Evaluation of Bare Metal CPU and Memory Performance of the Unikernel IncludeOS and Ubuntu

Martin Kot

Thesis submitted for the degree of
Master in Network and System Administration
30 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2019

Evaluation of Bare Metal CPU and Memory Performance of the Unikernel IncludeOS and Ubuntu

Martin Kot

Evaluation of Bare Metal CPU and Memory Performance of the Unikernel IncludeOS and Ubuntu

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

General purpose operating systems are widely in use for running many different applications on a big selection of hardware. Such an operating system must be quite complete in order to provide virtual memory, scheduling, services etc. for any application. They are designed with multitasking in mind, being able to run several tasks simultaneously, utilizing scheduling and time sharing. Any running application may at any time be interrupted if resources are requested by a different process. This will usually happen because of an interrupt or context switch from kernel to user mode, and can result in uneven performance.

A unikernel operating system is a different approach. It is just a single binary with only a single application running in kernel mode without any additional bloat. A lot of the general purpose software is not needed when running a virtual machine. This makes a unikernel OS a fast and lightweight option to general purpose operating systems for virtual machines.

The aim of this thesis is to compare the CPU and memory performance of a unikernel-based operating system and a general purpose operating system when running on bare metal and not on virtual machines. The goal is to avoid the operating system noise which is known to occur when using general purpose operating systems. In this case the comparison was done between the IncludeOS unikernel and Ubuntu Linux.

Contents

List of Figures

List of Tables

Preface

Acknowledgments

First of all, a huge thanks to my supervisor and mentor, Hårek Haugerud. This thesis would never become a reality without his long term support, technical knowledge, guidance, care and kindness. A truly great person, both as a scientist, and with fantastic people skills.

A big thanks to my friends and my sister for all their care and support, and not giving up on me.

I would also like to thank the administration at UiO and OsloMet for this chance and making it possible to complete.

Chapter 1

Introduction

Today most operating systems in use are general purpose operating systems. They try to cater for all kinds of purposes and must support a huge variety of hardware. In general, such an operating system has a lot of bloat and many running services. This will affect the performance if the objective is to just run a single application to solve a task as efficiently as possible.

1.1 Motivation

A general purpose operating system like Linux is unable to satisfy the requirements of applications needing immediate response time. Examples of such applications are trading algorithms, self-driving cars, aviation and aerospace systems etc. In Linux an application might be delayed because of kernel interrupts, paging, system services and other processes with a higher priority. Linux is known to have issues like unpredictable latencies and limited support for real-time scheduling [1].

In comparison a unikernel like IncludeOS is highly specialized and runs only a single task in a single thread. There are no other OS duties or interrupts to worry about. The assumption is that this will result in improved efficiency and fewer delays in completing application tasks.

How consistent is the response time compared with Linux? One way to analyze this is by testing CPU and memory performance. For applications like high frequency trading, the extra cost and possible delays due to virtualization should be avoided. Normally, unikernels are used in virtualized environments. The focus on efficiency in the design of a unikernel makes it a promising candidate as an operating system for applications in need of fast response and execution also when running directly on hardware. This is often described as running on bare metal, which means without virtualization of any kind.

1.1.1 High Frequency Trading

High Frequency trading (HFT) is a type of algorithmic trading. This typically involves high speeds, high turnover rates and a huge volume of stocks exchanged in a very short amount of time [2]. There are a few definitions of HFT, but typically they make use of extremely sophisticated algorithms and deploy them in short-term investments not humanly executable [3]. In order for an HFT to be effective, proprietary trading strategies carried out by computers in fractions of a millisecond are employed. HFT is often regarded as the primary form of algorithmic trading in finance [4] The most profitable company will in general be the one with the fastest hardware and the best trading algorithms [5] In general the competition in HFT is not from humans, but from other companies employing the same strategy. The typical algorithms and optimizations are therefore proprietary.

HFTs require low latency combined with high frequency programming. In almost all cases an HFT is run on a UNIX variant, typically Linux. Even if Linux is traditionally tuned to support a large number of running processes and to fairly provide each process a share of resources, it can be tuned in different ways. An HFT will typically require just one application running with the least amount of overhead for context switching and lowest possible latency from a very fast network link. Additionally, I/O speed should be prioritized more than memory usage, for instance by keeping databases in memory, cached up front etc. A unikernel is a type of an operating system that runs just a single task and nothing more. This should make it more suitable and efficient to run just the required task without interruptions and sharing resources with competing processes. In theory this should result in both better performance and noise.

1.2 Problem statement

Until recently the IncludeOS unikernel has been developed in order to run in a virtualized environment. In this case IncludeOS may use virtualized drivers and there has not been any need for developing drivers for all kinds of physical devices. When running directly on hardware, like most operating systems normally do, drivers for the actual hardware at hand are needed. Since no hardware drivers for IncludeOS have yet been developed, this study concentrates on comparing the performance of the CPU and memory of an Intel server. In these cases there is no need for hardware drivers and performance tests may be run using the existing version of IncludeOS. This leads to the first of the two research questions for this thesis:

> *What are the differences in performance of the CPU and memory when running workloads on a server controlled by the IncludeOS unikernel compared to the case where the same server hardware is controlled by*

the Ubuntu operating system?

In both cases the systems are running directly on the server hardware and they carry out the exact same computations.

Our hypothesis is that the Linux kernel occasionally will perform operations related to scheduling, paging and other duties which a general purpose operating system has to deal with, and that this will result in variances in the performance of applications running on the system. Our second research question has two parts:

> *To what extent do the performance results vary when very short CPU and memory operations are performed a large number of times? To what extent do delays in the execution of workloads of the multipurpose operating system occur compared to the unikernel system?*

Chapter 2

Background

2.1 Bare Metal

The definition of bare metal is physical hardware, and software being executed directly on it. Traditionally this has been the main way of running operating systems and applications, all the way from the earliest computers to sharing resources on a mainframe running UNIX. Computing hardware with good performance was quite expensive, but due to Moore's law and big leaps in performance vs. cost ratio, at one point in time it became apparent that commodity server and desktop hardware could easily run several operating systems simultaneously by utilizing a concept called virtualization. This is a viable option as most computers stay idle for long periods, wasting resources. Virtualization has come a long way, and in some cases the performance is almost equivalent to real hardware. Providing that the CPU has virtualization extensions, it is well supported in both different operating systems and platforms. However, it is undeniable that bare metal hardware might still yield better performance, especially for time critical applications like high frequency trading. At least this is the assumption.

2.2 Virtualization

Virtualization is a way of presenting virtual, rather than a physical, version of something. This could be both computer hardware, storage devices and networking. It is a layer of abstraction enabling a single computer to run multiple operating systems at the same time, each seeing its own virtual hardware subsystem. Quite often the terms *host* and *guest* are used: the host being the physical hardware with the software running directly on it, and the guest(s) being the virtualized hardware/software. The layer of abstraction is called a hypervisor, which is a piece of low level software usually responsible for creating the virtual machines and sharing resources like processing, memory etc.

Originally the typical classification of the hypervisors was either as Type 1 or Type 2

- Type 1 hypervisor: a bare metal hypervisor that runs directly on a host's hardware, often beneath the operating system
- Type 2 hypervisor: an application or process executed in the host operating system

The type definitions were created decades ago, and do not really apply to modern virtualization. Modern hypervisors like Xen, KVM and VMware are both type 1 and type 2 hypervisors. Both Xen and KVM have support in the Linux kernel turning the host into a type 1 hypervisor, but the host is still a fully usable operating system with VMs running as processes making it type 2.

2.3 Unikernels

Typically a standard operating system consists of a large kernel and user space tools with libraries. The operating system is developed with good support for different hardware (eg. drivers and modules) with time sharing between different processes in mind. This involves context switching and being able to react on interrupts (like input from a keyboard, network or I/O). While this makes the system great for running many different applications at once and out of the box, it imposes an overhead and a lot of unneeded functionality when all that is required is a single application. Unikernels are specialized library operating systems. A library operating system is a self contained entity with just a single purpose. Instead of making use of existing operating system features, providing libraries and modules, you start in the other end by creating an application first, then wrapping an operating system around it. Essentially you link and compile an application with just the necessary functions from the required libraries, then add a boot loader. The result is an operating system that boots and runs just a single application in kernel mode. There is no time-sharing, multi-threading, unused libraries or modules to worry about. Just the bare minimum for a single threaded application.

A unikernel is not a new invention, already in 1995 the Exokernel [6] was a description of a similar approach. The term unikernel, however, was invented fairly recently in a paper by Madhavapeddy et. al. [7], where MirageOS is presented as a modern library operating system for the cloud. With the development of virtualization and good support for it in both hardware and the operating systems, it is now very easy to deploy VMs in the cloud. However, it is very common that VMs run a general operating system with just a single application, and the unikernel has become a viable option to make efficient use of resources. Since a unikernel is a very small binary with extremely fast start-up times, it is very easy to scale VMs

horizontally by executing thousands of small instances very quickly and efficiently.

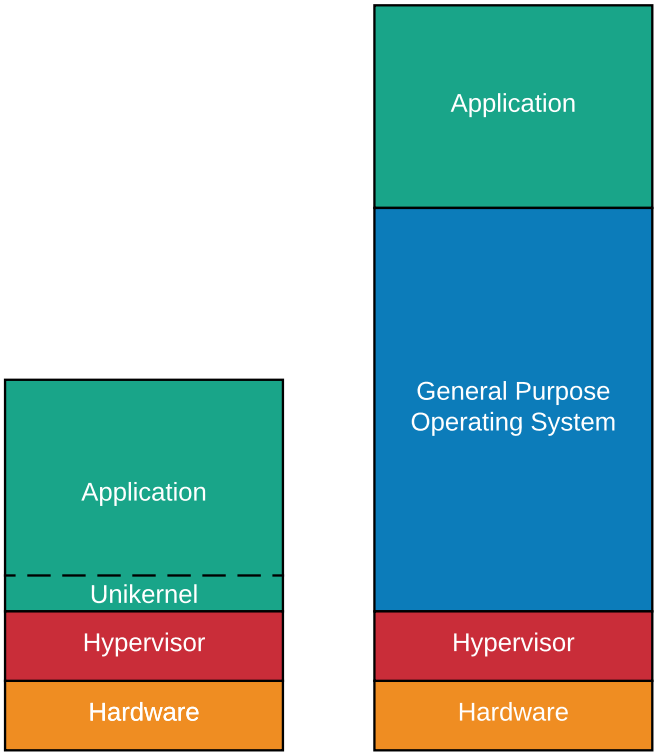

Figure 2.1: A traditional virtualized setup showing both a unikernel and a general purpose operating system

A typical setup is shown in Fig. 2.1, with the hypervisor running a virtual operating system on top of it. This has been a very common configuration for many years, as it has become quite simple to deploy VMs. Unikernels are very often made with virtualization in mind. The emphasis here is to show the larger overhead and resource usage from a general purpose operating system compared with a unikernel.

A progression from the virtual environment is shown in Fig. 2.2. In this case both operating systems run directly on hardware. This is to remove any bottlenecks imposed by a hypervisor, to improve both latency and resource usage. For the general purpose operating system this is a standard configuration, but for a typical unikernel a new and more complex scenario. A unikernel is an OS made from ground up, not being based on existing operating systems. Drivers and hardware support must be coded from scratch.

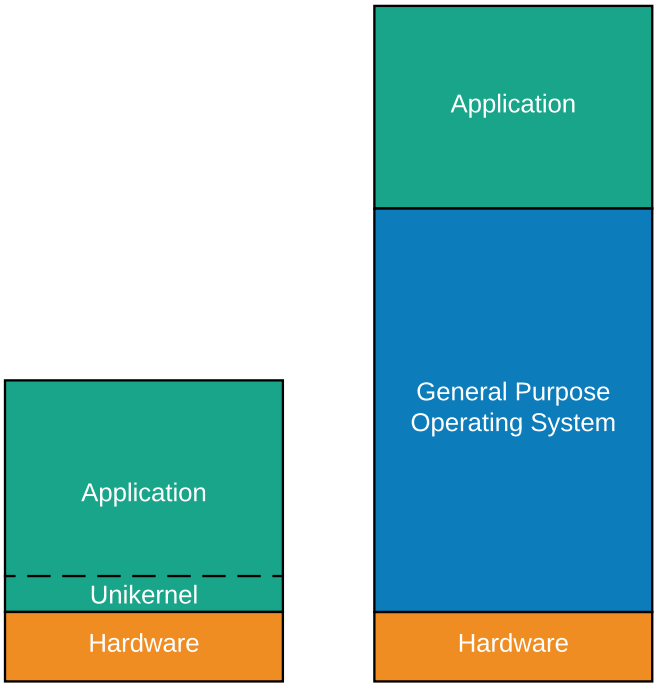

Figure 2.2: Comparison of a unikernel and a general purpose operating system running directly on hardware (bare metal)

2.4 IncludeOS

IncludeOS [8] is a library operating system with minimal system requirements. This was a research project started by Alfred Bratterud at Oslo University College (now Oslo Metropolitan University), but is now a standalone project developed by the independent company IncludeOS AS [9]. IncludeOS is designed from the ground up as an efficient single-tasking operating system for virtualized environments. It is developed in C++ and aims to support C/C++ applications, referred to as services. A typical service starts with `#include <os>` and this will include a tiny operating system at link-time. The resulting binary links in just the required OS object files with a boot-loader and a ready to run image file that supports modern hypervisors. An overview of the IncludeOS build system is shown below in Fig. 2.3

IncludeOS can run fully virtualized on both KVM, VirtualBox and VMWare. The main development and testing are done on KVM and VMWare Fusion/ESXi. This makes it possible to run IncludeOS virtualized both on a single desktop or server, but also easily in the cloud like OpenStack or Google Compute Engine [10].

