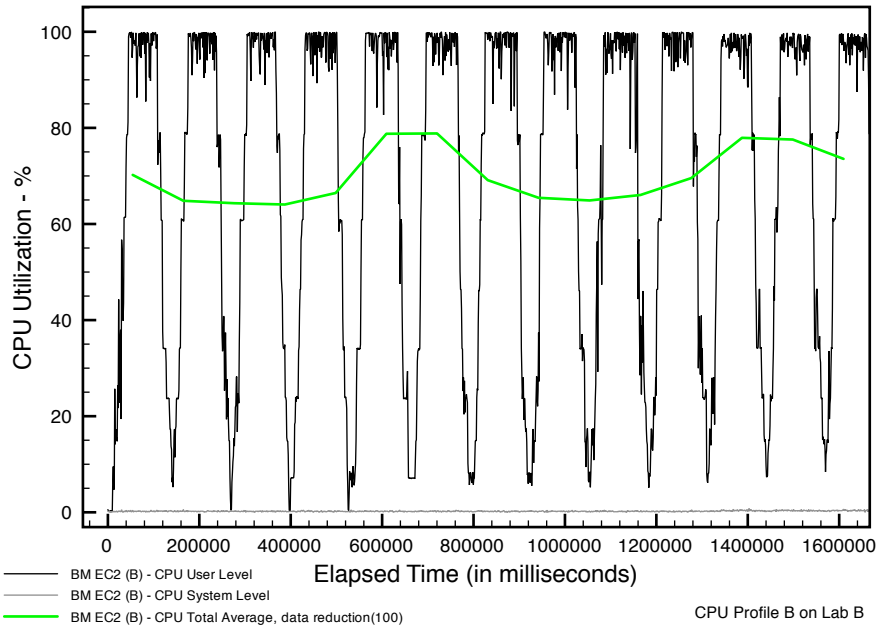

Figure N.2: Graph showing the CPU utilization of BareMetal OS at user and system level with 10 VMs testing pattern B on the cloud lab.

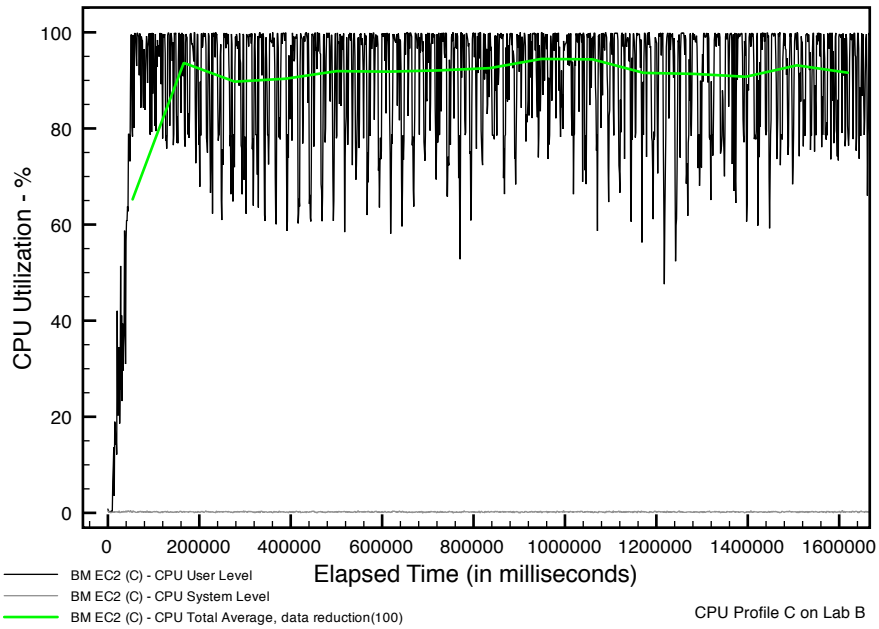

Figure N.3: Graph showing the CPU utilization of BareMetal OS at user and system level with 10 VMs testing profile C on the cloud lab.

