8-25-2016

Analysis of Performance and Power Aspects of Hypervisors in Soft Real-Time Embedded Systems

John Guthrie

John Guthrie
Candidate

Electrical and Computer Engineering
Department

This thesis is approved, and it is acceptable in quality and form for publication:

Approved by the Thesis Committee:

Dr. Gregory Heileman, Chairperson

Dr. Chris Lamb

Dr. Wennie Shu

ANALYSIS OF PERFORMANCE AND POWER ASPECTS OF HYPERVISORS IN SOFT REAL-TIME EMBEDDED SYSTEMS

by

JOHN GUTHRIE

B.S., ELECTRICAL ENGINEERING, UNIVERSITY OF CALIFORNIA DAVIS, 2013

THESIS

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science
Computer Engineering

The University of New Mexico
Albuquerque, New Mexico

July, 2016

DEDICATION

I would like to dedicate this thesis to my father, mother and my two brothers for supporting me throughout my education and my life. Their encouragement enabled me to reach farther than I would have dreamed of. My father especially stressed the importance of higher education and this thesis is the result of his persuasion to keep at it.

I would also like to dedicate this to my fiancé, Helen Rickey, for supporting me throughout graduate school and putting up with evenings of me busy with classes and this thesis effort. Her encouragement and love helped me carry through one of the toughest challenges of my life.

ACKNOWLEDGEMENTS

I heartily acknowledge Dr. Gregory Heileman, my advisor and thesis chair, for supporting me through my thesis experiments and the review of my experiments and thesis. His guidance will help me throughout my professional career.

I also thank my committee members, Dr. Lamb and Dr. Shu, for their assistance in this technical study and assistance in my professional development. An extra thanks goes to Dr. Lamb for working personally with me to ensure that my technical experiments and thesis were of professional quality.

Gratitude is extended to the Air Force Research Laboratory for the funding to pursue the research and their professional guidance. Finally, I thank Mr. Calvin Roman, whose support and professional guidance helped make this thesis a reality.

ANALYSIS OF PERFORMANCE AND POWER ASPECTS OF HYPERVISORS IN SOFT REAL-TIME EMBEDDED SYSTEMS

by

John Guthrie

B.S., Electrical Engineering, University of California Davis, 2013

M.S., Computer Engineering, University of New Mexico, 2016

ABSTRACT

The exponential growth of malware designed to attack soft real-time embedded systems has necessitated solutions to secure these systems. Hypervisors are a solution, but the overhead imposed by them needs to be quantitatively understood. Experiments were conducted to quantify the overhead hypervisors impose on soft real-time embedded systems. A soft real-time computer vision algorithm was executed, with average and worst-case execution times measured as well as the average power consumption. These experiments were conducted with two hypervisors and a control configuration. The experiments showed that each hypervisor imposed differing amounts of overhead, with one achieving near native performance and the other noticeably impacting the performance of the system.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

1.1 Introduction

The embedded systems industry has grown exponentially in the last 20 years. The application of embedded systems varies widely from the aerospace and automotive industries to cellular phones and other mobile devices that consumers use on a daily basis. The applications have been just as diverse, from providing pilots and drivers information about the status of their systems to providing internet capabilities to consumers away from their computers at home. In many cases, the functionality of these systems is based on real-time capabilities in which algorithms must perform under a specified time constraint. Failure to meet these real-time requirements can result in a variety of adverse effects. One of the more mild consequences is degradation in the performance of the embedded system. On the other end of the spectrum, failure to meet real-time constraints can result in a catastrophic failure of the system. Meeting timing constraints is a must for real-time systems.

Complicating matters further is the reality that security threats to embedded systems have become more commonplace, with the rate of "malware strains discovered increas(ing) by 77 percent in 2014" (Chickowski). By exploiting security vulnerabilities, attackers have become more skilled at conducting attacks ranging from Denial of Service attacks to stealing data. This has driven the need to cyber-harden embedded systems, often resulting in performance reductions due to the overhead imposed by security solutions. This thesis focuses on one kind of security solution, the hypervisor. This thesis will also focus what quantifiable effects hypervisors have on performance for real-

time embedded systems and if it is worthwhile to use hypervisors, which exchanges overhead for application isolation, process segregation, and other security benefits.

1.2 Motivation

Quantifying the performance drawbacks of hypervisors is important because knowing exactly what a hypervisor brings to an embedded system and what performance overhead it imposes allows for the usage of trade studies to determine which software environments may benefit from these virtualization schemes. Not all software environments need to be protected; blind implementation of security schemes results in unnecessary computing overhead. Careful analysis needs to be made to see which parts of an embedded system are critical to the success of the embedded environment, whether it is needed for critical operations or its compromise would result in a failure of the embedded system's functionality. By enabling this analysis, embedded systems can succeed at protecting critical information in exchange for minimal overhead.

1.3 Thesis Outline

The rest of this thesis will be presented in the following manner. Chapter 2 will present background information concerning real-time systems and virtualization and hypervisors. Chapter 3 will present a review of related literature of how researchers are improving hypervisors to support real-time capabilities. Chapter 4 will present the methodology for conducting controlled experiments to characterize the performance overhead imposed by hypervisors. The experiments will use a control set-up and two different hypervisors to conduct the experiments. Chapter 5 will present the results of the

experiments. Finally, chapter 6 will be a discussion of the results, limitations of the study, and how the study could be used for future research.

Chapter 2

Background

2.1 Real-time systems

A real-time system is a system "that functions within a time frame" that is considered "immediate or current" (Rouse). The performance of the system is measured by its ability to perform continuous operations, with certain actions occurring before a pre-specified latency. This latency, known as the worst-case execution time (WCET), determines whether the system is real-time or not. Real-time systems can be further categorized into either soft real-time or hard real-time. With a soft real-time system, if the execution time of an algorithm exceeds the WCET, the system is able to continue operating. The system will suffer performance penalties, such as degraded operation, but the system can continue to function. Hard real-time, on the other hand, will experience a system failure if the WCET is not met. Steps must be taken to put the system back to normal operation, such as a system reset.

2.2 Virtualization/Hypervisors

The terms "hypervisor" and "virtualization" can refer to many different implementations and schemes for abstracting software and services from the underlying physical resources. In this thesis, we use the term "virtualization" to discuss the technique or act of hardware/resource abstraction, while we use "hypervisor" to refer to a product that supplies virtualization. Virtualization involves adding a software layer somewhere above the firmware on the software stack. The hypervisor acts as a mediator between the OS layer and the firmware layer. This allows multiple virtual environments,

called virtual machines (VMs), to share a hardware pool. Virtualization is used for a variety of purposes, such as supporting multiple OSs on a single hardware host and enhancing resilience through logical segregation and/or geographic separation. Virtualization can be implemented using a type-1 or type-2 hypervisor.

2.2.1 Type-1 (bare-metal) hypervisor

A type-1 hypervisor has the virtualization layer reside below the operating system. The hypervisor lies above the firmware layer and interacts with traffic coming from the above software stack down to the hardware. Because of this, the hypervisor runs in a "privileged" status and takes control of the hardware instead of the software above it. If the hypervisor deems that an incoming hardware instruction is "privileged," meaning that the instruction would execute assuming it has exclusive access to the hardware, the hypervisor will trap the instruction and execute the instruction instead, allocating hardware resources to the caller as necessary (VMware Inc.).

2.2.2 Type-2 (hosted) hypervisor

A type-2 hypervisor executes similarly to a type-1 hypervisor, except that the hypervisor runs above a host operating system layer instead of below it. To the host operating system, the hypervisor looks like another application running on the system. The hypervisor can then launch operating systems above it that have to go through the hypervisor in order to access hardware resources (VMware Inc.).

2.2.2.1 Example: A Bare-Metal Hypervisor Implementation

A Type 1 hypervisor implementation involves inserting the hypervisor between the OS layer and the hardware layer. The OS and user applications are instantiated above the hypervisor, providing isolation from the underlying hardware. As in normal

computing systems, the applications interact with the OS and with other applications. However, instead of the OS being able to make direct calls to memory and the CPU, hardware requests now flow to the hypervisor. This design ensures that the hypervisor enforces proper behavior in the OS and the applications above the OS. In addition, if a request acts contrary to protocol, the hypervisor can intervene and block the OS from accessing the hardware. Figure 1 illustrates a bare-metal hypervisor implementation.

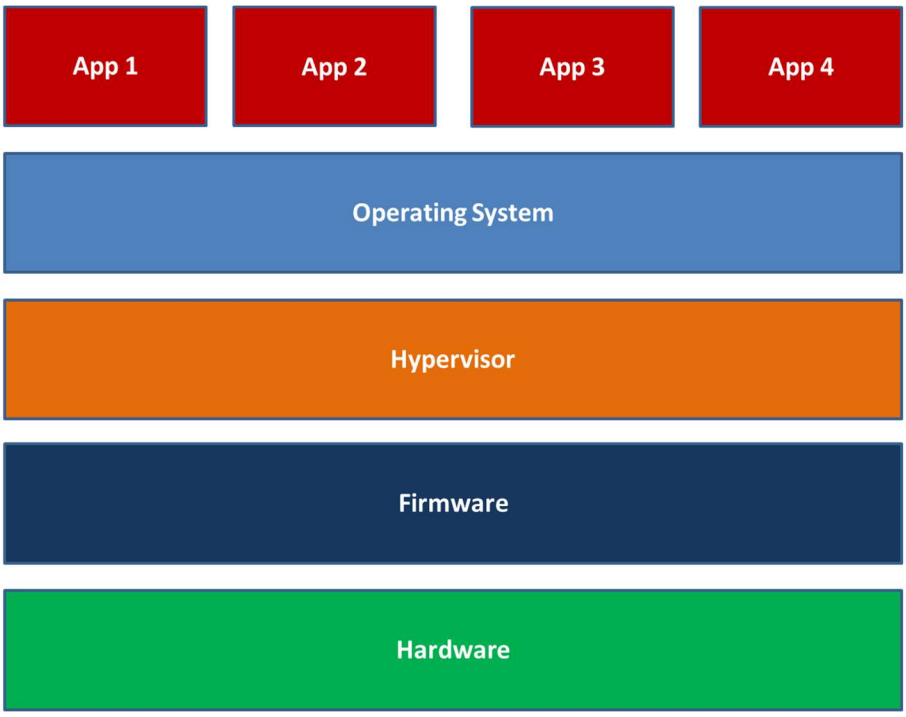

FIGURE 1: A BARE-METAL HYPERVISOR IMPLEMENTATION

2.3 Virtualization Challenges and Implementation Schemes

With a virtualization layer, multiple OSs running in separate virtual machines (VM) may run concurrently while the virtualization layer dynamically allocates hardware resources to each VM. These VMs are logically segregated from other VMs in the

system. Virtualization can provide functional and security benefits to systems. With virtualization, hardware resources can be allocated more efficiently to multiple VMs. Unused hardware resources can be parsed to other VMs instead of being distributed statically to VMs that have the potential to be wasted. However, this hardware resource distribution comes with an overhead cost that impacts execution time of the system. This is a critical cost that will become important when looking at real-time systems. Virtualization's main security benefits are achieved via isolation and abstraction. Through isolation, exposure to other parts of the system is reduced, limiting the ability for malware to maneuver in the system. Virtualization adds an additional layer of abstraction, limiting direct access to the system's hardware and in turn inhibiting malware's ease of influencing other components. On the other hand, introducing a new layer poses the risk of increasing the attack surface of the system. The hypervisor needs to be implemented so that only essential actions can occur in the operation of hypervisors, reducing the hypervisor's attack surface. Another problem with this abstraction stems from the fact that OSs in these virtual machines assume that they have exclusive access to hardware resources, which makes translating OS calls with the virtualization layer in place a non-trivial task. To solve this problem, "full virtualization," "paravirtualization," and "hardware assisted virtualization" implementations have been used to successfully "virtualize" privileged instructions.

2.3.1 Full Virtualization

Full virtualization involves the virtualization layer translating virtual machine privileged instructions to a new set of instructions that achieve the same effect in the computer hardware. By doing this, the virtual machines are completely decoupled from

the hardware layer, with only the virtualization layer communicating with the hardware. The OSs do not need to be modified in order to be placed in a virtual machine and have no knowledge of virtual machines outside of its own. Full virtualization allows for a high degree of isolation among virtual machines. Virtual machines are unaware of other guest virtual machines, and the virtual machines are isolated from the hardware. Operating systems are also highly portable. Because the hypervisor does all the virtualization, operating systems can run normally and let the hypervisor trap sensitive instructions. On the other hand, full virtualization creates overhead that may hinder the performance of the system. Since extra work is necessary to virtualize sensitive instructions, the hypervisor will need extra time to execute privileged instructions, possibly reducing performance. In addition, the hypervisor will need to support various operating systems. If it needs to support multiple operating systems, this will increase the cost of developing a hypervisor.