The unikernel is developed in modern C++, in contrast to some other unikernels like MirageOS (OCaml) and OSv (Java) [11]. One of the main principles of the C++ language is the *zero-overhead principle* [12] and this

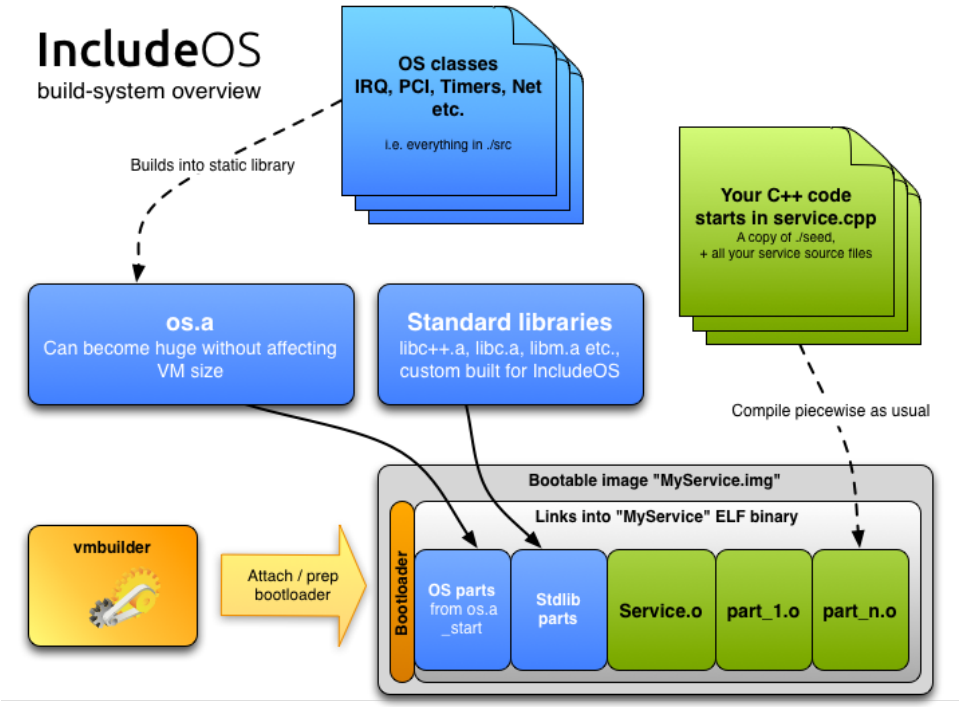

Figure 2.3: IncludeOS build system

is also an essential part of the IncludeOS model. It has full C++ language support when compiled with clang 5 (or later) and the standard library from LLVM. A compiled unikernel is only about 2.5 MB and boots in 0.3 seconds, in special cases it can even boot in 10 ms (with the Solo5/uKVM from IBM). A ”Hello World!” program in IncludeOS barely uses 8.45 MB of memory, while a minimal instance on Linux in OpenStack is around 300 MB .[8] The purpose is to use a library to add the required functions at link time, generating a single service. In essence an application or service is added with the necessary operating system components to create an ”operating system” image. This is in contrast to a general purpose operating systems which must provide all the necessary libraries and a lot of functionality in order to run any kind of applications.

In contrast, the resulting IncludeOS image contains only the necessary bits including a boot loader. The result is a single threaded unikernel running just a single service. There is no framework for virtual memory, multithreading, huge stacks and context switches. The whole focus is to run just a single application. There are no interrupts, the OS itself is event based. Modern hypervisors provide scheduling and sharing of resources, it is unnecessary to add another abstraction layer in the OS itself. Although very small, IncludeOS can still take advantage of multiple CPUs/cores and has support for threading. The hardware support is mainly for VMs using the Virtio framework, there is no need to include drivers for a wide range of devices. Recent development has enabled limited support for ”bare

metal" hardware. On physical hardware IncludeOS is very much a real time operating system (RTOS), since the latency is very predictable with no interrupts and pre-emption.

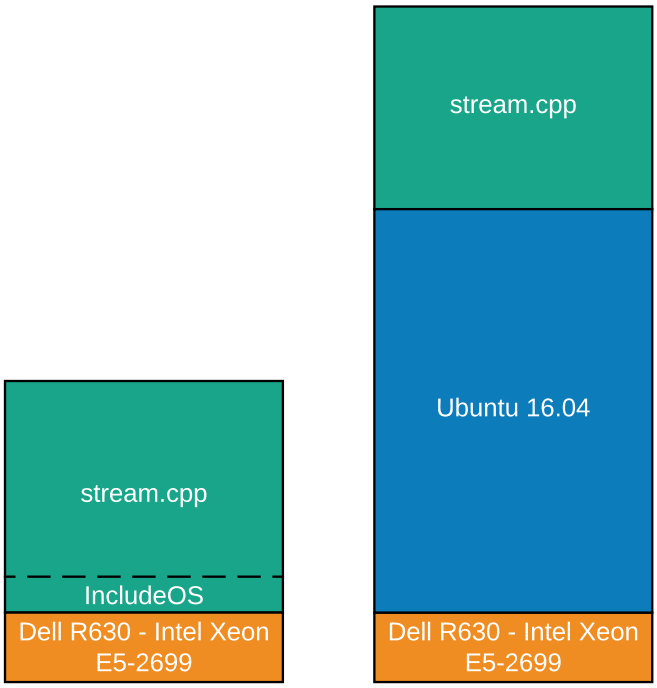

Figure 2.4: The experimental bare metal setup with IncludeOS unikernel and Linux running directly on Intel hardware

Figure 2.4 shows the experimental setup with both IncludeOS and Linux running on bare metal Intel hardware. The idea is to show the large resource usage by a GPOs compared with IncludeOS, even though the figure does not scale correctly. In reality IncludeOS memory footprint is typically just 1/35 of Linux for smaller applications [8].

Another great aspect of IncludeOS is security. The application is small and runs only in kernel mode and the hypervisor already provides good separation. Furthermore, the service is immutable and cannot be changed without being recompiled, which results in a very limited attack surface. With the advent of Internet of Things (IoT), security is even more important. But in order to still allow for a smooth update of services, IncludeOS includes the LiveUpdate feature. It is a way to update the service "on the fly" by saving state before replacing the old instance without any interruptions.

2.5 Real Time Operating Systems

Real Time Operating Systems (RTOS) are made for time critical application. An operating system like this is made with guarantees that a process will

complete or respond in a given time frame. Often used in the health industry, aviation, cellular networks and other time critical applications. Linux can be used as an RTOS, but since it is a general operating system with complex scheduling algorithms, interrupts and aiming for good pre-emptive multitasking and responsiveness it is not ideal for this kind of applications. In addition, a GPO is in constant change, and the state is unpredictable. It requires quite a lot of tweaking and knowledge to set up properly to behave as a RTOS. A unikernel is very static by default and much more predictable for this kind of use.

Another aspect is that a RTOS implies overhead. Guaranteeing that a process will complete a response within a given time frame means that other processes must be interrupted to let the guarantee be fulfilled. The guaranteed response time is the primary objective for a RTOS, not performance. For high performance, Linux is often an option, and it is being used in HFT. This often yields adequate performance, but the trade off here might be increased delays and noise due to unexpected interruptions and context switches. As opposed to RTOS, Linux has no response time guarantees and occasionally OS operations like paging may lead to relatively long glitches where the processing of the service is paused. However, a unikernel like IncludeOS may provide both high performance and low latency by avoiding interrupts from the OS performing its regular tasks.

2.6 Related Work

Several papers investigate how to reduce the system noise in High Performance Computing environments. A HPC system distributes a single task amongst a large number of CPU cores. Since each core is dedicated to this one task, it is beneficial to let this task run with the least amount of interruption from local timer interrupts. Those are often used for accounting, system time, preemption and scheduling.

Both [13] and [14] discuss how to reduce system noise in the Linux kernel. Here the definition of noise is any delay or interruption that comes from the kernel, and does not originate from the application itself. A point is made that the traditional way of pinning an application to a CPU will still let it be interrupted by system processes which are not pinned and can be scheduled to run on any CPU. They describe a solution with dedicated CPUs for the OS and applications, and the `isolcpus` Linux kernel option to disable SMP load balancing and prevent scheduling of any user-threads and kernel tasks/system services on the isolated cores. However, this does not prevent kernel threads and daemons to still be migrated to one of the isolated CPUs.

Additionally they show a modification of the Linux kernel for `tickless` operation to disable any timer interrupts on CPUs running the application, and designate specific OS cores for system critical clock ticks.

Chapter 3

Approach

The purpose of the experiments was to test the performance of both IncludeOS running on bare metal and Linux on the same hardware. Compiling and running code on Linux is straightforward, while it is a bit more challenging for IncludeOS. It is even harder to make IncludeOS run on bare metal, as it was initially designed for running in a virtualized environment only. With help from the designer of IncludeOS, Alfred Bratterud, we were able to build an ISO-image and load it on a server. In order to make the experiments as realistic as possible, we chose to run the experiments on a Dell PowerEdge R630 Rack Server with two Intel Xeon E5-2699 processors each containing 20 hyperthreading cores. An essential part of the project was to measure the time spent by the experiment workloads and the RDTSCP processor instruction that returns the number of cycles since the last reset was used for this purpose.

3.1 Compiling IncludeOS

The first step was to get hold of the IncludeOS source code and compile an ISO image with the STREAM benchmark. When running the experiments an IncludeOS version from an experimental branch updated in February 2017 was used. This revision of IncludeOS was compiled using g++ and could be executed successfully on bare metal. Later changes in the code requires a recent version of the Clang compiler, and for some reason the changes in the source code made it impossible to get any output on bare metal. To check out exactly the same revision from the repository, one can use the following git commands:

```
$ git clone https://github.com/fwsGonzo/IncludeOS/ ~/IncludeOS
$ git reset --hard b5087124aae0dd468434d7e60212fb5ff2571a0b
```

IncludeOS was then compiled without setting any CC or CXX variables in the shell, which led to g++ being used as the default compiler. The variable `$INCLUDEOS_PREFIX` must point to where the compiled binaries

and libraries will be installed. In addition it is a good idea to modify the path and add `$INCLUDEOS_PREFIX/bin` to it. An install.sh script is provided with IncludeOS that will execute the required steps in order to get a working IncludeOS environment.

A typical example of an IncludeOS service is shown below. This is the file `service.cpp` which is the main source file of an IncludeOS image. The code shows that the OS part of the program is literally included at the beginning. The file `stream.cpp` contains the actual workload and is compiled separately.

```
$ export INCLUDEOS_PREFIX=~/includeos/
$ export PATH=$PATH:$INCLUDEOS_PREFIX/bin
$ cd ~/IncludeOS && ./install.sh
```

To measure the memory performance the STREAM memory benchmark [15] was used. It is already included as an example project in the IncludeOS-directory. The followning is the file `service.cpp` which defines the IncludeOS service. It literally includes the OS-part needed to build a combination of the service and a kernel into a single executable.

```
// This file is a part of the IncludeOS unikernel - www.includeos.org
//
// Copyright 2015 Oslo and Akershus University College of Applied Sciences
// and Alfred Bratterud
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//     http://www.apache.org/licenses/LICENSE-2.0
//

#include <os>
#include <iostream>

void Service::start()
{
   printf("Running STREAM benchmark\n");
   extern int stream_main();
   stream_main();
}
```

The goal of the thesis was to find differences in performance and latency when compared to compiling and running the same `stream.cpp` code using Linux. One important option only possible using IncludeOS was to turn off interrupts. This was in order to run the workload as efficiently as possible on physical hardware. IncludeOS was tested with and without this feature and it was done by simply adding an instruction using inline assembly in the `service.cpp` source code of IncludeOS.

```
printf("Turning off interrupts (asm(\"cli\")\n");
asm("cli");
```

A simple way to test the benchmark in a virtual machine is to build and execute it using the IncludeOS boot command. Adding verbose output will also tell us something about the compiler flags being used. It is essential to compile the STREAM binary with exactly the same options in Linux to get a fair comparison.

```
$ cd ~/IncludeOS/examples/STREAM
$ boot -cv .
```

The following is the part of the output from the `boot -cv .` command containing the c++ flags

```
> /usr/bin/c++   -DARCH=\"x86_64\" -DARCH_x86_64
 ↪  -DINCLUDEOS_SINGLE_THREADED -DPLATFORM=\"x86_pc\"
 ↪  -DPLATFORM_x86_pc -D_LIBCPP_HAS_NO_THREADS
 ↪  -I/home/martink/includeos/includeos/api/posix
 ↪  -I/home/martink/includeos/includeos/x86_64/include/libcxx
 ↪  -I/home/martink/includeos/includeos/x86_64/include/newlib
 ↪  -I/home/martink/includeos/includeos/x86_64/include
 ↪  -I/home/martink/includeos/includeos/api
 ↪  -I/home/martink/includeos/includeos/include
 ↪  -I/home/martink/includeos/include  -MMD -msse3
 ↪  -mfpmath=sse -m64 -fstack-protector-strong
 ↪  -DOS_TERMINATE_ON_CONTRACT_VIOLATION -D_GNU_SOURCE
 ↪  -DSERVICE="\"stream_example\""
 ↪  -DSERVICE_NAME="\"STREAM Memory Benchmark Service\""
 ↪  -Wall -Wextra -nostdlib -fno-omit-frame-pointer -c
 ↪  -std=c++17   -o CMakeFiles/service.dir/service.cpp.o
 ↪  -c /home/martink/IncludeOS/examples/STREAM/service.cpp
```

We noticed that the compiler flags did not use any optimization mode, like -O2. This switch had to be added to get reasonable performance. This can be either done globally for compiling all IncludeOS projects, or locally for just a single project.

```
# globally in a cmake-file included by all projects
cat ~/includeos/includeos/post.service.cmake
set(OPTIMIZE "-O2")

# or locally an option can be added to CMakeLists.txt
# in the same directory as source code for the service
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -O2")
```

After adding the option to CMakeLists.txt just for the STREAM-project, it was then recompiled. The compile output from `boot -cv .` was then like this:

```
> /usr/bin/c++   -DARCH=\"x86_64\" -DARCH_x86_64
 ↪  -DINCLUDEOS_SINGLE_THREADED -DPLATFORM=\"x86_pc\"
 ↪  -DPLATFORM_x86_pc -D_LIBCPP_HAS_NO_THREADS
 ↪  -I/home/martink/includeos/includeos/api/posix
 ↪  -I/home/martink/includeos/includeos/x86_64/include/libcxx
 ↪  -I/home/martink/includeos/includeos/x86_64/include/newlib
 ↪  -I/home/martink/includeos/includeos/x86_64/include
 ↪  -I/home/martink/includeos/includeos/api
 ↪  -I/home/martink/includeos/includeos/include
 ↪  -I/home/martink/includeos/include  -MMD -msse3
 ↪  -mfpmath=sse -m64 -fstack-protector-strong
 ↪  -DOS_TERMINATE_ON_CONTRACT_VIOLATION -D_GNU_SOURCE
 ↪  -DSERVICE="\"stream_example\""
 ↪  -DSERVICE_NAME="\"STREAM Memory Benchmark Service\""
 ↪  -Wall -Wextra -nostdlib -fno-omit-frame-pointer -c
 ↪  -std=c++17 -O2   -o
 ↪  CMakeFiles/service.dir/service.cpp.o -c
 ↪  /home/martink/IncludeOS/examples/STREAM/service.cpp
```

