Faculty of Science and Technology
Department of Computer Science

Reconstructing Omni-kernel control flow

—

Christopher Lomsdalen Haugen

INF-3981 Master thesis in Computer Science - December 2014

“Computers are good at following instructions, but not at reading your mind. ”
–Donald Knuth

“Beware of bugs in the above code;
I have only proved it correct, not tried it.”
–Donald Knuth

Abstract

Today, with the prevalence of many- and multi-core systems has it been sparked a new interest for programming models that permits developer to exploit their resources. This has sparked renewed interest in creating larger event-based systems, systems where *stack ripping* occurs and with an obfuscated control flow. Both increases the complexity of debugging errors. During the development of the event-based experimental research OS and VMM Vortex saw we the need for tools that could aid developers to handle these challenges.

This thesis design and implements two tools that allow users to gain insights into an obfuscated control flow and see when and why a state change was done. We propose a design and implements two tools that are *simple*, *flexible*, and *lightweight* enough to live inside of the critical path of event processing in Vortex. First is a tool to observe the messages being passed to and from one resource, enabling the de-obfuscation of the control flow. The second tool uses the built-in debugging tool in modern CPU to tie state access and change to the processing of one event.

Both of these tools creates debug messages that are being visualized in an remote client.

Acknowledgements

I would like to express my great appreciation to my supervisor Dr Åge Andre Kvalnes for his time and help during the work with this thesis. His knowledge about Operating systems and Vortex is truly impressive and helped greatly with this thesis. I would also like to thank everyone at the iAD group for their input, discussions, and helping me with getting up to speed on Vortex during a short semester.

I would like to thank all of my classmates for their help, support, discussions, and for being great friends.

Finally, I would thank my parents and siblings for their support and patience through the years. Lastly, I would like to thank my girlfriend, Ragnhild Kosmo Holm, for her support and patience during my work with this thesis. Her help and patience has been much appreciated.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

API	Application programmable interface
CPU	Central Processing Unit
CR	Control Register
DE	Debugging Extensions
DR	Debug Register
GCD	Grand Central Dispatch
GPU	Graphical Processing Unit
HPC	High performance computing
HP	Hewlett-Packard
HTML	Hyper Text Markup Language
I/O	Input/Output
IAD	information Access Disruption
JSON	JavaScript Object Notation
JS	JavaScript
NIC	network interface card
OS	Operating system
TCP	Transmission Control Protocol

UiT	University of Tromsø
VMM	Virtual Machine monitor
VM	Virtual Machine

1 Introduction

Concurrent programming has become ever more important for increasing the performance of applications and systems after Central Processing Unit (CPU)-developers hit the power-wall and were unable to increase performance through clock-speed [1, 2]. Today, the prevalence of many- and multi-core systems has sparked a renewed interest in programming models to permit their exploitation.

There are two general programming models for achieving parallelism and concurrency: *threaded* and *event-based* programming [3, 4, 5]. Each model comes with its own benefits and drawbacks, exemplified by identifying shared data structures and protecting them [6] and *stack ripping* [5].

Stack ripping occurs when it is discovered that the processing of an event has to be postponed and continued at some later point. For example, processing of the event might require completion of some I/O or other processing. Intermediate processing of other events might then cause state changes or modifications which will make continued processing of the event difficult. Thus, event state typically has to be stored upon postponement and restored upon continued processing.

Postponement and continued processing of events also obfuscate control flow, making system behavior more complex and debugging more difficult [7]. Interleaved processing of messages makes it harder to deduce which message resulted in what change of state and from where the message originated. Thus, finding the offending entity causing faulty state might be a complex challenge that requires a large debugging effort.

The popularity of cloud services and availability of multi-core systems has resulted in a rebirth of virtualization technology [8, 9]. The workload of cloud customers are placed inside Virtual Machine (VM)s and Virtual Machine monitor (VMM)s handle multiplexing of the underlying hardware resources on physical machines. These demands have resulted in Operating system (OS)s being extended with VMM support [10, 11, 12].

Vortex is an experimental research OS and VMM created at University of Tromsø (UiT). Vortex implements the Omni-kernel architecture [13], which has been designed to control allocation of system resources. The Omni-kernel factors the OS kernel into multiple components that exchange messages in order to implement higher-level OS functionality. By relying on asynchronous message passing, the problems of stack ripping and obfuscation of control flow are present in the Vortex kernel.

This thesis focuses on developing tools and techniques to aid the Vortex kernel developer in tackling stack ripping and obfuscation of control flow.

1.1 Problem statement

The use of asynchronous message passing makes it difficult to deduce the cause of a state change for components in the Vortex kernel.

Specifically, this thesis investigates the problem of how to reconstruct control flow in the Vortex kernel. The thesis focuses on the design and implementation of a system that:

1. Can reconstruct message processing order at a Vortex kernel component.
2. Can deduce what message caused a specific state change in a Vortex kernel component.

1.2 Context

The thesis has been done as a part of the information Access Disruption (IAD) Centre[14] at UiT that targets core research for next generation precision, analytics and scale in the information access domain. Partially funded by the Research Council of Norway as a Centre for Research-based Innovation (SFI), iAD is directed by Microsoft Development Center (Norway) in collaboration with Accenture, Cornell University, University College Dublin, Dublin City University, BI Norwegian School of Management and the universities in Tromsø (UiT), Trondheim (NTNU) and Oslo (UiO). The iAD project investigates structuring techniques for future-generation large-scale information access applications. This includes fundamental research issues like, for instance, how to best partition an application into a set of cooperating modules, how to optimize interaction among them, how and where to deploy them, how to interact with the users, how to provide integrity, security and auditing, and how to ensure fault-tolerance.

1.3 Scope and limitations

The scope of this thesis is to devise and implement systems for gathering and reconstructing the order of messages so that control flow in an Omni-kernel system can be reconstructed. The thesis should also expand the debugging possibilities in the system by creating a system for controlling the debug registers that are present in a modern x86 CPU.

Limitations to this work is that the implementation for controlling debug register will contain an Application programmable interface (API) for developers to use and an exception handler for using the debug registers in combination with the system for reconstructing control flow. Other scenarios might exist when the exception handler created for this thesis is not the optimal solution however; this thesis leaves the task of creating exception handler based on their need up to the user.

1.4 Methodology

The final report of the ACM Task Force on the Core of Computer Science gives three major paradigms that Computer Science is divided into [15]:

Theory is rooted in mathematics. They approach problems characterize objects that are to be studied to defining a problem, create hypothesis and theorems about relationships between objects. Before they try to prove that the relationships are true. This is done to determine and interpret the results found.

Abstraction is rooted in the experimental scientific method. The approach is to investigate a phenomenon by forming hypothesis, which are turned into models and make a prediction. An experiments is designed and executed to collect data for analysis and interpret results.

Design is rooted in engineering. They solve a problem through construction of a system or device. Based on a set of stated requirements and specifications are a system designed and implemented. This system allows for testing and evaluation based on the set requirements and specifications.

Common for all paradigms is that they are split up into steps, which when needed is repeated, e.g. due to discovery of new information.

This thesis is placed into the design paradigm, given a problem we construct a system based on requirements and specifications set before testing the system.

1.5 Contribution

The contributions of this thesis is the design and implementation of:

- A tool for reconstructing control flow in the Omni-kernel architecture.
- A tool for enabling and controlling debug registers that are in the x86 processors.

1.6 Structure

The rest of the thesis is structured as follows:

Chapter 2 explains some of the technologies used in this thesis.

Chapter 3 details the architecture and implementation the tools.

Chapter 4 evaluates the architecture and details how this type of monitoring and debugging affects the performance of a Omni-kernel systems.

Chapter 5 described related work for this thesis.

Chapter 6 concludes the thesis and describes some use-cases for the tools.

2 Background

This chapter presents some of the background for the thesis and employed technologies.

2.1 Vortex

2.1.1 Virtual Machine Monitors

VMM technology has been around since the 1960s [8, 9]. The technology started out as a way to multiplex applications, providing users a way to run multiple applications at once on large expensive machines. With the decrease in cost of machines during 80s and 90s, there was corresponding decline in the usage of VMMs both in the industry and in academia. However, VMM technology became popular again during the 2000s, with the emergence of larger parallel systems and VMMs able to run on commodity hardware.

One can differentiate VMM, or hypervisor, software based on where it is situated in a system. Type-1 VMM, exemplified by Hyper-V, Xen, and Vortex, run directly on hardware or as part of a privileged OS. Type-2 VMMs [16]., on the other hand, run on top of an OS and typically implement execution environments for programs written in a specific language. Examples of type-2 VMMs include the JavaVM and the .Net environment. These two types of VMMs are illustrated in figures 2.1 and 2.2.

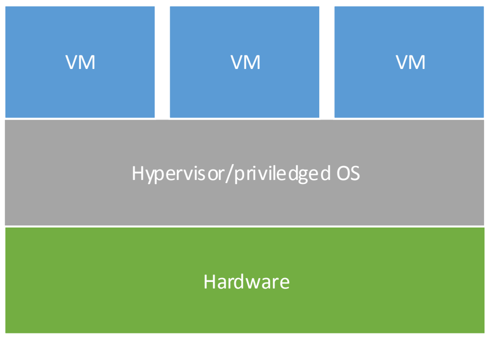


Figure 2.1: First type of Hypervisor, running directly on the hardware.

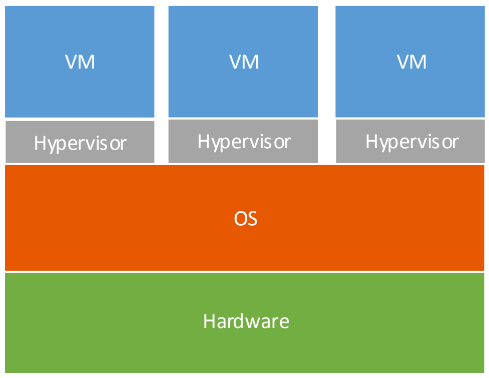


Figure 2.2: Second type of Hypervisor where each of them acts as applications.

Type-1 hypervisors were defined in [17] as bare-metal hypervisors. This type of hypervisor runs directly on the hardware or as a part of a privileged OS. The privileged OS then provide access to hardware resources such as drivers, abstractions, emulation of devices, and administrative tools. The hypervisor adds its own functionality on top of the privileged OS. Hypervisors of this type range from special purpose OS [9] to being implemented as a part of the current running OS of the machine, such as Xen[12] or Hyper-V [11].

Type-2 hypervisors run on top a host OS as a process [18]. A pure type-2 hypervisor will have to emulate all I/O calls and is unable to provide a VM with direct access to hardware. Examples of such hypervisors include Virtualbox [19] and VMware Workstation [20].

Type-1/2 hybrids exist, where some of the I/O calls are not emulated to optimize performance and limit overhead [21].

Running multiple VMs on a single machine is a way to increase the utilization of modern multi-core and multi-threaded machines. During the 2000s we saw the emergence of commodity multi-core systems, and with the many-core systems being developed today [22], is it ever more important to be able to utilize the resources in systems more efficiently.

In cloud environments, such as Amazon AWS[23], Google Cloud[24], and Microsoft Azure[25], customers are typically offered service guarantees by the particular provider. For the provider to satisfy such guarantees, the ability to control how resources are multiplexed is important. Since VMMs control access to physical resources, they are also charged with multiplexing them among hosted VMs according to some predefined policy. Not multiplexing resources according to policy at the VMM level hinders prioritization at other levels of the software stack[26].

2.1.2 Vortex

Fine-grained control over system resources is required from a VMM, and if the VMM is depending on a privileged OS, this requirement carries over to that OS. Control over resource allocation is the goal of the Omni-kernel architecture [13]. Vortex is implementation of the Omni-kernel architecture created at the UiT. Vortex is designed for visibility and opportunity for fine-grained control over resource allocations in a system and can function as a VMM.