2.3.2 Paravirtualization

Paravirtualization, on the other hand, enlists the help of the virtual machines in translating calls to virtual instructions. The OS inside the virtual machine is modified so that it replaces privileged instructions to calls that communicate directly with the virtualization layer without need for translation. Because of these calls, the virtual machine is aware of the virtualization layer and bears some of the workload in virtualizing instructions. Paravirtualization allows for the easy modification of current operating systems to work with the hypervisor. This allows for a short development time for hypervisors that need to support multiple guest operating systems. In addition, because the guest virtual machines assist in virtualizing privileged instructions, the imposed overhead can be potentially small. This overhead will depend on how

paravirtualization is implemented and the workload of the embedded system. A possible disadvantage to this virtualization scheme is that the modified operating systems are aware of the hypervisor. This lowers isolation between the hypervisor and guest virtual machines, which could be exploited by hackers to compromise the security of the virtualization scheme. In addition, operating systems need to be modified to work in a paravirtualization scheme. This means unmodified operating systems are not supported as is, and maintainability issues are introduced to the operating systems due to modified operating systems behaving differently than what is supported by operating system suppliers.

2.3.3 Hardware Assisted Virtualization

This technique incorporates the use of the hardware itself to create the virtualization environment. The hardware is specifically designed to have the hardware create a custom, privileged root mode where the hypervisor resides. This hypervisor can operate below the operating system software level. With the help of the hardware, privileged and sensitive instructions are automatically trapped to the hypervisor, removing the need for a software-based solution (VMware Inc.). Hardware assisted virtualization has the advantage of simplifying virtualization for software due to using what is prebuilt in the hardware. This allows software to dedicate resources to other tasks and let hardware handle the virtualization portion. On the other hand, it has the disadvantage of being only available to newer hardware. Hardware must be specifically designed to handle virtualization, which has only been commonly designed in hardware for the last ten years. This results in hardware virtualization being incompatible with legacy systems.

2.3.4 Balancing Features and Security Hardness

When implementing virtualization, careful consideration needs to be made in the complexity of the tool. Adding more software features that virtualization can support can improve the functionality of the system, but at the cost of increasing the complexity and attack surface area of the overall system. On the other hand, implementing virtualization that is minimally sized for the system will limit system functions in its design, but will reduce the attack surface area of the system. This enables the hypervisor to use a simpler process in proving that the system is acting as it should be without malicious interference. For security purposes, the virtualization implementation should only support what is critical to the system so as to maximize the security benefits of virtualization.

Chapter 3

Review of Related Literature

This chapter summarizes a literature review of research done in virtualization and hypervisors. Each paper will be summarized, with the main points relevant to this thesis emphasized. The literature focused on ways to analyze the performance of hypervisors as well as introducing new ways to implement virtualization schemes.

Hwang et al. (Hwang, Suh and Heo) explored putting the Xen bare-metal hypervisor on a mobile phone environment. They modified the phone to enable Xen 3.0.2 to run on an ARM environment. They conducted their performance benchmark using LMBENCH, which evaluated timing of basic system operations under differing workloads. They also explored what happened when multiple virtual machines were running at once. Finally, they did macro benchmarks by conducting common phone operations, such as loading time and image saving time. The paper concluded that for some micro benchmarks, there was some moderate overhead, while the macro operations did not see much overhead difficulties.

Xi et al. (Xi, Wilson and Lu) presented RT-Xen, which they toted as "the first hierarchical real-time scheduling for Xen". The paper presented 4 new schedulers that emulated real-time. The VCPUs were identified by budget, period and priority parameters and bucketed them in a ReadyQueue, a RunQueue, and ReplenishQueue. The paper then evaluated the performance of the schedulers by evaluating how many tasks failed to finish in time depending on varying scheduling compared quantum and compared how the overhead differed. This was measured in terms of scheduling latency and context switching. The paper concluded that the Deferrable Server delivered better

soft real-time performance than the other server algorithms, while the Periodic Server incurred high deadline miss ratios in overloaded situations.

Avanzini et al. (Avanzini, Valente and Faggioli) explored implementing a dual OS on the Xen hypervisor, one being Linux and the other being ERIKA, a RTOS. The idea was to have ERIKA handle the real-time, safety critical tasks while Linux handled the bulk of the non-safety critical tasks. The paper outlined its implementation scheme and ended the paper with a set of tests that will demo the end result at a later date.

Jing et al. (Jing, Guan and Yi) proposed a model for controlling shared memory accesses using the Xen hypervisor. This will improve the timing predictability of real-time applications. The monitoring process was implemented using the Performance Monitoring unit in the processor to budget a set number of memory accesses. If it goes over the budget, the PMU suspends the VM. The results showed that the execution time was kept low and stable regardless of memory access behavior, while memory access throttling stabilized the timing of the experiments but at the cost of average performance.

Yu et al. (Yu, Xia and Lin) explored modifications to the Credit Scheduler in the Xen hypervisor. They noticed that real-time operations were not distinguished from other operations, resulting in poor real-time performance. They tried to solve this by adding in real-time priority so that it goes to the head of the queue and when the processor goes into boost mode, real-time events jump to the front. In addition, they incorporated guest balancing for multiple real-time guests. The results showed that their enhancements improved the real-time performance of Xen by about 20%.

Cherkasova et al. (Ludmila Cherkasova) examined the three CPU schedulers in Xen in terms of performance. The three schedulers were Borrowed Virtual Time, Simple

Earliest Deadline First, and Credit. They used XenMon to analyze SEDF, which showed that Dom0 has a high overhead on I/O applications. They also conducted experiments using Iperf and Disk tests. They also incorporated Allocation Error Test to see if CPU resources were allocated correctly, which showed as much as 10% error. Through other tests, they concluded that Credit is the best for global balancing.

Masmano et al. (M. Masmano) explored the capabilities of XtratuM, which they described as a "bare hypervisor that uses paravirtualization". They based their performance evaluation on partition context switching time and measuring the cost of hypercalls, which they measured in microseconds. They evaluated the performance overhead imposed by XtratuM and concluded that the overhead was lower than 3% if the slot duration was higher than 1 millisecond.

Habib (Habib, Virtualization with KVM) introduced KVM, a hypervisor that used hardware-based virtualization to create virtual machines. It integrated hypervisor capabilities into the Linux kernel. This allowed the virtualized environment to incorporate work into the Linux kernel. It also used the QEMU emulator to provide a user-space and incorporate its effective I/O model. KVM was relatively small because it does not have to make its own kernel protocols, avoiding the complexity of Xen and VMware. Results showed that KVM allowed native execution using the KVM kernel for non-critical tasks while effectively isolating the other critical parts of the system.

Bruns et al. (Bruns, Traboulsi and Szczesny) investigated the performance of implementing a real-time operating system (RTOS) with a hypervisor (L4/Fiasco). They measured the performance by comparing system latencies with the RTOS without a hypervisor installed. In addition, they looked at hardware interrupts and analyzed their

impacts on real-time requirements. Results found that the average execution time increased between 30 to 50 percent when cache contention was imposed on the hypervisor as well as a higher variation of execution times.

Zhang et al. (Zhang, Chen and Zuo) investigated using a Kernel-Based Virtual Machine (KVM), which combined both Linux and VxWorks into a real-time operating system (RTOS). This unique virtualization took advantage of common Linux resources, but with added virtualization capabilities. The paper summarized their design and commented on the real-time performance analysis the RTOS had on the base system. They realized there were "harmful workloads," so they used real-time tunings such as prioritization and CPU shielding to lower down the latencies of these use cases. They evaluated these experiments and displayed their results, which achieved sub millesecond latency for harmful workloads.

Asberg et al. (Asberg, Forsberg and Nolte) proposed a type-2 hypervisor that did not require kernels running above it to be modified. This allowed real-time scheduling to be conducted. The paper proposed to have in the host kernel a RESCH core, so that it could support real-time task support. Their preliminary testing showed lower overhead than the KVM hypervisor and looked to conduct more experiments in the future.

Soltesz et al. (Soltesz, Pötzl and Fiuczynski) explained Container-based Operating System Virtualization. This virtualization runs on top of a host kernel and efficiently allocates host resources amongst "containers," which are separate instances of the kernel. This contrasts significantly from hypervisors because there only needs to be one host, rather than have a host run the hypervisor and each VM below run their own separate operating system and resources. The upside to this new technique is the fact that

significant amount of resources are saved due to saving on overhead. On the other hand, it does not support multiple operating systems running on the machine and it still needs a host kernel to run it, which remains exposed, since the containers run below the host kernel.

Xavier et al. (Xavier, Neves and Rossi) explored using container-based virtualization in high performance computing. They did a number of test comparing containers to the Xen hypervisor. Their experiments showed that container-based virtualization achieves performance close to native performance, while the Xen environment has a considerable amount of overhead of 4.3%. However, the paper admitted that this container-based virtualization shared a lot of common resources, hence lowering the security of the entire system.

Masrur et al. (Masrur, Pfeuffer and Geier) proposed a technical approach to designing a fixed-priority real-time scheduler for VMs in order to meet all real-time deadlines. They determined that the period of the VMs was determined by the minimum that needed to be scheduled on that VM. They also concluded from their experiments that response time improved when only one task is running on a VM, but memory was efficiently allocated when multiple tasks were running.

Lee et al. (Lee, Krishnakumar and Krishnan) explored improving the Credit scheduler in the Xen hypervisor in order to improve real-time task performance. They sought to improve it through introducing a laxity value given to all tasks. Low values would indicate a soft real-time task while a high one indicated a non-real time task. In addition, they introduced workload balancing by balancing CPU time amongst different real-time tasks so that it was roughly equal among multiple processors. In addition, they

introduced an algorithm to maintain cache coherence. Their results illustrated that a small laxity value improved real-time performance, while a large value had no effect on the system. Cache coherence results showed that the voice quality increased significantly compared to the baseline credit scheduler.

Chapter 4

Methodology

This chapter will describe the process to configure and execute the experiments that will quantify the performance and power consumption of virtualization schemes. The first section of this chapter goes into detail the configuration of the hardware and software. The second section details how the experiments themselves were executed and recorded.

4.1 Configuration of Software and Hardware

The experimental hardware and software was selected based on the following characteristics. The system will be soft real-time, able to operate at limited capability if deadlines are not met. The soft real-time system in question will be assumed to operate in a varied physical environment, running on a limited source of power. It would operate for long periods of time without easy access to maintenance resources, making hardware and software reliability a must. In addition, it would have stringent space requirements for the design of its system, motivating a need to choose a solution that gets the most processing power in relation to its power and space requirements. The security requirements of the system would depend on what kind of operations the system would conduct. In summary, the emulated system will need to need to be reliable for long periods of time, have soft real-time requirements for its operation, have flexible security requirements depending on its usage, and have restrictive power and space requirements with the goal to maximize the processing capability of the system while meeting these space and power requirements. With the above system specifications in mind, the first

subsection will go into more detail the hardware specifications of the experiments, while the second subsection will delve into the software specifications of the experiments.

4.1.1 Hardware Specifications

The hardware selected for the emulated system consists of the following parts. The processing unit is a MinnowBoard MAX development board. Its CPU is an E3825 dual-core processing unit with 1.33 GHz of processing power each. Its GPU features integrated Intel HD Graphics which can be displayed via interface with the micro HDMI connector on the board. Each core has 1 GB of DDR3 RAM. Audio is also outputted from the board via HDMI. The other I/O ports on the board are a micro SD port, 1 SATA2 port, 1 USB 3.0 port, 1 USB 2.0 port, 1 serial debug port, and an Ethernet port. Other notable features include it being only 99x74mm, an operating temperature range of 0-70 degrees C, it needing to be powered with a 5 V DC source, and the capability to host a Linux operating system. The development board is shown below in Figure 3.

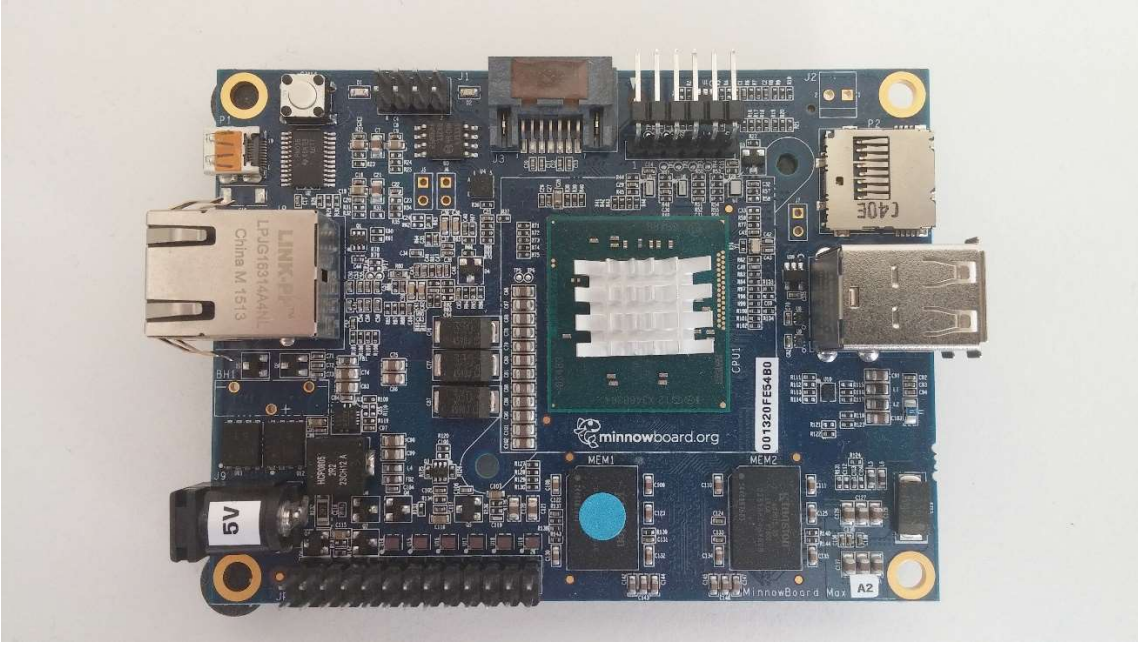

FIGURE 2: THE MINNOWBOARD MAX DEVELOPMENT BOARD USED IN THE EXPERIMENTS

Other peripherals are connected to the MinnowBoard MAX development board. 2 USB root hubs are connected to the two USB ports in the board. In one of these root hubs, a Fujitsu 250 GB hard drive is connected via a Super Top SATA bridge. In the other root hub, a standard USB keyboard and mouse is attached. Finally, a C310 Logitech webcam is attached. The webcam can support resolutions ranging from 160x120 pixels to 1280x720 pixels. The peripherals to the development board are shown below in Figure 4.

