CHALMERS

Virtual Machine Based Program Modification

Code Rewriting and Instrumentation using LLVM

Master of Science Thesis in the Program Secure and Dependable Computer Systems

Tracy Meyers

Chalmers University of Technology

University of Gothenburg

Department of Computer Science and Engineering

Göteborg, Sweden, September 2011

Virtual Machine Based Program Modification
Code Rewriting and Instrumentation using LLVM

Tracy Meyers,

Examiner: Arne Dahlberg

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden September 2011

Table of Contents

Chapter 1 – Introduction

A challenging aspect of system level virtual machine development is how to combine a system that executes the guest's code at a high level of performance, yet is capable of detailed instrumentation and modification of the guest code. A significant amount of research in just in time (JIT) compilers exists and a number of existing products make this an interesting possibility to improve the performance. However, the performance penalty imposed by machine abstraction may outweigh the additional benefits abstraction creates. Can these techniques can be adapted for machine code to machine code translation and thereby prove to be capable optimizers that decrease the performance penalty of machine abstraction?

System level virtual machines are capable of providing a certain level of detail of what their systems are doing, but they are insufficient both in detail and in their capabilities to react to the system's behavior. Other specialized tools exist that are very similar to VMs [12], but are designed for individual programs instead of system wide instrumentation. The requirements we expect out of an instrumentation tool are that it must be system wide and it must be capable of introducing arbitrary code modification. The goal is to provide a base where system designers or administrators can introduce logging, security checks, or any other desired modification. How can we build a VM that is capable of introducing this level of arbitrary instrumentation into any code executing on the system?

The solution being sought by this paper is to determine the effectiveness of producing a system level virtual machine in such a way that it simplifies the process of runtime behavior modification and decreases the extra costs of running software on a VM. This will be accomplished by translating machine code through a JIT compiler. The translation process will introduce monitoring and arbitrary code changes.

The approach we have taken is to alter a portion of the traditional computing platform such that the software and its developers are unaware of the changes that are made. The transformation of a piece of software begins with some form of high level source code, such as C or Haskell. If we ignore the case of this code executing through a process virtual machine, such as uncompiled Haskell, a compiler will transform the high level code into machine code. This can be done with any compiler product and is not limited to a specific tool chain which this project uses. The machine code is then given to a physical CPU where the program is executed as one expects. This interaction can be seen in figure 1.1.

In our system the transformation from source to machine code remains unaltered but deviates from the traditional path at the point where the machine code is given to the physical CPU. Figure 1.2 shows a filtering screen after the X86 code block and another filtering screen just before the code is passed to the CPU. Between these two screens is what this paper will cover in detail. The first screen effectively breaks apart the code into its individual instructions and groups them into basic blocks. The basic blocks are then emulated and additional behavior may be injected into them depending on the goals of the policy. The expected goals of the policy are up to the designer or administrator, but some examples are to increase security through runtime checks, or to introduce logging to a specific portion of the system under execution. After this the emulated and transformed code is modified once again by a just in time compiler before it is placed in a code cache and executed by the CPU. This last step represents the second filtering screen in figure 1.2. Figure 1.3 presents a graphical representation of this process.

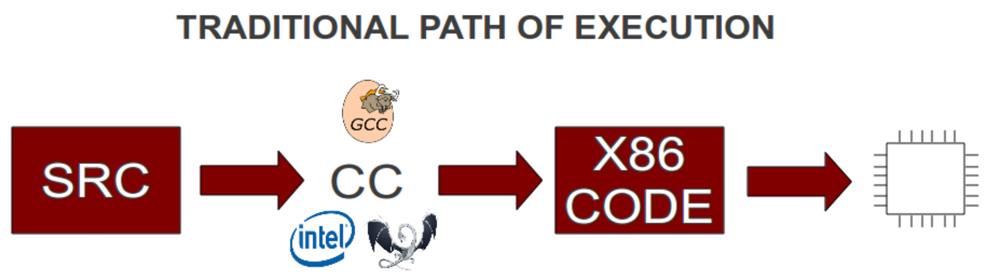

Figure 1.1 - The traditional path of execution from high level source code to execution on a CPU.

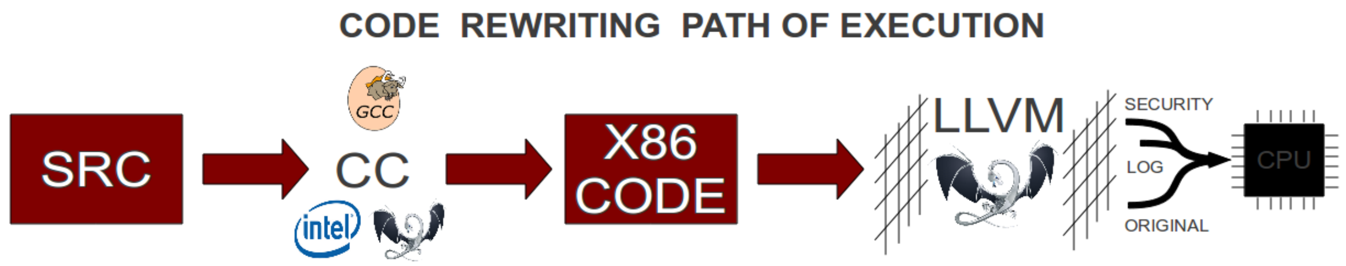

Figure 1.2 - The modified path of execution using code rewriting and virtualization.