The output shows that the compiler options for g++ now include the -O2 flag. When successfully built, a resulting IncludeOS binary should appear in the build directory. In our case it was named `build/stream_example` This binary can be run directly in a virtualized environment, like qemu. The boot command can be run directly to execute a qemu environment with parameters specified an JSON-file, usually vm.json in the project directory

```
root@intel1:~/IncludeOS/examples/STREAM# cat vm.json
{
  "mem" : 4096,
  "cpu" : {"model": "host"}
}
```

These options will make the boot command create a qemu VM with the specified amount of memory and the CPU model passthrough feature - essentially by identifying the CPU in the VM as exactly the same as on the physical hardware. A simple test of the benchmark is done by just executing `boot .`, but in general if only a compilation is required and executing qemu is unnecessary, then the appropriate command is just `boot -b .`

Sample output when running IncludeOS with STREAM in a qemu vm:

```
martink@intel1:~/IncludeOS/examples/STREAM$ boot .
....
....
....
Found /home/martink/includeos/includeos/chainloader Type:  ELF 32-bit
LSB executable, Intel 80386, version 1 (SYSV), statically linked, stripped
[ WARNING ] Running with sudo
```

```
[sudo] password for martink:
================================================================================
 IncludeOS v0.9.3-5077-gb508712 (x86_64 / 64-bit)
 +--> Running [ STREAM Memory Benchmark Service ]
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Running STREAM benchmark
-------------------------------------------------------------
STREAM version $Revision: 5.10 $
-------------------------------------------------------------
This system uses 8 bytes per array element.
-------------------------------------------------------------
Array size = 10000000 (elements), Offset = 0 (elements)
Memory per array = 76.3 MiB (= 0.1 GiB).
Total memory required = 228.9 MiB (= 0.2 GiB).
Each kernel will be executed 100000 times.
-------------------------------------------------------------
```

3.2 Running IncludeOS on bare metal

The next step was to run the IncludeOS binary on physical hardware. To make it run on bare metal is not trivial, since writing drivers from ground up is a huge effort. Being an OS developed completely from scratch, where the main objective is to run it virtualized in the cloud, it is reasonable to spend the effort on virtualized drivers. Virtualized hardware is quite generic in its nature, making it possible to maintain just a few drivers. This is in contrast with bare metal, where the amount of drivers is virtually unlimited.

First of all, the hardware support of IncludeOS is limited to serial and VGA output. However, VGA is not desirable for parsing output. One option could be to send the benchmark data through the network, but IncludeOS has no hardware support for any physical network cards, it only supports Virtio and vmxnet3 drivers used for virtualization. Networking on physical hardware would require an IncludeOS network driver written from scratch for just the specific hardware. In addition, even with networking available, some extra setup for a network/server application is necessary.

However, IncludeOS has support for output to a serial console and this is enabled by default. The Dell server used in the experiments is configured with out-of-band-management [16]. Most hardware manufacturers use various names for such devices, Dell refers to it as iDRAC. Essentially this is a small device with its own cpu, controller and networking that works independently of the server, with its own physical connection and IP address. In broad terms, a tiny computer controls the server, providing services like management of the hardware, monitoring and remote access with a full view of the server monitor, keyboard input, remote shutdown and reboot etc. The iDRAC itself can be accessed through a web interface making it convenient to manage the server. In addition iDRAC has a very

useful feature for serial console redirection, making it possible to access the serial port over a network.

Another useful feature provided by the iDRAC is booting from virtual hardware. By uploading a CD-ROM or removable disk image in the web interface, the server can then boot from it remotely. This makes running different tests much easier, everything can be done remotely without physically swapping the server's disks or images.

Using IncludeOS' multiboot support, Qemu/KVM can boot directly without a boot loader. On bare metal a boot loader is necessary, and one option is to use GRUB boot loader. A utility named `grub-mkrescue` is provided with GRUB, which assists with the creation of a bootable CD-ROM image (ISO) from a directory that contains the binary to be booted. If the utility is installed, the process can automated with a simple script.

`grub-mkrescue` expects a special directory structure to create an ISO from. This can be done with `mkdir -p iso/boot/grub`. The IncludeOS binary is copied to the `iso/boot/` sub directory, and a simple GRUB configuration file is placed in the `iso/boot/grub/` directory. This process is described in the GRUB manual [17]

The GRUB configuration file configures GRUB to boot a single menu entry and specifies the timeout to 0, speeding up the boot process.

```
# cat iso/boot/grub/grub.cfg
set default=0
set timeout=0

menuentry "IncludeOS/STREAM 100 times" {
  multiboot /boot/stream_example
}
```

Once an IncludeOS binary is compiled, it is copied over to the `iso/boot/` directory. The next step is to run `grub-mkrescue` to create a bootable ISO with the content. The ISO-file is transferred to a desktop computer, and then uploaded through the iDRAC web interface. A small helper script to simplify the process:

```
#! /bin/bash

cp build/stream_example iso/boot/
grub-mkrescue -o inc.iso iso
scp -4 inc.iso   user@desktop.example.com:~/
```

Once the ISO-file is copied back to a desktop computer, it is uploaded through the iDRAC web interface. The next step is to launch a Virtual

Console from the web interface. This can simply be done from the overview screen in iDRAC, and choosing launch a Virtual Console, as shown in Fig. 3.1. The same server overview screen can be used to reboot the server. There are more ways to achieve this, like through IPMI.

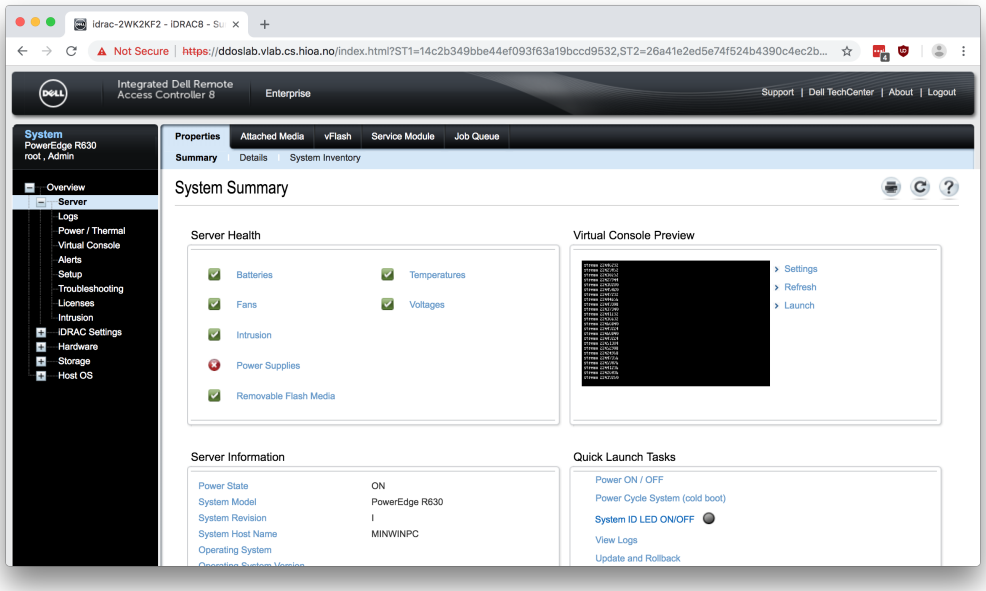

Figure 3.1: Dell iDRAC - launch virtual console

In Fig. 3.2 an ISO image is uploaded from a local computer, and this image is mapped as a virtual CD-ROM. The server can then be rebooted and started up from this image.

It is also possible to map a virtual CD-ROM image from the shell. This is convenient for a totally headless operation. Dell provides management utilities as part of their OpenManage software. For Ubuntu a utility called `vmcli` is provided as part of OpenManage. The software can be quite easily installed from Dell's Linux repositories [18] For remote mapping only the package `srvadmin-idrac-vmcli` was necessary. Typical usage is like this:

```
$ vmcli -r idrachost -u user -p pass -c /home/martink/inc.iso
```

The image remains mapped as long as the utility is running. Further management of the server from the shell was necessary, since the server was to be rebooted several times and benchmark results fetched through a redirected serial console. One solution is to use IPMI. Intelligent Platform Management Interface [19] is an open standard for out-of-band-management, monitoring, logging and recovery. It is supported by most large system vendors and provides management and monitoring independently of the host CPU, firmware and operating system. Being standardized, it can be applied across different hardware. There is an open source utility named `ipmitool` for managing and configuring devices from

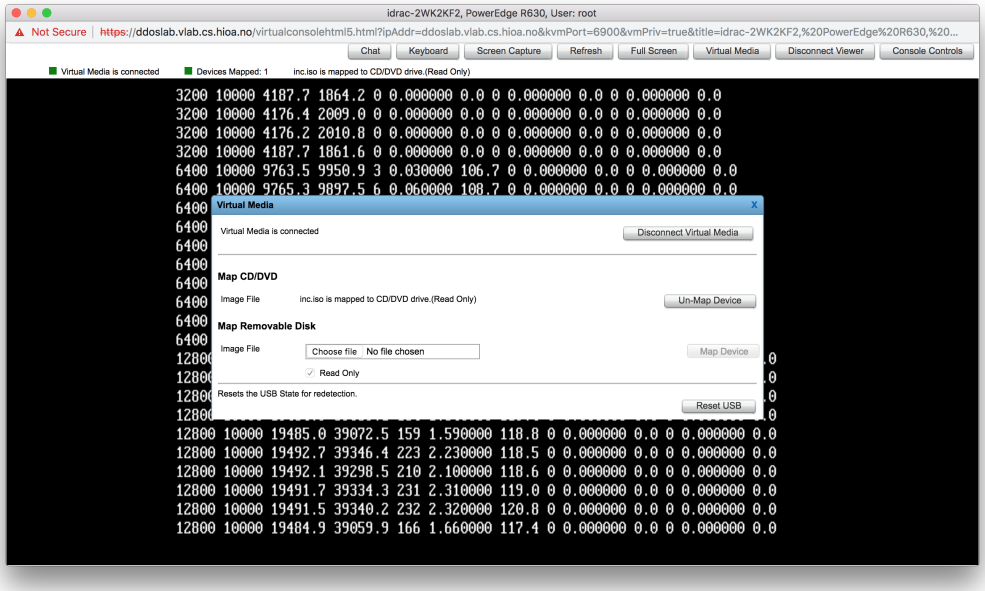

Figure 3.2: Dell iDRAC - map a virtual CD-rom

the shell.

Just a few commands were used, one for rebooting the server and another one for activating the serial console. The communication is done through SSL/TLS to the iDRAC, no further ports need to be opened in a firewall. When omitting the password option `-P`, a password can be entered interactively.

```
$ ipmitool -I lanplus -H idrachost -U user -P pass chassis power reset
$ ipmitool -I lanplus -H 192.168.0.120 -U root sol activate | tee log.txt
$ ipmitool -I lanplus -H 192.168.0.120 -U root sol deactivate
```

In the listing above a warm boot is performed, and serial console (SOL) is activated. The console output is logged to log.txt. The STREAM benchmark and other compiled C++ programs used in the experiments are mostly run from the console and output text only. This is practical for later parsing and statistical analysis.

Before performing a reboot, it is possible to set the boot device through IPMI, which is very useful for comparing runs between a CD-ROM image with IncludeOS and Linux. However, we just changed the boot sequence making the server start up from an IncludeOS image if it was mapped, and booting Linux otherwise.

3.2.1 RDTSCP

It was important to accurately measure how much time the processor spends executing a given workload. In order to do this, the RDTSCP assembly instruction was called before and after the workload instructions of the code, giving the exact time used.

RDTSCP - Read Time-Stamp Counter and Processor ID is a processor instruction that returns the number of cycles since the last reset. The counter is incremented on every CPU cycle, even if the CPU is halted, and the instruction puts it into the EDX:EAX register. It is described in chapter 17.15 of the Intel® 64 and IA-32 Architectures Developer's Manual: Vol. 3B [20]. In general, most x86 CPUs implement the TSC instruction, but RDTSCP is a serialized version of it that reads the timestamp from the `IA32_TSC` register (into EDX:EAX) as well as a signature value from the `IA32_TSC_AUX` register (into ECX) in an atomic way. This ensures that no context switches occur when reading the values. This is important on modern CPUs that support Out-of-branch-execution, as they differ from the RDTSC instruction.

However, care has to be taken when using the instruction. First of all, multicore CPUs have a different counter for each core, and those are not in sync. Ensuring that the process is pinned to certain core/CPU is one strategy. In addition, frequency scaling functions of the CPU should be disabled. Intel ensures that the counter is incremented at a constant rate, but this rate could differ from the core-clock to bus-clock ratio or maximum resolved frequency set at boot time. Intel also states in chapter 18.18.2 that the maximum resolved frequency is not necessarily the same as the processors base frequency. The TSC operate at close to the maximum non-turbo frequency. The exact formula is given in chapter 18.18.3 of the Intel manual.