FIGURE 3: THE PERIPHERALS USED IN THE EXPERIMENTS, FROM LEFT TO RIGHT: C310 LOGITECH WEBCAM, FUJITSU HARD DRIVE, POWER METER, TWO USB ROOT HUBS

4.1.2 Software Specifications

The software specifications of the emulated system are broken up into three separate implementations. They all share some common software but with differing layers in their respective stacks. Common to each stack is the firmware layer, operating system layer, and application layer. These layers will first be described then the implementation of each software stack will be explained. The firmware layer is at the bottom of each stack. It is unmodified and takes care of communication between software and hardware. The operating system layer contains Ubuntu 14.04, a popular Linux-based operating system. Ubuntu is installed with typical packages essential for its functionality such as build-essential. In addition, packages necessary for the software in the application layer above it are installed here. Ubuntu's desktop environment is replaced with XFDE, a light-weight desktop, in order to minimize the impact of desktop functionality on the program. The application layer contains two pieces of software,

OpenCV and cvBlob. OpenCV is a library of functions geared toward executing real-time computer vision. Its features range from image processing to motion tracking. CvBlob is a program that executes real-time tracking of red objects using a webcam connected to the board. It puts forward a hardware resource request to retrieve a captured picture from the webcam, analyzes the image for a red object, draws a square around the object if found, and displays the image on the screen. It will continue displaying a stream of images dictated by the webcam's framerate until the user terminates the program. CvBlob heavily borrows from the functionality of OpenCV to complete its computing tasks. This application was chosen due to the persistent privileged hardware requests the application makes to the web camera, creating a workload that can stress implemented hypervisors. With the common software in each software stack explained, the specific ordering of each stack will be explained.

The software stacks of the control configuration, KVM configuration, and Docker configuration differ in distinct ways. The control has the firmware layer at the bottom, the operating system layer next, and lastly the application layer on top of its stack. However, the KVM and Docker configuration have an additional layer in their stacks called the virtualization layer. In this layer lies KVM or Docker, depending on the configuration. KVM is a type-1 hypervisor which lies between the firmware layer and the operating system layer. It executes hardware-assisted virtualization to initialize virtual machines that contain operating system and application layers. On the other hand, Docker is a lightweight type 2 hypervisor, implementing what is known as container-based virtualization. It lies between the operating system and application layer. It initializes light weight virtual machines called containers that contain a variety of

software, from applications to operating systems. Docker was chosen as a configuration because it represents a lightweight type-2 hypervisor, while KVM was chosen to represent a type-1 hypervisor. This will show the variance of overhead that is possible when hypervisor solutions are considered. In summary, the KVM software stack is ordered, from bottom to top, with the firmware layer, the virtualization layer containing KVM, operating system layer and application layer, while the Docker software stack has the firmware layer, operating system layer, the virtualization layer containing Docker, and the application layer from bottom to top. Figure 5 provides an illustration of the three software stacks.

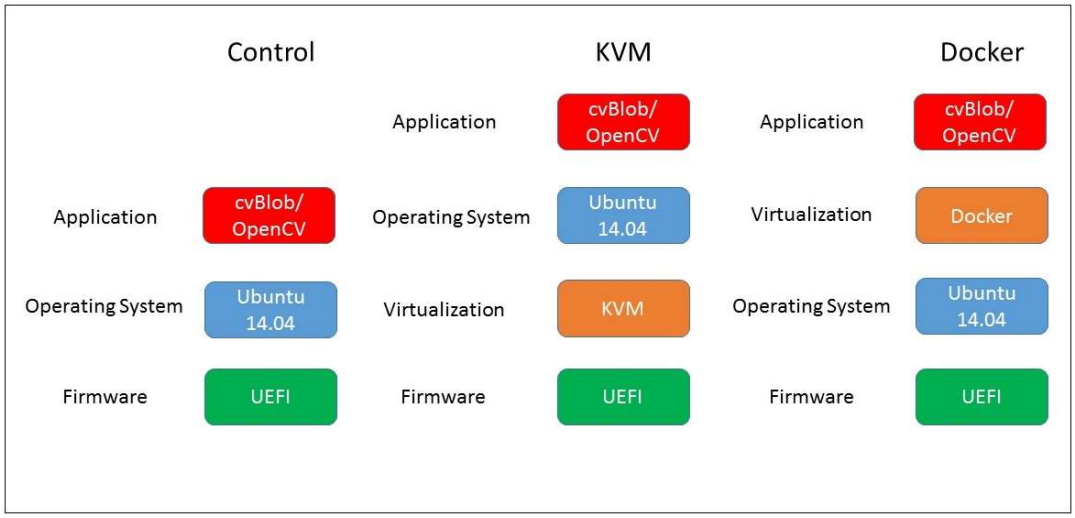

FIGURE 4: THE SOFTWARE STACK FOR EACH OF THE THREE TESTED CONFIGURATIONS

4.2 Procedure

The experiments were conducted using the control, KVM, and Docker configurations. The environment was first set up by plugging the Ethernet cable, HDMI cable, and USB peripherals into the development board. In particular, the webcam had its own dedicated USB port for minimal interference, while the external drive, mouse and keyboard were plugged into a root hub. The board was then powered on using the 5V

power supply. Further procedures were a function of which software configuration was under test.

When KVM was being tested, the Virtual Machine Monitor would be launched next. Once complete, the KVM virtual machine was initialized, which would launch its own computing environment. When Docker was being tested, Docker would need to be initialized. It would start an Ubuntu 14.04 container and finish setting up its own computing environment. From here, each configuration would execute the following instructions. The OpenCV library would be modified so that the webcam captured frames at a particular resolution. This resolution was defined as height and width variables in the library. These variables were restricted to the webcam's supported resolutions. Once the height and width variables were changed to the appropriate values, the OpenCV library was rebuilt to reflect the new camera resolution. Once complete, the red object tracking program was recompiled to link the OpenCV library to the program. Finally, the cvBlob was executed.

The time was measured using a function called rdtsc, an Intel-specific hardware call that returned the counter of the processor. This resulted in a highly accurate measurement of the current time of the program in processing cycles. The counter was recorded before a camera frame was processed and after the frame was processed and displayed. Five hundred measurements were recorded for the pre-specified camera resolution to accurately record the steady state of the video stream. The counter values for the beginning and end of each frame processing were subtracted from each other to get the difference, and this was divided by the core frequency of a single core to get the total time of processing an image from the web camera in seconds. This is because the

object detection program executed serially, so it was allocated a single core for its execution. Once the time was determined for each frame, the average of the samples was calculated to determine the average execution time of processing and displaying one frame from the web camera for a given frame resolution. The worst-case execution time was found in the samples by looking for the maximum execution time value. The first 10 frames were thrown out due to the program being in the start-up phase. The average frames per second value and worst-case frames per second value were calculated by taking the multiplicative inverse of the average execution time and worst-case execution time. This was done because the units of execution time are seconds per frame, so to get frames per second the reciprocal of the execution time was calculated. The power consumption of each configuration for a given resolution was measured using a Watt meter. The power supply was plugged into the meter and the power consumption was displayed in Watts to 100mW resolution. While the program was processing and displaying frames in a steady state, the average power over 15 seconds of program execution was recorded. The above steps for recording time and power metrics were repeated for 17 frame resolutions and for the three experiment configurations for a total of 25,551 samples. The frame resolutions tested were 160x120, 176x144, 320x176, 320x240, 352x288, 432x240, 544x288, 640x360, 640x480, 752x416, 800x448, 800x600, 864x480, 960x720, 1024x576, 1184x656, and 1280x720 pixels.

Chapter 5

Results

This chapter details results obtained from collecting data measuring average execution time, average frames per second, worst-case execution time, worst-case frames per second, as well as the average power consumption resulting from the control, Docker, and KVM configurations.

5.1 Average execution time and average frames per second

This subsection states the results of average execution time for each configuration and the average frames per second. It will first list the results using the control configuration, then the Docker configuration, and finally the KVM configuration.

5.1.1 Control Results

The results were collected with the program processing the frame with the pre-specified resolution captured from the web camera. The data for the execution time was plotted using the number of pixels processed as the x-axis and the average execution time as the y-axis. The data for the frames per second was plotted using the number of pixels processed as the x-axis and the average frames per second on the y-axis. The data collected is displayed in Table 1 and the graphs illustrated in Figure 6 and Figure 7. The data shows that when the resolution is 176x144 and below, the average execution time is roughly constant at .0337 seconds, but then increases linearly as the number of pixels is increased. The data also shows that when the resolution is 176x144 and below, the average frames per second is roughly constant at 29.7 FPS, but then decreases inversely as the number of pixels is increased.

Resolution (Width/Height)	Number Pixels	Average execution time(s)	Average Frames per second
160/120	19200	0.03369689	29.67632918
176/144	25344	0.033739097	29.63920463
320/176	56320	0.046733162	21.3980813
320/240	76800	0.055633007	17.97494067
352/288	101376	0.06647477	15.04330139
432/240	103680	0.067207373	14.87931986
544/288	156672	0.088719267	11.27150881
640/360	230400	0.121937956	8.200891936
640/480	307200	0.154353826	6.478621398
752/416	312832	0.156423007	6.392921471
800/448	358400	0.176856959	5.654286977
800/600	480000	0.22880332	4.370565952
864/480	414720	0.201997386	4.950559113
960/720	691200	0.322712799	3.098730522
1024/576	589824	0.277576639	3.602608647
1184/656	776704	0.36317172	2.753518363
1280/720	921600	0.41673399	2.39961228

TABLE 1: THE AVERAGE EXECUTION TIME AND AVERAGE FRAMES PER SECOND FOR A GIVEN RESOLUTION USING THE CONTROL CONFIGURATION

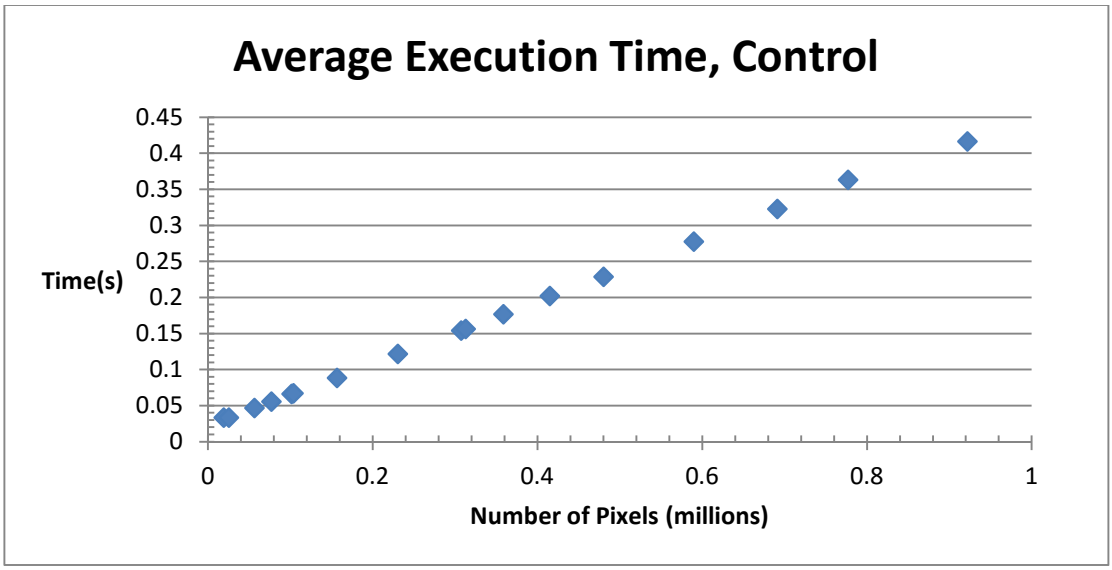

FIGURE 5: GRAPH PLOTTING AVERAGE EXECUTION TIME FOR A GIVEN RESOLUTION USING THE CONTROL CONFIGURATION

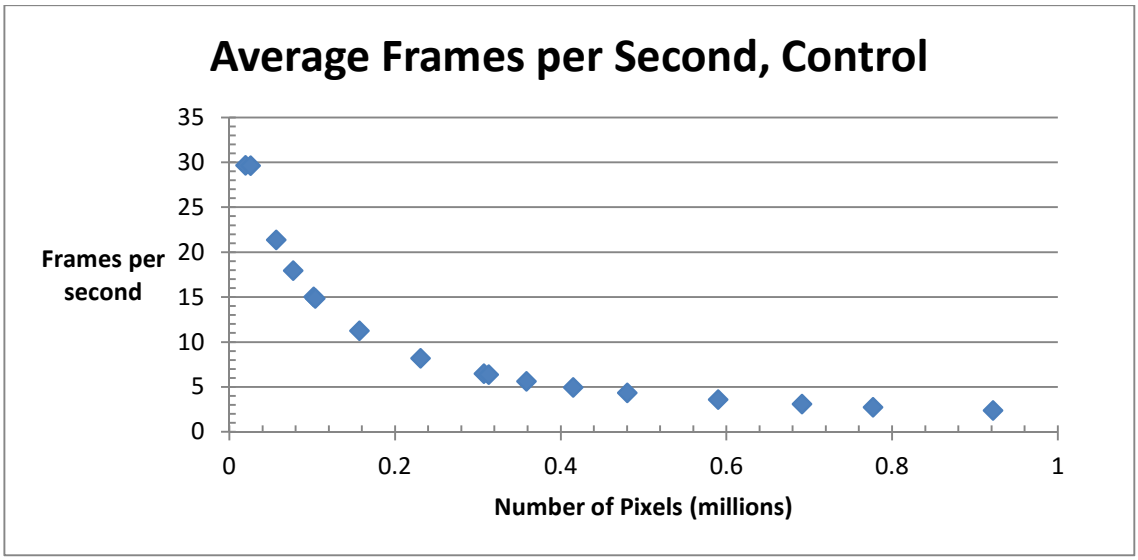

FIGURE 6: GRAPH PLOTTING AVERAGE FRAMES PER SECOND FOR A GIVEN RESOLUTION USING THE CONTROL CONFIGURATION

5.1.2 Docker Results

The results were collected with the program processing the frame with the pre-specified resolution captured from the web camera. The data was plotted using the number of pixels processed as the x-axis and the average execution time as the y-axis.