The Omni-kernel architecture was made with three design principles in mind:

- Measure all resource consumption
- Identify the unit to be scheduled with the unit of attribution
- Employ fine-grained scheduling

Vortex enables cloud providers to accurately attribute resources consumption to different activities, thus enabling fine-grained billing to be generated for any tenant that share a host[27]. Fine-grained attribution makes Vortex unique: every resource and system device, such as files, disk, processors, memory and I/O controllers, can be controlled by schedulers and have their usage monitored.

Vortex divides the OS kernel into components, all residing in a single address space, that communicate using messages with schedulers interpositioned on communication paths. These schedulers control the order in which messages are processed at components, thereby prioritizing among system activities. The activities might be processes, services, database transactions, VMs or other units of execution.

The use of messages passing and schedulers is the basis of the Omni-kernel and the Vortex implementation of the architecture. Messages are passed between resources in the Omni-kernel, where a resource is any software component that exports an interface for accessing other components, e.g. hardware such as I/O devices, or software components such as the TCP/IP stack or a file system.

Resources uses other resource through sending a *resource request message*. This message specifies arguments and a function to be invoked in the interface specified for that receiving resource. The sender of the messages is not delayed and the message is posted into a *request queue* associated with the target resource. Between two resources there are schedulers that control the order in which messages are processed.

These resources are configured into grids to implement higher-level kernel features and abstractions. Different resources have different roles inside of the grid; some resources are producers and other are consumers. Some resource will have both roles depending on the operation that is being processed. E.g. the network interface card (NIC) will produce message based on what it is being received on the network port, and consume message that should be sent out.

In Vortex, processes run within the confines of a compartment. Compartments are organized hierarchically and define separate namespaces for system resources such as Transmission Control Protocol (TCP) ports etc. Hardware resources are initially assigned to compartments and then subdivided among hosted processes. A process within a compartment can create a new subcompartment and transfer to it fractions of available resources. This gives rise to a hierarchy, where all system resources are assigned to a *root* compartment. Compartments are used to group and separate VMs belonging to different tenants. One tenant can have a compartment as her tenant-specific 'root' compartment and create subcompartments that host different VMs. This allows her to monitor and control resource usage at the level of individual VMs and to aggregate resource usage across compartments for more holistic views.

2.2 CPU debug support

Our mechanism for identifying state changes with specific messages make use of hardware-supported debugging features. Specifically, we make use of the debugging features of Intel-based CPU[28]. Similar features exist in other architectures, for example in the MIPS [29] and SPARC [30] architectures. Support for use of Intel debugging registers was added to Vortex as part of the work presented in this thesis.

The Intel debugging features were introduced in the 80386 as a set of hardware debug registers along with a debug exception. Generally, these registers allow the setup of watch points on specific memory addresses; when a memory access or instruction fetch matches a watched memory location, the processor triggers an exception.

Today, Intels x86 platform have 8 debug registers, Debug Register (DR)0 through DR7. These registers function as interfaces to the following features and functionality:

DR0 through DR3 are available for storing the address of currently watched memory address, either variables or functions. This provides debug registers for watching up to four memory addresses at any given time.

DR4 and DR5 are reserved and any attempt to move values into these register will cause Invalid-opcode exception.

Table 2.1: Bit configuration of register DR6 [28].

Bit	Name	Meaning
32-15	N/A	
15	BT	Task switch
14	BS	Single Step
13	BD	Breakpoint Debug Access detected
12-4	N/A	
3	B3	Breakpoint #3 condition detected
2	B2	Breakpoint #2 condition detected
1	B1	Breakpoint #1 condition detected
0	B0	Breakpoint #0 condition detected

DR6 reports conditions at the time a debug exception was generated. This includes which debug register that caused the exception to be created, as shown in table 2.1. Bit 15 details if the exception was triggered from a task switch. Bit 14 details if the exception was triggered by the **single-step** execution mode. Bit 13 indicates if the next instruction is one that will access any of the debug

registers (DR0 through DR7). Bit 3-0 details which one of the breakpoint that was the cause of this debug exception.

Table 2.2: Bit configuration of register DR7 [28].

Bit	Name	Meaning
31-30	LEN3	Length of DR3
29-28	R/W3	Read/write DR3
27-26	LEN2	Length of DR2
25-24	R/W2	Read/write DR2
23-22	LEN1	Length of DR1
21-20	R/W1	Read/write DR1
19-18	LEN0	Length of DR0
17-16	R/W0	Read/write DR0
15-14		Not used
13	GD	General Detect enabled
12-10		Not used
9	GE	Global exact breakpoint enable
8	LE	Local exact breakpoint enable
7	G3	Global breakpoint enable
6	L3	Local Breakpoint enable
5	G2	Global breakpoint enable
4	L2	Local Breakpoint enable
3	G1	Global breakpoint enable
2	L1	Local Breakpoint enable
1	G0	Global breakpoint enable
0	L0	Local Breakpoint enable

DR7 specifies the forms of access that will generate an exception and the data size that each memory address covers. This register hold most of the configuration that is used for controlling the behavior of the debug registers.

Table 2.2 describes the bits of DR7. Bits **31-30**, **27-26**, **23-22**, and **19-18** detail the size of the memory location that the specified debug register points to. These fields are interpreted as following:

1. Value: 00 - 1-byte (Also used for triggering on instructions)
2. Value: 01 - 2-byte
3. Value: 10 - 8-byte
4. Value: 11 - 4-byte

Bits **29-28**, **25-24**, **21-20**, and **17-16** detail the breakpoint condition that are needed for triggering the specified breakpoint. These fields are interpreted as following:

1. Value: 00 - Break on instruction execution only.
2. Value: 01 - Break on data-write
3. Value: 10 - Break on I/O reads or writes
4. Value: 11 - Break on data reads or writes, but not instructions fetches.

Bit **13** enables the debug-register protection. This will cause a debug exception be a created if any MOV instruction will access a debug register. Bit **9-8** is no longer supported in modern CPUs, and should be set to the value $0x1$ due to backwards compatibility. Bits **7, 5, 3, 1** enables the breakpoint condition for a specified debug register. This is a global flag and is **not** cleared by the CPU meaning that the breakpoint can be triggered by other tasks in the system Bits **6, 4, 2, 0** enables the breakpoint condition for a specified debug register. This is a local flag and is cleared by the CPU. This avoids unwanted breakpoints in other tasks.

It is possible to set DR registers to point at either variables or functions. But for variables, the debug exception is generated after the memory has been accessed. For functions, the exception is generated before access.

2.2.1 Enabling and disabling debug registers

Enabling and disabling the debug registers are done through writing the correct values to the correct register.

There some steps that has to be done for the debug register to be enabled. First, the Debugging Extensions (DE) flag in Control Register (CR)4 must have been enabled earlier. The DE flag is bit 3 of the CR4 register. The next actions are:

1. Moving the address for the debug condition into a debug register.
2. Enabling that debug register and debug condition in DR7.
3. Registering an exception handler for the interrupt vector 1.

The processes of disabling the debug registers involve:

1. Disabling the Debug register in DR7.

2. Removing exception handler.

2.3 Programming model

Today, developers have a choice between two different programming models when creating parallel systems: threaded and event-based programming. These two programming models enable developers to execute parallel systems and applications that utilizes the many-core systems that are popular today. However, the properties that they provide is very different, and designing a solution for one model might make it very challenging to use the other model. This means that choosing the correct model is important and can impact the design and performance of the program to a large degree [31, 32, 33].

2.3.1 Threaded Programming

Threads is a commonly used method to create parallel programs. Threads allow the developer to split his program into parts that are being executed in parallel, and then pass any needed information between the different threads that make up the program, as shown in figure 2.3. This method requires the developer to identify the parts of the program that would benefit from being executed in parallel. The parts that can or needs to be running in a separate thread is split from the rest of the function into a function of their own and a thread started with that function as its main function. This allows the program to stay responsive even when starting large and long running tasks.

Creation of a thread is an operation that typically has been expensive and taken time, resulting in that the code-parts that should be threaded has to be bigger than in e.g. event based programs [6]. Work in this area has created thread packages that are lightweight and scales to 100,000 threads [4]. However, being able to quickly and efficiently create many threads is important for those times when there are many small tasks that should be executed, e.g. in web servers as shown in [7]. But threads do not scale well when having more CPU-intensive threads than the number of cores in the system [32].

Input/Output (I/O) operations are very often the reason for creating multi-threaded programs, to achieve the ability to continue processing whilst waiting for I/O operations to finish. Whilst the I/O operation often does not take up very much CPU, they can take up time and resources in other devices such as Graphical Processing Unit (GPU), network, and disk. Being able to

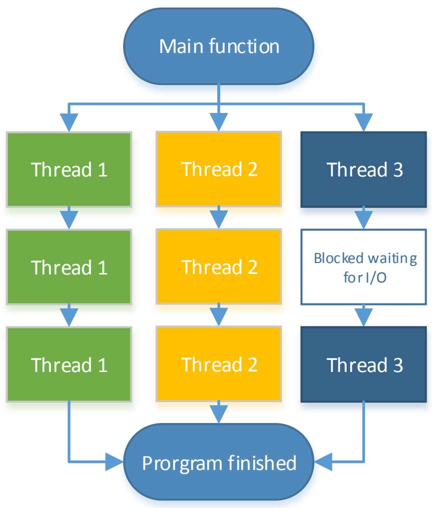


Figure 2.3: The threaded programming model.

efficiently distribute resources among all threads in system is often a challenge, as each program does not have overview of other programs threads and their operations [6].

Threads also has challenges in creating safe programs where all data access to shared data is handled correctly [6], where scheduling is done as efficiently as needed [32], and being able to debug race conditions and corner cases is difficult [34].

Thread-based programming requires the developer to identify parts that can be split into another thread to achieve parallelism as can be seen in figure 2.3. Each thread will execute I/O requests and block while waiting for results. While one thread is blocked, another can be executed and this way one achieves concurrency or parallelism [31].

The drawback of threaded programming is the need to identify shared memory and protect those parts with locks. This process can be complex and improper handling of shared memory can create data races and dead-locks. For example, there are many situations where dead-locks can occur in the Linux kernel [35].

Another challenge with threaded programming is choosing the correct number of threads. As shown in [32], threads executed in parallel will affect each other and running too many threads can degrade performance. The challenge is that when the program is running alone, there will be enough resources for it to be running with a high number of threads, as was often the case before when parallel programming was used for High performance computing (HPC). However, modern parallel programming is on multi-core systems which is being shared with many other applications and a high number of other threads running in parallel. This limits the number of resource available for each application.

2.3.2 Event-based programming

Event-based programming is the alternative to thread based programming and is another way to achieve parallel execution. The model is based around asynchronous execution of small jobs, where the issuer typically is informed of job completion through a call-back or by waiting for a completion-code associated with a token/future. This allows a program to continue executing whilst waiting for operations to finish.

One event is a happening of interest [36, 37], e.g. a state change in a component. The component issues a notification that describes the event, these notifications are moved from the producers and to the consumers. The consumers will register for a specific type of notification that they will receive and handle. A event might be a block of code as in Grand Central Dispatch (GCD) [6], input to function or closures as in Vortex [27]. The event are sent to the consumers that will based on the type and content of the event process it and return the produced result.

Supporting asynchronous I/O operations has become more and more popular, as it has been shown to be a good way of increase performance [38, 39]. It has been added to the Linux Kernel [40], and ssynchronous I/O is also the basis of Node-js [41], a event-driven platform built on Chrome's JavaScript (JS) runtime. Common across these is that I/O takes time; a disk is slow and network operations is even slower. Instead of having a full thread blocking for every I/O that the program has to do, can the operations be created using events and asynchronous I/O operations. This will allow the program to continue while waiting for the result. Node JS has taken this as a basis for their design [42] and each Node application is only a single process, never executing its core logic in parallel. Most of the operations is implemented as events leaving very little logic in the main loop.

As shown in figure 2.4, the main program will run the logic of the program inside e.g. a main loop, creating events based on user input or other events.