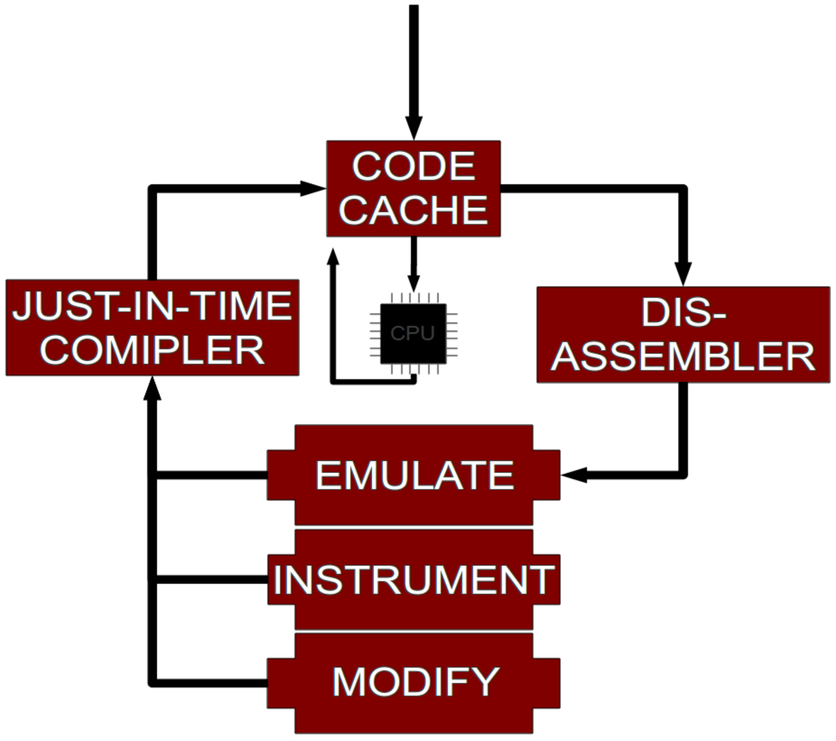

Figure 1.3 - Basic model of the system

1.1 Background

Every physical processor inside a machine has a defined set of instructions it operates on. These instructions are executed by the processor's internal logic which effectively emulates the conceptual view of what the instruction should do within the confines of the system. This abstraction requires significant optimizations before it can reach the levels of executing millions and even billions of instructions per second. Modern processor design has become so effective that the abstraction can now be moved into the software stack without suffering from unusable performance degradation. Abstraction of the machine within software has been approached in many different ways and each contributes a multitude of benefits to their execution.

Abstraction of a machine in software is labeled a virtual machine. Whatever form or implementation the VM may take, it provides a platform with beneficial properties when executing its guest code on the host machine. In the server space, the typical VM executes a virtualized machine on top of the same type of machine it is emulating. The benefit of doing this is to logically separate every VM from all other VMs as well as the physical machine itself. The first separation protects one kind of service and its user or processes from directly affecting the resources and data of another unrelated service on the system. This way every service can receive a dedicated machine from which its users can operate on without complicated control mechanisms between them. The second separation allows a VM to be separated from the physical machine it executes on and transitioned from one machine to another machine. The software executing on the VM will be unaware of the transition which allows it to continue offering the service it provided without interruption. These are benefits to a service being consumed by other users or machines.

A more focused benefit of virtual machines can be found in how they allow dynamic control of the code being executed on them. Unlike a physical machine where the behavior is typically fixed, a VM can be configured to apply safety checks or behavior modifications at runtime. A typically example of this is a Java virtual machine which applies bounds checking to ensure indexing into an array is kept within its defined boundaries. This is effectively a security control to prevent buffer overflow exploits related to overrunning a fixed size array. Other VMs are capable of instrumenting the code they are executing. This type of behavior modification can provide detailed information regarding what the software is doing at arbitrary levels of detail.

Every modification the VM supports must be capable of doing its work without committing an unintentional change in the state of the system.

1.1.1 Virtual Machines

A virtual machine is an implementation of a machine, whether real or conceptual, in software that is capable of executing a stream of the machine's instructions. The execution of the instruction stream must conform to the behavior of the machine, which may include unexpected side effects in order to support arbitrary code. Additionally, the software executing inside the VM must be limited to the bounds of the VM and be incapable of breaking out of the VM's sandbox. Current technology provides two levels of virtual machines: a high level VM, typically modeling a conceptual machine, and a low level VM, typically modeling a physical machine.

1.1.1.1 Process Virtual Machines

A process virtual machine targets the execution of its machine code as a process inside the confines of an operating system. The machine code it executes always targets a conceptual machine with its own instruction set in order to abstract away the actual hardware or operating system. This choice provides a clean slate from which code targeting the machine can be executed upon a variety of hardware and OS configurations. This is a very popular choice of development platforms and languages, such as Oracle's Java and Microsoft's .NET platform. While this paper does not focus on this type of VM, its concepts are still applicable.

The widespread use of process virtual machines means that a great deal of research has been invested into how to increase the speed and efficiency of the VMs. Since these types of machines execute code specific to their conceptual machine, the typical usage is to have an interpreter that executes arbitrary code and a JIT compiler that optimizes frequently executed code into a stream of the native machine's instructions. Therefore, interesting work into both when to transition the code to the native machine's code and how to optimize it with as little overhead as possible has been invested. Andreas Gal's thesis [1] provides an interesting look at how to turn typically expensive traditional static compiler optimization techniques into more efficient runtime optimizations through the use of hot trace selection. The insights he provides focus on how to structure the code traces encountered at runtime in an efficient way in order to ease the JIT optimization overhead.

1.1.1.2 System Virtual Machines

While research into process level virtual machines can be applied to this paper, the majority of relevant information comes from system level virtual machines. This type of machine is built to exhibit the behavior of a specific machine at the instruction level. The particulars of a machine below the intended behavior of its individual instructions, such as hardware bugs or the implementation of the logic at the silicon level, are typically not emulated. This level of virtualization is sufficient to allow an entire operating system to run completely unaltered on a host machine. The virtualized machine will share all resources with its host machine, but typically does so without any realization of this fact. This allows multiple OSes to run concurrently without any specialized hardware.