The second set of data was plotted using the number of pixels processed as the x-axis and the average frames per second on the y-axis. The data collected is displayed in Table 2 and the graphs illustrated in Figure 8 and Figure 9. The execution time data shows that the average execution time increases linearly as the number of pixels is increased. The frames per second data shows that the average frames per second decreased inversely as the number of pixels is increased.

Resolution (Width/Height)	Number Pixels	Average execution time (s)	Average frames per second
160/120	19200	0.033709	29.66568
176/144	25344	0.038589	25.91442
320/176	56320	0.039757	25.15281
320/240	76800	0.045748	21.85866
352/288	101376	0.055583	17.99127
432/240	103680	0.057194	17.4845
544/288	156672	0.08026	12.45958
640/360	230400	0.112671	8.875374
640/480	307200	0.146043	6.847285
752/416	312832	0.148222	6.74665
800/448	358400	0.168911	5.920281
800/600	480000	0.220982	4.52526
864/480	414720	0.192091	5.205856
960/720	691200	0.311483	3.210447
1024/576	589824	0.267246	3.741867
1184/656	776704	0.352752	2.834849
1280/720	921600	0.408609	2.447325

TABLE 2: THE AVERAGE EXECUTION TIME AND AVERAGE FRAMES PER SECOND FOR A GIVEN RESOLUTION USING THE DOCKER CONFIGURATION

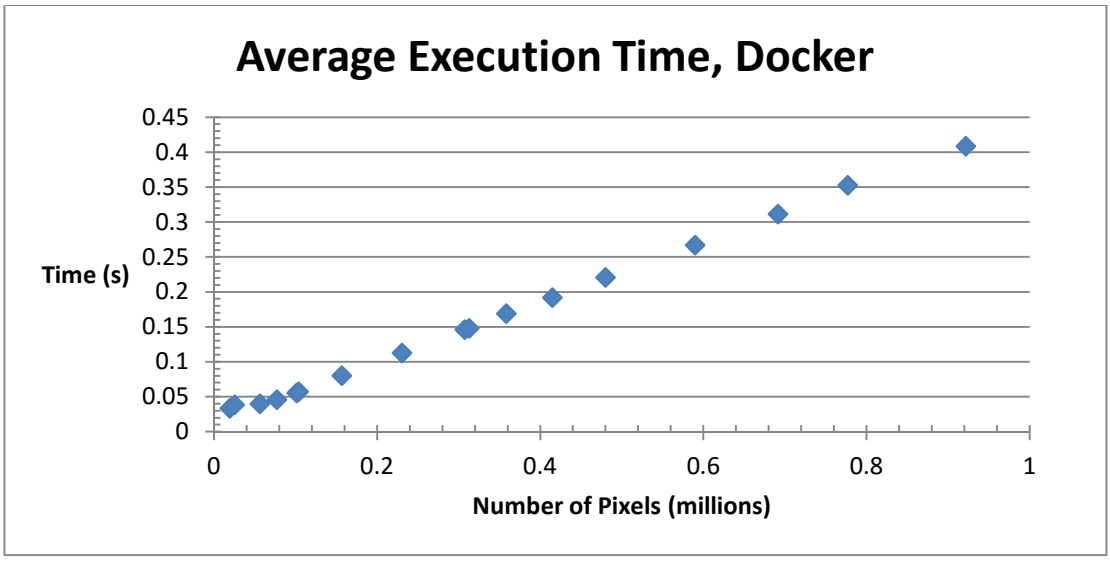

FIGURE 7: GRAPH PLOTTING AVERAGE EXECUTION TIME FOR A GIVEN RESOLUTION USING THE DOCKER CONFIGURATION

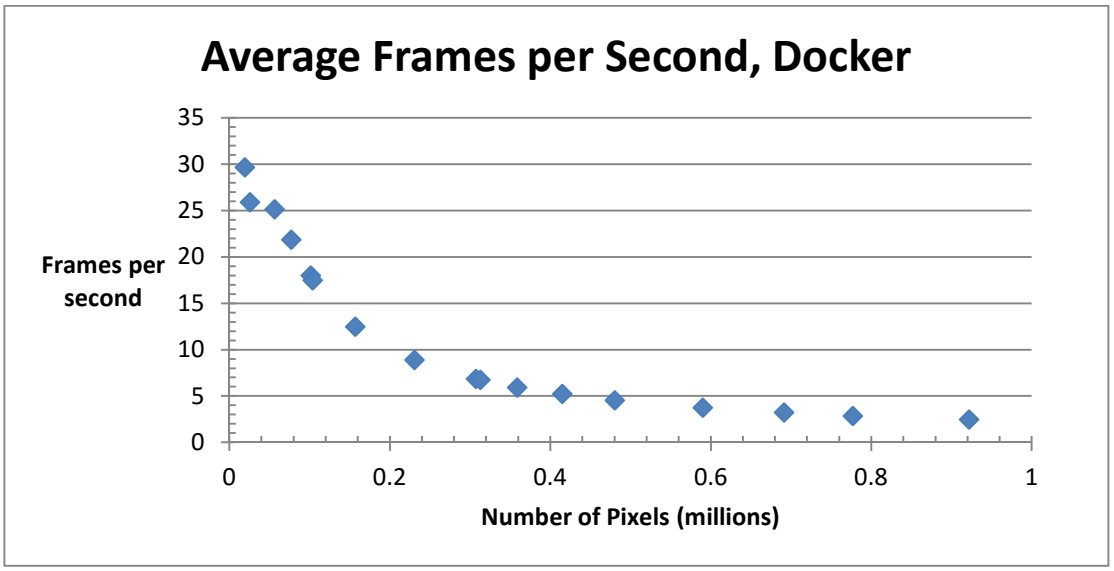

FIGURE 8: GRAPH PLOTTING AVERAGE FRAMES PER SECOND FOR A GIVEN RESOLUTION USING THE DOCKER CONFIGURATION

5.1.3 KVM Results

The results were collected with the program processing the frame with the pre-specified resolution captured from the web camera. The data was plotted using the number of pixels processed as the x-axis and the average execution time as the y-axis. The second set of data was plotted using the number of pixels processed as the x-axis and the average

frames per second on the y-axis. The data collected is displayed in Table 3 and its graph illustrated in Figure 10 and Figure 11. The execution time data shows that when the resolution is 176x144 and below, the average execution time is roughly constant at .0417 seconds, but then increases linearly as the number of pixels is increased. The frames per second data shows that when the resolution is 176x144 and below, the average frames per second was roughly constant at 23.9 frames per second, but then decreased inversely as the number of pixels was increased.

Resolution (Width/Height)	Number of Pixels	Average Execution Time (s)	Average Frames per second
160/120	19200	0.041688	23.98757
176/144	25344	0.041965	23.82916
320/176	56320	0.059266	16.87309
320/240	76800	0.0673	14.85884
352/288	101376	0.0771	12.97017
432/240	103680	0.086679	11.53679
544/288	156672	0.143876	6.950446
640/360	230400	0.196286	5.094594
640/480	307200	0.218281	4.581249
752/416	312832	0.223956	4.46517
800/448	358400	0.2591	3.859507
800/600	480000	0.325151	3.075491
864/480	414720	0.30565	3.271715
960/720	691200	0.488825	2.045721
1024/576	589824	0.414864	2.410428
1184/656	776704	0.519967	1.923198
1280/720	921600	0.634971	1.574874

TABLE 3: THE AVERAGE EXECUTION TIME AND AVERAGE FRAMES PER SECOND FOR A GIVEN RESOLUTION USING THE KVM CONFIGURATION

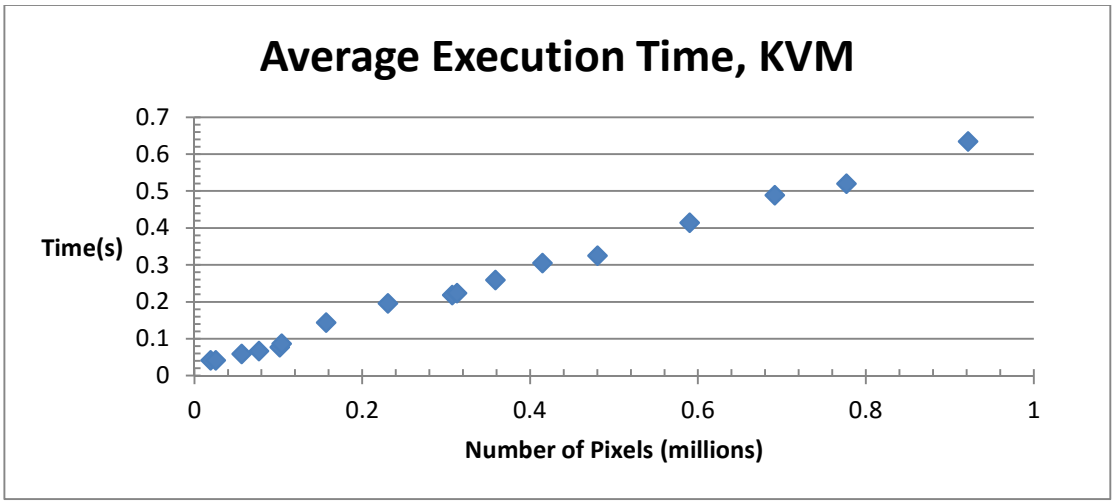

FIGURE 9: GRAPH PLOTTING AVERAGE EXECUTION TIME FOR A GIVEN RESOLUTION USING THE KVM CONFIGURATION

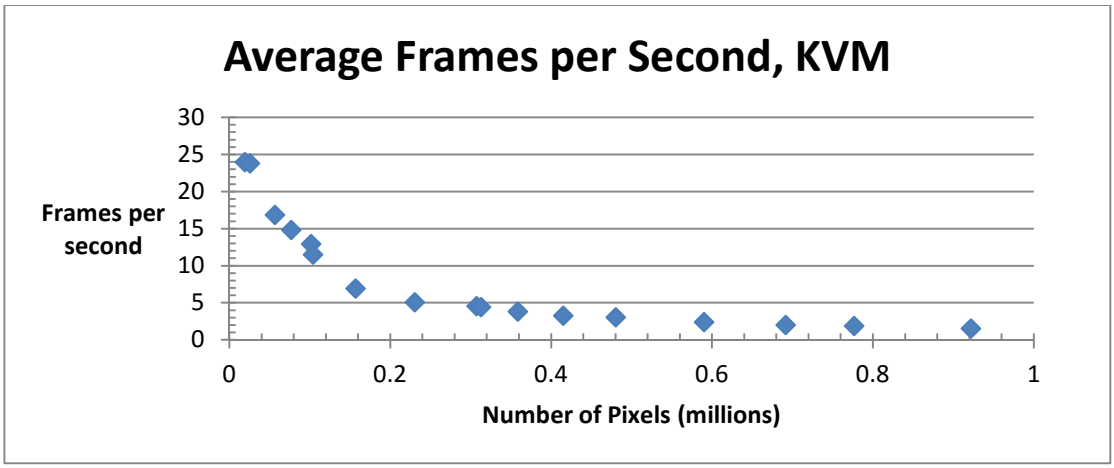

FIGURE 10: GRAPH PLOTTING AVERAGE FRAMES PER SECOND FOR A GIVEN RESOLUTION USING THE KVM CONFIGURATION

The average execution time and average frames per second graphs are consolidated in Figure 12 and Figure 13.