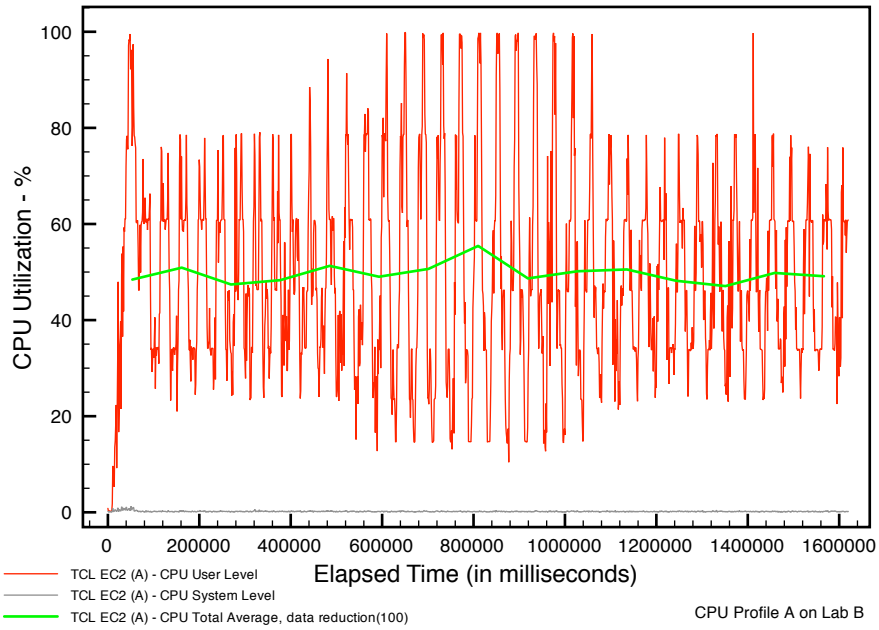

Figure N.4: Graph showing the CPU utilization of TinyCore Linux at user and system level with 10 VMs testing pattern A on the cloud lab.

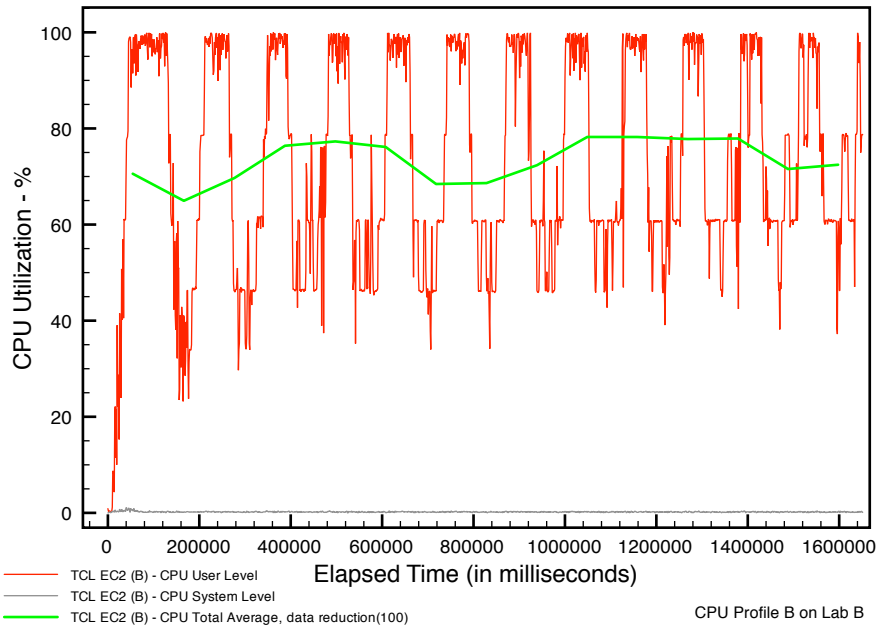

Figure N.5: Graph showing the CPU utilization of TinyCore Linux at user and system level with 10 VMs testing pattern B on the cloud lab.