Chapter 4

Results

4.1 Prerequisites for hardware and OS configuration

All the experiments were performed with the server running at maximum CPU frequency with speedstep disabled. This was to ensure that all the experiments were performed under the same conditions without varying CPU speed and power management. Fig. 4.1 is an overview of the different BIOS settings. The CPU was set to maximum frequency and with power

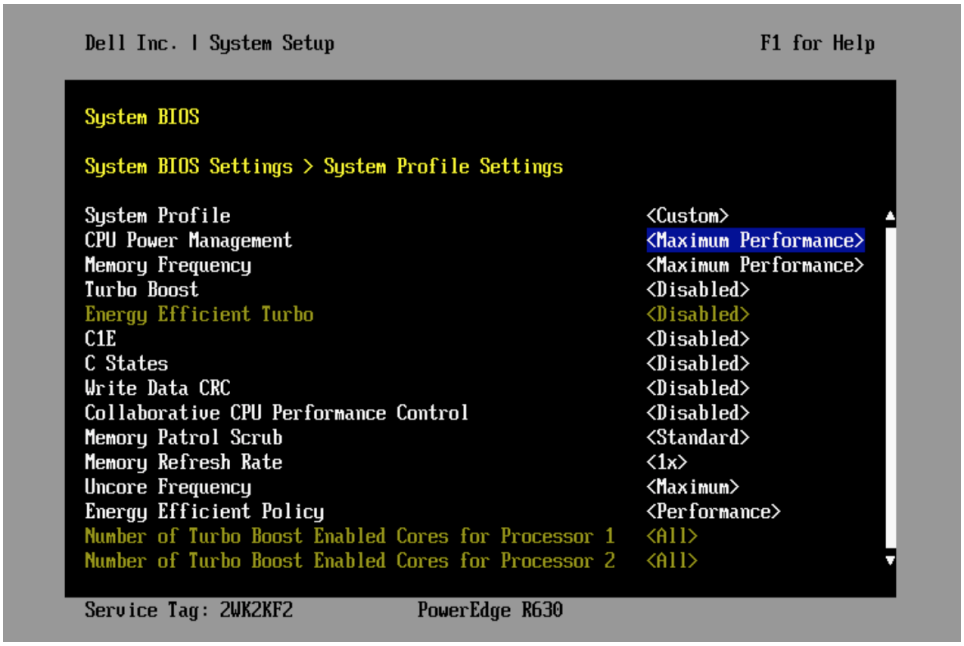

Figure 4.1: Configuration of CPU frequency and power management in the Dell BIOS setup.

management disabled. All features like automatic adjustment of CPU frequency like turbo boost were disabled, in addition to C1 power states and making sure that memory ran at maximum frequency.

Linux may schedule processes to run on different CPUs. The `isolcpus` kernel option was used to isolate cores from running processes, and pinning the experiment to specified cores. This isolates the cores from SMP load balancing and general scheduling algorithms of user-threads. By using this feature no user processes and threads can be run on the isolated cores, except the specified ones, although kernel threads and system services may still be scheduled to run on those cores, like timer interrupts and IRQ handlers.

4.2 CPU bound experiments

The first part of performance benchmarking was done with CPU bound experiments running on both IncludeOS and Linux. The experiments consisted of calculating the number of prime numbers below a given number. A small assembly routine and a C++ program was written for this purpose. The C++ program was used to run a specified number of iterations, running the same experiment a large number of times. When all the runs were done, the collected data with accompanying statistics were printed to the screen.

The prime number calculations were done in the assembly routine to avoid any kind of memory access – using registers only. This assures that no memory usage at all takes place and the experiments measure delays which are not caused by cache or memory operations. The elapsed time was measured in cycles using the RDTSCP instruction to avoid unnecessary instructions in between, and to avoid out-of-order execution. In essence, the assembly routine was the real experiment and the main program just a wrapper for automated execution.

```
#define RUNS    100
#define NTIMES  10000000

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <stdint.h>

// assembly routine for calculating prime numbers
// and measuring time in number of cycles
extern "C" {
   uint64_t assum(int max);
}

int stream_main()
{
    prime=22;
```

```
        // Running experiment NTIMES
        for (k=0; k<NTIMES; k++)
          {
             cycles[k] = assum(prime);
          }

}
```

The code excerpt above is from the main program performing a calculation in CPU cycles for all prime numbers below number 22. The same experiment was executed 1 million times using IncludeOS, IncludeOS with interrupts turned off and Linux.

Fig. 4.2 shows the result of this experiment where the number of cycles of each of the experiments is recorded.

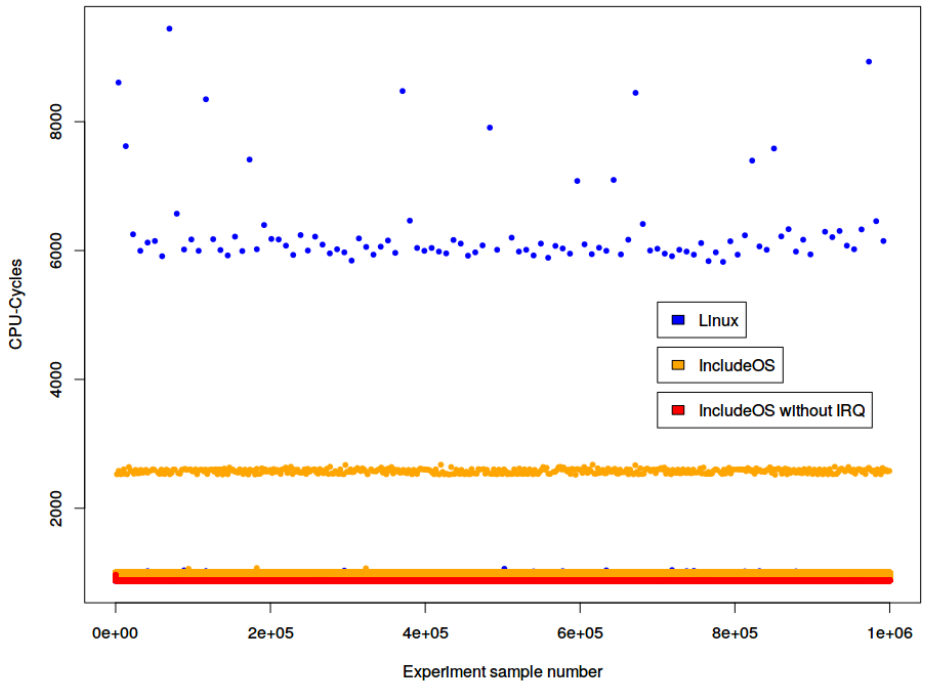

Figure 4.2: The noise of Linux and IncludeOS compared to IncludeOS with interrupts turned off when running a CPU bound workload. Results from 1 million experiments.

The figure shows a similar output to the graphs with delays presented in the [13] paper. The numbers for both IncludeOS and Linux are quite consistent. Running on Linux gives the biggest spread in performance and with the largest delays. This is due to operating system noise, like other tasks running at the same time or interrupts. There is a minor deviation with interrupts turned on for IncludeOS, but the deviation is always in the same range and is very linear.

IncludeOS with interrupts turned off, shows a very linear performance almost without any kind of delays, with just a handful of exceptions during a million experiments. A closer look at how much time the CPU bound experiments used is shown in in Fig. 4.3 which is a view zoomed in on the results from the first 500 experiments. The graph shows the interval

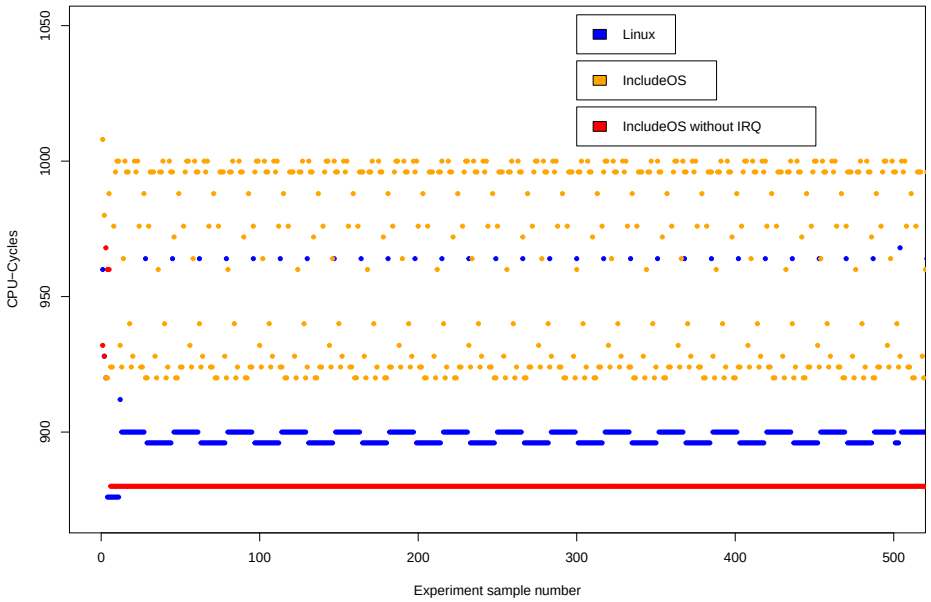

Figure 4.3: The noise of Linux and IncludeOS compared to IncludeOS with interrupts turned off when running a CPU bound workload. The results from the first 500 experiments are shown.

where most of the values occurred when calculating prime numbers below 22, without the big delays. On average the experiments in Linux used 906.263 cycles, with a 90% confidence interval of +- 0.006, IncludeOS with interrupts used 964.618 cycles, with a 90% confidence interval of +- 0.003 and IncludeOS without interrupts used 880 CPU cycles with a 90% confidence interval of 0. Almost all the runs on IncludeOS without interrupts were completed without any system noise. It is interesting to note that Linux had overall slightly lower latency than IncludeOS with interrupts.

The Fig. 4.4 is a histogram showing the frequency of CPU cycles spent in both IncludeOS and Linux, and is the overall distribution showing the noise of both Linux and IncludeOS. Even when completing 1 million experiments, then by far most results were unaffected by system noise. In order to show both the average results and delays in the same graph, the vertical axis is logarithmic. Results with system noise are just a small number of the total runs. However, as shown in the histogram, the larger delays occurred only in Linux.

The results from IncludeOS without interrupts are worth a closer look,

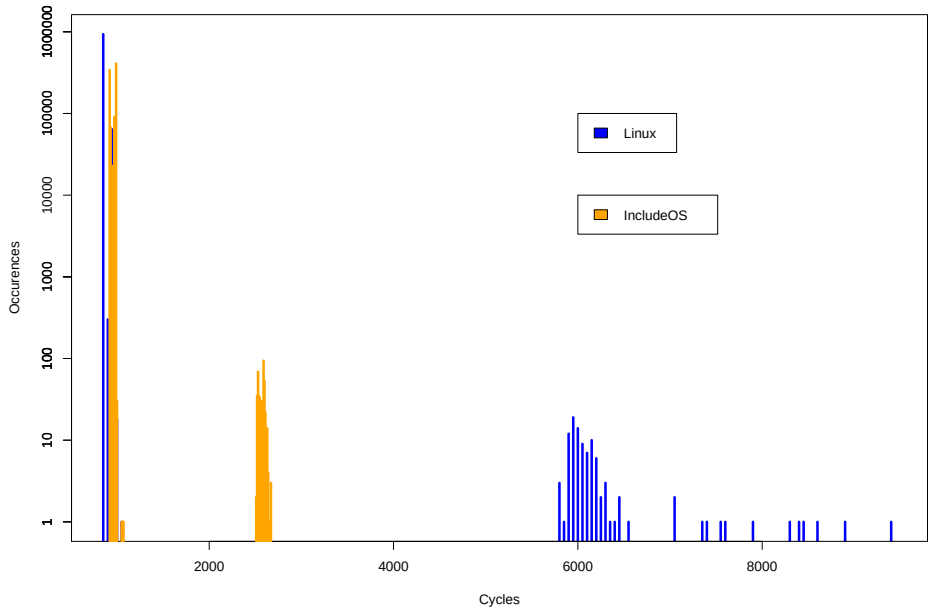

Figure 4.4: Histogram of the noise of Linux and IncludeOS when running a CPU bound workload. Results from 1 million experiments.

and Fig. 4.5 shows the distribution of CPU cycles spent with this unikernel. Here again the vertical axis is logarithmic, to emphasize that almost the whole million of experiments ran at 880 cycles with literally just a handful barely outside this number and for all practical concerns, no system noise.

The results from the varying lengths of prime number calculations can be seen in Fig. 4.6. This figure shows that CPU cycles scale very linearly with integer divisions when performing the CPU calculations for different prime numbers. When looking at 84 integer divisions, it required 1000 CPU cycles, and doubling the number of integer divisions results in 2000 CPU cycles.

```
int stream_main()
{
for (prime=10; prime<39; prime +=4)
   for (r=0; r<RUNS; r++)
     {
        // Running experiment NTIMES
        for (k=0; k<NTIMES; k++)
          {
             cycles[k] = assum(prime);
          }
    }
}
```

The code sample above shows the prime number calculations for prime

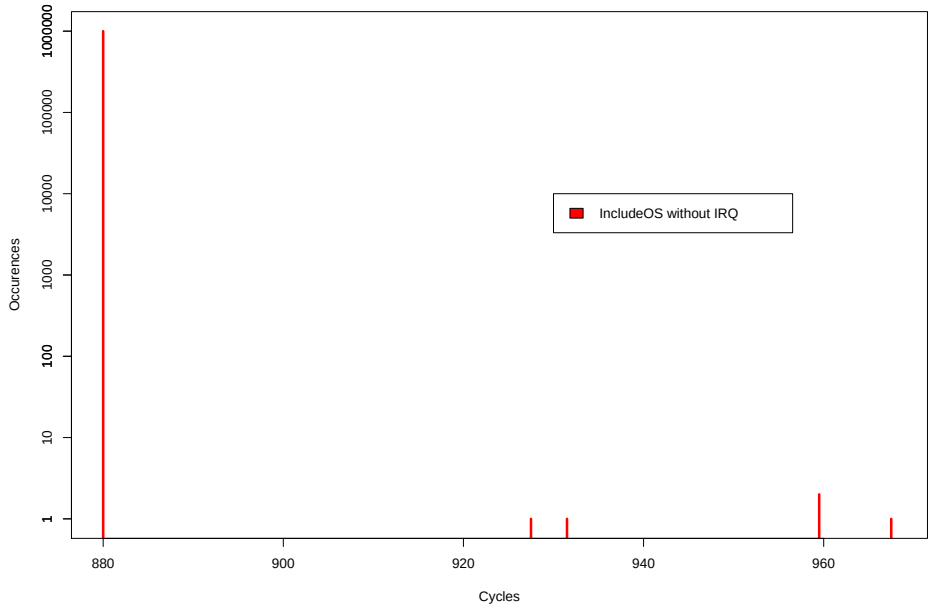

Figure 4.5: Histogram of the noise of IncludeOS with interrupts turned off when running a CPU bound workload. Results from 1 million experiments.

numbers below 39, starting at 10. In the first run, the upper limit was 10, sent as a parameter to `assum(prime)`. This would calculate all prime numbers below and including 10 and doing it 1 million times in 100 runs. Then the upper limit was increased by 4, now running a longer calculation and more CPU cycles spent but repeating it the same number of times.