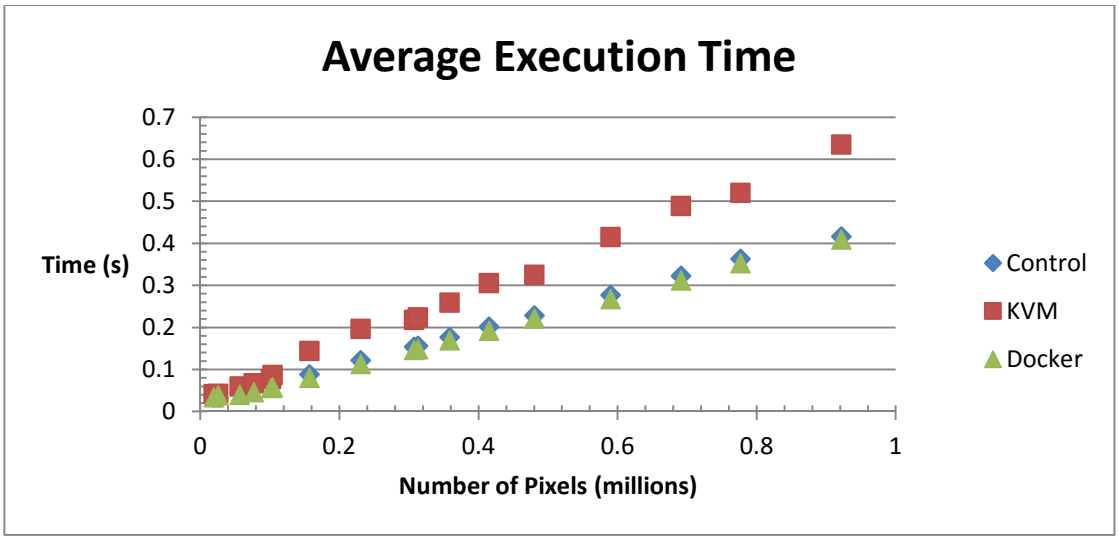

FIGURE 11: AVERAGE EXECUTION TIME FOR ALL THREE CONFIGURATIONS

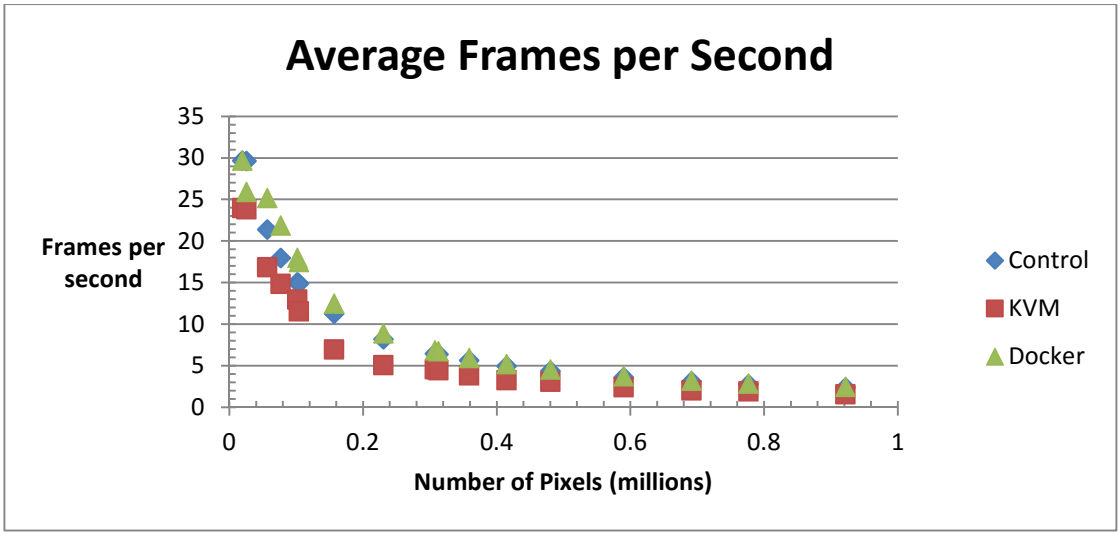

FIGURE 12: AVERAGE FRAMES PER SECOND FOR ALL THREE CONFIGURATIONS

5.2 Worst-case execution time and frames per second

This subsection states the results of the measured worst-case execution time for each configuration and the resulting frames per second. It will first list the results using

the control configuration, then the Docker configuration, and finally the KVM configuration.

5.2.1 Control Results

The results were collected with the program processing the frame with the pre-specified resolution captured from the web camera. The data for the execution time was plotted using the number of pixels processed as the x-axis and the average execution time as the y-axis. The data for the frames per second was plotted using the number of pixels processed as the x-axis and the average frames per second on the y-axis. The data collected is displayed in Table 4 and the graphs illustrated in Figure 12 and Figure 13. The data shows that as the resolution increases, the worst-case execution time increases linearly and the resulting frames per second decreases inversely.

Resolution (Width/Height)	Number Pixels	Worst-case execution time (s)	Worst-case frames per second
160/120	19200	0.040972	24.40696
176/144	25344	0.042135	23.73344
320/176	56320	0.048494	20.62111
320/240	76800	0.068577	14.58215
352/288	101376	0.079682	12.54996
432/240	103680	0.088109	11.34964
544/288	156672	0.091912	10.87993
640/360	230400	0.135711	7.368582
640/480	307200	0.185371	5.394598
752/416	312832	0.176946	5.65144
800/448	358400	0.213617	4.681283
800/600	480000	0.261898	3.818279
864/480	414720	0.218917	4.56795
960/720	691200	0.342829	2.916907
1024/576	589824	0.284723	3.512189
1184/656	776704	0.463986	2.155237
1280/720	921600	0.527674	1.895111

TABLE 4: THE WORST-CASE EXECUTION TIME AND FRAMES PER SECOND FOR A GIVEN RESOLUTION USING THE CONTROL CONFIGURATION

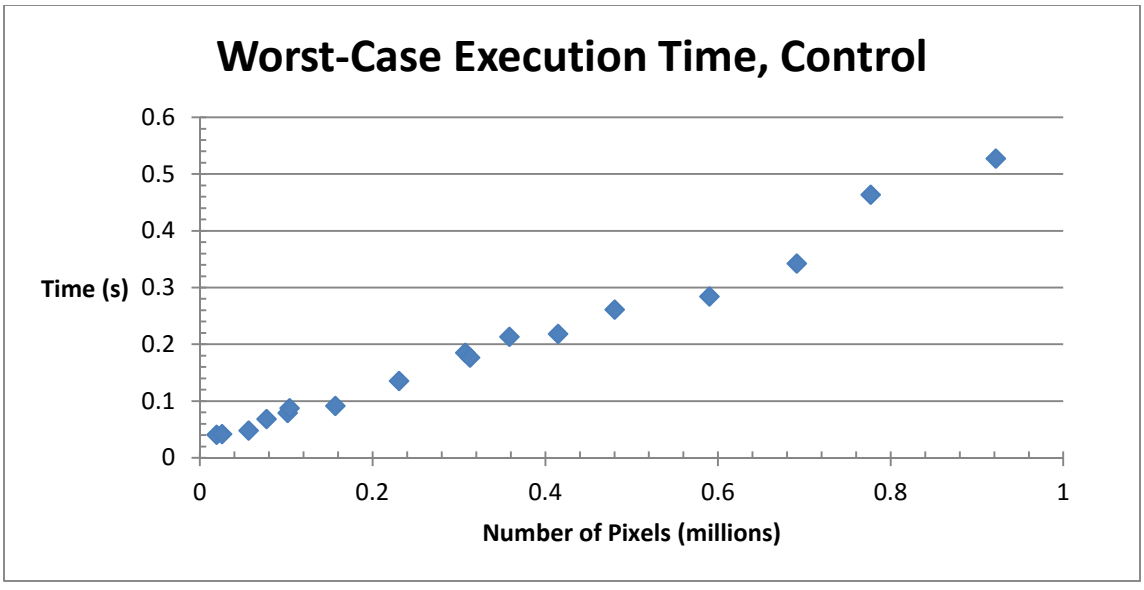

FIGURE 13: GRAPH PLOTTING WORST-CASE EXECUTION TIME FOR A GIVEN RESOLUTION USING THE CONTROL CONFIGURATION

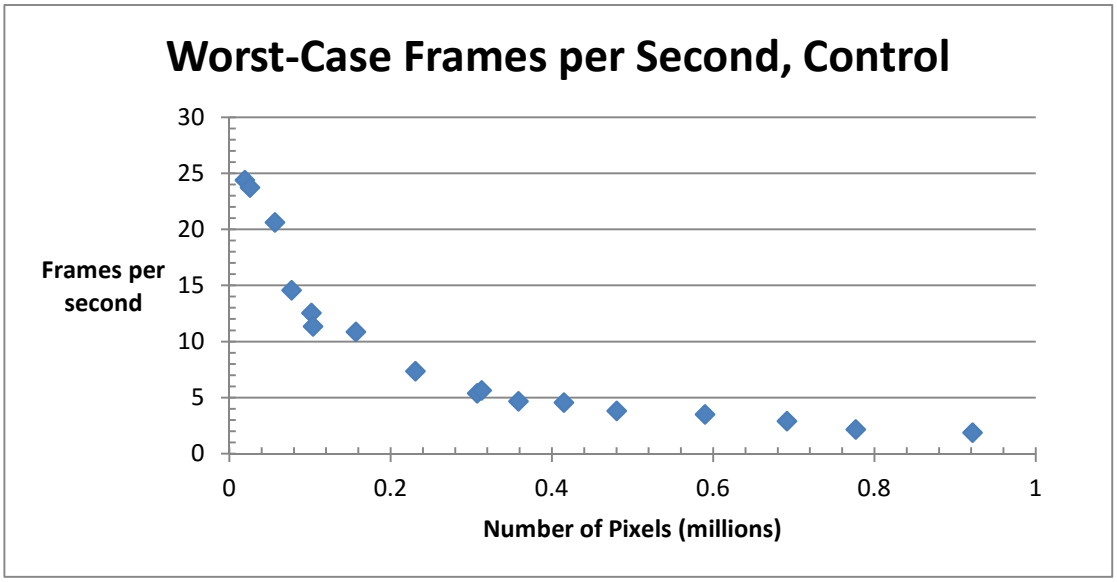

FIGURE 14: GRAPH PLOTTING WORST-CASE FRAMES PER SECOND FOR A GIVEN RESOLUTION USING THE CONTROL CONFIGURATION

5.2.2 Docker Results

The results were collected with the program processing the frame with the pre-specified resolution captured from the web camera. The data for the execution time was plotted using the number of pixels processed as the x-axis and the average execution time as the y-axis. The data for the frames per second was plotted using the number of pixels processed as the x-axis and the average frames per second on the y-axis. The data collected is displayed in Table 5 and the graphs illustrated in Figure 14 and Figure 15. The data shows that as the resolution increases, the worst-case execution time increases linearly and the resulting frames per second decreases inversely.

Resolution (Width/Height)	Number Pixels	Worst-case Execution Time	Worst-case Frames per Second
160/120	19200	0.044513	22.46535
176/144	25344	0.109603	9.123842
320/176	56320	0.04621	21.64032
320/240	76800	0.0488	20.49176
352/288	101376	0.086385	11.57606
432/240	103680	0.062398	16.0262
544/288	156672	0.167896	5.956074
640/360	230400	0.115319	8.671579
640/480	307200	0.152809	6.544118
752/416	312832	0.155746	6.420715
800/448	358400	0.182754	5.471848
800/600	480000	0.229165	4.363665
864/480	414720	0.201617	4.959896
960/720	691200	0.334239	2.991872
1024/576	589824	0.299616	3.337608
1184/656	776704	0.363382	2.751928
1280/720	921600	0.428831	2.331922