However, this type of execution has a clear disadvantage when compared to the same software executing on a real machine. The emulation will severely affect the performance of the software executing in the VM. A system level VM must be heavily optimized so that the code running on it is executed efficiently. Therefore, it must perform as close to physical hardware on a wide range of operating systems and their applications before it will gain acceptance by users.

There are generally two types of transparent system level virtual machines. The first is a system that fully virtualizes the guest machine entirely as a process in the host OS [5]. This means that every instructions that executes, from the first instruction at boot up to the last during shutdown, is executed via some type of emulation by the VMs software. However, since this type of VM runs as a process on the host system and because the X86 machine instruction set contains privileged instructions, it requires special handling to process those privileged instructions as their original behavior requires. Therefore, these types of VMs must inspect every instruction executed in ring 0 and rewrite the privileged instruction to manufacture the expected behavior.

An example of an instruction that must be specially handled is POPF [6]. When this instruction is executed in user mode the flags are set according to the value on the stack except for bits that are privileged. However, this same instruction when executed in ring 0 is capable of also setting the privileged bits. Consider a VM whose state has the interrupt flag (IF), a privileged bit, currently set and the value on the top of the stack has the IF bit unset. When the POPF instruction is encountered as a user level process in the VM then the instruction executes as expected because the VM itself is executing as a user level process. When the POPF instruction is encountered as a root level process in the VM then the instruction will keep the IF flag set even though the flags value on the stack has the IF bit unset. This is contrary to the expected behavior of the original code where a root level process is expected to be able to modify the privileged IF bit. In this example, the root level process in the VM executing in a virtualized privileged state is unable to perform the privileged action. Therefore, direct execution of instructions in a VM on an X86 processor cannot be done in a VM executing as a process on the host. Privileged instructions must be caught and translated to perform the expected behavior.

The second general type of transparent system level VM is one that fully virtualizes the guest machine with the assistance of hardware that traps privileged instruction for emulation by the virtual machine monitor. When a privileged instruction is executed in a non-privileged mode hardware state then a hardware trap is issued. The trap can then use the state of the OS to determine the appropriate action to emulate the instruction. For X86 machines, the trap and emulate approach is a relatively new development, existing only since 2006 [5]. Unfortunately trap and emulation introduces significant overhead which limits it from becoming the dominant implementation of a virtual machine for X86.

There is another type of system level VM that provides high performance but is not considered transparent. Paravirtualization is a virtual machine monitor that requires the guest OSes to be modified to be aware of the VMM. This awareness permits complicated operations to be executed more efficiently than the trap method used in hardware assisted VMs. The guest OS also removes as many complicated instructions as possible, but if they must be executed then they are passed onto the VMM which verifies and executes the instruction [7]. To achieve the highest performance while abstracting away the hardware, paravirtualization allows guests to access hardware devices at near native speeds through drivers.

The typical benefit attached to the use of virtual machines is server consolidation. Without using a VMM, it is reasonable to setup every service with its own machine to run on. This separation keeps the services isolated from each other, increasing fault tolerance and protecting data and resources of the individual services. However, since a VMM is able to manage multiple VMs while maintaining isolation, many of the services can be moved from individual machines to individual VMs operating on a single machine. This migration reduces the cost and storage of physical hardware and typically increases the CPU utilization per machine. A VM instance can be migrated to another physical machine with relative ease unlike an OS running directly on a physical machine. Migration from one set of hardware to another set increases the overall uptime by permitting the hardware the VM is running on top of to be serviced without reducing or eliminating the services the software provides. Finally, a virtual machine monitor can inspect the internal workings of the machine state, such as view the state of the CPU, as well as log the CPU and I/O usage of the guest machine. This information can be used to dynamically assign resources to the VMM in order to effectively and efficiently make use of available hardware.

Most VMMs focus on performance, but there also exists the need for detailed monitoring and even alteration of an executing VM. The inspection of a VM's internal state is a benefit, but the granularity of it is severely limited. VMs provide an opaque view of anything inside the emulated machine. For instance, a VMM cannot determine which process inside the VM is using the majority of the machine's allotted CPU time. Likewise the usage of I/O is limited to analyzing its use in the system as a whole. This is a problem because the view of guest OS's data structures and applications is very limited to nonexistent. If one of the goals of the VMM is to be able to directly inspect the internal workings of the VM, then additions must be made to the VMM. In addition to the limitation of inspection of VMs, there is also a limitation in the ability to introduce additional security. If the security of an operating system inside a VM is in question, the VMM is not capable of reconfiguring the OS at runtime in order to patch the vulnerability.

1.1.2 Dynamic Code Modification

All but the most basic of CPUs modify the code at runtime through reordering in order to take advantage of CPU pipelines. Likewise, dynamic optimizations can be made in software through the analysis of usage patterns in executing software. The usage patterns can either be analyzed and optimized immediately or can be deferred until their usage is frequent enough to outweigh their cost.

1.1.2.1 Byte Code Modification

Byte code modification is performed on process virtual machines. Due to the limited scope of a process machine, this is the most commonly researched and well understood code modifier. Most byte code executing in the machine is actually interpreted in software to perform the expected behavior of the instruction [13]. However, the VM may decide it is worth the cost to transform a chunk of the operating code and turn it into a form that is much more optimized. This modification is akin to a static code compiler, but typically employs cheaper optimizations to reduce overhead in both time and space [14]. The benefit to the system is an increase in performance in the code that occupies most of the program's execution time.

1.1.2.2 Machine Code Modification

A machine code modifier expects the code stream it encounters to already have been highly optimized. However, despite this there is still room for runtime improvement or introduction of new code into the stream. A highly optimized stream of instructions can still be improved upon through the knowledge of its use at runtime. Dynamo [4] has shown that speculative trace generation techniques, such as most recently used tail, can yield increased performance for many applications. This is effectively identical to what byte code modification does. Additionally, new behavior can be added by inserting instructions into the stream without altering the expected output, or the behavior can be altered if the intention is to patch the code.