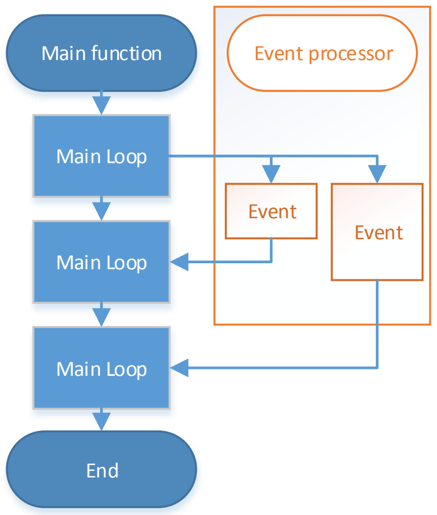


Figure 2.4: Event-based programming. A main loop that contains the logic and event handlers that handle I/O or longer requests.

For any operations that can block or might take time, the main loop spawns an event. This event is passed over to the event processor, that handles the event which might consist in passing it to another resource or directly replying with a result. For many systems, this results in one thread running the program and one or more threads handling all of the events being generated. The developer does not see the threads that handle the events, so for him is there only one thread.

Event-based programs has the benefit of an possible less memory overhead, as the number of threads needed can be lower or better utilized [43]. This also has the benefits of allowing the OS to build the event-based framework directly into the OS as has been done with Vortex [27] and GCD [6]. Having the frameworks as integral parts of the system allows the system to handle the work of balancing the load between the different applications that generates events and the resource that they have been allocated. In thread-based programming is it up the to developer to ensure that his threads does not consume more resource than needed, or in such a manner that they negatively affect the system. Using event-based frameworks built-into the OS allows the OS to control the resource usage of application and ensure that starvation does not happen. The framework will implement a set of queues and schedulers running on those queues; every event created is inserted into a queue and based on the

the event, the sender and the receiver is the event scheduled. Different queues might e.g. be used for different levels of priority or different types of events. This queue and scheduler design allows the framework to effectively control, optimize and attribute resources that the events needs on a system level.

There have been many different frameworks and packages create to enable and help developers implementing event-based programs, and handle those events in optimized way [44, 6, 43, 3]. One of these is the GCD [6] implemented in OS X 10.6 Snow Leopard. GCD is a framework for event-based programming built in to the OS X OS, enabling developers to create blocks of code that will be executed as an event. As a part of GCD they built-in queues for different priority levels, which allows multiple programs to be running at once with all of them producing events and have them execute as fast as the hardware is able to do. This allows developers to create as many events as they need without having to think about how many threads they can create and how many system resources that are available, as this is handled by the framework which can optimize the amount of events running in parallel based on the number of applications and available system resources. This also allows the system to prioritize some events above other based on the wanted behavior. These frameworks hides the challenges that are present with the event-based programming model such as *obfuscated control flow* and *stack ripping*.

One of the challenges in creating event-based system is the *obfuscation of control flow* [7]. In these systems, one method *calls* a method in another resource/module by sending a message or event and expect an answer using the same event method. This requires the developer to keep track of where each of the call/return methods are and which resource they represent when these can be in very different parts of the code-base. In addition to these call/return methods, the developer has to save and restore the state in the process called *stack ripping*, as he has to *rip* out the needed state for his event. The complexity and challenges of *stack ripping* is discussed in more detail in Chapter 2.3.2.

The obfuscation of control flow is showing when looking at the processing order of events. Messages being sent from resource to resource and interleaved with messages from the other resources in the system can make it hard to see exactly why one state in a resource was achieved instead of the expected one. The processing order of messages in one resource will be affected by how many other resources is using it. This means that bugs can be very complex and intermittent, increases the debugging challenge and the amount of developing work to solve bugs. Bugs might require a given sequence of events or states to happen and thus be complex and intermittent, this can make them very complex and complicated to debug.

Designing event-based systems might be hard as they often rely on a flat structure [36, 37]. This means that as these systems grow, it becomes harder and harder for the developer to keep track of the code base and design. The less overview that he has of the system, the easier it is for a change to result in faults.

There has been done work to unify the two different models and creating hybrid solutions that combines the best from both models [45]. These types of systems allows the developer to use the threaded and event based model in the same program.

Stack ripping

One of the most prominent challenges when creating event-based systems is *stack ripping* [5]. Adya et al discussed the concept and introduced two different styles of stack management: automatic and manual stack management. They claim that the style of task management (cooperative or preemptive) is orthogonal to its style of stack management. *Stack ripping* is done in event based programming where parts of the program are *ripped* out and saved. The event handlers have to *rip* out the needed state for the programs, so that once the event trigger the program is able to continue the computations.

The process of *ripping* is needed to save the state for the currently running program, as the sender often continues to execute once the event is sent, and this might change or remove state that is needed.

Consider requests received by a web server. One request might come in from a web browser for a Hyper Text Markup Language (HTML) page. This request will result in multiple events being created from a high-level view. First, one event to process the incoming packet, then reading the block with the web page, and an event for sending packet back to the client. Each of these events might create multiple events of their own to be able to produce the wanted result. Reading the disk block might be one event for initiating the disk I/O, and one event for reading and for returning the block to the web server. The imagined web-server can whilst waiting for the events to return continue to operate and handle more incoming requests.

In [5] they argue that this approach breaks up the control flow between many different events, possible different languages. The result of this is that the scoping features of the programming language is disregarded. This increases the complexity that process of stack handling, requires more manual stack management, and it can become harder to deal with as the complexity of the application or system increases.

Adya et. al. introduced the concept of *stack management* and its two different types: *manual* and *automatic*. *Automatic* stack management is the type that is being used in multi-threaded programming where the developer does not have to think about the stack or saving state between two threads. This is done automatically for him. *Manual* stack management is the type of stack management that message based systems provides. The developer has the responsibility to store enough state for the message to be able to execute once processed. The result of this is that more responsibility is placed on the developer and provides new avenues for bugs.

Preemptive task management is often for HPC, so that task can be interleaved or overlap on the multi-core processors that are used in these systems. This allows the programs to utilize the resources available. *Serial* task management is each task is run to completion before starting a new one. The benefit of *serial* is that it does not have any shared state conflicts, as those that might arise in *preemptive*. This because only one task being executed at once.

Choosing the correct type of task management is important as it will affect the performance of the systems. For systems with multiple CPUs or with some slow tasks and many small quick tasks, the usage of *serial* can limit the performance that one would get out of the system or increase delays.

A third type of task management is *cooperative*. Whilst being similar to *serial* where each task run is executed alone, it has an advantage over *preemptive* where tasks are able to yield on specified points during its execution. One example of a point to yield is whilst waiting for I/O. This means that there are only specified set of points where state has to be saved, so that it can be resumed once the task resumes. *Cooperative* with its yielding is more complicated than *serial* due to the need to save state. It gets more complicated if the state of the task is dependent on global state, as the global state might have changed once the task resumes. This is the same problems that *preemptive* suffers from.

However, the type of stack management that is used in the system influences the type of task management. According to [5], the combination of these types of stack management and task management can be seen as the basis of the two types of programming models that we currently have. *Event driven* and *multi-threaded*. *Event driven programming* is the combination of *manual stack management* and *cooperative task management* whereas multi-threaded is the combination of *automatick stack management* and *preemptive task management*.

The processes of *stack ripping* is a challenge in event based systems, such as Vortex, as it hides the control flow in an application or system [4]. This results in systems where it is hard to understand the cause and effect of state in various resources, to see why one resource was in the state that it was. *Stack ripping* and the process of storing and restoring state can be troublesome to debug, and creates new often intricate avenues that intermittent bugs might be created from. The developer has to match the *call* and *return* methods in his head, whilst ensuring that he saves and restore the state correctly.

3 Design and implementation

The presence of stack ripping and obfuscated control flow poses challenges for Omni-kernel developers. This chapter describes the design and implementation of tools to help Omni-kernel developers handle and overcome these problems.

3.1 Design goals

In a message-based system, it can be hard to determine the originating activity of a message. The Omni-kernel design goal of being able to attribute all message processing to an activity alleviates this problem somewhat; all messages carry an identifier that ties the message to a particular activity. Still, knowing the originating activity of a message is often not sufficient when debugging complicated bugs that might require replicating or knowing the history of messages leading up to the bug triggered when a particular message is processed.

The overall design goal for this thesis is to create a system that can monitor message processing at a particular resource in the Vortex Omni-kernel implementation, and create message processing traces sufficiently detailed to reproduce the history of messages leading up to a particular state change in a resource.

We approach this goal by adopting the general design principles of attempting creating a *simple* and *lightweight* system. The system should filter message sent to and from the watched a resource and store just enough information about each message so that it can be identified and connected to the sending process.

We also desire for the system to be *flexible* and *easy-to-use*. The design should allow developers to expand on the information captured from messages, and have accessible ways to view message histories.

We have tried to meet these design goals by creating a set of tools. The first tool enables a developer to track messages processed by an Omni-kernel resource. The tool can e.g. be used to reconstruct the flow of messages originating from a particular system activity. The second tool aids the developer in identifying a message with a particular state change. By activating CPU debugging support upon message dispatch, access to a variable or invocation of a function can be intercepted and tied to a particular message. An common infrastructure handles the information produced by these two tools. Last, an external gui-tool can be used to retrieve and visualize message flow to a resource.

The design principle of creating a *simple* and *lightweight* system is shown throughout the system, where each part has been designed to handle a high amount of messages during the task processing path without affecting Vortex negatively. The tools are small and with an implementation that limits the amount of computations done for each message.

Designing debugging tools requires them to be *flexible* and *easy-to-use*, as they will be used by other developers, in scenarios that might differ greatly from ours. They might be required to work under different circumstances and hardware combinations that we use today with a greatly increased number of cores and resources. Creating *flexible* and *easy-to-use* tools are achieved by allowing developers to change different parts of the tools easily and making tools that can be used on different levels based on need.

The tools designed for this thesis fits with the demands set and provides the ability to reconstruct the control flow for an kernel-resource.

3.2 Architectural overview

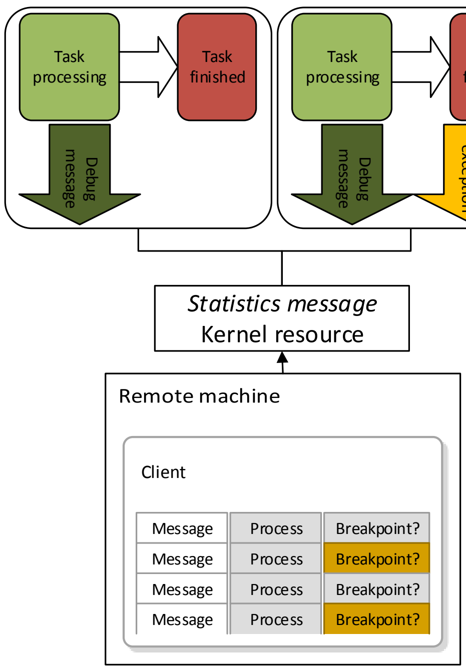


Figure 3.1: Architectural overview of the message gathering tool.