TABLE 5: THE WORST-CASE EXECUTION TIME AND FRAMES PER SECOND FOR A GIVEN RESOLUTION USING THE DOCKER CONFIGURATION

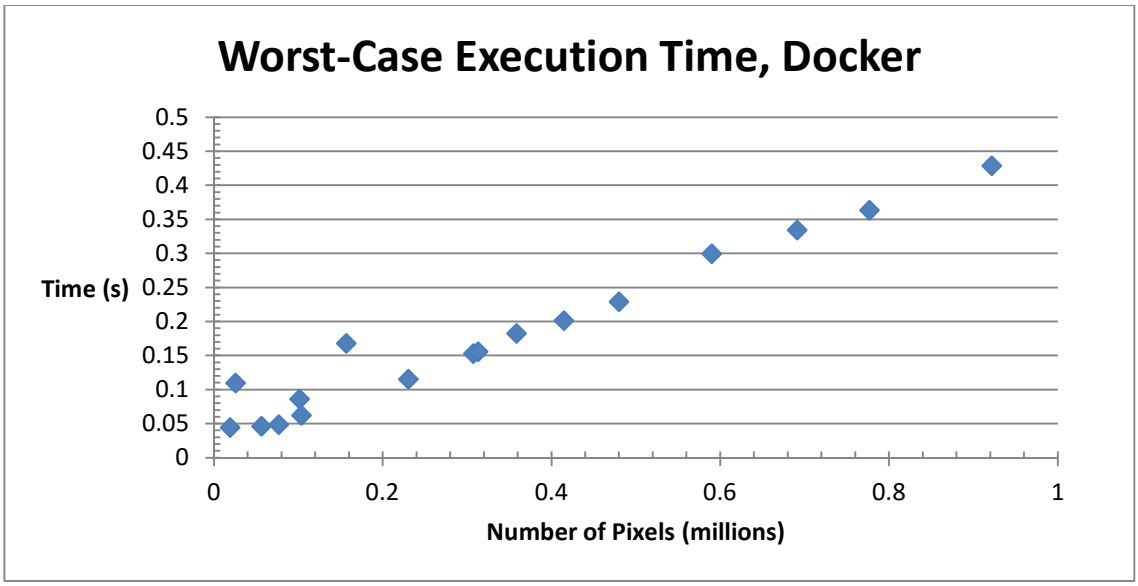

FIGURE 15: GRAPH PLOTTING WORST-CASE EXECUTION TIME FOR A GIVEN RESOLUTION USING THE DOCKER CONFIGURATION

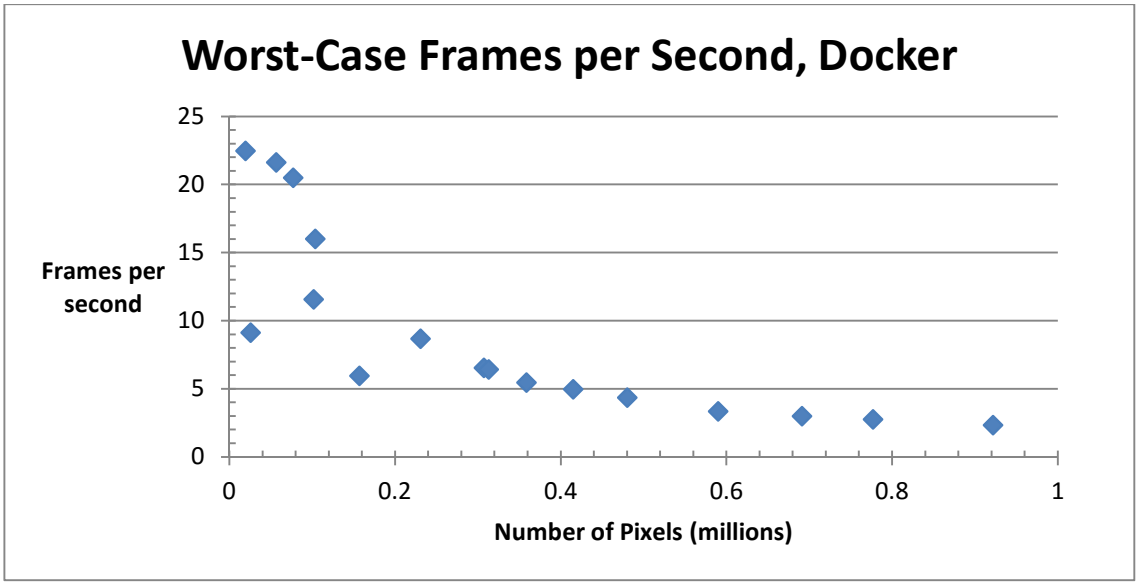

FIGURE 16: GRAPH PLOTTING WORST-CASE FRAMES PER SECOND FOR A GIVEN RESOLUTION USING THE DOCKER CONFIGURATION

5.2.3 KVM Results

The results were collected with the program processing the frame with the pre-specified resolution captured from the web camera. The data for the execution time was plotted using the number of pixels processed as the x-axis and the average execution time

as the y-axis. The data for the frames per second was plotted using the number of pixels processed as the x-axis and the average frames per second on the y-axis. The data collected is displayed in Table 6 and the graphs illustrated in Figure 16 and Figure 17. The data shows that as the resolution increases, the worst-case execution time increases linearly and the resulting frames per second decreases inversely.

Resolution (Width/Height)	Number of Pixels	Worst-case Execution Time	Worst-case Frames per Second
160/120	19200	0.102516	9.754597
176/144	25344	0.131045	7.630941
320/176	56320	0.446	2.242153
320/240	76800	0.367	2.724796
352/288	101376	0.534	1.872659
432/240	103680	0.204905	4.880317
544/288	156672	0.926605	1.079209
640/360	230400	0.802903	1.24548
640/480	307200	0.43555	2.29595
752/416	312832	0.48654	2.055332
800/448	358400	0.475937	2.101119
800/600	480000	0.553952	1.805211
864/480	414720	0.599279	1.668671
960/720	691200	0.751013	1.331534
1024/576	589824	0.669749	1.493097
1184/656	776704	0.962022	1.039478
1280/720	921600	1.020919	0.97951

TABLE 6: THE WORST-CASE EXECUTION TIME AND FRAMES PER SECOND FOR A GIVEN RESOLUTION USING THE KVM CONFIGURATION

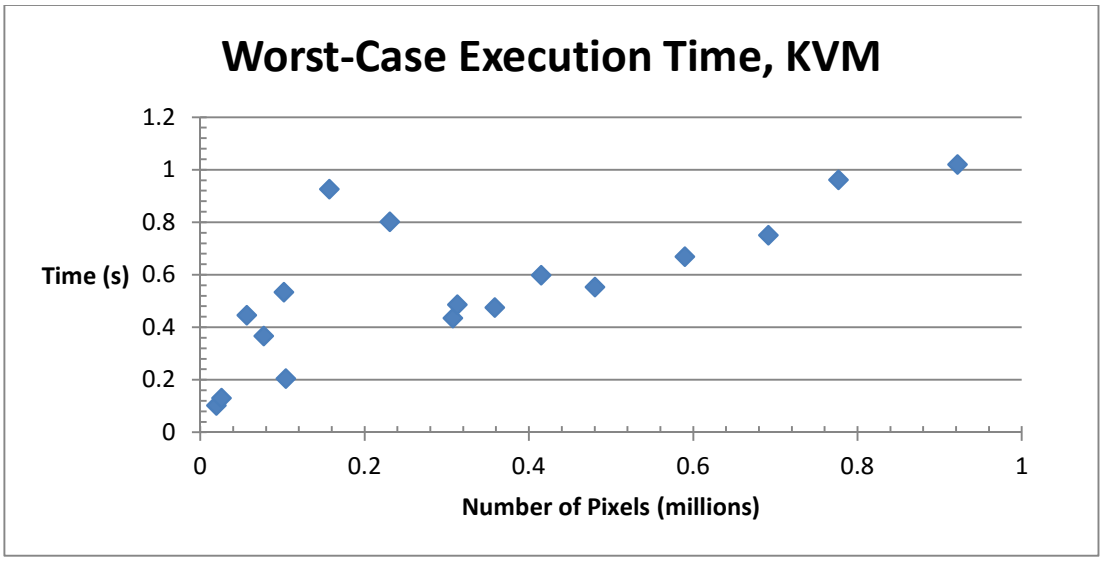

FIGURE 17: GRAPH PLOTTING WORST-CASE EXECUTION TIME FOR A GIVEN RESOLUTION USING THE KVM CONFIGURATION

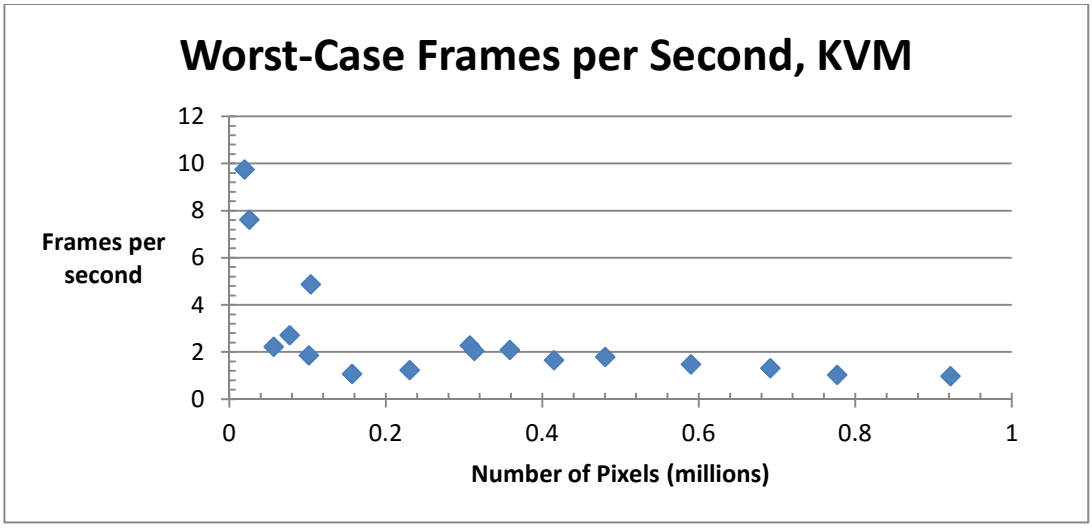

FIGURE 18: GRAPH PLOTTING WORST-CASE FRAMES PER SECOND FOR A GIVEN RESOLUTION USING THE KVM CONFIGURATION

The worst-case execution time and worst-case frames per second graphs are consolidated in Figure 20 and Figure 21.

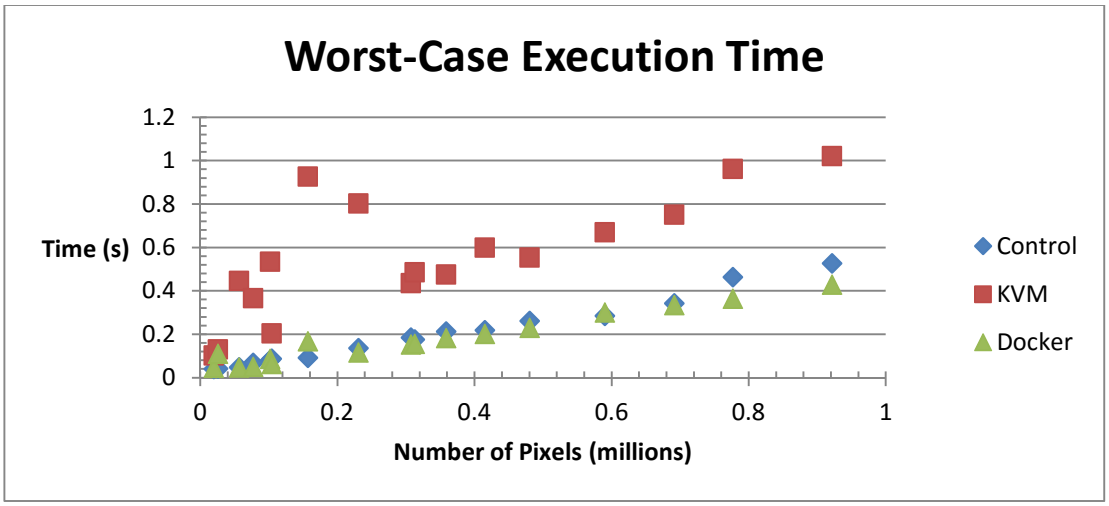

FIGURE 19: WORST-CASE EXECUTION TIME FOR ALL THREE CONFIGURATIONS

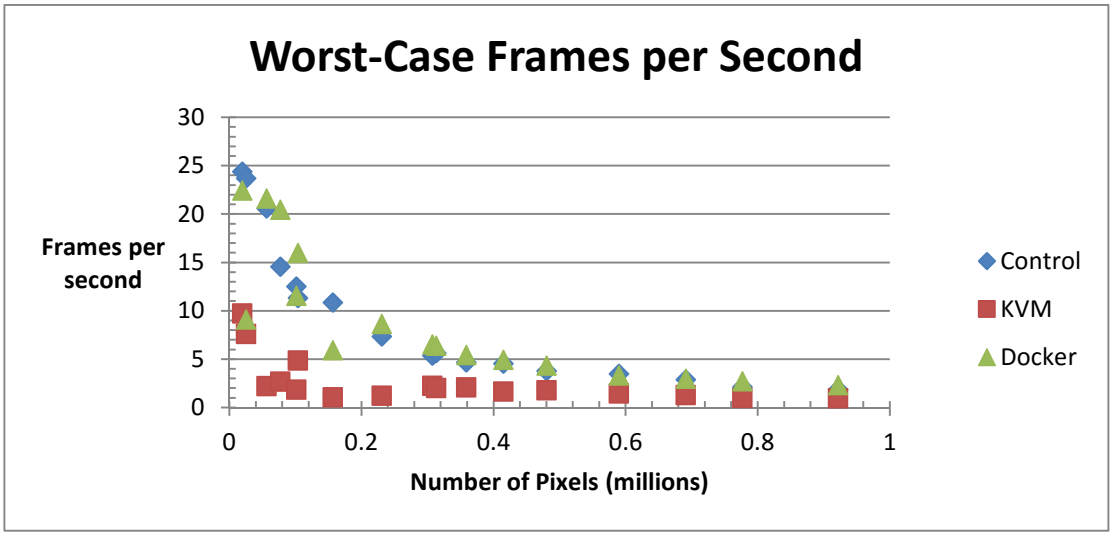

FIGURE 20: WORST-CASE FRAMES PER SECOND FOR ALL THREE CONFIGURATIONS

5.3 Average Power Consumption Results

This subsection states the results of the measured average power consumption for the KVM, Docker, and control configurations. The results were collected with the program processing the frame with the pre-specified resolution captured from the web camera. The data for the average power was plotted using the number of pixels processed as the x-axis and the average power value as the y-axis. The data collected is

displayed in Table 7 and the graphs are illustrated in Figure 18, Figure 19, and Figure 20. The data shows that for the KVM configuration, the power increases slightly as the number of processed pixels increases, then remained roughly constant at 6.6 watts. For the control configuration, the power decreased slightly as the number of pixels increased, then remained roughly constant at 6.2 watts with some variation. For the Docker configuration, the power increased slightly as the number of pixels increased, then remained constant at roughly 6.1 watts with some variation.