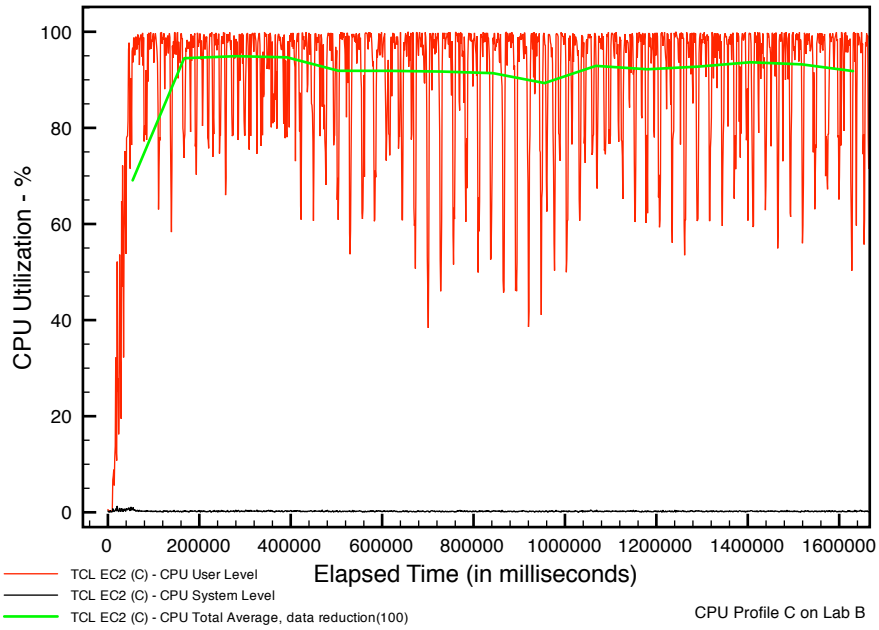

Figure N.6: Graph showing the CPU utilization of TinyCore Linux at user and system level with 10 VMs testing pattern C on the cloud lab.

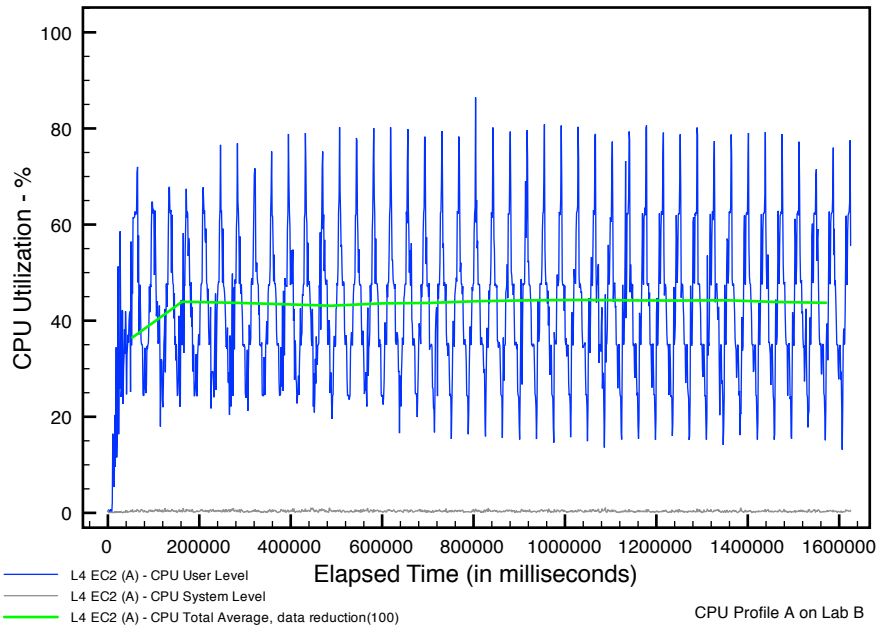

Figure N.7: Graph showing the CPU utilization of L4 Pistachio at user and system level with 10 VMs testing pattern A on the cloud lab.