The next three figures show the occurrences of delays for Linux and IncludeOS with CPU bound calculations of prime numbers. The number of delays is shown in conjunction with integer divisions, since integer divisions is more appropriate to show the linearity of CPU cycles rather than using just prime numbers on the x-axis. When looking at 4.7 the short delays occurred only in IncludeOS with interrupts enabled, but the larger delays in Fig. 4.8 and the very large delays in Fig. 4.9 appear only in Linux. There were more or less no delays in IncludeOS without interrupts and this is very consistent with the frequency of system noise shown in the earlier histograms.

Fig. 4.10 shows the total percentage of time the CPU spends doing work not related to the actual prime number calculation. This is obtained by adding together all the events where there is a delay of 400 cycles or more. These are considered to be some systematic kind of delay due to handling interrupt events by the operating system. For Linux, almost all of these delay events are part of delays larger than 2000 cycles. For IncludeOS, the total amount of time spent in delays is a bit larger than Linux, except for the very short jobs. But when there is a delay, it is always smaller than 2000 cycles. This is completely consistent with the results seen in the histogram

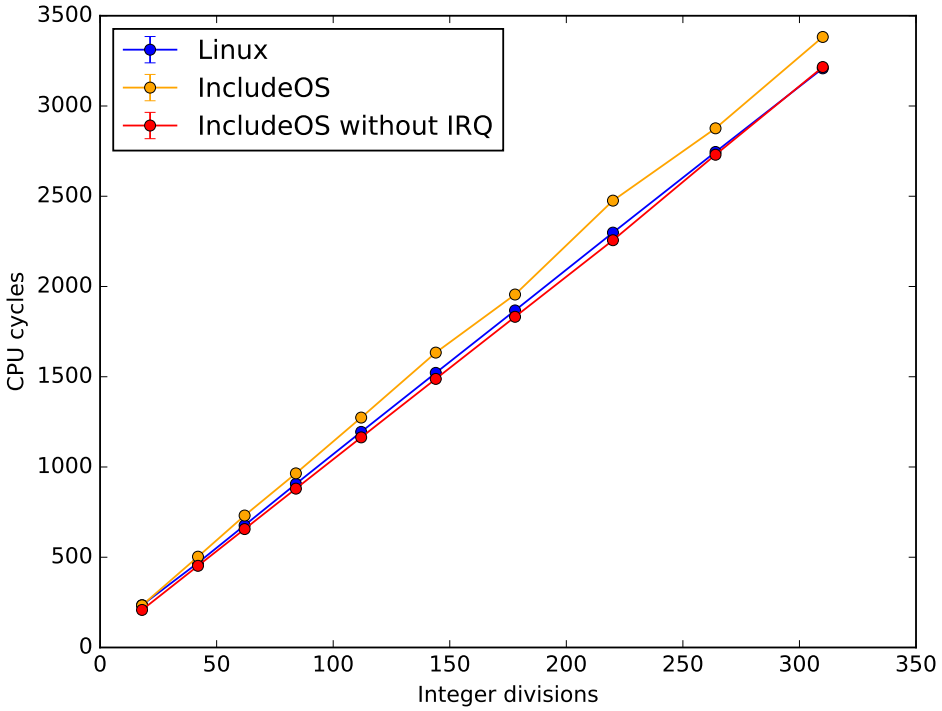

Figure 4.6: Number of CPU cycles used for a CPU bound calculation of prime numbers.

of Fig. 4.4. However, the histogram was based on 1 million experiments, while the averages which Fig. 4.10 are based on, is from 100 samples where each sample is based on 10 million experiments.

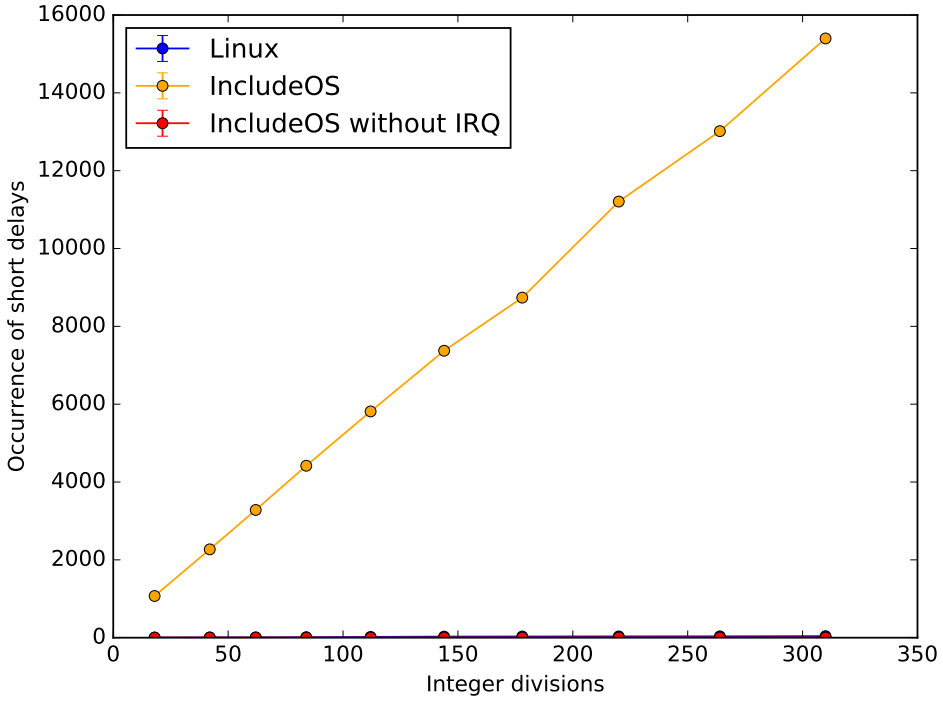

Figure 4.7: Occurrence of short delays between 400 and 2000 cycles for a CPU bound calculation of prime numbers.

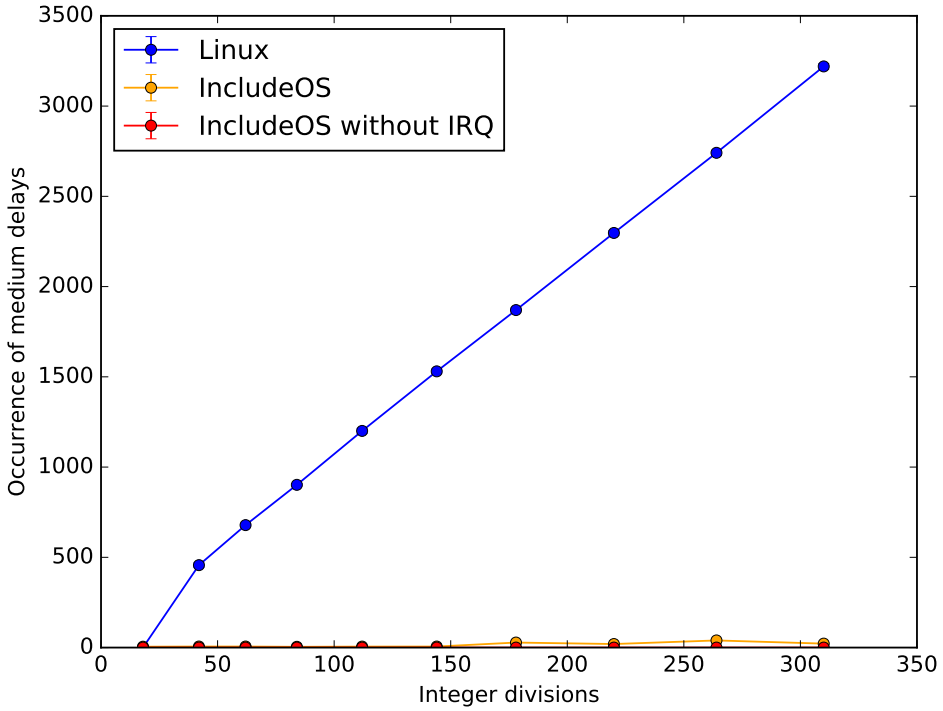

Figure 4.8: Occurence of medium sized delays between 2000 and 6000 for a CPU bound calculation of prime numbers.

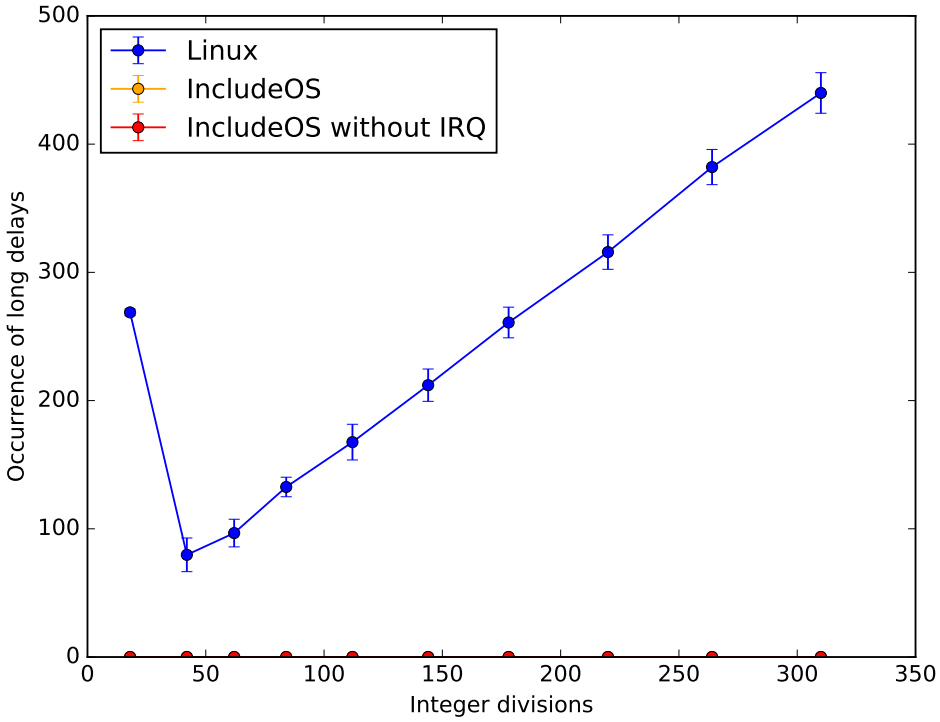

Figure 4.9: Occurrence of delays larger than 6000 cycles for a CPU bound calculation of prime numbers.

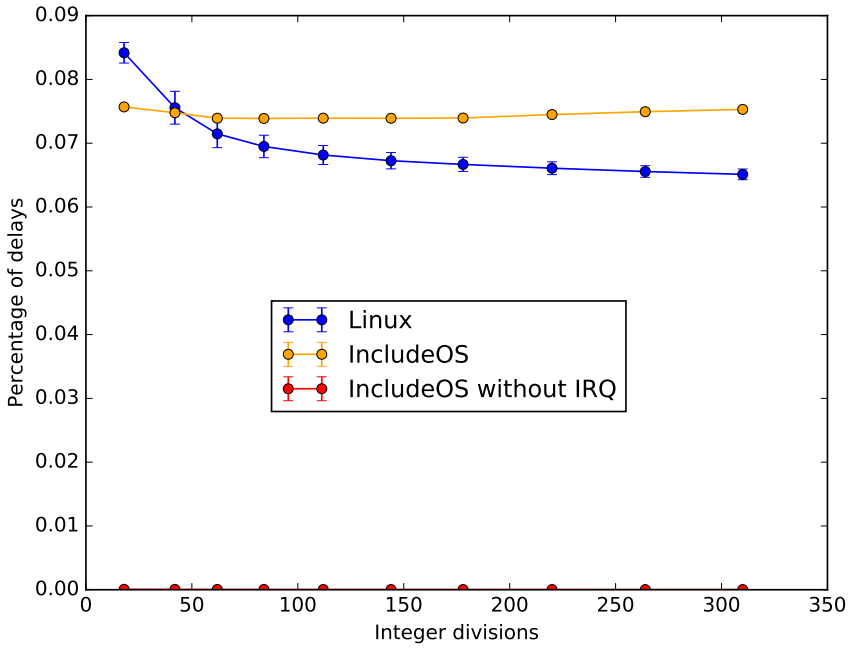

Figure 4.10: Percentage of execution time spent in delays larger than 400 cycles for a CPU bound calculation of prime numbers.

4.3 Memory experiments

Memory experiments were done with small C programs based on the STREAM memory benchmark [15]. Only one of the tests in the benchmark was used, essentially just a function that copy the content of a small array. The experiments were run 1 million times and elapsed time for each experiment was again measured with the RDTSCP assembly instruction.

Fig. 4.11 shows the number of CPU cycles of a million experiments. The

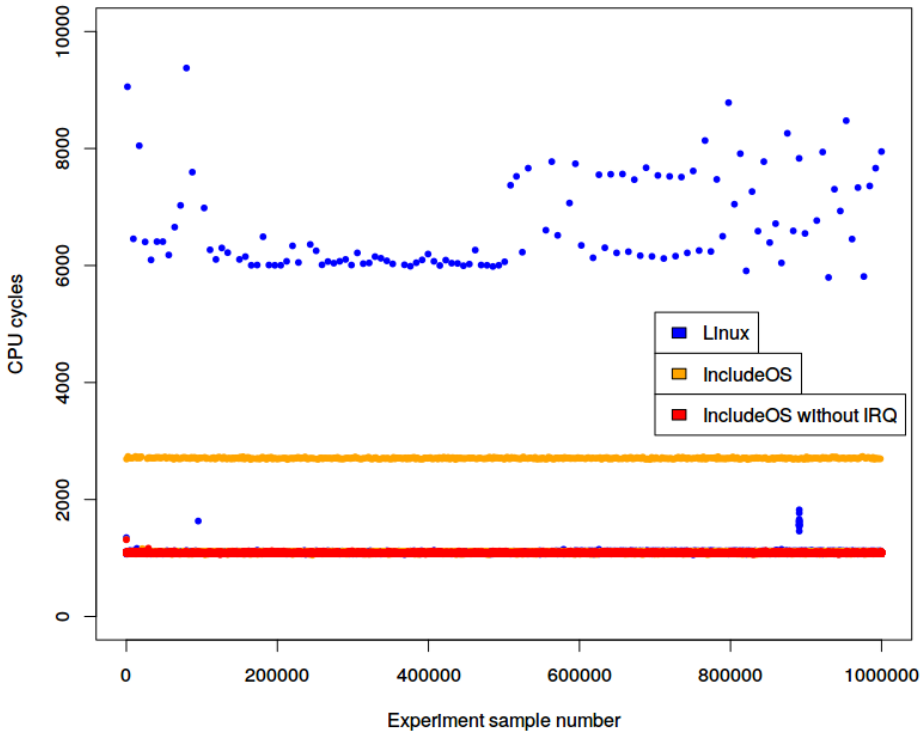

Figure 4.11: The noise of Linux and IncludeOS compared to IncludeOS with interrupts turned off when copying between two arrays. Results from 1 million experiments.

experiments were done by copying between arrays and the graph shows that IncludeOS is very consistent when doing memory operations, the number of CPU cycles is almost always the same. It takes somewhat longer and more CPU cycles to perform the memory operations as compared with the CPU bound experiments, this is as expected since accessing memory must be accessed outside the CPU. Interesting to note that IncludeOS with interrupts enabled, more or less always shows the same amount of system noise, there is no change between two states as in the CPU experiments. Every experiment just spends around 1000 cycles longer than with interrupts turned off.