1.1.2.3 Dynamic Code Modification Components

All dynamic code modification systems contain a basic set of components that make up their design. At the beginning of the process, code is read from some location in the system as a stream of instructions to the machine. These instructions are executed, whether directly or indirectly through emulation on the host machine, and their result continuously modifies the state of the machine. All of this is done in such a way that the host is only altered through expected paths of change.

The first major component of these systems is an interpreter. It is needed in order to make sense of the instruction under execution and then execute it so that the state of the machine is correctly altered. Typically an instruction is interpreted when the system assumes the cost to compile it is greater than the cost to emulate it (see Dynamo [4]). Since this is executing inside a virtual machine, interpretation is an easy way to safely alter the state of the VM without worrying about drastically affecting the state of the real machine. However, for some systems the interpreter is not used and is effectively replaced by a component that simply reads in instruction groups called basic blocks (see DynamoRIO [11] and Pin [15]). A basic block is a series of instruction that end in a branch to another location.

If the interpreter encounters a section of code that is executed frequently, then the cost to interpret the code is often higher than the cost to convert it to a non-interpreted format. This component involves the use of a just-in-time compiler that optimizes and outputs code in the host machine's instruction set architecture. In most systems that use an interpreter and a compiler, the compiled code typically corresponds to a loop. For machine code modification systems, the compiler is also used to aid linking of basic blocks into traces known as trace linking [15]. The newly generated native code now duplicates the original which adds to the overall memory cost of the system.

In order to effectively manage the compiled and duplicated code the system requires a special cache to manage their reuse. This component may be split into a variety of specialized caches depending on type of system and how separating caches improves performance (e.g. DynamoRIO uses a basic block cache and a trace cache). The cache never stores code that is going to be interpreted. This component is far from being simple as it must account for such difficulties as cache invalidation from dynamic code modification and virtual to physical memory mappings in the machine codes.

1.1.3 Just-In-Time Compilation using LLVM

The Low-Level Virtual Machine (LLVM) is a compiler framework originally developed at the University of Illinois at Urbana-Champaign [8]. It features a large set of tools and features that make it very powerful to analyze and generate code. The compiler is built upon an intermediate representation that is primarily focused upon static single assignment (SSA) form [9] for dataflow and control flow graphs. The optimizations can then generate highly efficient code from this IR.

The LLVM package contains an extensive C++ API for building a wide variety of tools that take advantage of its optimizations. The API spans the entire compiler toolset, from disassembling machine code to generation of optimized machine code from its original. Additionally, the API provides a complete implementation for generating LLVM's IR. This way new languages can easily be supported with a ready to go and robust compiler back end for optimization.

Ready-made tools built upon LLVM's API make up the typical compiler tool chain set, such as a disassembler, static compiler and linker. Additionally it contains an interpreter and dynamic compiler tool that operates on LLVM's bitcode format. LLVM's bitcode is the encoding of LLVM IR into LLVM's container format. A dynamic compiler can then use this format to access its library functions for inclusion to the actual program it wishes to optimize. Using this technique a simple process type of virtual machine can easily be created from any toy language.

1.1.4 Xen

Xen is a virtual machine monitor that originally was built to support paravirtualized virtual machines. As of Xen 3.0, it also supports full system virtualization using either AMD or Intel's hardware assisted virtualization features. A single privileged guest operating system executing in Xen's dom0 domain is allowed to have direct access to the hardware. This guest OS must be modified to be aware of the Xen hypervisor as well as be modified to operate as Xen's dom0. All other guest OSes operate in domU, where the either the OS is modified to operate on Xen's hypervisor or it is unmodified because the physical machine supports hardware VMs.

In order to host a guest OS, Xen provides a sample kernel called Mini-OS and uses newlib for its C library. Mini-OS is designed specifically to demonstrate how a kernel interacts with the hypervisor so that other OSes can reference its usage. However, it does provide a minimal set of features that can be used to execute additional services or applications and it has been extended for such purposes [10]. For the standard ANSI C library, Xen uses Redhat's newlib. This implementation is intended for embedded systems and lacks POSIX compliance and GNU extensions found in glibc. Therefore, any software ported from glibc to run on Mini-OS requires additional changes.

1.2 Purpose

System level virtual machines monitors provide a minimal amount of information about the software they are executing. They can know and log how much of the CPU or I/O is being used at any point during the runtime. Unfortunately, their knowledge of what services or processes are using these resources is limited or nonexistent and their ability to limit the usage of these resources does not exist. This lack of inspection and behavioral changes is intentional because these actions severely limit the performance of the software under execution. Otherwise such changes oppose a VM's goal to provide performance that is as close to bare metal performance as possible.

The need to provide inspection and behavioral alteration is possible outside the confines of the typical virtual machine monitor. These solutions typically work at the user process level and expect intimate knowledge of the application under inspection. They are nearly unlimited in their ability to instrument the code such that programming errors and performance limitations can be detected. One can even consider a debugger, such as GDB, to be an example of an instrumentation tool that also allows live updates of the executed code. While most of these tools limit their instrumentation to user level processes, tools like PinOS [2] exist that instrument the entire software stack. These tools are effectively virtual machines themselves as they are the first layer of software in the stack and limit the OS they execute.

In order to turn a VMM into a highly instrumented and modifiable software platform, we must determine an effective way to introduce concepts used in the instrumentation tools without severely detracting from the already reduced performance incurred by the VM. An approach similar to a debugger could be taken where soft breakpoints are inserted into the locations where instrumentation is desired. The effect of this on performance can be witnessed by running a debugger with a conditional breakpoint in code that is often executed. This choice can immediately be disqualified as the cost to handle such an instruction is typically very high and therefore restricts its usefulness. Another approach is to take advantage of the machine's memory management unit to detect read or writes to specific memory locations. While this is sufficient if the number of memory locations to instrument is limited to a reasonable level, it is too broad in scope and slow for large scale instrumentation. Finally, emulation of the whole system or a specific region of code is another alternative. The former has already been widely used in projects like QEMU [3] but insufficiently equipped for instrumentation. The latter has been used in project like PinOS and is sufficiently equipped but executes on bare metal.