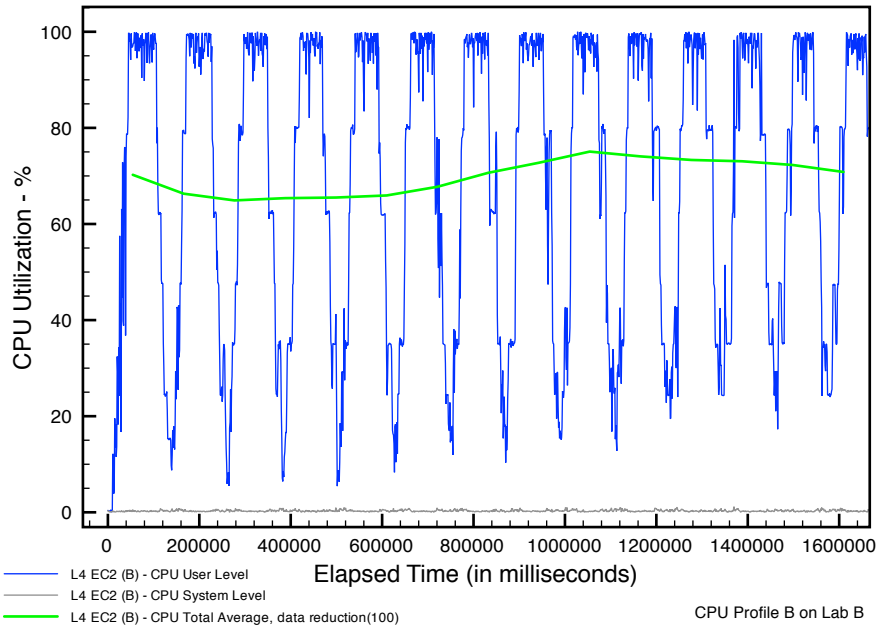

Figure N.8: Graph showing the CPU utilization of L4 Pistachio at user and system level with 10 VMs testing pattern B on the cloud lab.

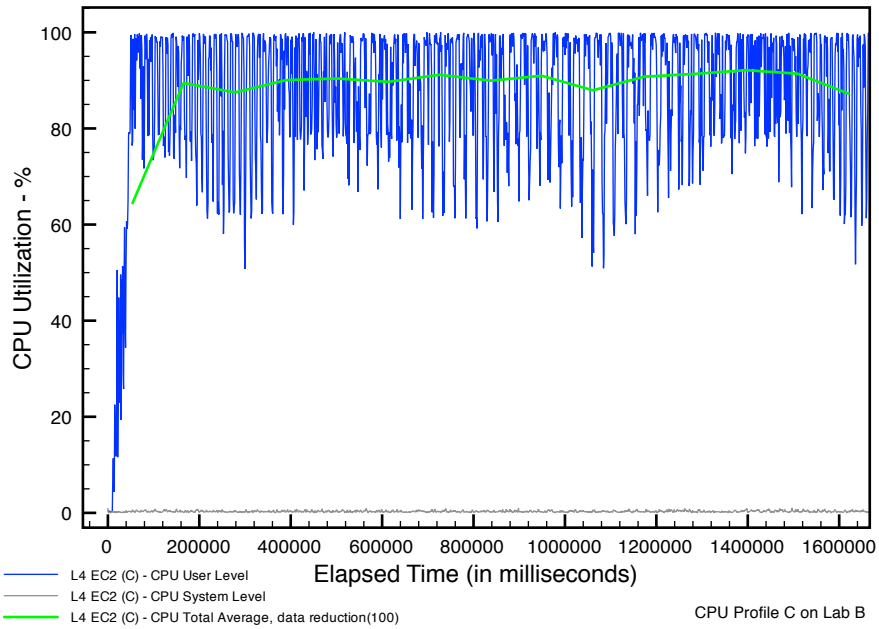

Figure N.9: Graph showing the CPU utilization of L4 Pistachio at user and system level with 10 VMs testing pattern C on the cloud lab.

Appendix O

L4 Pistachio Usage Patterns, L4_patterns.cc

```
1  /*********************************************************************
2   *
3   * Author 2012, Hvard Ostnes
4   *
5   * File path: /
6   * Description: OLD CPU patterns for the L4Ka::Pistachio
7   * Sleep value of 1 000 000 = 1 second
8   * Pattern specific settings:
9   * Pattern A: fibsleeper = 20000000; i = 40;
10  * Pattern B: fibsleeper = 40000000; i = 43;
11  * Pattern C: fibsleeper = 5000000; i = 40;
12  ********************************************************************/
13 #include <l4io.h>
14 #include <l4/ipc.h>
15
16 int fib(int n);
17
18 int main(void)
19 {
20     L4_Time_t fibsleeper;
21     fibsleeper = L4_TimePeriod(20000000);
22
23     for(int i = 40; i <= 50; i++)
24       {
25           if( i == 41)
26             {
27                 fib(i);
28                 L4_Sleep(fibsleeper);
29             }
30
31           if( i == 50)
32             {
33                 // Resetting loop.
34                 i = 39;
35             }
36       }
37     return 0;
38 }
```

```
39
40 int fib(int n)
41 {
42     if (n==0 || n==1)
43         return 1;
44     else
45         return fib(n−1)+fib(n−2);
46 }
```