Fig. 4.12 is a zoomed in view of 500 selected experiments. The trend

of the number of cycles measured varies a bit and the picture may look slightly different if choosing another 500 experiments.

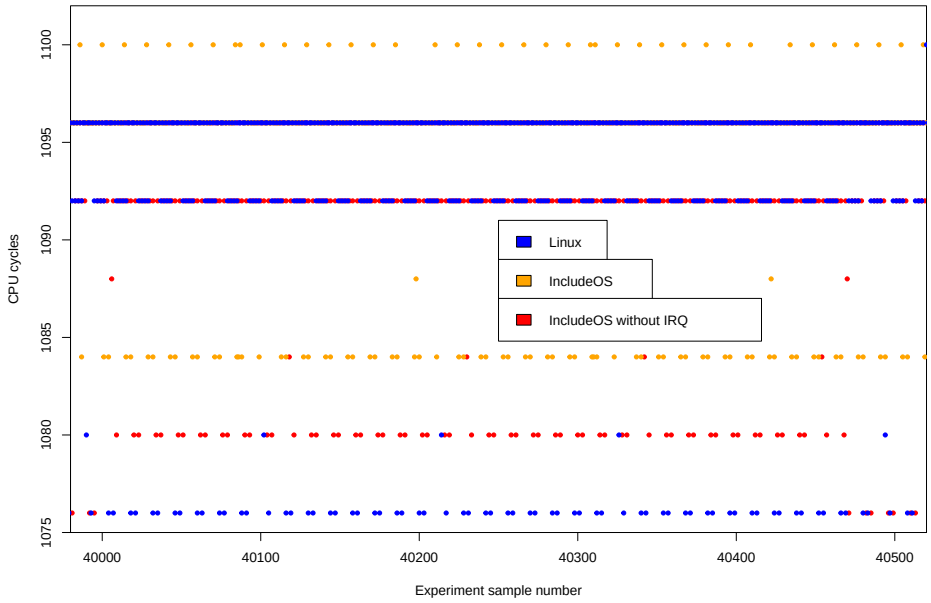

Figure 4.12: The noise of Linux and IncludeOS compared to IncludeOS with interrupts turned off when copying memory. The results from 500 selected experiments are shown.

The distribution of delays when doing memory operations is shown in Fig. 4.13 and shows the time spent in cycles for both IncludeOS and Linux. In order to show both the overwhelming amount of short delays and the longer ones in the same graph, the scale is logarithmic. Very similar to the CPU experiments shown in Fig. 4.4, the longer delays were only seen in Linux, and in IncludeOS almost all the experiments used the average amount of cycles without bigger spikes. Typically a larger delay in Linux of 10.000 cycles may add almost 5 microseconds in execution time to an already running calculation. In addition, Linux had the highest percentage of execution time spend on large delays and this is shown in Fig. 4.20

A closer look at the distribution of delays in IncludeOS with interrupts turned off is shown in 4.14, which shows an array copy operation performed 1 million times for an array size of 800 bytes. With interrupts turned off most of the experiments were done in around 1100 CPU cycles, with some deviation around this number. Still the results were quite consistent, just as when running the CPU bound experiments shown in 4.5 there are no large delays. The deviation in the memory experiments are probably due to cache.

The figures below show the occurence of delays when performing memory operations. Fig. 4.7 shows the occurrence of short delay events, while Fig. 4.17 is about the occurence of medium sized events. The last figure 4.18 show the occurence of delays larger than 6000 cycles, and those

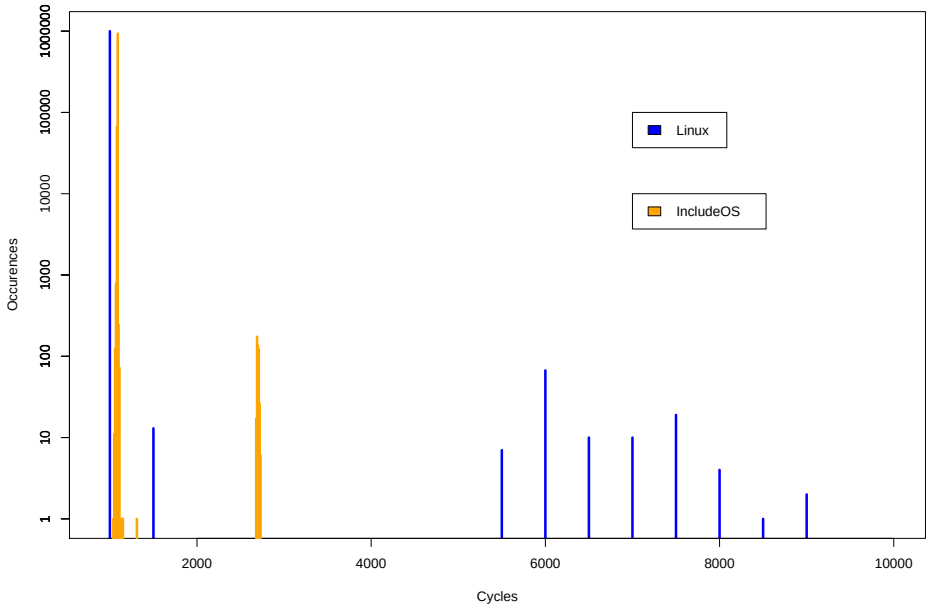

Figure 4.13: Histogram of the noise of Linux and IncludeOS when copying between two arrays. Results from 1 million experiments.

were only encountered in Linux.

Fig. 4.19 shows the total percentage of time that the CPU spends doing work in delays when copying data between two arrays in memory. This is obtained by adding together all the events where there is a delay of 400 cycles or more. As opposed to the case of CPU bound calculations, there are now some delay events even when interrupts are turned off in IncludeOS. This could be due to occasional cache misses forcing the CPU to wait for more data.

When comparing these results to the results of Fig. 4.10, the large Linux delays are roughly three times higher for memory based calculations. This difference is probably due to both variances in cache performance and due to paging in the Linux operating system.

When only showing delays of 2000 cycles or more in Fig. 4.20, there are only Linux events and the results looks very similar to the results of Fig. 4.10. If only events of delays of more than 2000 cycles were shown in that figure, only the Linux events would be present, as all IncludeOS events are for less than 2000 cycles.

In Fig. 4.10 one can see that the percentage of Linux delays in this range is quite similar when executing a CPU-bound workload. So it seems that the amount of such long delays is independent of the type of application run by the Linux operating system. Both memory-bound and CPU-bound workloads lead to the same amount of OS-noise.

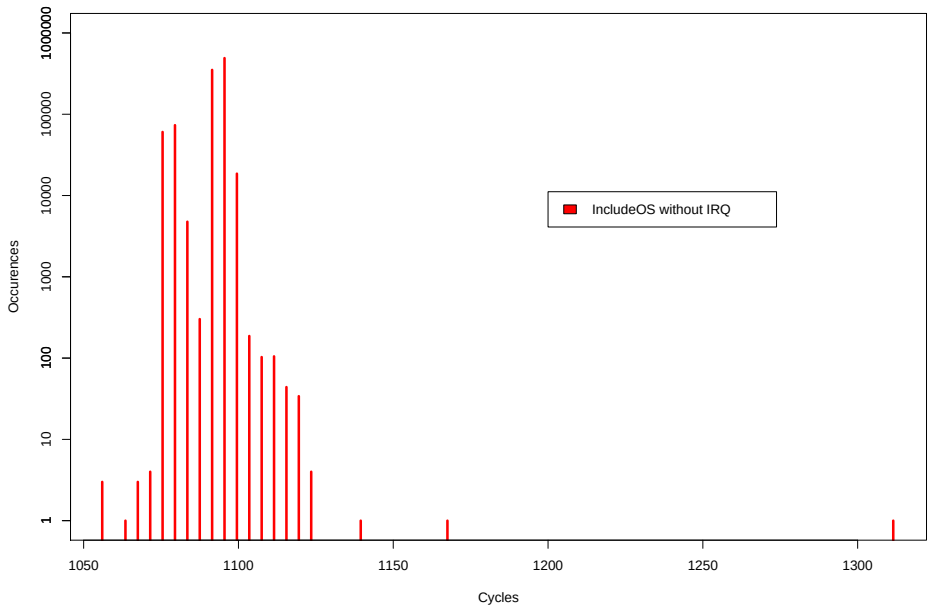

Figure 4.14: Histogram of the noise of IncludeOS with interrupts turned off when copying between two arrays. Results from 1 million experiments.

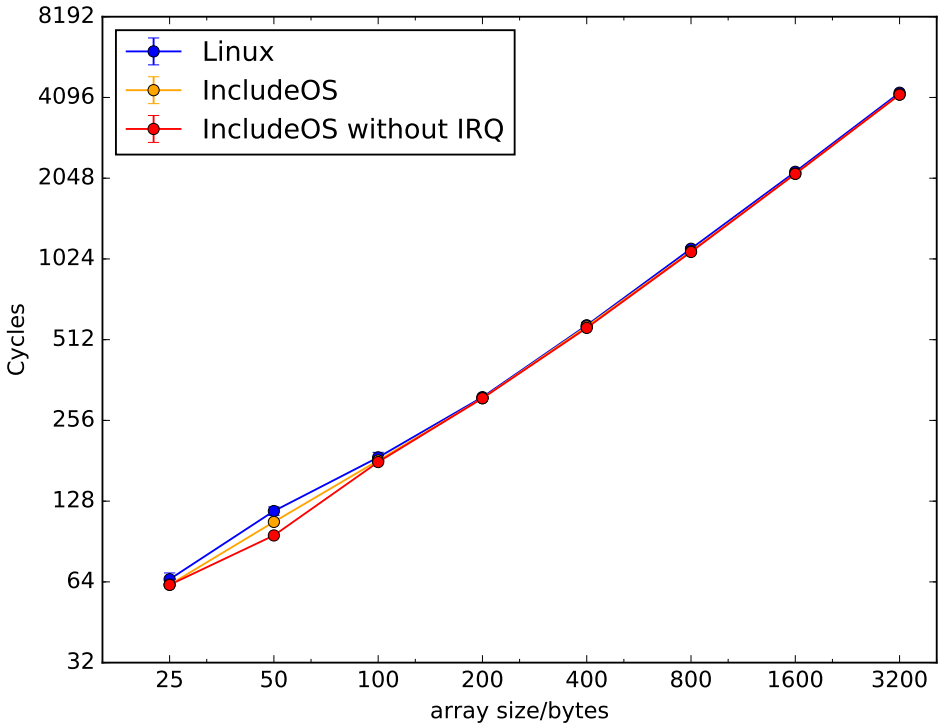

Figure 4.15: Number of CPU cycles used when copying between two arrays.

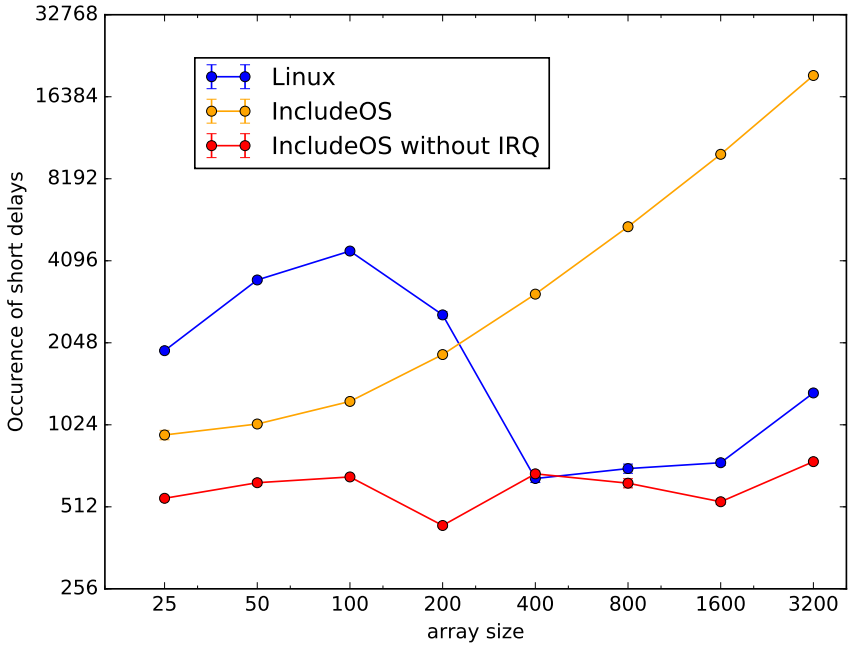

Figure 4.16: Occurence of short delay events between 400 and 2000 cycles when copying between two arrays.