Resolution (Width/Height)	Number of Pixels	Average power, KVM (W)	Average power, Docker (W)	Average power, control (W)
160/120	19200	6.3	5.9	6.5
176/144	25344	6.3	5.9	6.6
320/176	56320	6.4	6	6.5
320/240	76800	6.4	6.1	6.4
352/288	101376	6.4	6.2	6.4
432/240	103680	6.5	6.1	6.4
544/288	156672	6.4	6.2	6.4
640/360	230400	6.4	6.1	6.3
640/480	307200	6.5	6.1	6.3
752/416	312832	6.6	6	6.2
800/448	358400	6.6	6	6.1
800/600	480000	6.6	6	6.2
864/480	414720	6.6	6.3	6.2
960/720	691200	6.6	6.1	6.2
1024/576	589824	6.6	6.2	6.1
1184/656	776704	6.6	6.1	6.2
1280/720	921600	6.5	6.1	6.3

TABLE 7: THE AVERAGE POWER FOR A GIVEN RESOLUTION USING THE KVM, DOCKER, AND CONTROL CONFIGURATIONS

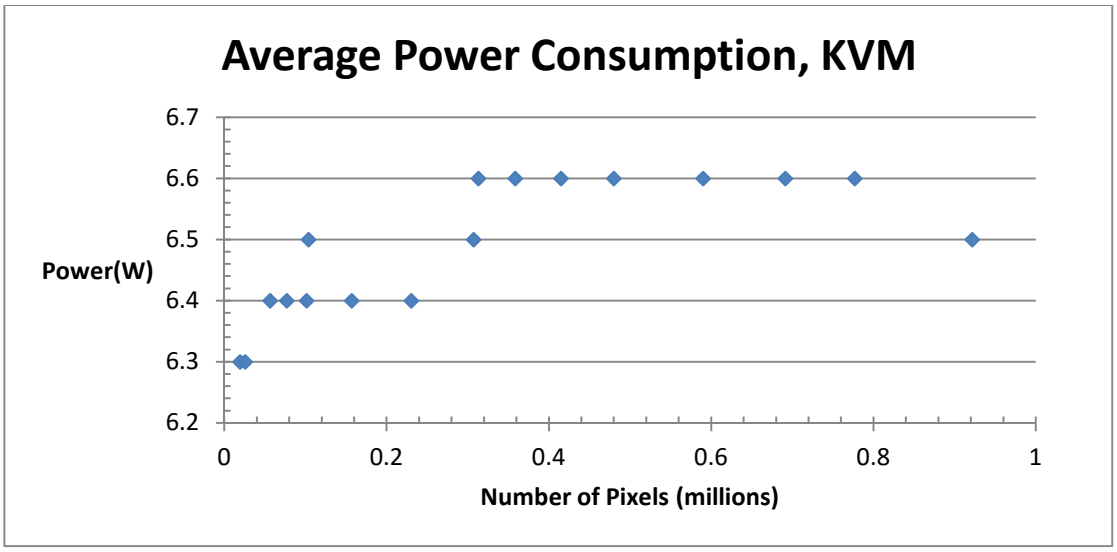

FIGURE 21: GRAPH PLOTTING AVERAGE POWER CONSUMPTION FOR A GIVEN RESOLUTION USING THE KVM CONFIGURATION

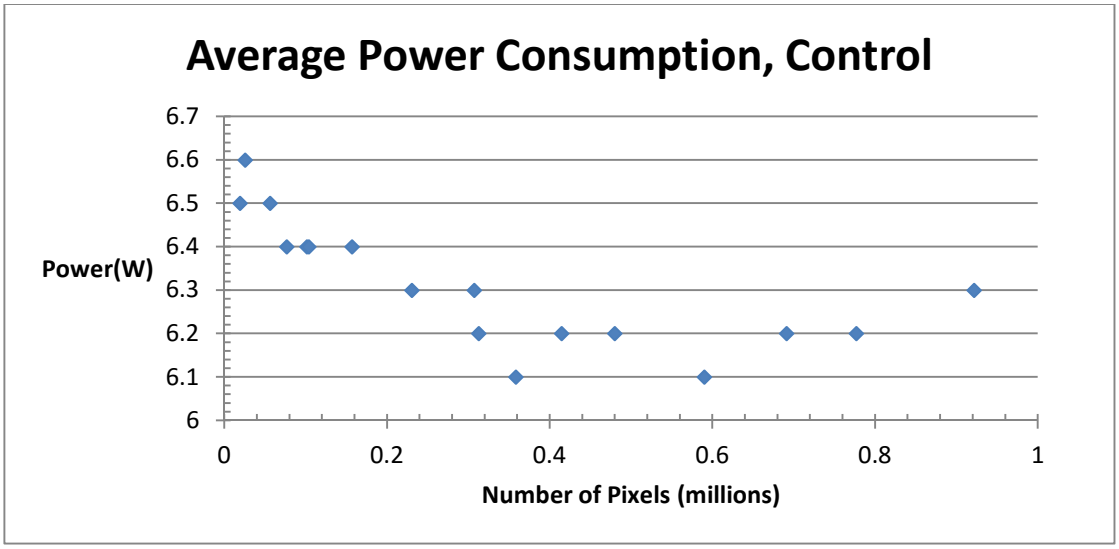

FIGURE 22: GRAPH PLOTTING AVERAGE POWER CONSUMPTION FOR A GIVEN RESOLUTION USING THE CONTROL CONFIGURATION

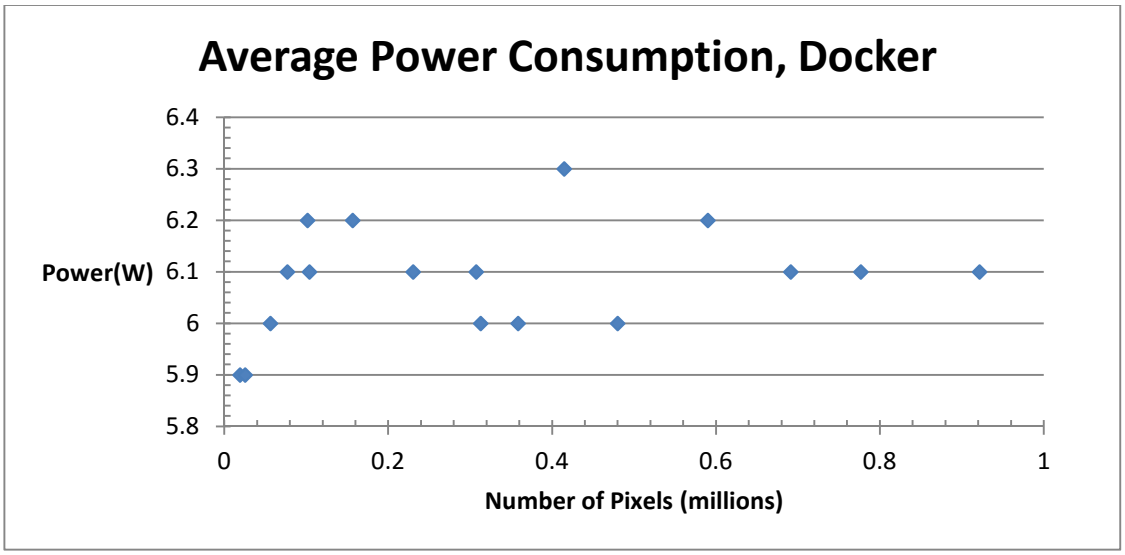

FIGURE 23: GRAPH PLOTTING AVERAGE POWER CONSUMPTION FOR A GIVEN RESOLUTION USING THE DOCKER CONFIGURATION

The average power graphs are consolidated in Figure 25.

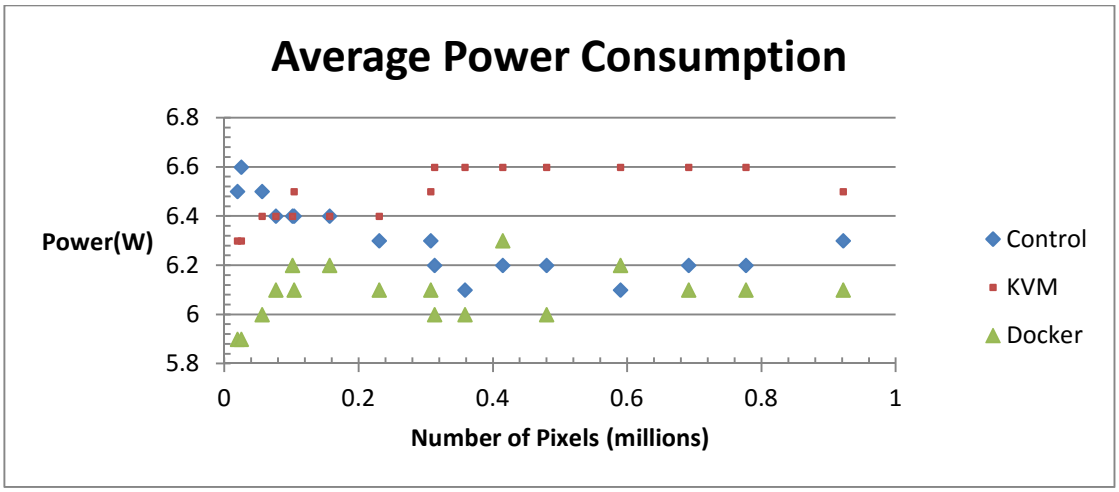

FIGURE 24: AVERAGE POWER CONSUMPTION FOR ALL THREE CONFIGURATIONS

Chapter 6

Discussion

This chapter analyzes the results and their implications for future research. The methodology and results will first be summarized. Next the results will be discussed. The limitations of the study will be noted and implications for future research will be analyzed. Finally, concluding remarks will be made on the results and analysis.

6.1 Summary of Results

The average execution time and the resulting frames per second were calculated and plotted. The control data shows that when the resolution is 176x144 and below, the average execution time is roughly constant at .0337 seconds, but then increases linearly as the number of pixels is increased. The control data also shows that when the resolution is 176x144 and below, the average frames per second is roughly constant at 29.7 FPS, but then decreases inversely as the number of pixels is increased. The Docker execution time data shows that the average execution time increases linearly as the number of pixels is increased. The Docker frames per second data shows that the average frames per second decreased inversely as the number of pixels is increased. The KVM execution time data shows that when the resolution is 176x144 and below, the average execution time is roughly constant at .0417 seconds, but then increases linearly as the number of pixels is increased. The KVM frames per second data shows that when the resolution is 176x144 and below, the average frames per second was roughly constant at 23.9 frames per second, but then decreased inversely as the number of pixels was increased.

The worst-case execution time and resulting frames per second were calculated from the sampled data and plotted. The control data shows that as the resolution increases, the worst-case execution time increases linearly and the resulting frames per second decreases inversely. The Docker data shows that as the resolution increases, the worst-case execution time increases linearly and the resulting frames per second decreases inversely. The KVM data shows that as the resolution increases, the worst-case execution time increases linearly and the resulting frames per second decreases inversely.

The data shows that for the KVM configuration, the power increases slightly as the number of processed pixels increases, then remained roughly constant at 6.6 watts. For the control configuration, the power decreased slightly as the number of pixels increased, then remained roughly constant at 6.2 watts with some variation. For the Docker configuration, the power increased slightly as the number of pixels increased, then remained constant at roughly 6.1 watts with some variation.

6.2 Discussion of the Results

This section discusses in detail the results of the experiments and the factors involved in affecting the end results of the experiments. The first subsection will discuss the average execution time and frames per second as well as worst-case execution time and frames per second. The second subsection will discuss the average power consumed.

6.2.1 Performance Metrics Discussion

The average execution time increased as the number of pixels per frame increased across all three configurations. The rationale behind this is simple. With

more pixels to process per frame, the image processing algorithm takes longer to process, increasing the time to go through one iteration of processing a frame. The control and Docker configurations start at a max value of roughly 29.67 frames per second, while the KVM configuration starts out at roughly 24 frames per second. The control and Docker start at roughly 30 frames per second because that is the maximum frames per second at which the webcam can capture frames. Since the time to process a 160x120 image was less than the capture rate of the web camera, the program had to wait until a new frame was captured by the web camera. This allowed for optimal usage of the system, and this would be considered the real-time requirements to complete the processing of one frame before another frame was captured. With the KVM configuration and increasing resolution under the control and Docker configurations, 30 frames per second was not achieved. KVM did not achieve this with the tested resolutions because KVM is a type-1 hypervisor. The program was forced to interface the web camera through the hypervisor layer in addition to the operating system layer, creating additional overhead to process a captured frame. This delay violated the real-time requirement of processing an image before .033 seconds had elapsed, the necessary time needed to maintain 30 frames per second. The real-time requirement for the KVM, control, and Docker configurations were repeatedly violated when the number of pixels processed increased. As the data shows, 30 frames per second is a soft real-time requirement. When the program is faced with violating the 30 frames per second requirement, it chooses to continue processing the current frame instead of fetching a new frame from the web camera. This results in the degraded performance of the displayed

video stream from the web camera at a sub-optimal frame rate. The Docker configuration also appears to have a slightly better execution time and frames per second than the control configuration. This deviation is due to the different steps involved to interface with the web camera. With the control configuration, when it makes a request to access the web camera, the application does not have permission to directly access the web camera. Instead, it calls a trap instruction, which saves the program on the kernel stack and raises the privilege level to kernel mode. The trap is handled by the operating system and executes the privileged instruction. Once complete, the operating system executes a return-from-trap instruction, which restores the program from the kernel back to the user program and lowers the privilege level back to user mode. The application then completes the trap execution and continues onto the next instruction. This series of steps introduces significant overhead when the control attempts to retrieve a frame from the web camera. However, Docker bypasses all these steps. When Docker is initialized, USB devices need to be defined in order for Docker containers to access them on their own. Due to the way Docker works with the host operating system, all Docker requests to the specified hardware are deemed to be in kernel mode by the operating system, not user mode. This completely bypasses the previous steps of saving the program to the kernel, going into kernel mode, executing the instruction, saving the program back to the application, going back into user mode and continuing execution. The time required to retrieve a frame from the web camera is orders of magnitude faster with Docker than the control configuration. The experiments showed that Docker reduces the time it takes to retrieve a frame from

the web camera by 0.007 to 0.012 seconds per frame compared to the control configuration. This is evident when looking at the resulting execution times between the Docker and control configurations. When this difference in frame retrieval time is taken into account, the rest of the time it takes to process a frame under the Docker configuration is nearly identical to the control configuration's execution time due to the slight overhead the Docker configuration introduces.