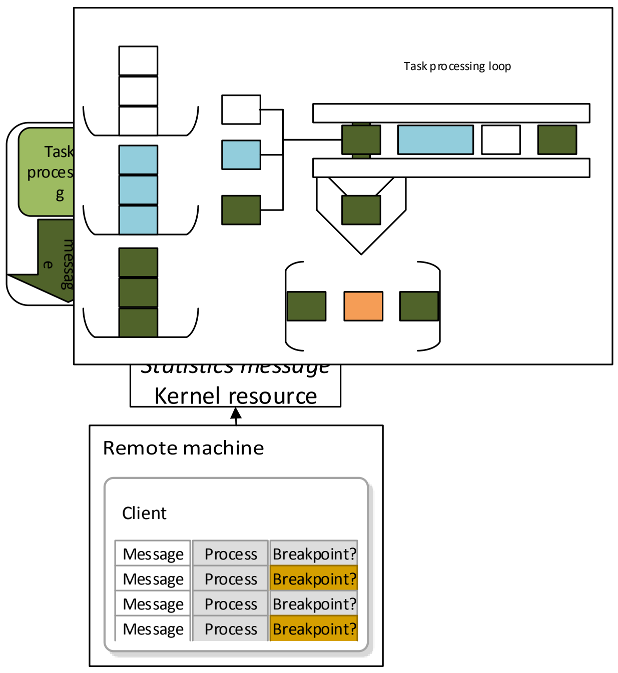
Figure 3.1 depicts the overall architecture of our system. The tools for tracking message flow and identifying messages with state changes both produce *debug messages* in their operation. These messages describe e.g. pertinent details of a message processed by a resource or that a particular message has caused access to a specific variable. Each message contains enough information to discover the intent and ownership of the original event.

Debug messages are in turn handled by a common infrastructure that consists of both kernel- and user-side components. The kernel-side components handle message queueing and provides access to queued messages through the general Vortex resource interface.

The user-side components interface with the kernel to retrieve debugging messages, to expose them externally through a network-based interface.

Last, the infrastructure provides a client-side tool for visualizing and creating views on a set of retrieved debug messages. The view will recreate the control flow for the given kernel resource, showing every message sent to and from that resource.

3.3 Filtering messages



Message	Process	Breakpoint?
Message	Process	Breakpoint?
Message	Process	Breakpoint?
Message	Process	Breakpoint?

Figure 3.2: Filtering of events and creation of debug messages

All work in the Omni-kernel is represented as a message. Thus any state change in the kernel occurs as a result of processing a message. Every message targets an Omni-kernel resource and our first tool allows for the capture and profiling of messages sent to specific resources.

This results in a very large number of messages being sent and processed from every type of resource and program in the system. When active, the message filtering tool inspects, at the time of dispatch, all incoming messages for a resource and produces a corresponding *debug message* as shown in figure 3.2. Creation of debug messages thus occur in the critical path of Omni-kernel message processing. This means that to reduce overhead, debug message creation should be very lightweight and the debug message should only include pertinent details. Trying to gather too much information would negatively affect the running system.

Listing 3.1: Contents of a debug message.

```
// Gathered from resource and request
to:%s,            // Receiving Resource
from:%s,          // Sending resource
timestamp:%D,     // Timestamp of message
function:%s,      // Function to be called
functionType:%d,// What type of request
cl_argc:%d,       // Number of arguments
cl_fmt:%s,        // Type of argument, reference
    counted object or not
cpu_id:%d,        // CPUID
msg_id:%d,        // ID of message
// Gathered from compartment and process struct
cmp_name:%s,      // Name of compartment
pid:%d            // Sending process ID
// Added if message caused by breakpoint
dr:%d,            // Was this message sent
    because of debug exception
dr_reg:%d,        // Which debug register caused
    the exception
```

Figure 3.1 shows the contents of a debug message. The information captures salient aspects of a message, such as which function the message targets and what process caused the creation of the message. This information is recorded both from the corresponding message data structure itself as well as information drawn from other sources.

To capture information not part of the message data structure, our code traverses many Vortex kernel data structures. For example, a message does not include a process or compartment identifier, but rather an *IOshare* identifier. The IOShare identifier refers to an instance of the IOshare abstraction. Internally in the Vortex kernel, IOshare instances are the clients, or activities, among which I/O resources are multiplexed and shared. An IOshare instance is associated with each instance of the *IOAggregate* abstraction. The IOAggregate abstraction is exposed in the Vortex system call interface, and from the perspective of a process, instances of this abstraction are the entities among which I/O resources are multiplexed and shared. Code for tying an IOShare identifier to IOAggregate to process was not readily available, and something we added to the Vortex kernel.

Choosing which information to save can greatly affect the tools ability to scale with load. Gathering too much information or information that takes time will affect the overhead of the tools. The information we gather is enough to gain the insights that was needed, and for the tools to help developers. One have to be careful when increasing the amount of information for each debug message. As the amount of messages being processed is very high, we have seen 43000 messages every second being sent to and from one resource, can one extra entry cause the overhead to increase so much that it negatively affect the system.

3.4 Monitoring state access and change

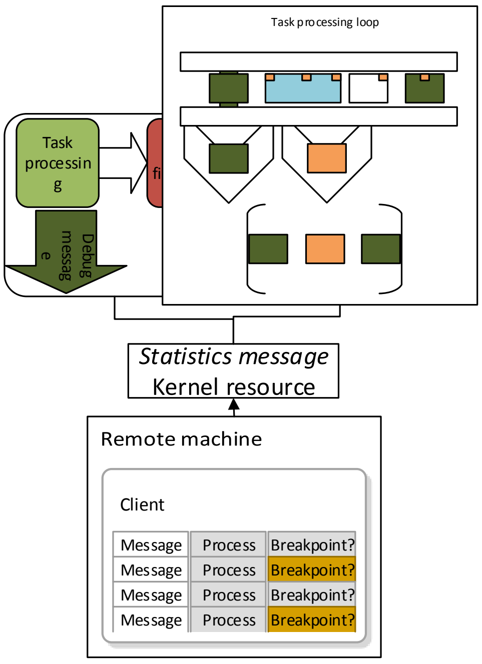


Figure 3.3: Monitoring access to state and creating debug messages.

Our second tool can be used to track accesses and changes to the state of a Vortex kernel resource. Specifically, this monitoring tool makes it possible to identify a particular message with some state access or update. This is accomplished by allowing the developer to set breakpoints on specific memory addresses and have a debug exception be generated when that memory address is being used as shown in figure 3.3. This enables a developer to ascertain when a variable is being accessed or when a function-call happens.

To perform its task, the tool makes use of the built-in debugging support of Intel

CPUS. Functionality for use of these CPU debugging facilities was added to the Vortex kernel as part of our work. Since we expected that the functionality could be of interest beyond its use in our tool, the functionality is designed and implemented separately from our tool.

Listing 3.2: Low level APIs for enabling and disabling an individual debug register.

```
vxerr_t dr_add_debug_registers(vaddr_t address,
    variable_size_t size,
debug_register_flags_t flag, debug_registers_t dr)
vxerr_t dr_remove_debug_registers(debug_registers_t dr)
vxerr_t dr_set_exception_handler(vx_vaddr_t function)
vxerr_t dr_remove_exception_hander()
```

Listing 3.3: Default exception handler.

```
vxerr_t predefined_exp_handler(excdetail_t *exc)
```

The interface for controlling the CPU debugging registers is shown in Listing 3.2. It consists of four functions:

1. *dr_add_debug_registers* enables the specific debug register identified by the dr argument. The address and size arguments identifies a memory address range. The flag argument specifies what to monitor:

   - INSTRUCTION_EXECUTION
     Triggers only on instruction fetched from the address. This will trigger before the instruction is executed

   - DATA_WRITE
     Triggers on writing to the address. This will trigger after the value has been updated.

   - IO_READ_WRITE
     Triggers on IO reads and writes

   - DATA_READ_OR_WRITE
     Triggers on reads and writes to the address. This will trigger after the values has been updated.

   For example, enabling the debug register to trigger on function call should a size of one byte and the flag *INSTRUCTION_EXECUTION* should be used.

2. *dr_remove_debug_registers* disables a specific debug register.

3. *dr_set_exception_handler* is called to specify a handler that will be invoked when a debugging exception is generated. The function has to have the signature as detailed in Listing 3.3, with the return value of *VXERR_OK* as this function is called from the general exception handler in Vortex.

4. *dr_remove_exception_hander* removes the previously set exception handler for the debug exception.

Listing 3.4: breakpoint data structure

```
struct breakpoint_t{
        vaddr_t                 address;
        variable_size_t         size;
        debug_register_flags_t  flag;
        debug_registers_t       dr;
        bool_t                  set;
};
```

Listing 3.5: API for controlling all of the debug registers.

```
vxerr_t dr_set_breakpoint_regs(rbs_t* rbs)

vxerr_t dr_reset_breakpoint_regs(rbs_t* rbs)

vxerr_t dr_remove_breakpoint_regs(rbs_t* rbs)

rbs_t *create_breakpoint_structure()
```

The monitoring tool is built on top of the debugging interface and provides an interface centered around a breakpoint abstraction. The interface is shown in Listing 3.5.

Each breakpoint is described by the data structure shown in Listing 3.4. The breakpoint describes an address to monitor and how it should be monitored. When a user sets the breakpoints by invoking *dr_set_breakpoint_regs*, are all enabled breakpoints set.

Listing 3.6: rbs data structure

```
struct rbs_t{
        breakpoint_t    dr[DR_MAX];     // One slot per
            debug register
        bool_t          DR_set;         // Is debug
           register enabled
        object_t        obj;            // Object for
           mutex-locking
        bool_t          local;          //Is this set
            to look at a stack address?
};
```

Information about current debug register allocations is described by the data structure shown in Listing 3.6. Much of this information corresponds to what is needed when the debug interface functions are invoked. Some additional information is stored, however. A boolean value identifies if the register should be enabled. Thus, a debug register can be allocated but not active. The data structure also contains a bool to tell if this address is a local or global variable. Local stack variables are unique for each function due to the scoping of the C language, however, the stack memory location is reused because of unwinding and subsequent function calls. This can cause problems when a stack variable is monitored (see Section 3.4.1).

In short, the functions in the interface shown in Listing 3.5 can be invoked to:

1. *dr_add_debug_registers* is invoked to enable a debug register.
2. *dr_set_breakpoint_regs* enables the debug registers according to the parameters described by the rbs pointer argument.
3. *dr_reset_breakpoint_regs* disables debug registers according to the rbs argument descriptions. The allocation of registers is kept, however.
4. *dr_remove_breakpoint_regs* disables and removes all debug register allocations and configurations.
5. *create_breakpoint_structure* allocates a new rbs data structure.

The tool code will set up its own debug exception handler by invoking the *dr_set_exception_handler* debug interface function. When an exception invokes the handler, will it (1) determines what debug register caused the exception, and (2) generates a debug message that describes the exception. The debug message is shown in 3.1. Only the fields *dr*, *msg_id* and *dr_reg* will be populated. Sending the whole debug message was chosen to simplify the receiving in clients, as every message is of the same size and data structure.

3.4.1 Breakpoints on variables local to a function

C compilers typically produce code that places variables local to a function on the stack. As the stack contracts and expands due to function calls and returns, particular memory locations on the stack are potentially reused multiple times to hold different types of state. This causes complications when a developer wishes to use our debugging functionality to monitor access to a specific local variable in a function.

If a local variable in a function is monitored, the number of debug exceptions might be higher than expected due to stack memory reuse. For example, we observed that as many as six to eight debug exceptions would be generated when our code seemingly only contained one reference to a monitored variable.

To address this problem we included an extra boolean in the rbs data structure (see Listing 3.6). If this variable is set to *TRUE*, the developer informs that a breakpoint is set up for a local variable and that he is aware of the challenges.

If the variable is set to *FALSE* and the developer tries to set up debug registers on local variables, will an critical syslog message be generated. This message will halt the Vortex system, halting was chosen as enabling debugging on local variables can have very negatively effect on a Vortex system if not done correctly, and might change system behaviour.

3.5 Storing debug messages

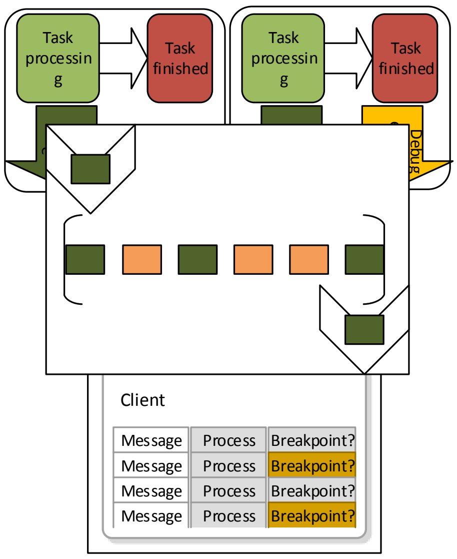


Figure 3.4: Storing of debug messages in rounded buffer.