Appendix P

L4 Pistachio Usage Patterns, L4_patterns.cc

```
1  /*********************************************************************
2   *
3   * Author 2012, Hvard Ostnes
4   *
5   * File path: /
6   * Description: NEW CPU patterns for the L4Ka::Pistachio
7   * Sleep value of 1 000 000 = 1 second
8   * Pattern specific settings:
9   * Pattern A: fibsleeper = 20000000; i = 41;
10  * Pattern B: fibsleeper = 40000000; i = 44;
11  * Pattern C: fibsleeper = 5000000; i = 41;
12  ********************************************************************/
13 #include <l4io.h>
14 #include <l4/ipc.h>
15
16 int fib(int n);
17
18 int main(void)
19 {
20     L4_Time_t fibsleeper;
21     fibsleeper = L4_TimePeriod(20000000);
22
23     for(int i = 40; i <= 50; i++)
24       {
25           if( i == 41)
26             {
27                 fib(i);
28                 L4_Sleep(fibsleeper);
29             }
30
31           if( i == 50)
32             {
33                 // Resetting loop.
34                 i = 39;
35             }
36       }
37     return 0;
38 }
```

```
39
40 int fib(int n)
41 {
42     if (n==0 || n==1)
43         return 1;
44     else
45         return fib(n−1)+fib(n−2);
46 }
```

Appendix Q

BareMetal OS Usage Patterns, BM_patterns.c

```
1  /*************************************************************************
2   *
3   * Author 2012, Hvard Ostnes
4   *
5   * File path: /
6   * Description: Usage Pattern A for BareMetal OS
7   *   A sleep value of 8 is 1 second.
8  *  Example: 8 = 1 second, 80 = 10 seconds
9   *
10  * Settings for all patterns:
11  * Pattern A: sleeper = 160; i = 40;
12  * Pattern B: sleeper = 320; i = 43;
13  * Pattern C: sleeper = 40; i = 40;
14  *************************************************************************/
15 #include "libBareMetal.h"
16
17 int fib(int n);
18
19 int start(void)
20 {
21     // 8 == 1 second, 160 == 20 sec
22     int sleeper = 160;
23     int i;
24
25     for(i = 40; i <= 50; i++)
26       {
27         if( i == 40 )
28           {
29             fib(i);
30             b_delay(sleeper);
31           }
32
33         if( i == 50 )
34           {
35             i = 39;
36           }
37       }
38     return 0;
```

```
39 }
40
41
42 int fib(int n)
43 {
44     if (n==0 || n==1)
45         return 1;
46     else
47         return fib(n−1)+fib(n−2);
48 }
```

Appendix R

Tiny Core Linux Usage Patterns, TCL_patterns.cc

```
1  /*****************************************************************
2   *
3   * Author 2012, Hvard Ostnes
4   *
5   * File path: /
6   * Description: Usage Pattern A for Tiny Core Linux
7   *
8   * Settings for all patterns:
9   * Pattern A: sleeper = 20; i = 40;
10  * Pattern B: sleeper = 40; i = 43;
11  * Pattern C: sleeper = 5; i = 40;
12  *****************************************************************/
13 #include <stdio.h>
14 #include <unistd.h>
15
16 int fib(int n);
17
18 int main(void)
19 {
20     // Change i and sleeper to achieve the desired usage pattern
21     int sleeper = 5;
22
23     for(int i = 40; i <= 50; i++)
24       {
25         if( i == 40)
26             {
27                 fib(i);
28                 sleep(sleeper);
29             }
30
31          if( i == 50)
32             {
33                 // Resetting loop.
34                 i = 39;
35             }
36       }
37     return 0;
38 }
```

```
39
40 int fib(int n)
41 {
42     if (n==0 || n==1)
43         return 1;
44     else
45         return fib(n−1)+fib(n−2);
46 }
```