The worst-case execution time and resulting frames per second followed a similar, though rougher, trend illustrated with the average execution time. As the number of pixels processed increased, the worst-case execution time tended to increase, displaying a positive correlation between the number of pixels processed and the resulting worst-case execution time. This is logical, since more pixels would impact the worst-cast execution time as well the average execution time. The variance in data, however, can be attributed to a couple of factors. The first factor was the absence of an isolated system running on the development board. Although the system was implemented to emulate a dedicated frame processing system, other essential functions originating from the operating system ran in the background. The operating system could unexpectedly run when the program was processing a frame due to its higher priority, severely impacting the execution time of a particular frame. The second factor is the limited processing power available for frame processing. Although there were two cores available to the program, only one core was utilized due to the program running serially. The lack of parallelism misses out on a possible optimization and quickly saturates the usage of one core, exposing the possibility of overhead if another system procedure takes away the

core the program is working on for its own use due to it being higher priority. The impact of these factors on the worst-case execution time can be observed from the data. When the average frames per second is at its highest number, the overhead imposed by losing the CPU for a period of time lowered the calculated frames per second for that particular frame by 5 or 12 when compared to the average frames per second. As the average execution time increased, however, operating system overhead made less of an impact due to the overhead remaining relatively constant due to the operating system's needs remaining the same no matter what workload was being executed. Looking more closely at each configuration, the KVM configuration had overall higher worse case execution times compared to the other two configurations due to its average execution times being higher due to the higher overhead imposed by the KVM virtualization layer when interfacing between the firmware and operating system layers. The control and Docker worst-case execution times exhibited similar worst-case execution times, though there was some variance due to the Docker configuration having privileged access to the web camera and also due to the previously mentioned factor of the operating system taking the CPU away from the program when a particular frame was processed due to the operating system's higher priority. In the end, the impact of one of these worst-case execution times on the overall operation of the program was minimal due to the localized nature of operating system calls and the soft real-time requirements of the system. Although the operating system imposed overhead, it did so on a tiny window of time. This resulted in only one frame being impacted by operating system calls. In addition, the soft real-time requirements of the system

meant that the operating system overhead would only impact the frame currently being processed and not the execution time of future frames. This is because, as stated previously, when the real-time requirement is violated by the processing time of a particular frame, the processing continues for the current frame until complete. This is a subtle point to consider because in a system such as a hard real-time system, violating the real-time requirements could result in severely degraded performance or cause the system to fail. In this particular system, violations are forgiven, resulting in a few frames in the collected samples being slowed down by the CPU being taken away by the operating system while the vast majority of frames being unaffected since the operating system calls only run in small intervals of time.

6.2.2 Power Discussion

Across all three configurations, the power consumption varied slightly. The overall average across all average power data for the control was 6.311 watts, the overall average across all average power data for the Docker configuration was 6.082 watts, and the overall average across all average power data for the KVM configuration was 6.488 watts. Between the farthest data point and the corresponding average, all average power data for the control fell within 4.76% of its overall power average, all average power data for the Docker configuration fell within 3.27% of its overall power average, and all average power data for the KVM configuration fell within 3.08% of its overall power average. The power consumption varied slightly with a varied workload, but not enough to be considered significant. However, some factors can account for the variance in power in a varied workload. The first factor is the usage of the CPU cores. Each core

will draw a certain amount of power depending on its usage. For the control and Docker, one core was dedicated towards program execution while one core stayed mostly idle, capping the power drawn once one core was fully utilized. For the KVM, however, the system delegated the KVM virtualization overhead to the other core, resulting in more power being drawn with both cores being kept busy. This was why KVM on average consumes more power than the control and Docker. Another factor is the time to get a frame from the web camera. With the Docker configuration having an optimized hardware pass-through to the web camera, less CPU resources were dedicated to getting access to the web camera and getting the frame from the camera. This resulted in less power being drawn from the CPU, resulting in the slight decrease in power seen when comparing the Docker and control configurations. The last factor is the utilization of the GPU. The GPU drew less power when it had to display less frames on the screen. This resulted in slightly decreased power being drawn from the GPU as the frames per second decreased with an increased workload. With all this in mind, the difference in power consumption between Docker and KVM on average was only about 0.4 watts, with the control being in the middle of this range. Docker is the best if power consumption is a high priority, but the differences in power amongst all three configurations is minimal.

6.3 Limitations of the Experiments

There were various limitations that limited what could be performed in the experiments. The web camera could only support 17 resolutions, limiting what experimental data could be collected. The application program ran serially, leaving one

of the two cores mostly in idle mode for the control and Docker configurations. The scarcity of available open-source free hypervisors limited what could be tested for each software configuration. The limited amount of funds available limited testing to one development board. Time constraints limited quantitative experiments to performance and power metrics in one soft real-time embedded system.

6.4 Implications for Future Research

These experiments provided a starting point for quantitatively analyzing the performance of hypervisors in a soft real-time system. There are a number of different ways that future research can build from this starting point. Future research can focus on higher resolutions like ones associated with 1080p and 4K resolutions. As video streams progress to higher resolutions, conducting experiments with these resolutions would prove beneficial. Another focus point could be on parallel applications operating in configurations involving virtualization. The interaction of parallel applications in a virtualized setting could provide interesting experiments where one virtualized application would be able to interact with multiple cores and see if there are tradeoffs to this approach. The experiments could be extended to different hypervisors such as Xen or VirtualBox. These experiments could provide more quantitative data on how the experiments operate in different software configurations. Further researchers could also conduct the experiments on different hardware. Such experiments could further quantify how each software configuration would behave with weaker or more powerful hardware. Finally, research could be conducted in quantifying the security benefits of virtualization schemes. Such experiments could conduct classes of malware attacks that try to access particular resources in the software or hardware and determine which groups of attacks

are mitigated with certain virtualization schemes. These proposed ideas would further expand the data collected in the experiments presented in this thesis.

6.5 Conclusions

This thesis conducted experiments to quantify the performance impacts virtualization schemes have on a soft real-time embedded system. The experiments were conducted with hardware chosen to emulate a power and performance-limited embedded system with the desire to balance performance with security requirements. The best configuration for this proposed system depends on the explicit security and performance requirements for the system. If security is a high priority, the KVM configuration provides the highest-tested security capabilities due to it being a type-1 hypervisor at the cost of overhead that noticeably impacts the performance of the system. If performance is a higher priority over security, the Docker configuration provides nearly identical performance to the control configuration and provides isolation between the application and operating system layers due to it being a container-based hypervisor. What is clear is that completely unprotected real-time embedded systems are an increasingly risky option for their users. With options like Docker that achieve almost identical performance to non-virtualized systems, the designer gives up little in performance to implement some level of cyber-protection in the system. As embedded malware increases in number, so will the need to incorporate security measures into embedded systems to mitigate the threat of malware. Hypervisors provide an attractive option within the trade-space in which performance can be exchanged for enhanced protection against malware in soft real-time embedded systems.

References

Andrew Zeliff, Cal Roman, Sam Allen. "State of Space Cyber, Midterm Report." 2014.

Asberg, M., et al. "Towards real-time scheduling of virtual machines without kernel modifications ." *Emerging Technologies & Factory Automation (ETFA), 2011 IEEE 16th Conference on* . Toulouse: IEEE, 2011. 1 - 4 .

Avanzini, A., et al. "Integrating Linux and the real-time ERIKA OS through the Xen hypervisor ." *Industrial Embedded Systems (SIES), 2015 10th IEEE International Symposium on* . Siegen: IEEE, 2015. 1 - 7 .

Bruns, F., et al. "An Evaluation of Microkernel-Based Virtualization for Embedded Real-Time Systems." *Real-Time Systems (ECRTS), 2010 22nd Euromicro Conference on*. Brussels: IEEE, 2010. 57-65.

Chen, Huacai, et al. "Adaptive Audio-aware Scheduling in Xen Virtual Environment." *Computer Systems and Applications (AICCSA), 2010 IEEE/ACS International Conference on* . Hammamet: IEEE, 2010. 1-8.

Gupta, Diwaker, et al. "Enforcing Performance Isolation Across Virtual Machines in Xen." *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2006. 342-362.

Habib, Irfan. "Virtualization with KVM." *Linux Journal* (2008): Article 8.

Hwang, Joo-Young, et al. "Xen on ARM: System Virtualization using Xen Hypervisor for ARM-based Secure Mobile Phones." *Consumer Communications and Networking Conference*. Las Vegas, NV: IEEE, 2008. 257-261.

Jablkowski, Boguslaw and Olaf Spinczyk. "CPS-Xen: A Virtual Execution Environment for Cyber-Physical Applications." Jablkowski, Boguslaw and Olaf Spinczyk. *Architecture of Computing Systems – ARCS 2015* . Porto: Springer International Publishing, 2015. 108-119.

Jing, Wei, Nan Guan and Wang Yi. "Performance isolation for real-time systems with Xen hypervisor on multi-cores." *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2014 IEEE 20th International Conference on* . Chongqing: IEEE, 2014. 1-7.

Lee, Jaewoo, et al. "Realizing Compositional Scheduling through Virtualization ." *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th* . Beijing: IEEE, 2012. 13 - 22.

Lee, Jeong Gun, Kyung Woo Hur and Young Woong Ko. "Minimizing Scheduling Delay for Multimedia in Xen Hypervisor." *Communications in Computer and Information Science*. Springer Berlin Heidelberg, 2011. 96-108.

Lee, Min, et al. "Supporting soft real-time tasks in the xen hypervisor." *Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments* . New York: ACM, 2010. 97-108.

Ludmila Cherkasova, Diwaker Gupta, and Amin Vahdat. "Comparison of the Three CPU Schedulers in Xen." *ACM SIGMETRICS Performance Evaluation Review* 2007.

M. Masmano, I. Ripoll, and A. Crespo. "XtratuM: a Hypervisor for Safety Critical Embedded Systems." *Eleventh Real-Time Linux Workshop*. Dresden, Germany: Real-Time Linux Foundation Working Group, 2009. 263-272.

Masrur, Alejandro, et al. "Designing VM schedulers for embedded real-time applications." *CODES+ISSS '11 Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis* . New York: ACM, 2011. 29-38 .

Ongaro, Diego, Alan L. Cox and Scott Rixner. "Scheduling I/O in virtual machine monitors." *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments* . New York: ACM, 2008. 1-10.

Rouse, Margaret. *Real-time application (RTA) definition*. n.d. SearchUnified Communications. December 2015. <http://searchunifiedcommunications.techtarget.com/definition/real-time-application-RTA>.

Soltesz, Stephen, et al. "Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors." *EuroSys '07 Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*. New York: ACM, 2007. 275-287.

Vaughan-Nichols, S.J. "New Approach to Virtualization Is a Lightweight." Vaughan-Nichols, S.J. *Computer*. IEEE, 2006. 12-14.

VMware Inc. "Understanding Full Virtualization, Paravirtualization, and Hardware Assist." 2007. *VMware.* VMware Inc. 13 February 2015. <http://www.vmware.com/files/pdf/VMware_paravirtualization.pdf>.

Xavier, M.G., et al. "Container-Based Virtualization for High Performance Computing Environments ." *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on* . Belfast: IEEE, 2013. 233 - 240 .

Xi, Sisu, et al. "RT-Xen: Towards real-time hypervisor scheduling in Xen ." *International Conference on Embedded Software*. Taipei: IEEE, 2011. 39-48.

Yu, Peijie, et al. "Real-time Enhancement for Xen Hypervisor ." *Embedded and Ubiquitous Computing (EUC), 2010 IEEE/IFIP 8th International Conference on*. Hong Kong: IEEE, 2010. 23 - 30 .

Zhang, Jun, et al. "Performance analysis towards a KVM-Based embedded real-time virtualization architecture." *Computer Sciences and Convergence Information Technology (ICCIT), 2010 5th International Conference on* . Seoul: IEEE, 2010. 421 - 426.

Zonghua Gu, Qingling Zhao. "A State-of-the-Art Survey on Real-Time Issues in Embedded Systems Virtualization." *Journal of Software Engineering and Applications* (2012): 277-290.