To store debug messages, we associate a circular buffer with every Vortex kernel resource. Debug messages are deposited in the appropriate buffer, and the buffer is emptied upon user-level software requesting profiling data via the Vortex statistics message interface (see Section 3.6). This is shown in figure 3.4.

A debug message will be dropped if the circular buffer is not emptied at a rate at least matching that of message insertion. Another design option would be to start overwriting old debug messages if no room to store them. We decided on the policy of dropping messages because both of the options would result missing information. By dropping new messages are the old ones kept so that the history up to where any messages was dropped is complete.

The circular buffer is represented using a fixed size array. The size of the array may be changed at compile-time, but remains fixed in a running system. Using fixed size arrays decreases complexity and ensures that the overhead of gathering messages is kept low: a circular buffer allows both inserting and removing from the buffer to happen at constant time with a complexity of $O(1)$ as there is no need to resize, lookup or traverse the data structure.

Another reason for a fixed size array is the need for performance; store of a debug message is on the critical path in the Vortex kernel. For example, resizing the array during message processing would have added overhead, which might have affected system performance substantially. This design lowers the complexity and overhead of the buffer operations whilst ensuring that buffer is of sufficient size at the drawback of requiring developer interaction.

3.6 A programmatic user-level interface to our tools

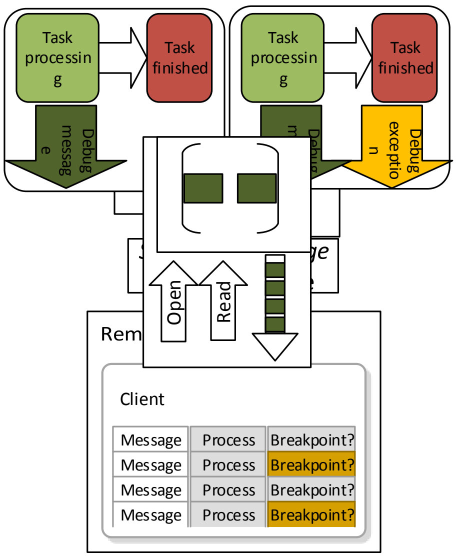


Figure 3.5: Opening the kernel side resource and reading any buffered messages.

To fulfill our design goals of creating a flexible and easy-to-use system, we provide user-level processes with a programmatic interface to our tools. A goal with this interface is to allow run-time configuration of the tools. This promotes flexibility and makes use of the tools more convenient, as opposed to static kernel-side options and configuration that would have required re-compilations of the kernel and reboots for activation.

Vortex exposes most abstractions through a namespace akin to the conventional hierarchical file system namespace. For example, a process can create a new client TCP connection by performing a *vx_aopen* system call with the path "/network/tcp/client" as an argument.

Our first step in exposing our tools to user-level processes was to add a new resource to the Vortex kernel, the *statistic message* resource. Creation of a kernel resource in turn allows the resource to be registered as accessible through the Vortex namespace. This enables a process to perform *vx_aopen* system calls with the path "/statistics/message" as argument. The *vx_aopen* calls end up in an *open* handler in our profile resource, where arguments to the call are extracted [1]. This process is shown in figure 3.5.

The open call arguments identify what Vortex kernel resource to monitor and how, causing creation of the circular buffer described in Section 3.5. Breakpoints and debug register configurations has to initiated manually by the programmer. Note that multiple monitoring sessions can be active concurrently.

Beyond exposing itself in the Vortex namespace, a kernel level resource can register itself as capable of being the source and/or target of asynchronous I/O. This means that the resource identifier, or RID, returned from the *vx_aopen* call can be used as an argument when creating a Vortex flow. A Vortex flow essentially is an asynchronous write operation, with a source providing data and a sink consuming data.

Our profile resource registers itself as capable of being an asynchronous I/O source. This allows a user-level process to issue *vx_aopen* call to activate monitoring for some resource. Then, the process can use the returned RID to create a flow that reads from the profile resource. These reads provide a mechanism and interface for the process to retrieve debugging messages—when the profile resource receives a AIO_READ message [2], it can respond with data from the circular buffer containing debugging messages.

1. Vortex allows the namespace path string to also encode arguments; these can be utf_8 encoded strings and integers.
2. Kernel resources involved in I/O receive different types of read and write messages that they have to respond to.

3.7 User-level profiling service

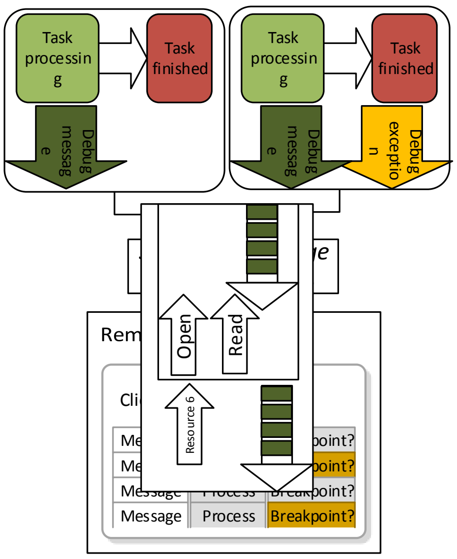


Figure 3.6: User-level service allowing for external retrieval of debug messages.

We have designed and implemented a user-level service that allows profiling to be configured and profile data to be retrieved externally from the machine running Vortex.

The profiling service runs as a user-level process on Vortex. Its services are exposed through TCP, allowing external software to interface with the service over a conventional network connection.

For external software, setting up a profiling session for a specific Vortex kernel resource involves connecting to the profiling service and communicating the identifier of the kernel resource that should be monitored.

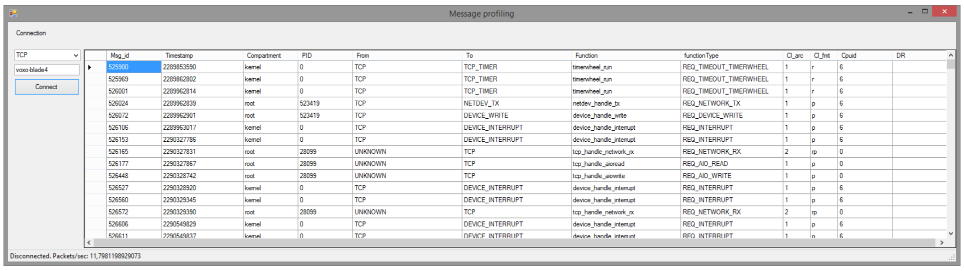
Having received the appropriate parameters, the service uses the system call interface described in Section sec:prginterface to activate the monitoring. Then, the service sets up a long running asynchronous read operation to retrieve debug messages. Retrieved debug messages are translated into JavaScript Object Notation (JSON) objects, shown in 3.1, and written to the TCP connection to the external software.

Use of JSON to represent debug messages is convenient because of wide platform and language support, simplifying the creation of applications or tools that process profiling messages. It also has the benefit of being easily extensible, allow for easily adding new items to a debug message.

For our tools have we opted to choose *scalability* and *simplicity* whenever design choices came up. One of those choices is security. The user-level service does not require the clients to authenticate and does not encrypt the stream of messages. Both of these can result in unwanted clients connecting or gaining access to the debug messages. However, these tools as a part of the development for Vortex and should not be running while the system is in production. They are designed to be used for specific situations and not continuously. To keep the overhead and resource usage of the tools down, opted we to not have any security implemented into the tools.

3.8 Visualizing monitoring data

To demonstrate the applicability of our tools, we have created an application that visualizes profiling data. This application is implemented in C# on .Net platform and uses Windows Forms for its graphical elements. The application allows developers to trace message sent to a resource, thereby seeing what process instigated the message, what other resource sent the message, and so on.



Msg_id	Timestamp	Compartment	PID	From	To	Function	functionType	CI_arc	CI_fmt	Cpuid	DR
525900	2289853590	kernel	0	TCP	TCP_TIMER	timerwheel_run	REQ_TIMEOUT_TIMERWHEEL	1	r	6	
525969	2289862802	kernel	0	TCP	TCP_TIMER	timerwheel_run	REQ_TIMEOUT_TIMERWHEEL	1	r	6	
526001	2289962814	kernel	0	TCP	TCP_TIMER	timerwheel_run	REQ_TIMEOUT_TIMERWHEEL	1	r	6	
526024	2289962839	root	523419	TCP	NETDEV_TX	netdev_handle_tx	REQ_NETWORK_TX	1	p	6	
526072	2289962901	root	523419	TCP	DEVICE_WRITE	device_handle_write	REQ_DEVICE_WRITE	1	p	6	
526106	2289963017	kernel	0	TCP	DEVICE_INTERRUPT	device_handle_interrupt	REQ_INTERRUPT	1	p	6	
526153	2290327786	kernel	0	TCP	DEVICE_INTERRUPT	device_handle_interrupt	REQ_INTERRUPT	1	p	6	
526165	2290327831	root	28099	UNKNOWN	TCP	tcp_handle_network_rx	REQ_NETWORK_RX	2	rp	0	
526177	2290327867	root	28099	UNKNOWN	TCP	tcp_handle_aioread	REQ_AIO_READ	1	p	0	
526448	2290328742	root	28099	UNKNOWN	TCP	tcp_handle_aiowrite	REQ_AIO_WRITE	1	p	0	
526527	2290328920	kernel	0	TCP	DEVICE_INTERRUPT	device_handle_interrupt	REQ_INTERRUPT	1	p	6	
526560	2290329345	kernel	0	TCP	DEVICE_INTERRUPT	device_handle_interrupt	REQ_INTERRUPT	1	p	6	
526572	2290329390	root	28099	UNKNOWN	TCP	tcp_handle_network_rx	REQ_NETWORK_RX	2	rp	0	
526606	2290549829	kernel	0	TCP	DEVICE_INTERRUPT	device_handle_interrupt	REQ_INTERRUPT	1	p	6	
526611	2290549837	kernel	0	TCP	DEVICE_INTERRUPT	device_handle_interrupt	REQ_INTERRUPT	1	p	6	

Disconnected. Packets/sec: 11,7981198929073

Figure 3.7: The remote client.

After being configured with what Vortex kernel resource to profile, the application connects to the Vortex profiling service over a TCP connection and requests profiling data. Figure 3.7 exemplifies how the application performs visualization when instructed to profile the Vortex kernel TCP resource. Shown in the figure are messages from a live system, detailing which process, compartment, and resource that sent each message.

The application uses a grid view to lists messages. Windows forms and grid view have poor draw performance, which means that drawing the grid is a costly process. For scenarios where Vortex experiences high load, a correspondingly high number of profiling messages will be produced. For example, we have experienced sessions with over 43000 messages per second. Re-drawing the grid for every message proved to scale poorly, resulting in the drawing lagging behind message receipt.

We initially attempted to solve this problem by creating separate grid drawing and message receipt threads in the application. This improved the situation, allowing drawing to scale with message receipt until TCP experienced about 50% of the load needed to saturate a gigabit network interface card. Beyond that point, drawing started to lag behind message receipt. Our final solution to the drawing problem was to batch messages and initiate grid drawing only when a batch was full. For a heavily loaded Vortex system batches had to contain 40 messages for drawing to keep up with message receipt.

We also took steps to limit memory usage, which became a problem when monitoring heavily loaded Vortex systems. This steps was to limit the number of messages that are stored and presented to 4096 messages. Not having this limit would result in the client filling up its memory.

3.9 Summary

In this chapter have we presents the design and implementation of the two tools and their properties. We look at how they are designed and fit into the Omni-kernel. For the message-gathering tool we look at what types of information it gathers and how to gather it during event processing without affecting system performance. For the tool to control the debug registers we look at how to create a *flexible* and *simple* layered API that enables developers to utilize the debug registers in different debugging scenarios.