We desire a system that is high performance, highly instrumentable without limitations in its scope, and operates on top of a VMM for wide deployment. While all of the previously listed options can meet at least some of these requirements, all but the last one that provides emulation are too slow or insufficiently detailed to provide complete instrumentation. This is why projects like PinOS have employed this form. This paper will further explore this domain by including a ready-made just-in-time compiler to optimize the emulated machine code in much the same way as a process virtual machine does. We will place the LLVM libraries and JIT into a dynamic code modification system to optimize the machine code to determine the effectiveness of this approach. The eventual goal is that this system can be used to dynamically alter the runtime behavior of a system in order to improve the overall security of the services provided by it.

1.3 Applications

It is helpful to gain an understanding of how this type of system is useful. The most obvious use case is for those using it for software development or server administration. However, it can also be a boon for those managing the deployment of their services to a customer through the level of invisible control it gives them. We will first present the behavioral goals of the project in order to drive the understanding of their application.

Transparency is a requirement for most types of system level virtual machines. The obvious exception to this is a paravirtualized one where the OS knows it is executing above another software layer. While this paper has already stated the assumption that it is built on top of Xen and the OS knows this (assuming it is not an HVM guest), the presence of a machine code modifier should not be known to it. This simple goal is difficult because the code is transformed into another form (further discussion upon its limitations is covered in more detail in section 3.2.3). This affects the applications of this system because any software running above it must not be aware of its presence. If it is aware, then it is possible that given enough knowledge of the underlying software the security of an application or the entire system could be compromised.

Inspection is a requirement for the already obvious reason that we desire to look at the inner workings of the software. While this wasn't possible for the typical VM, this system must be able to do so if given enough knowledge of the software executing on top of it. This ensures us the capability to fully monitor the state of the machine and provide data to determine an action to take if needed.

Control is a requirement in order to determine when an action should take place that alters the runtime behavior of the system. Given enough knowledge of the underlying software and given sufficient inspection of it, the machine code modifier must be able to transform the currently executing code in order to inhibit, correct, or alter the software.

Transformation is a requirement that performs the action that inhibits, corrects or performs any other alteration to the system's normal behavior. The actual action taken may be of any scope and may alter the underlying software in any way.

Flexibility is a requirement imposed by all of the above. The instrumentation and modifications must be allowed to take place in arbitrary code. The instrumentation and modifications themselves must be able to be placed at arbitrary, yet relevant, points in the system.

Of the five goals above, the first and last cannot be directly shown as specific use case benefits. We could consider them the characteristic traits that the other three possess in order to provide the desired behavior. Therefore, the use cases presented here are limited to inspection, control, and transformation.

1.3.1 Inspection

Inspection of the system is a passive behavior that either logs what the administrator is interested in or is used as a trigger to the other behavioral traits. Depending on the level of insider knowledge of the OS or application under inspection, this may be a significant drain on the system resources or it could be acceptable. In most cases the administrator requires insider knowledge of both the OS and the application it tracks in order to confine the inspection to a specific task.

A useful software analysis tool that can be created is to log the execution traces of an application at all levels of the software stack. The trace could then be used by a developer to analyze performance improvements or to understand the impact of a particular system call.

A very intrusive and draining case is one that tracks every memory read or write. Clearly this is so intrusive that it will bring the system to a virtual halt, but it could be limited to a specific memory range. One could think of this as being similar to a debugger's break on memory read or write, where the developer wants to know who is accessing that memory location.

1.3.2 Control

Security checks and enforcement could be the greatest benefit of this type of system. As established in the inspection section, if we have enough knowledge we can stop, alter or completely replace the behavior of a piece of code transparently with regards to the executing system's knowledge.

The integrity of the system's software is critical in order to prevent unauthorized code modifications from executing. Many mobile phone chipsets implement this in hardware in order to prevent unauthorized changes to the phone's firmware. However, this can also be done in software by enforcing this as a control policy in this system. This may be limited to a specific application or driver, but it could also be placed as a requirement on the entire system. For example, when an application is loaded into memory but has not yet executed, the integrity of the application could be verified through a hash code check. If the known hash, accessed via Xen's dom0, is the same as the one generated then the system permits the application to execute. However, if the hashes differ then the controlling software intervenes to prevent the code from executing. A message could then be sent to notify an administrator of the unauthorized code change.

Active monitoring of licensed software is another way this system can control the virtualized system. In this scenario, consider a limitedly licensed piece of software that processes SMS messages for a mobile phone service provider. If the software is licensed to processes no more than 50,000 messages in a day, then the control policy can be used to enforce this instead of the software itself. Therefore, the software can be written in such a way that enforcing the licenses is not part of its consideration and is left to an external policy developed by the administrator of the virtual system. A user logging in to the virtual system will be unable to manipulate the policy because it is outside its logical control.

1.3.3 Transformation

Transformation may be performed based on the results of the inspection or the result of control actions required by a policy. For example, it may inhibit the actions of the SMS processing application by removing the code that routes the SMS traffic. Alternatively, it may correct the behavior of a piece of code that contains a bug or to temporarily alter the runtime behavior. The following are development focused usage examples.

Hot patching is the action of applying patches to software without restarting the software or the system. If a bug is found in a piece of code, this type of virtual machine could simply replace the afflicted area with new code. A simple example that should be hot patchable is a common off by one error. Simply changing a JL with a JLE instruction in the emulated instruction cache is a simple way to transform the runtime behavior without loading new binaries or altering the binary code itself.