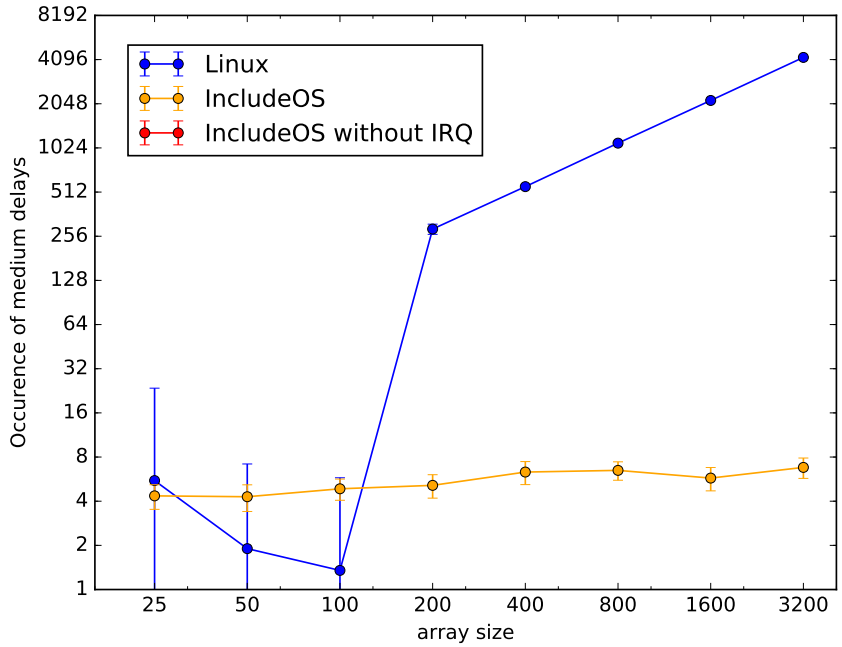

Figure 4.17: Occurence of medium sized delay events between 2000 and 6000 when copying between two arrays.

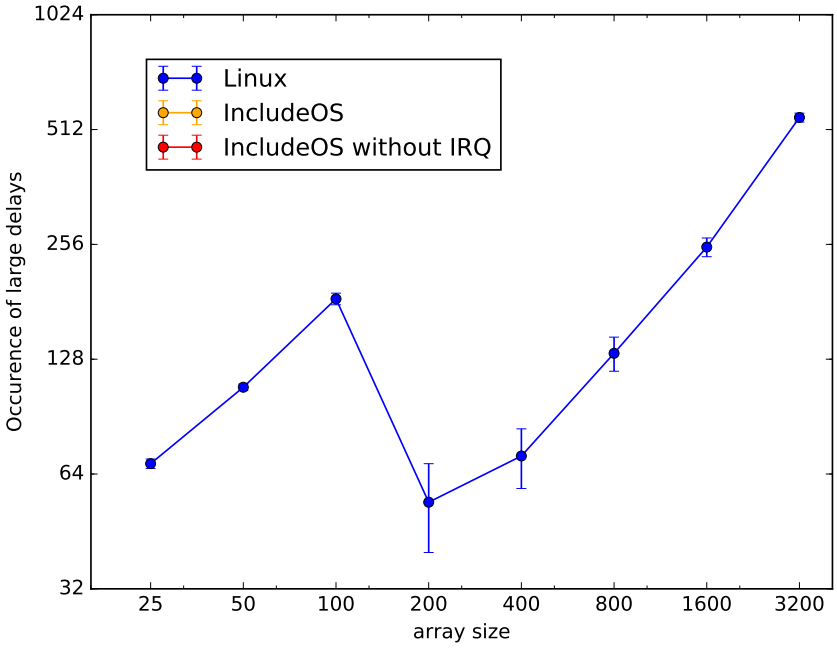

Figure 4.18: Occurence of delay events larger than 6000 cycles when copying between two arrays.

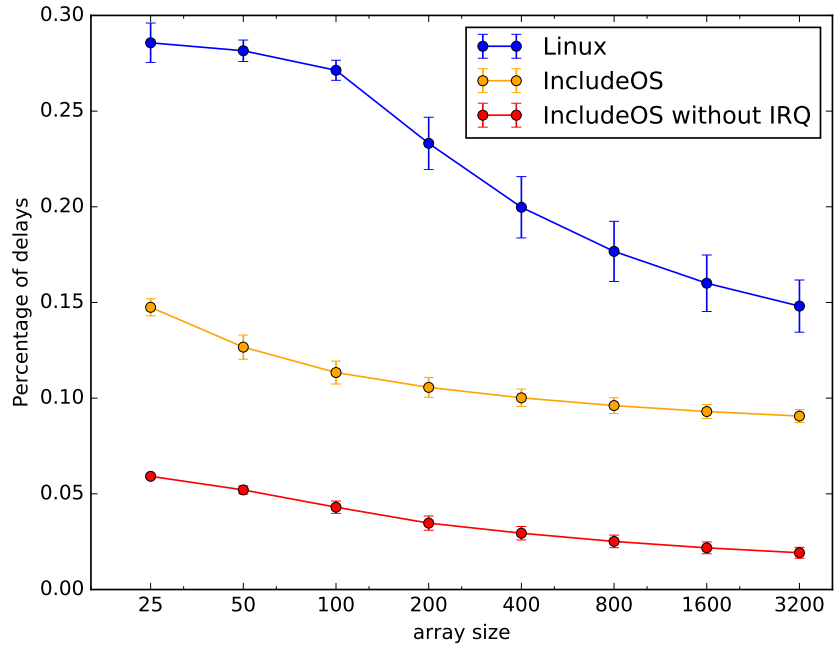

Figure 4.19: Percentage of execution time spent in delays larger than 400 cycles when copying data from one array to another.

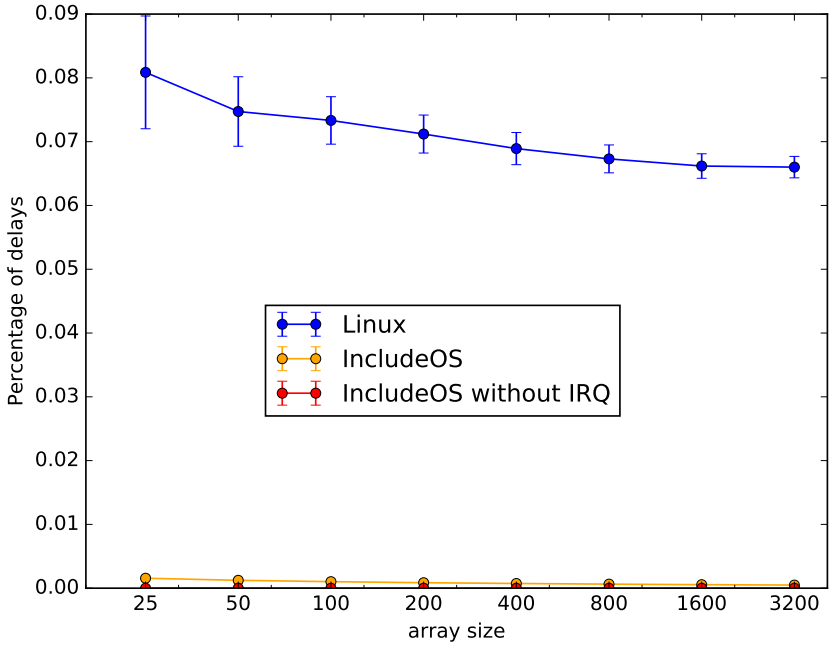

Figure 4.20: Percentage of execution time spent in delays larger than 2000 cycles when copying data from one array to another.

Chapter 5

Discussion

Looking at the different results from both the CPU and memory experiments, larger delays occur in Linux due to system noise. An unexpected delay like this can be critical for time sensitive applications. For example, in High Frequency Trading, delays of a few microseconds can result in a lost opportunity to complete a transaction. In the experiments of a CPU-bound process, the experiment corresponding to 84 divisions takes roughly 900 cycles to complete. From Fig. 4.10 one can see that in 0.07 percent of the cases, when making such a calculation on Linux, there will be a delay in the range 2000-6000 cycles. Given that such a transaction is performed by a High Frequency Trading algorithm, then there is almost a chance of one per thousand for a delay in the range of 2000 - 6000 cycles, which potentially may make one out of a thousand transactions fail.

Taking into account the same hardware as the experiments were run on, with a 2.2Ghz CPU being the same a 2.2×10^9 cycles per second, then 1000 cycles takes

$$\frac{1000\,cycles}{2.2 * 10^9 cycles/seconds} = 0.45 \times 10^{-6} seconds$$

Such a transaction would then take half a microsecond, and a delay of 2000 - 6000 cycles would add 1 - 3 microseconds to the calculation time of the transaction and this could mean that the competing algorithms would win a round of trading. For IncludeOS there would be similar delays, but they would all be less than 2000 cycles and hence less than a microsecond. Shielding the algorithm from interrupts by turning interrupts off in IncludeOS would avoid such delays completely.

Looking at the CPU bound experiments, IncludeOS with interrupts turned off, shows a very linear performance almost without any kind of delays, with just a handful of exceptions during a million experiments. The CPU cycles scale very linearly with integer divisions when performing the CPU calculations for different prime numbers. When looking at 84 integer divisions, it required 1000 CPU cycles, and doubling the number of integer divisions results in 2000 CPU cycles.

The number of delays is shown in conjunction with integer divisions, since integer divisions is more appropriate to show the linearity of CPU cycles rather than using just prime numbers on the x-axis. The short delays occurred only in IncludeOS with interrupts enabled, but the larger delays and the very large delays appear only in Linux. There were more or less no delays in IncludeOS without interrupts and this is very consistent with the frequency of system noise.

The memory experiments were done by copying between arrays and IncludeOS is very consistent when doing memory operations, the number of CPU cycles is almost always the same. It takes somewhat longer and more CPU cycles to perform the memory operations as compared with the CPU bound experiments, this is as expected since accessing memory must be accessed outside the CPU. Interesting to note that IncludeOS with interrupts enabled, more or less always shows the same amount of system noise, there is no change between two states as in the CPU experiments. Every experiment just spends around 1000 cycles longer than with interrupts turned off.

Very similar to the CPU experiments, the longer delays were only seen in Linux, and in IncludeOS almost all the experiments used the average amount of cycles without bigger spikes. Typically a larger delay in Linux of 10.000 cycles may add almost 5 microseconds in execution time to an already running calculation. In addition, Linux had the highest percentage of execution time spend on large delays.

In the memory experiments, there are now some delay events even when interrupts are turned off in IncludeOS. This could be due to occasional cache misses forcing the CPU to wait for more data.

With IncludeOS and interrupts turned off most of the memory experiments were done in around 1100 CPU cycles, with some deviation around this number. Still the results were quite consistent, just as when running the CPU bound experiments, there are no large delays. The deviation in the memory experiments are probably due to cache.

When comparing the large Linux delays to the CPU experiments and percentage of delays, they are roughly three times higher for memory based calculations. This difference is probably due to both variances in cache performance and due to paging in the Linux operating system.

If only events of delays of more than 2000 cycles were shown, only the Linux events would be present, as all IncludeOS events are for less than 2000 cycles. One can see that the percentage of Linux delays in this range is quite similar when executing a CPU-bound workload. So it seems that the amount of such long delays is independent of the type of application run by the Linux operating system. Both memory-bound and CPU-bound workloads lead to the same amount of OS-noise.

The large Linux delays are roughly three times higher for memory based calculations, as compared with CPU based calculations. This difference is probably due to both variances in cache performance and due

to paging in the Linux operating system.

It would be interesting to perform the tests on a very recent version of IncludeOS, and compare the results with the Februrary 2018 version used here. Many changes have been made to the codebase since then. Especially the results with interrupts enabled may be a little different.

In addition, testing a Linux kernel with one of the `CONFIG_NO_HZ*` options set might also be an interesting comparison to see how it affects interrupts and possibly yields less noise. The different `NO_HZ` kernel options are described in the Linux kernel documentation.

Chapter 6

Conclusion

Looking at the experiments, we can conclude that IncludeOS has a clear reduction in system noise compared with Linux, both in CPU bound and memory intensive applications. Although delays occur in IncludeOS with interrupts enabled, they are much shorter, and virtually non-existent with interrupts disabled.

Even with interrupts enabled, there is an essential reduction in system noise when using IncludeOS, the delays are short and systematic, and probably related to how IncludeOS deals with interrupts. However, running IncludeOS with interrupts disabled results in very low latency and almost no system noise at all. With Linux, interrupts are unavoidable, and even when taking measures like dedicating CPUs solely to the operating system, interrupts and system noise may appear due to SMP load balancing and scheduling of kernel threads and interrupt handlers. The larger delays were found in Linux during both CPU and memory bound experiments. In some cases such a delay may add up to 5 microseconds to an already running process, which may be unacceptable for time critical algorithms like those used in HFT trading.

Bibliography

[1] Juri Lelli et al. "Deadline scheduling in the linux kernel". In: *Software: Practice and Experience* 46.6 (2016), pp. 821–839.

[2] Irene Aldridge. *High-frequency trading: a practical guide to algorithmic strategies and trading systems*. Vol. 604. John Wiley & Sons, 2013.

[3] G Lins and T Lemke. "Soft Dollars and Other Trading Activities". In: *The New Financial Industry. Alabama Law Review* 2 (2016), p. 31.

[4] Tom C.W. Lin. "The new financial industry". English. In: *Alabama Law Review* 65.3 (2014), pp. 567–623. ISSN: 0002-4279.

[5] MIT Technology Review. *Trading Shares in Milliseconds*. 2009. URL: https://www.technologyreview.com/s/416805/trading-shares-in-milliseconds/ (visited on 01/21/2019).

[6] Dawson R Engler, M Frans Kaashoek, et al. *Exokernel: An operating system architecture for application-level resource management*. Vol. 29. 5. ACM, 1995.

[7] Anil Madhavapeddy et al. "Unikernels: Library Operating Systems for the Cloud". In: *SIGPLAN Not.* 48.4 (Mar. 2013), pp. 461–472. ISSN: 0362-1340. DOI: 10.1145/2499368.2451167. URL: http://doi.acm.org/10.1145/2499368.2451167.

[8] A. Bratterud et al. "IncludeOS: A Minimal, Resource Efficient Unikernel for Cloud Services". In: *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*. Nov. 2015, pp. 250–257. DOI: 10.1109/CloudCom.2015.89.

[9] IncludeOS AS. URL: https://www.includeos.com (visited on 01/31/2019).

[10] IncludeOS project. *Trading Shares in Milliseconds*. 2019. URL: https://github.com/hioa-cs/IncludeOS (visited on 01/31/2019).

[11] Avi Kivity, Dor Laor Glauber Costa, and Pekka Enberg. "OS v—Optimizing the Operating System for Virtual Machines". In: *Proceedings of USENIX ATC'14: 2014 USENIX Annual Technical Conference*. 2014, p. 61.

[12] Bjarne Stroustrup. *The C++ programming language*. 4th ed. Addison-Wesley, 2013, p. 10.