Appendix S

L4 Pistachio Performance Test Application, L4_context_apps.cc

```
1  /*********************************************************************
2   *
3   * Author 2012, Hvard Ostnes
4   *
5   * File path: /
6   * Description: CPU Simulation application for L4 Pistachio
7   *
8  * Population sizes and number of laps used:
9   * 4 VMs: laps = 384;
10  * 8 VMs: laps = 192;
11  * 16 VMs: laps = 96;
12  * 32 VMs: laps = 48;
13  * 64 VMs: laps = 24:
14  *
15  *********************************************************************/
16 #include <l4io.h>
17 #include <l4/ipc.h>
18
19 int fib(int n);
20
21 int main(void)
22 {
23     int input = 41;
24     int laps = 384;
25     int i;
26     L4_Time_t fibsleeper;
27     // Sleep 60 seconds
28     fibsleeper = L4_TimePeriod(60000000);
29     L4_Sleep(fibsleeper);
30     // SLeep 2000 seconds
31     fibsleeper = L4_TimePeriod(2000000000);
32     for(i = 0; i < laps; i++)
33       {
34
35           fib(input);
36           if(i == laps-1)
37             {
38                 //sleeps to show test is done
```

```
39                  L4_Sleep(fibsleeper);
40              }
41
42          }
43      return 0;
44  }
45
46
47  int fib(int n)
48  {
49      if (n==0 || n==1)
50          return 1;
51      else
52          return fib(n-1)+fib(n-2);
53  }
```

Appendix T

BareMetal OS Performance Test Application, BM_context_apps.c

```
1  /*************************************************************************
2   *
3   * Author 2012, Hvard Ostnes
4   *
5   * File path: /
6   * Description: CPU Simulation application for BareMetal OS
7   *
8   * Population sizes and number of laps used:
9   * 4 VMs: laps = 384;
10  * 8 VMs: laps = 192;
11  * 16 VMs: laps = 96;
12  * 32 VMs: laps = 48;
13  * 64 VMs: laps = 24:
14  *
15  *************************************************************************/
16 #include "libBareMetal.h"
17
18 int fib(int n);
19
20 int start(void)
21 {
22     int input = 40; // Takes ~3 seconds
23     int laps = 12; // 300 laps == 900 seconds
24     int i;
25
26     // Sleep 60 seconds
27     b_delay(480);
28
29     for(i = 0; i < laps; i++)
30       {
31
32           fib(input);
33           if(i == laps-1)
34             {
35                 //sleeps to show test is done
36               b_delay(16000);
37             }
38
```

```
39        }
40     return 0;
41  }
42
43
44  int fib(int n)
45  {
46     if (n==0 || n==1)
47        return 1;
48     else
49        return fib(n−1)+fib(n−2);
50  }
```

Appendix U

Tiny Core Linux Performance Test Application , TCL_context_apps.cc

```
1  /*****************************************************************
2   *
3   * Author 2012, Hvard Ostnes
4   *
5   * File path: /
6   * Description: CPU Simulation application for Tiny Core Linux
7   *
8   * Population sizes and number of laps used:
9   * 4 VMs: laps = 384;
10  * 8 VMs: laps = 192;
11  * 16 VMs: laps = 96;
12  * 32 VMs: laps = 48;
13  * 64 VMs: laps = 24:
14  *
15  *****************************************************************/
16 #include <stdio.h>
17 #include <unistd.h>
18
19 int fib(int n);
20
21 int main(void)
22 {
23     int input = 40; // Takes ~3 seconds
24     int laps = 384; // 300 laps == 900 seconds
25     int i;
26
27     // Sleep 60 seconds
28     sleep(60);
29
30     for(i = 0; i < laps; i++)
31       {
32
33           fib(input);
34           if(i == laps-1)
35              {
```

```
36                  //sleeps to show test is done
37                  sleep(2000);
38              }
39
40          }
41      return 0;
42  }
43
44
45  int fib(int n)
46  {
47      if (n==0 || n==1)
48          return 1;
49      else
50          return fib(n-1)+fib(n-2);
51  }
```