A common problem in debugging and testing cycle is to alter the behavior of software under test without altering the source code. For example, rather than execute a very time expensive function that is called before and is not relevant to the piece of code we are interested in, we could eliminate the call to the function itself. This transformation requires a policy that defines what should be removed when writing the new code to the cache and it will be done every time the targeted code is executed.

Chapter 2 – Problem Statement

The main problem we are solving is how to create a system level virtual machine that is capable of high performance yet is able to modify the executing machine code in order to introduce instrumentation and behavior modification. We will explore this problem from two related points. Firstly, are existing JIT compilers capable enough to improve the inherent performance penalty of machine abstraction? Secondly, can we build a VM that is capable of introducing arbitrary instrumentation or machine code modification into any executing code?

2.1 Known Problems and Solutions

System level virtual machines monitors are limited in the detail of the inner workings of the software executing on them. While they are able to log coarse grained resources utilization, such as CPU utilization or network bandwidth, they are unable to determine the utilization of specific software services. Additionally, they are unable to directly limit the capabilities of specific processes or services. However, the lack of insight into the software executing on the VM is intentional due to its effect on performance. Being able to peer inside the execution of software to determine what is happening is intrusive and comes at a great cost in time. After all, a VM must be fast enough otherwise no one would use it.

In this section we will cover a set of known possible solutions and discuss the benefits and problems associated with each. The solutions are not necessarily restricted to virtual machine monitor implementations but they all require fine grained control. Their outcome must seek to develop a system whose performance is sufficient and its capability to transparently inspect, control, and transform the executing software is significantly higher than today's VMM. In order to turn a VMM into a highly instrumented and modifiable software platform, we must determine an effective way to introduce concepts used in the instrumentation tools without severely detracting from the already reduced performance incurred by the VM.

2.1.1 Possible Solutions

2.1.1.1 Interrupt Solution

The first proposed solution is to insert X86 INT instructions or similar at locations in the executing code that we are interested in. Then when the instruction is executed it can notify the appropriate inspection handler to perform its duties. This simple approach is relatively easy to implement. Effectively this is what a debugger does when it a programmer requests a break point to be inserted at a specified location. The downside to this implementation is that it cannot maintain a reasonable level of performance. This is readily apparent when a programmer places a break point in a frequently executed piece of code. Even conditional break points severely hamper performance as their condition must still be checked even when the debugger determines control should not be restored to the operator. Using this to gain control of the system under execution to monitor and transform its behavior is too expensive.

2.1.1.2 Memory Read / Write Detection

Another approach is to take advantage of the machine's memory management unit to detect read or writes to specific memory locations. This provides the ability to inspect, control and transform how code uses the system's memory. However, it is readily apparent that the scope of this system is limited. For example, the possible transformations are likely limited to whatever data is being read or written instead of arbitrary code changes. Additionally, this suffers from performance issues if the memory region being read or written to is often used by the system. Such cases will likely make the executing software too slow to be useful due to the broadness of this type of inspection.

2.1.1.3 Emulation

Emulation is a simple idea that can be applied as a solution to a variety of problems. In the scope of our problem it is used in order to have complete control over the software executing on it. Whatever changes required can easily be made because each step of the system is entirely controlled by our software. Therefore it is a solution that easily solves the problem, but its costs are very high. Even a simple add instruction quickly expands to significantly many more instructions when it is emulated. This means the performance is likely too large of a factor and will prevent its use.

2.1.1.4 Code Caching

An approach that is well explored is one where code caching is applied. This uses a scheme where the code's instruction stream is only slightly modified in order to insert it into a cache or apply transformations to it. This method is well researched by many projects and typically has adequate performance. The benefit of this type of system is that it is highly expressive and most of the code is executed natively without modification. The system is expressive because at nearly any point it can decide to insert new code into the system that may provide instrumentation of a software service or transformation of binary code at runtime. Without any transformations applied to the code, it simply needs to be glued together so that it can execute entirely out of a code cache instead of the original and unaltered code. The downside to this is that there is an immediate upfront cost to convert the code to a cache and the state of the machine is more difficult to save than in emulation.

2.1.1.5 Emulation, Code Caching, and JIT Compilation

The final proposed solution combines some of the previous ones with the idea that their detracting qualities are mitigated through their combined benefits. As noted previously, emulation is highly expressive but its costs are very high. However, if the emulation is recompiled through a JIT compiler and inserted into a code cache then it may reduce the performance issues to a point where the system runs sufficiently fast enough. The JIT will be able to use constant folding, dead code elimination, loop unrolling and other qualities of the code that can be taken advantage of by the compiler. In many regards this will be similar to the advantages of a process level virtual machine. The issue with this type of system is mostly a question of performance. We do not know how much overhead can be eliminated so that it is faster than an equivalent system that is pure emulation. Additionally, the code cache requires a significant upfront cost we hope will be mitigated once the system reaches a stable state where newly encountered code is limited.

2.2 Related Work

2.2.1 Machine Code Modification

2.2.1.1 Dynamo and DynamoRIO

A significant amount of research into efficient machine code optimization and modification has been performed by Hewlett Packard in their Dynamo [4] project and Bruening's DynamoRIO [11]. These systems work off the idea that the cost of interpreting and optimizing code that is executed at least once will eventually outweigh the startup penalty involved. This goal is in line with that of this paper and the results and lessons learned from these projects were beneficial to this project's development.

2.2.1.1.1 Dynamo

The least interesting and highest cost portion of Dynamo [4] is its interpreter. Just like a process virtual machine, Dynamo is designed to interpret every instruction it executes until it discovers a hot spot in the code. The role of the interpreter is important because it permits Dynamo to peer into the instruction stream in order to determine when a piece of code ought to be optimized. This is a trade-off between Dynamo's reduced performance for an infrequently executed code block versus the performance and memory costs required to optimize every piece of code.