4 Evaluation

The tools implemented for this thesis are placed in the critical path for the Omni-kernel, and can therefore have a large effect on the performance of the system it is monitoring or debugging. The effect that the tools might have on a Vortex is important to measure, so as to verify that it does not hinder the system or change behavior.

This chapter describes the results of a set of measurements performed on a live Vortex system, with the goal of quantifying the resource use of our monitoring tools.

All experiments were done with Vortex running on Hewlett-Packard (HP) BL460c G1 blade with two Intel Xeon X5355 Quad-Core processors. Each of the cores run at 2.66Ghz and are connected to 16GB of DDR-2 memory running at 667Mhz. Vortex is running with a RAM-based file system.

Two different types of machine load are used during these experiments; *idle* and *busy*. An *idle* machine is a machine that is only running the Vortex performance profiling framework. This framework collects detailed performance data on a running Vortex systems and presents the data in graph form, on a Windows client machine.

A *busy* machine runs the Apache 2 web-server with a client continuously requesting a 4MB large file using Siege 3.0.5, with no delay between requests. We configure our tools to profile the Vortex kernel TCP resource, as this resource is heavily loaded due network traffic. By selecting this resource we will also observe load on an idle machine, because of the performance profiling framework requesting performance data samples.

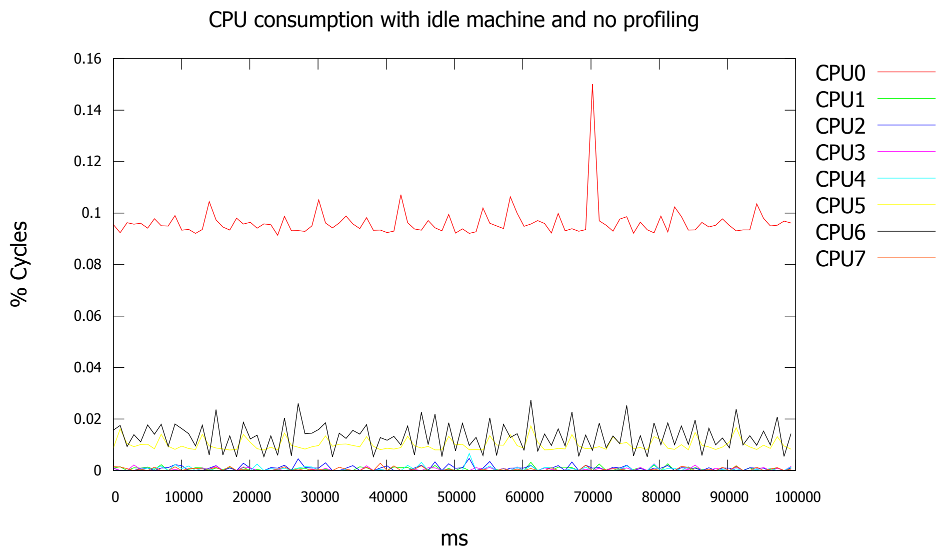


Figure 4.1: Idle machine running only the statistics framework

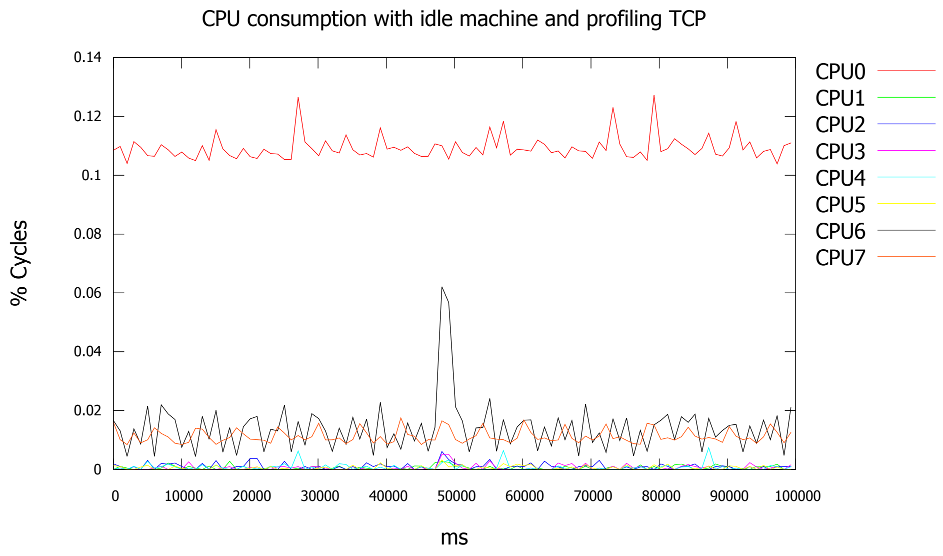


Figure 4.2: Idle machine running only the statistics framework while profiling the TCP kernel resource

Our first experiment quantifies the overhead of our tools in an idle system. This experiment establishes the base resource usage of our tools.

Figure 4.1 and Figure 4.2 show the impact of gathering message of an idle system. The figures show that the impact of doing message profiling in an idle system is very small and that overhead should not impact the performance and behavior of the system.

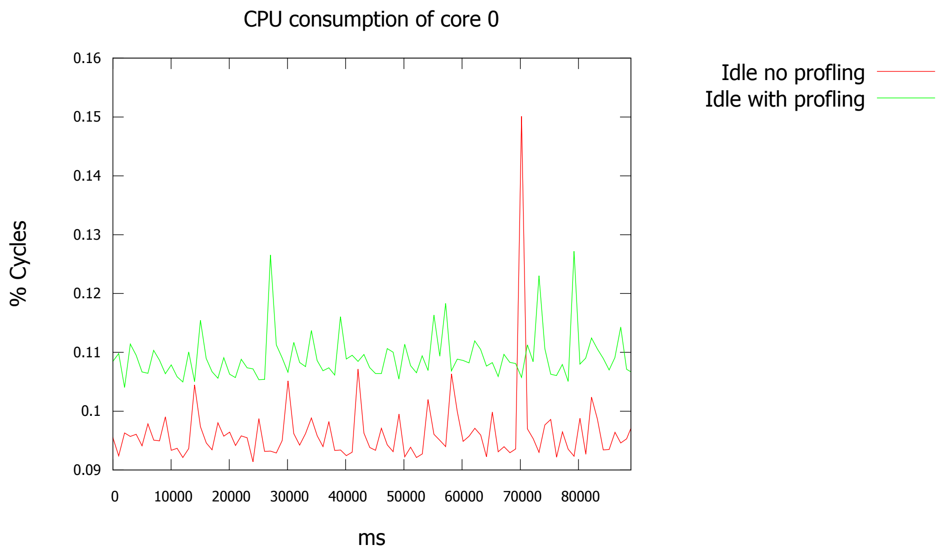


Figure 4.3: Core zero of an idle machine with and without profiling.

Although resource use in an idle system is low, our tools are involved on the critical path in the Vortex kernel scheduling of message process. To determine if our tools significantly impact performance in a busy system, we compare CPU consumption with and without our tools enabled.

Figure 4.3 shows load for core 0 in a busy system, as this it is the most loaded core. Whilst with profiling the core does have a load spike at around 7000 ms, the percentage of CPU consumption is so low that any extra system maintenance being done will show up like spikes. This figure shows that a very limited amount of extra resources is use when our tools are active; the base cost of message profiling is low. The same can be observed on other cores.

Figure 4.4 shows the CPU consumption of our kernel-side profiling resource in an idle system. As described in Section 3.6, this resource is involved in asynchronous read operations, to transfer debug messages to the user-level profiling process. From the figure we can see that CPU consumption is very low in an idle system.

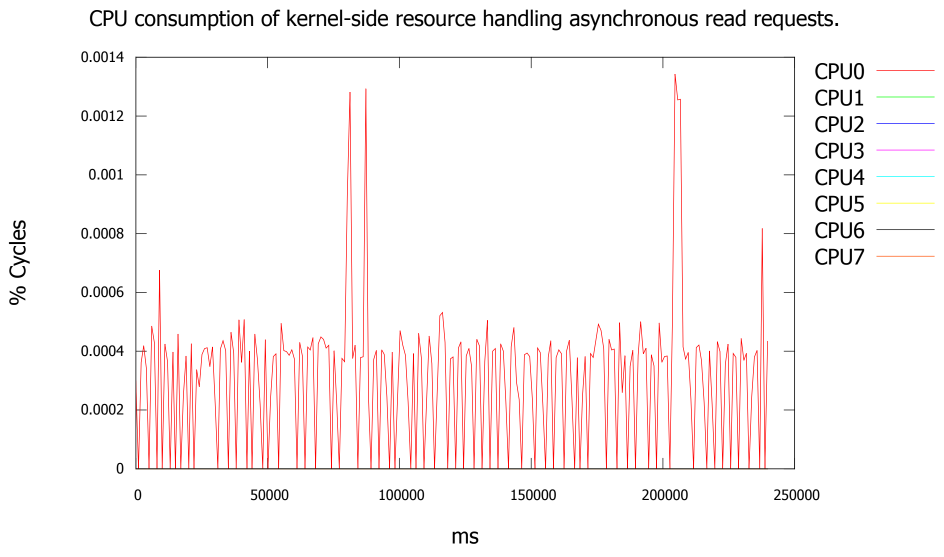


Figure 4.4: Showing the CPU usage of our kernel-side profiling resource on an idle machine running only the statistics framework while profiling the TCP kernel resource

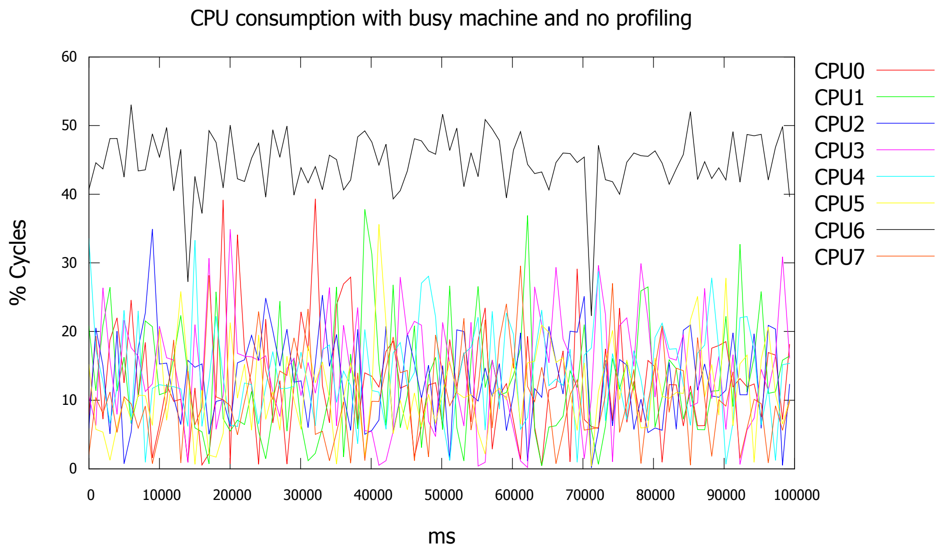


Figure 4.5: Busy machine running the statistics framework and Apache.