[13] Hakan Akkan, Michael Lang, and Lorie Liebrock. "Understanding and isolating the noise in the Linux kernel". In: *The International Journal of High Performance Computing Applications* 27.2 (2013), pp. 136–146.

[14] Hakan Akkan, Michael Lang, and Lorie M Liebrock. "Stepping towards noiseless Linux environment". In: *Proceedings of the 2nd international workshop on runtime and operating systems for supercomputers*. ACM. 2012, p. 7.

[15] John D. McCalpin. *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. Tech. rep. A continually updated technical report. http://www.cs.virginia.edu/stream/. Charlottesville, Virginia: University of Virginia, 2007. URL: http://www.cs.virginia.edu/stream/.

[16] Wikipedia. *Out-of-band management — Wikipedia, The Free Encyclopedia*. 2019. URL: https://en.wikipedia.org/w/index.php?title=Out-of-band_management (visited on 02/11/2019).

[17] GNU Project. *Making a GRUB bootable CD-rom*. 2019. URL: https://www.gnu.org/software/grub/manual/grub/html_node/Making-a-GRUB-bootable-CD_002dROM.html.

[18] Dell. *Dell EMC OpenManage Ubuntu and Debian Repositories*. 2019. URL: http://linux.dell.com/repo/community/openmanage/ (visited on 02/14/2019).

[19] Wikipedia. *Intelligent Platform Management Interface — Wikipedia, The Free Encyclopedia*. 2019. URL: https://en.wikipedia.org/wiki/Intelligent_Platform_Management_Interface (visited on 02/14/2019).

[20] Intel. *Intel® 64 and IA-32 Architectures Developer's Manual: Vol. 3B, chapter 17.15*. URL: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.pdf (visited on 02/15/2019).

Appendices

Appendix A

Source code

Code Listing A.1: Making a bootable ISO

```
cd examples/STREAM/

vim stream.cpp # change from 100000 to 10 rounds
boot .                # building and testing
    mkdir -p iso/boot/grub
cp build/stream_example iso/boot/

cat > iso/boot/grub/grub.cfg

set default=0
set timeout=30
#serial --unit=0 --speed=9600
#terminal_input serial; terminal_output serial

menuentry "IncludeOS/STREAM First successful version" {
multiboot /boot/stream_example

}

grub-mkrescue -o i1.iso iso

scp i1.iso   martink@intel2.vlab.cs.hioa.no:~/

# Entries in CMakeLists.txt

set(DRIVERS
   disable_paging
   vga_output
   vga_emergency
   # virtionet   # Virtio networking
   # virtioblock # Virtio block device
   # ... Others from IncludeOS/src/drivers
   )
```

Code Listing A.2: prime.cpp

```
#define RUNS     100
#define NTIMES   10000000

#include <stdio.h>
```

```
#include <stdlib.h>
#include <math.h>
#include <stdint.h>

extern "C" {
   uint64_t assum(int max);
}

static uint64_t cycles[NTIMES];

int stream_main()
{

   int k,r;

   FILE* stream_data = stdout;

   uint64_t sum=0,square=0;
   int countA,countB,countC,countD,prime;
   double aver,var,sd,aSize,bSize,cSize,dSize,d;
   //fprintf(stream_data, "Starter\n");
   fprintf(stream_data, "prime,NTIMES,aver,sdev,countA,aSize,countB,bSize,countC,cSize,

   for (prime=10; prime<47; prime +=4)
   for (r=0; r<RUNS; r++)
     {
        // Running experiment NTIMES
        for (k=0; k<NTIMES; k++)
          {
             cycles[k] = assum(prime);
          }
        // Stats for experiment:
        aver = 0;
        var = 0;
        square = 0;
        sum = 0;
        aSize = 0;
        bSize = 0;
        cSize = 0;
        dSize = 0;
        countA = 0;
        countB = 0;
        countC = 0;
        countD = 0;
        for (k = 0; k < NTIMES; k++) {
           sum += cycles[k];
           square += cycles[k]*cycles[k];
        }
        aver = sum/(1.0*NTIMES);
        var = square/(1.0*NTIMES) - aver*aver;
        sd = sqrt(var);

        for (k = 0; k < NTIMES; k++) {
                d = cycles[k] - aver;
                if (d > 150 && d <= 400)
                  {
                     countA++;
                     aSize += d;
                  }
```

```
                else if (d > 400 && d <= 2000)
                  {
                     countB++;
                     bSize += d;
                  }
                else if (d > 2000 && d <= 6000)
                  {
                     countC++;
                     cSize += d;
                  }
                else if (d > 6000)
                  {
                     countD++;
                     dSize += d;
                  }
          }
          if(countA > 0){aSize = aSize/(1.0*countA);}
          if(countB > 0){bSize = bSize/(1.0*countB);}
          if(countC > 0){cSize = cSize/(1.0*countC);}
          if(countD > 0){dSize = dSize/(1.0*countD);}
          fprintf(stream_data, "%d %d %.2lf %.1lf %d %.1lf %d %.1lf %d %.1lf %d %.1lf \n",prim
      }

   return 0;
}
```

Code Listing A.3: assum.s

```
.globl assum

assum:                  # Standard

push %rbx
# %rdi is first param
mov %rdi,%r11

rdtscp

shl $32,%rdx    # %edx contains left 32 bits of counter
add %rdx,%rax   # %rax now contains 64 bit counter
mov %rax,%r10

# Payload, counting prime numbers, registers only

# movl $30,%r8d # Count primes below this number

movl $1, %ebx
movl $0, %r9d

start:          # 1 -> MAX loop (num)
inc %ebx
cmpl %ebx,%r11d  # Until first param
je finished
movl $2, %ecx

innerLoop:
movl $0,%edx
movl %ebx,%eax
divl %ecx       # eax/ecx, edx = reminder
```

```
cmpl $0, %edx    # compare, is edx equal 0?
je checkprime    # prime if cx=bx equal
incl %ecx
cmpl %ebx,%ecx
jle innerLoop

checkprime:
cmpl %ecx,%ebx
jne start

incl %r9d
jmp start

finished:
#movl %r9d, %eax

rdtscp

shl $32,%rdx
add %rdx,%rax  # %rax now contains second reading
sub %r10,%rax  # Subtracts first counter

pop %rbx

ret  # Cycles returned in rax

# not using cpuid, don't need push %rbx
```

Code Listing A.4: stream.cpp

```
#define STREAM_ARRAY_SIZE 12800
#define RUNS     100
#define NTIMES   10000000

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <stdint.h>

extern "C" {
   uint64_t assum(int max);
}

inline uint64_t tsc() {

        uint64_t high = 0, low = 0;
        asm __volatile__(
                "rdtscp;"
                : "=d" (high), "=a" (low)
                :
                : "rcx"
        );

        return ((high << 32) | low);
}

#define STREAM_TYPE uint64_t
static STREAM_TYPE       a[STREAM_ARRAY_SIZE],
```

```
                                c[STREAM_ARRAY_SIZE];
static uint64_t cycles[NTIMES];

int stream_main()
{
    int k,r,m;
    int arrSize[10];
    int j;
    FILE* stream_data = stdout;

    uint64_t sum=0,square=0;
    int countA,countB,countC,countD;
    double aver,var,sd,aSize,bSize,cSize,dSize,d;
    fprintf(stream_data, "MemSize,NTIMES,aver,sdev,countA,aSize,countB,bSize,countC,cSize,co


    arrSize[0] = 25;
    arrSize[1] = 50;
    arrSize[2] = 100;
    arrSize[3] = 200;
    arrSize[4] = 400;
    arrSize[5] = 800;
    arrSize[6] = 1600;
    arrSize[7] = 3200;
    arrSize[8] = 6400;
    arrSize[9] = 12800;
    // Varying size of mem-array

    // One initial loop
    // Initializing array
    for (j=0; j<STREAM_ARRAY_SIZE; j++) {
       a[j] = 1 + j;
       c[j] = 2 + j;
    }
    // Running experiment NTIMES
    for (k=0; k<NTIMES; k++)
      {
         cycles[k] = tsc();
         for (j=0; j<STREAM_ARRAY_SIZE; j++)
           {
              c[j] = a[j];
              a[j] += 1;
           }
         cycles[k] = tsc() - cycles[k];
      }

    for (m=0; m<10; m++)
      {
        for (r=0; r<RUNS; r++)
          {
            // Initializing array
            for (j=0; j<STREAM_ARRAY_SIZE; j++)
              {
                a[j] = 1 + j;
                c[j] = 2 + j;
              }
            // Running experiment NTIMES
            for (k=0; k<NTIMES; k++)
              {
```

```
      cycles[k] = tsc();
      for (j=0; j<arrSize[m]; j++)
        {
          c[j] = a[j];
          a[j] += 1;
        }
      cycles[k] = tsc() - cycles[k];
    }

   // Stats for experiment:
  aver = 0;
  var = 0;
  square = 0;
  sum = 0;
  aSize = 0;
  bSize = 0;
  cSize = 0;
  dSize = 0;
  countA = 0;
  countB = 0;
  countC = 0;
  countD = 0;
  for (k = 0; k < NTIMES; k++) {
    sum += cycles[k];
    square += cycles[k]*cycles[k];
  }
  aver = sum/(1.0*NTIMES);
  var = square/(1.0*NTIMES) - aver*aver;
  sd = sqrt(var);

  for (k = 0; k < NTIMES; k++) {
    d = cycles[k] - aver;
    if (d > 150 && d <= 400)
      {
        countA++;
        aSize += d;
      }
    else if (d > 400 && d <= 2000)
      {
        countB++;
        bSize += d;
      }
    else if (d > 2000 && d <= 6000)
      {
        countC++;
        cSize += d;
      }
    else if (d > 6000)
      {
        countD++;
        dSize += d;
      }
  }
  if(countA > 0){aSize = aSize/(1.0*countA);}
  if(countB > 0){bSize = bSize/(1.0*countB);}
  if(countC > 0){cSize = cSize/(1.0*countC);}
  if(countD > 0){dSize = dSize/(1.0*countD);}
  fprintf(stream_data, "%d %d %.2lf %.1lf %d %.1lf %d %.1lf %d %.1lf %d %.1lf \
}
```

```
        }
}
```

Code Listing A.5: runExp.sh

```
#! /bin/bash

#type=CPU # assembly-code
type=MEM

killall tail
exp="${type}Runs100Iter10M4slice"
dir="/root/IncludeOS/examples/STREAM"
file=$dir/experiments/$exp.res
fileIntel=$dir/experiments/Intel$exp.res

killall ipmitool
sleep 2
tail -f /dev/null | ipmitool -I lanplus -H ddoslab.vlab.cs.hioa.no -U root -P "pw" sol activate > $file 2> /dev/null &

cp serviceIRQ.cpp service.cpp
boot -b .
cp build/stream_example iso/boot/
grub-mkrescue -o inc.iso iso
sleep 2
ipmitool -I lanplus -H ddoslab.vlab.cs.hioa.no -U root -P "pw" chassis bootdev cdrom
killall vmcli
sleep 10
killall screen
sleep 2
screen -S vmcli -m -d /opt/dell/srvadmin/bin/vmcli -r ddoslab.vlab.cs.hioa.no -u root -p "pw" -c /root/IncludeOS/examples/STREAM
sleep 2
ipmitool -I lanplus -H ddoslab.vlab.cs.hioa.no -U root -P "pw" chassis power cycle

sleep 200 # Now IncludeOS should have started writing

while [ 1 ]
do
   ok=$(grep --text "Initializing plugins" $file)
   if [ "$ok" ]
   then
      echo "File $file contains IncludeOS output, proceeds...."
      break
   else
      killall ipmitool
      sleep 4
      tail -f /dev/null | ipmitool -I lanplus -H ddoslab.vlab.cs.hioa.no -U root -P "pw" sol activate > $file 2> /dev/null &
      sleep 4
      ipmitool -I lanplus -H ddoslab.vlab.cs.hioa.no -U root -P "pw" chassis power cycle
      sleep 200 # Now IncludeOS should have started writing
   fi
done

lines=$(cat $file | wc -l )
while [ 1 ]
do
   sleep 120 # Takes some time to boot
   lines2=$(cat $file | wc -l )
   if (( $lines2 == $lines ))
   then
      echo "File $file stopped increasing, proceeds...."
      break
   else
      echo -n "."
      lines=$lines2
   fi
done

cp serviceNoIRQ.cpp service.cpp
boot -b .
cp build/stream_example iso/boot/
grub-mkrescue -o inc.iso iso
ipmitool -I lanplus -H ddoslab.vlab.cs.hioa.no -U root -P "pw" chassis bootdev cdrom
killall vmcli
sleep 10
killall screen
sleep 2
screen -S vmcli -m -d /opt/dell/srvadmin/bin/vmcli -r ddoslab.vlab.cs.hioa.no -u root -p "pw" -c /root/IncludeOS/examples/STREAM
sleep 2
ipmitool -I lanplus -H ddoslab.vlab.cs.hioa.no -U root -P "pw" chassis power cycle

cd /root/IncludeOS/examples/STREAM/intel

if [ "$type" = "CPU" ]
then
   ./as # Assembly compiling
else
   ./c2
fi
```

```
lines=$(cat $file | wc -l )
while [ 1 ]
do
    sleep 120 # Takes some time to boot
    lines2=$(cat $file | wc -l )
    if (( $lines2 == $lines ))
    then
        echo "File $file stopped increasing, proceeds...."
        break
    else
        echo -n "."
        lines=$lines2
    fi
done

# IncludeOS finished

ipmitool -I lanplus -H ddoslab.vlab.cs.hioa.no -U root -P "pw" chassis bootdev disk
ipmitool -I lanplus -H ddoslab.vlab.cs.hioa.no -U root -P "pw" chassis power cycle

while [ 1 ]
do
    sleep 20
    res=$(ssh intel2.vlab.cs.hioa.no uname 2> /dev/null)
    if [ "$res" ]
    then
        break
    fi
done

ssh -4 root@intel2.vlab.cs.hioa.no  /root/stopServices.sh
sleep 10
scp -4 run root@intel2.vlab.cs.hioa.no:
sleep 2
ssh -4 root@intel2.vlab.cs.hioa.no "taskset -c 10 /root/run > /root/linux.res"
sleep 1
scp -4 root@intel2.vlab.cs.hioa.no:/root/linux.res  $fileIntel

echo "Finished!"
```