The interpreter executes all instructions as if they were running on physical hardware. When a branch instruction is interpreted, Dynamo performs supplemental operations for the instruction in order to determine when the code should be optimized. When a section of code ought to be optimized it is called a trace. So called "hot traces" are points in the instruction stream where Dynamo has determined the trace is executed often enough that it is likely more beneficial to analyze and optimize than simply interpret. Loops are the prototypical hot trace and are detected via a record of counters that monitor the execution of a branch's target address. Once a counter for a target address is exceeds a value, the point when the amortized cost of optimization over interpretation, then the trace is compiled and ready for execution directly by the machine. This is a common design pattern of virtual machines of any flavor and is a consideration for specific areas of the execution loop for this paper. However, our approach eliminates the interpreter and instead recompiles the interpreted code.

However, before a trace that is identified as hot can be optimized Dynamo must discover the possible exit points in the trace. Similar to identifying a trace's beginning, a backwards branch terminates the process. Recorded traces, what Dynamo identifies as fragments, are then optimized and ready for direct execution by the machine. In the event a trace becomes too long before an exit from the trace is discovered then Dynamo will cut the trace short.

Once a trace is optimized and ready for direct execution by the machine then Dynamo must be prepared to execute it often. A newly generated fragment is stored in a cache of fragments for future execution. One possible scenario during the generation of the fragment is for the trace to have exits into other locations in the fragment cache. These possible destinations are linked together for increased speed by permitting the direct execution of code on the machine will not be interrupted. Now a newly compiled trace can be executed directly by other fragments in the cache without the need for interpretation.

As the fragment cache grows and becomes filled with a majority of the programs runtime code, the overhead incurred by Dynamo is reduced. Linked fragments are able to run directly on the machine with infrequent uses of the interpreter, optimizer and linker. Over time the overhead incurred during start-up decreases to a point where its cost is negligible. This paper also seeks to amortize the cost of new code generation at start-up, with specific optimizations, in order to reduce the cost of its behavioral transformations. The technique of linking fragments is one we hope can be employed by our project in the future.

One major cost reduction in Dynamo's optimized code generator is that traces eliminate the overhead of call instructions and other types of branches. This allows the optimizer to effectively inline subroutines which reduces the overhead costs of Dynamo's system. Unfortunately, this method could not be pursued in this project, but Dynamo provides a good example of a potential cost reduction. This decision was influenced by other research [1] which has shown that super-blocks, similar to Dynamo's fragments [4], are not influential enough to reduce costs.

Overall Dynamo's goal is to increase the time performance of unaltered machine code. The benchmarks used [4] show performance increases as high as 22% over the same binary executed without Dynamo's optimizations. This is an impressive feat, but unfortunately the goals of Dynamo and that of this paper are different enough that this type of performance is unlikely. Specifically, this paper seeks to optimize and transform the code in order to alter its behavior. This means any behavioral transformations will result in code that has increased in size and complexity instead of the opposite.

2.2.1.1.2 DynamoRIO

The author of DynamoRIO [11] considers Dynamo to be the precursor to his system. Bruening vastly grows the goals, and implementation to encompass a system that maintains high performance optimizations but also introduces arbitrary code transformations. A program currently executing under the full control of the target machine as well as any program that could be executed are able to be run in DynamoRIO. As is the case with Dynamo and this project, DynamoRIO holds no requirements on the application under execution and from this requirement introduces a number of others.

DynamoRIO operates under the assumption that the overhead introduced by its modifications will eventually be insignificant when averaged over time. An efficient system that approaches or exceeds native performance is a requirement in order to promote significant adoption. However, efficiency is useless if the system fails to produce results identical to that of the program executing directly by the machine. The system must support arbitrary code transformations in order to build up additional logging or introduce stricter security checks. Finally, the system must be supported on commonly used machines and operating systems and must work on any type of program in order to promote adoption.

Unlike its predecessor this system does not employ an interpreter but instead solely executes the program from a code cache. Bruening determined that the cost to create and store all executed code is insignificant over time. Therefore, we will follow this design choice and cache executed code after we've modified it via our JIT compiler step instead of interpreting any of it. Bruening's code cache introduces a basic block cache to copy each basic block unit of the program that is executed for observation and manipulation.

DynamoRIO defines a basic block as an entry point to a control transfer. Every instruction will be executed natively with the exception of control transfer instructions. If these instructions point to code that has not been cached yet then they produce stubs that return control back to DynamoRIO. At this point the system is able to generate new code so that the application can continue executing.

Once the connected block is generated and added to the basic block cache then it is time to link the two blocks together. Bruening discovered that each link, whether incoming or outgoing, must be kept track of for efficiency and consistency with regards to insertion and deletion of the links targeted. He also discovered that placing the exit stubs in a separate cache was beneficial when deleting and reintroducing the exit stubs as well as to reduce the overall size of the basic block cache.

An instruction cache that stores every basic block in an address different than that of the original creates a challenging problem to solve. Since the application is expected to have a state identical to that of the original, an indirect branch that is altered to use the new address of its target in the instruction cache has a more difficult process to reverse this change. Bruening attempted to two distinct techniques to implement this correctly but found complete replacement too challenging. Specifically, a return address on the stack may be read by another instruction and its value used as part of its operation. This presents the need to watch every read of the application to make sure it is supplied with the correct address. Bruening considered direct replacement promising in terms of performance but practically was infeasible to implement efficiently while maintaining correctness.

Indirect branches may still maintain correctness while achieving satisfiable performance. DynamoRIO ended up using an inlined look-up with a hash table to determine the required translation. Bruening discusses a number of possible theoretical hash table optimizations and how their performance affected DynamoRIO in reality. It is interesting to note his observations regarding the theoretical performance expectation and the performance observed as the machine's architecture design affects the algorithm choice. However, the performance result is noted as being the largest overhead of DynamoRIO.