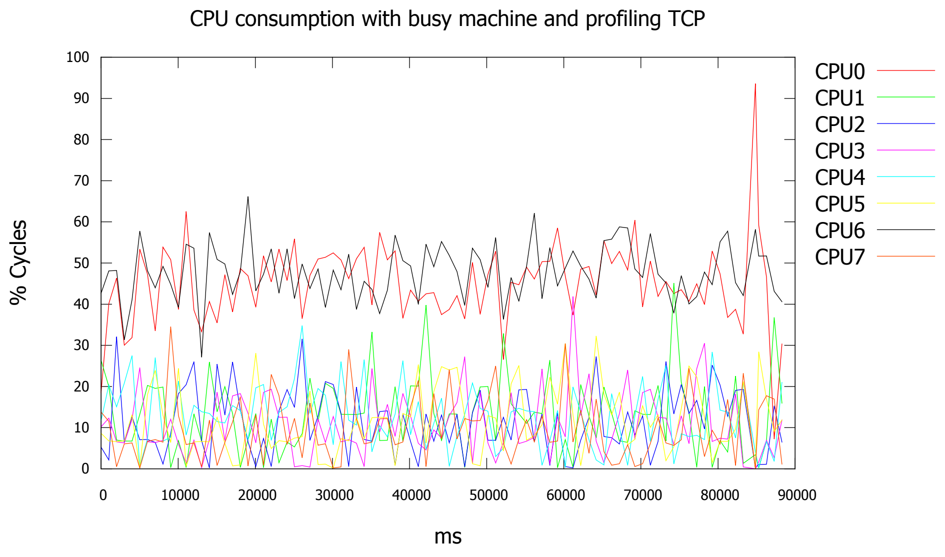


Figure 4.6: Busy machine running the statistics framework and Apache whilst profiling the TCP kernel resource.

Some bugs and errors might require high load for them to trigger, or that a specific system state has to be reached. This means that the message profiling tool has to scale with the load of the system. Using few resources on an idle system is good, but once the number of messages increases, load due to our tools should not be so high as to negatively affect the system.

Shown in figure 4.5 and figure 4.6 are results of profiling of messages on a system with around 50% load. By comparing the figures one can see that even for a busy system, additional load due to our tools is low.

Shown in Figure 4.7 and Figure 4.8 is the CPU consumption of core 0 and 6 in the experiment presented in Figure 4.5 above. This further validates that use of our tools do not result in substantial additions to system load. Further, Table 4.1 shows the number of messages processed by Vortex is relatively high, underlining that the overhead impact of our tools is low.

Comparing Figure 4.9 with Figure 4.4 shows how the CPU usage of our kernel-side profiling resource increases with additional message load. The increase from 19 to 43.000 message per second, as shown in Table 4.1, results in a moderate increase in CPU consumption.

We have demonstrated that the overhead of our tools is low, irrespective of system load. One interesting question is how an increase in load affects the

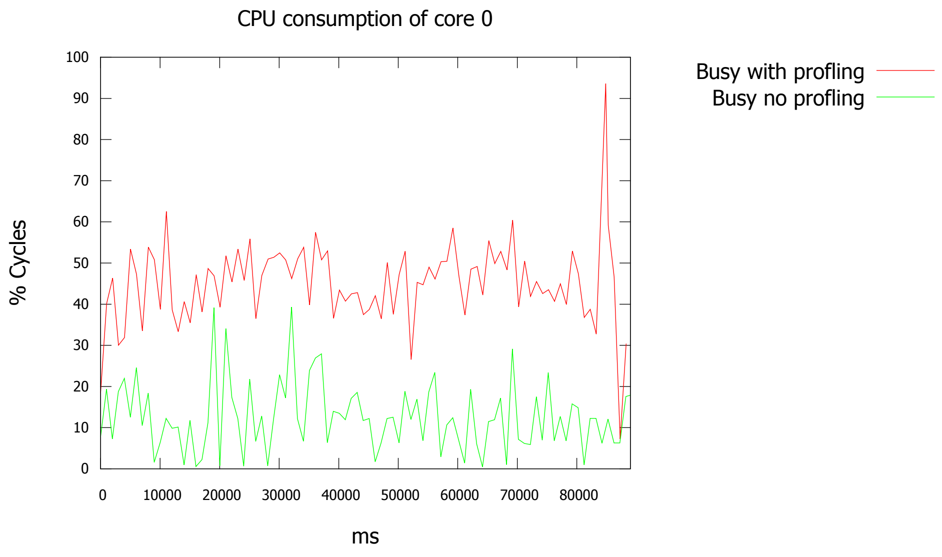


Figure 4.7: Core zero of a busy machine with and without profiling.

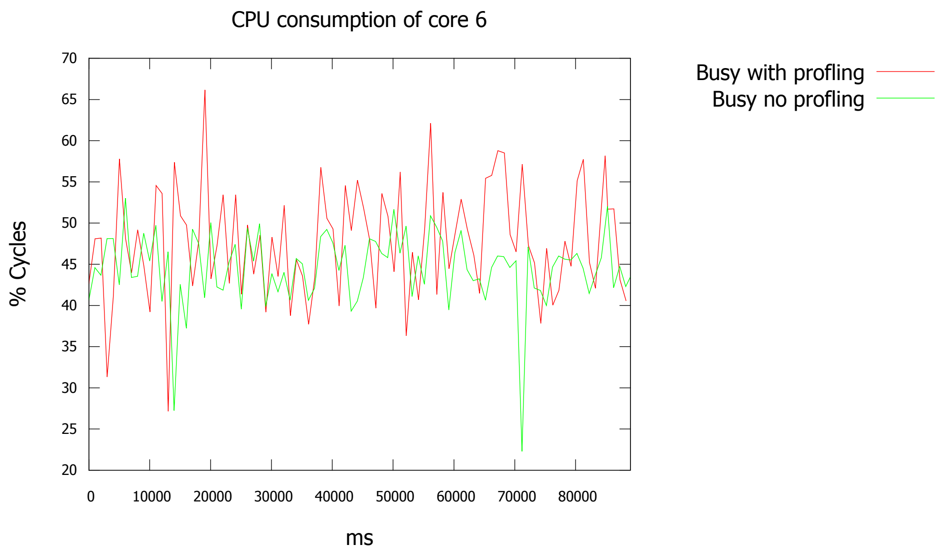


Figure 4.8: Core six of a busy machine with and without profiling.

number of messages processed by the Vortex kernel, since use of our profiling tools also result in the production of messages.

Shown in table 4.1 is the number of messages per second that is gathered for different system loads. Scaling from 19 message per second and up to 43.000 is a challenge, and results in a very high number of messages in a short amount

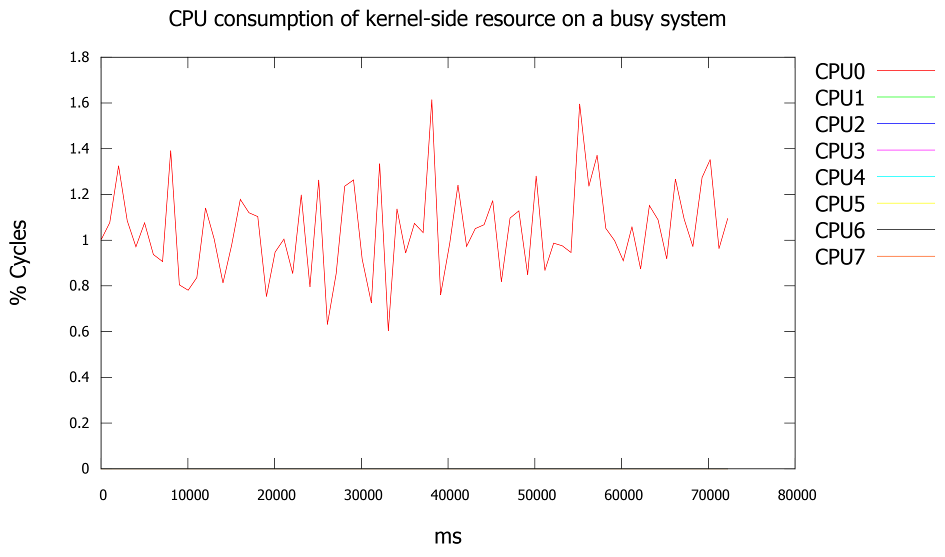


Figure 4.9: The CPU usage of the kernel-side profiling resource on a busy machine running Apache whilst profiling the TCP kernel resource.

Configuration	**Message per second**
Idle machine profiling TCP	19
Busy machine profiling TCP	43017
Busy machine profiling TCP with debug registers	32000

Table 4.1: The number of messages per second during message profiling.

of time. This means that for problems that requires a very high load, there a challenge in storing and presenting the message in a manner that allows the developer to gain insight from them. This affects the remote client as it is unable to store every message sent; it will store the latest 4096 messages to limit the amount of memory that are being used. 4096 was chosen as that is large enough so that some history is kept but the amount of resources spent is not too high. The number of kept message can be increased if needed.

When use of debugging registers is enabled, the number of messages drop noticeably. Since load is generated in a closed-loop fashion, with new connections only established after completion of the last one, this shows that use of debug registers and their accompanying debug exceptions is costly and does affect the system negatively.

Shown in figure 4.10 is the CPU consumption of a busy machine whilst the debug registers are enabled. They are triggered as a part of the TCP AIO write function, which happens every time the machine writes data in the TCP kernel

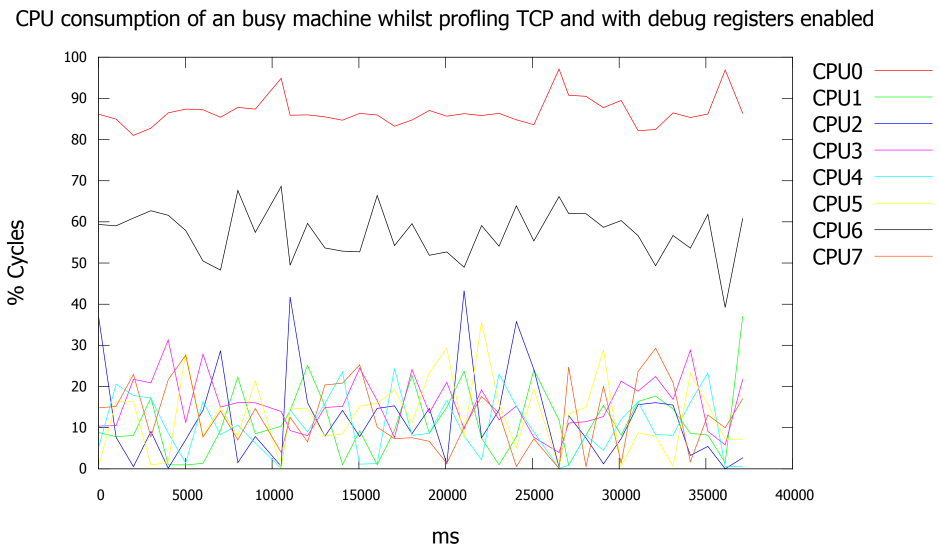


Figure 4.10: Showing the CPU of a busy machine with debug registers enabled.

resource. As seen in the figure, the CPU consumption increased drastically. Comparing the figure with Figure 4.6, one can see one core going from around 50% CPU consumption to almost 100% CPU consumption. This fits very well with table 4.1, as the number of messages per second drops by almost 10.000 messages whilst the load is still the same.

4.1 Summary

In this chapter have we have presented the results of experiments designed to quantify the overhead of our tools under various system loads. The evaluation shows that the process of profiling messages does require some system resource and that the amount of resources needed scales with the load of the system.

The experiments show that overhead is very high if use of debug registers is enabled. But our tools were not made to be used on a system during normal operation. They are development tools, and were designed to be used in development and debugging use and sessions. Still, it is important that they do not affect the system too much, preventing e.g. bugs from occurring.

5 Related work

This chapter presents some related work for this thesis.

Debugging OSs are often more complex than normal application due to their standalone, long running nature. In [46] they introduce the concept of a time-travelling VM to help debug non-deterministic bugs that requires long runtime to trigger. Time-travel allows the developer to more forwards and backwards in the execution path.

5.1 Event based debugging

One problem with larger systems is how to keep track of everything and simplifying debugging. Being able to group code together can help the developer to keep track and narrow down where in the code the fault might be. Using events as an abstraction is not new, and the event-based programming model has been around for many years.

Events creates a very useful level of abstraction that allows developers to structure events and their dependencies in their head. This has also earlier been used to aid developers in debugging [47, 48, 49]. Allowing the creation of events and having the ability to group code makes large systems and applications more orderly and transparent. Vortex is already an event-based system. This allows us to present some of the same information as these systems. e.g. in [48] they created events based on the behaviour they expect and compare them to the actual behaviour. Any event not behaving correctly can then quickly and easily be identified and it narrows down potential bug locations.

In [50] they use events to create global break-points by grouping code in events that allows the user to see high level patterns and then dig deeper on the faulty event. Others, as in [51], use events to debug large scale applications running on massive parallel systems, systems that generated huge amount of data that would have to analysed. Using events can limit the amount of data that has to be analysed and simplifies the debugging process. These systems are often designed to help developers debug distributed and large scale systems.

Similar to the work done in this thesis is [49], where they also uses events to debug object/actions programs in distributed systems. Their own *Clouds* distributed system has the notions of objects/actions implemented as a part of the OS. Action in *Clouds* is a unit of work, similar to events in the event-based model and Vortex.

6 Conclusion

In this thesis, we have shown that it is possible to create a tool for recreating the control flow, and creating new tools for debugging by visualizing messages sent to and from resources. We show that these tools can be created and used whilst the system is active and can help a developer with debugging errors and bugs in an event-based system. We also see that these systems can be enabled during normal systems operations with an affect that scales with the load of the system. This means that the tools can be used to find faults that might require time and different levels of load to trigger.

The tools created in this thesis expand the possible methods that a developer has when debugging bugs in event-based systems and show how to use the control-flow as an important step during debugging.

6.1 Use cases

6.1.1 Expand debugging possibilities

Developing OS and large event-based systems requires tools and properties to aid when debugging the complex and intermittent bugs that will be in such systems. Being able to add to tools to the tool-belt that developers has available is important and this thesis creates two more tools that can be used to debug errors.

Vortex is a separate OS and running on a machine without the possibility to attach a debugger to it. This means that it is not possible to set breakpoints in the code as one would normally do, and that way see the when different parts of the code is executed. Creating a tool that enables controlling the debug registers enables the developer to add breakpoints in his code and provide more of a normal debugging environment. The tool for controlling the debug registers enables developers to set break-points in the code, and bring their own exception handler. This allows them to use some of the normal debugging techniques, e.g. checking the value of variables during run-time and see how they change.

6.1.2 Connect one message with the sending process

One important question when debugging situations where one message caused the system to fail or a bug to emerge is who sent the message. Being able to identify the sending process is important is identifying the circumstances that caused the bug to appear and the ability to reproduce. The tools created in this thesis enables the user to see more information about each message and that way being able to connect a given message to a process. Being able to tie a specific fault to state created by one process can help to narrow down the set of circumstances that caused the fault and reduce the complexity in the debugging.

By using the debug registers is the developer also able to see which message it is that updates the offending variable or makes the function-call. This enables him to see which process sent the messages and the effect of the message.

Bibliography

[1] C. Mims. (2010, October) Why cpus aren't getting any faster. Online. MIT Technology Review. Accessed 18.11.14. [Online]. Available: http://www.technologyreview.com/view/421186/why-cpus-arent-getting-any-faster/

[2] J. Teubner and L. Woods, *Data Processing on FPGAs*, ser. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2013. [Online]. Available: http://books.google.no/books?id=qbJdAQAAQBAJ

[3] M. N. Krohn, E. Kohler, and M. F. Kaashoek, "Events can make sense." in *USENIX Annual Technical Conference*, 2007, pp. 87–100.

[4] R. Von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer, "Capriccio: scalable threads for internet services," in *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5. ACM, 2003, pp. 268–281.

[5] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur, "Cooperative task management without manual stack management." in *USENIX Annual Technical Conference, General Track*, 2002, pp. 289–302.

[6] K. Sakamoto and T. Furumoto, "Grand central dispatch," *Pro Multithreading and Memory Management for iOS and OS X*, pp. 139–145, 2012.

[7] J. R. von Behren, J. Condit, and E. A. Brewer, "Why events are a bad idea (for high-concurrency servers)." in *HotOS*, 2003, pp. 19–24.

[8] M. Rosenblum and T. Garfinkel, "Virtual machine monitors: Current technology and future trends," *Computer*, vol. 38, no. 5, pp. 39–47, 2005.

[9] E. H. Sibley, "Virtual machine monitors," *Encyclopedia of Computer Science and Technology: Volume 14-Very Large Data Base Systems to Zero-Memory and Markov Information Source*, 1980.

[10] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the linux

virtual machine monitor," in *Proceedings of the Linux Symposium*, vol. 1, 2007, pp. 225–230.

[11] A. Velte and T. Velte, *Microsoft virtualization with Hyper-V*. McGraw-Hill, Inc., 2009.

[12] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 164–177, Oct. 2003. [Online]. Available: http://doi.acm.org/10.1145/1165389.945462

[13] Å. Kvalnes, D. Johansen, R. v. Renesse, F. B. Schneider, and S. Valvåg, "Omni-kernel: An operating system architecture for pervasive monitoring and scheduling," *Parallel and Distributed Systems, IEEE Transactions on (Volume:PP , Issue: 99)*, 2014. [Online]. Available: http://dx.doi.org/10.1109/TPDS.2014.2362540

[14] iAD. (2014, October) iad centre. Online. iAD. Accessed 23.10.14. [Online]. Available: http://site.uit.no/iad/

[15] D. E. Comer, D. Gries, M. C. Mulder, A. Tucker, A. J. Turner, P. R. Young, and P. J. Denning, "Computing as a discipline," *Communications of the ACM*, vol. 32, no. 1, pp. 9–23, 1989.

[16] B. Armstrong. (2006, July) Vmms versus hypervisors. Online. MSDN. Accessed 30.11.14. [Online]. Available: http://blogs.msdn.com/b/virtual_pc_guy/archive/2006/07/10/661958.aspx

[17] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Communications of the ACM*, vol. 17, no. 7, pp. 412–421, 1974.

[18] M. Fenn, M. A. Murphy, J. Martin, and S. Goasguen, "An evaluation of kvm for use in cloud computing," in *Proc. 2nd International Conference on the Virtual Computing Initiative, RTP, NC, USA*, 2008.

[19] VirtualBox. (2014, November) Oracle vm virtualbox. Online. Accessed 14.11.14. [Online]. Available: https://www.virtualbox.org/

[20] Vmware. (2014, November) Vmware workstation. Online. Accessed 14.11.14. [Online]. Available: http://www.vmware.com/products/workstation

[21] Q. Lin, Z. Qi, J. Wu, Y. Dong, and H. Guan, "Optimizing virtual machines

using hybrid virtualization," *Journal of Systems and Software*, vol. 85, no. 11, pp. 2593–2603, 2012.

[22] S. Borkar, "Thousand core chips: a technology perspective," in *Proceedings of the 44th annual Design Automation Conference*. ACM, 2007, pp. 746–749.

[23] Amazon. (2014, October) Amazon ec2. Online. Amazon. Accessed 27.10.14. [Online]. Available: https://aws.amazon.com/ec2/

[24] Google. (2014, October) Google cloud computing. Online. Google. Accessed 27.10.14. [Online]. Available: https://cloud.google.com/

[25] Microsoft. (2014, October) Azure: Microsoft cloud platform. Online. Microsoft. Accessed 27.10.14. [Online]. Available: http://azure.microsoft.com/en-us/

[26] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu, "Ioflow: A software-defined storage architecture," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 182–196.

[27] Å. A. Kvalnes, "The omni-kernel architecture: Scheduler control over all resource consumption in multi-core computing systems," Ph.D. dissertation, Universitetet i Tromsø, 2014.

[28] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual 5 vols*, 052nd ed., Intel, Ed. Intel Corporation, September 2014, vol. 3A.

[29] D. Sweetman, *See MIPS Run*, ser. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2010. [Online]. Available: http://books.google.no/books?id=kk8G2gK4Tw8C

[30] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torrellas, "iwatcher: Efficient architectural support for software debugging," in *ACM SIGARCH Computer Architecture News*, vol. 32, no. 2. IEEE Computer Society, 2004, p. 224.

[31] F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazieres, and R. Morris, "Event-driven programming for robust software," in *Proceedings of the 10th workshop on ACM SIGOPS European workshop*. ACM, 2002, pp. 186–189.

[32] S. Peter, A. Schüpbach, P. Barham, A. Baumann, R. Isaacs, T. Harris, and T. Roscoe, "Design principles for end-to-end multicore schedulers," in *2nd Workshop on Hot Topics in Parallelism, Berkeley, CA, USA*, 2010.

[33] A. Gustafsson, “Threads without the pain,” *Queue*, vol. 3, no. 9, pp. 34–41, 2005.

[34] A. Ayers, R. Schooler, C. Metcalf, A. Agarwal, J. Rhee, and E. Witchel, “Traceback: first fault diagnosis by reconstruction of distributed control flow,” *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 201–212, 2005.

[35] D. Engler, B. Chelf, A. Chou, and S. Hallem, “Checking system rules using system-specific, programmer-written compiler extensions,” in *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*. USENIX Association, 2000, pp. 1–1.

[36] L. Fiege, “Visibility in event-based systems,” Ph.D. dissertation, TU Darmstadt, 2005.

[37] L. Fiege, G. Mühl, and F. C. Gärtner, “Modular event-based systems,” *The Knowledge Engineering Review*, vol. 17, no. 04, pp. 359–388, 2002.

[38] C. M. Patrick, S. W. Son, and M. T. Kandemir, “Enhancing the performance of mpi-io applications by overlapping i/o, computation and communication,” in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. ACM, 2008, pp. 277–278.

[39] P. Heidelberger and K. S. Trivedi, “Queueing network models for parallel processing with asynchronous tasks,” *IEEE Transactions on Computers*, vol. 31, no. 11, pp. 1099–1109, 1982.

[40] M. T. Jones. (2006, August) Boost application performance using asynchronous i/o. Online. IBM. IBM.com. Accessed 29.11.14. [Online]. Available: http://www.ibm.com/developerworks/library/l-async/

[41] NodeJS. (2014, November) Nodejs. Online. Node-js. Nodejs.org. Accessed 29.11.14. [Online]. Available: http://nodejs.org/

[42] S. Tilkov and S. Vinoski, “Node. js: Using javascript to build high-performance network programs,” *IEEE Internet Computing*, vol. 14, no. 6, pp. 0080–83, 2010.

[43] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, “Protothreads: simplifying event-driven programming of memory-constrained embedded systems,” in *Proceedings of the 4th international conference on Embedded networked sensor systems*. Acm, 2006, pp. 29–42.

[44] R. Cunningham and E. Kohler, “Making events less slippery with eel.” in

HotOS, 2005.

[45] P. Haller and M. Odersky, "Scala actors: Unifying thread-based and event-based programming," *Theoretical Computer Science*, vol. 410, no. 2, pp. 202–220, 2009.

[46] S. T. King, G. W. Dunlap, and P. M. Chen, "Debugging operating systems with time-traveling virtual machines."

[47] R. A. Olsson, R. H. Crawford, and W. W. Ho, "A dataflow approach to event-based debugging," *Software: Practice and Experience*, vol. 21, no. 2, pp. 209–229, 1991.

[48] P. C. Bates, "Debugging heterogeneous distributed systems using event-based models of behavior," *ACM Transactions on Computer Systems (TOCS)*, vol. 13, no. 1, pp. 1–31, 1995.

[49] C.-C. Lin and R. J. LeBlanc, "Event-based debugging of object/action programs," *ACM SIGPLAN Notices*, vol. 24, no. 1, pp. 23–34, 1989.

[50] S. Shende, J. Cuny, L. Hansen, J. Kundu, S. McLaughry, and O. Wolf, "Event and state-based debugging in tau: A prototype," in *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*. ACM, 1996, pp. 21–30.

[51] D. Kranzlmuller, "Dewiz-event-based debugging on the grid," in *Parallel, Distributed and Network-based Processing, 2002. Proceedings. 10th Euromicro Workshop on*. IEEE, 2002, pp. 162–169.