The most effective performance increase comes from DynamoRIO's trace cache. Like Dynamo before it, the trace cache follows a long series of instructions that together are often executed. In the case of DynamoRIO the traces are composed of a series of basic blocks that are executed together. It also slightly departs from its predecessor's next executing tail (NET) scheme by ignoring backwards indirect branches when determining if a branch target should be a trace head. Naturally this will increase the length of a trace and likely improve the overall performance, but it will also increase the usage of the trace cache. Additionally, the trace size must be limited to prevent unlimited loop unrolling. Bruening discovered his scheme to improve performance by as much as 10% over the standard NET implementation.

One of the DynamoRIO's most straight forward and impressive optimizations is to directly connect code that makes an unconditional branch. Bruening calls this eliding unconditional control transfers. When fetching instructions to complete a basic block, an unconditional branch does not stop the process but instead is removed. The basic block that the branch pointed to is then appended to the current block thereby eliminating the cost of the branch. Bruening notes a few issues with this optimization as it may increases memory, pose a problem for infinite loops, and invalid memory at branch destination. Nevertheless a performance increase was seen by most test applications and was above 10% for some. The simplicity of this optimization makes it useful for this project.

2.2.2 Machine Code Instrumentation

2.2.2.1 Pin and PinOS

Machine code instrumentation is similar to pure machine code modification. The most significant additional challenge is how to transform the machine code in a way that is completely transparent. Pin [15] and PinOS [2] cover many problems encountered complete with solutions that are applicable to the original ideas behind this project. The largest differentiation between the two systems is that Pin focuses on application level instrumentation while PinOS, much like this project, focuses on system level instrumentation. It is clear there are significant challenges for any system that intends to virtualize and modify the runtime behavior of the system.

2.2.2.1.1 Pin

A single application can be instrumented at any location and with any kind of inspection using Pin [15]. In fact, an application that is already in progress can have Pin attached to it, be inspected for a period of time, then have Pin detach without any modifications to the original application binary. This is accomplished via a JIT compiler that inserts the instrumentation into the application's own code. All of this is done completely transparent to the application, thereby allowing Pin to operate on any type of application. While the instrumentation is interesting and eventually applicable to this paper's work, the use of the JIT compiler is its most interesting aspect at this time.

Just like Dynamo and DynamoRIO, Pin takes a series of instructions that are supposed to be executed and stores the instructions in a cache. Each exit from the block goes to a stub code which returns control directly to Pin. Once the exit's target is placed in the cache, then the stub call is removed if it is a direct branch. Otherwise Pin transforms the branch instruction into a predictive look-up to search for the appropriate cache entry in a very similar way to how DynamoRIO does it. Connecting blocks of code together so that a return to the dispatcher is unnecessary clearly increases performance and will be considered for this project.

However, unlike DynamoRIO, Pin uses a JIT to transform every block into newly compiled code. This occurs regardless of whether any instrumentation is necessary, which corresponds to the design of this project. One of the useful optimizations Pin’s JIT used is liveness analysis for the flags register where a write to the flags register that is never read before another write occurs can be optimized away. We will evaluate the performance implications of such an optimization as part of this paper in chapter 6.

2.2.2.1.2 PinOS

PinOS [2] is an interesting experiment into how to apply all of the instrumentation concepts developed by Pin for a single application into an entire system. Predictably the system wide approach encounters numerous highly complex transparency issues previously undiscovered by Pin. To simplify the implementation, Bungale and Luk used a modified Xen virtual machine monitor because it provides a number of features convenient for their type of instrumentation system.

One requirement of transparency, just as this paper requires, is that the operating system under execution must not be modified nor be aware it is executing by another layer of software. Additionally, the instrumentation tool must be independent of any services, such as standard library functions or memory allocation, provided by the OS. Therefore, in order to accomplish this goal the memory used by PinOS must not be accessible nor be known of by the OS. Xen makes this a simple task because it can place the guest OS on top of the PinOS layer in such a way that the guest OS assumes it is running on a real machine. This is accomplished via a Xen driver that performs memory stealing, attaching/detaching from a running system, and I/O services. Likewise our system is designed identically in order to support the following behavior.

The process of stealing memory from the guest OS is critical for transparency. Unfortunately this requires a modification to Xen in order to support this behavior. Xen maintains domain separation by keeping a map of every page a given domain is able to access. PinOS exploits this behavior by allocating a special hidden chunk of the domain's physical memory for its purposes and which is not known by the OS. Virtual addresses are more involved as they require stealing the pages from the kernel and detecting if the kernel ever tries to access or write into these regions.

This instrumentation tool does not run on top of an OS which provides plenty of services to store its observations to disk. Therefore, it must find another way to divert its plethora of logs into long term storage. PinOS is able to perform I/O operation with the help of a process running on Xen's domain-0. Xen supports the ability to map memory between the domains which allows the given process to store the logs. This simplifies what would otherwise require a significantly more costly development approach without Xen.

Interrupts and exceptions are problems common to any system that attempts to virtualize away the machine. PinOS solves interrupts with a simple check at the beginning of each trace to query the virtualized interrupt controller to check if any have occurred and then handle them. All interrupts that do occur are queued in order to allow the currently executing code to be safely free from any instrumentation code. Unfortunately the solution to exceptions is significantly more complex for any system that compiles the original machine instructions. Since the machine code is transformed by a compiler that means the original behavior is intact but the exact steps involved in reaching the end result may be mixed around. Therefore, whenever PinOS generates multiple instructions from a single instruction it will emulate these instruction in order to revert the state of the machine back to what it should have been if the original instruction had been executed. This is a major issue to tackle for any machine code optimizer that operates on the system level rather than the application level and produces highly optimized and mixed up code. Unfortunately, this paper must leave this exploration to future work, but the lessons from PinOS should be leveraged there.

Another obstacle to system level virtualization on X86 machine code is called the irreversible segmentation problem by Bungale and Luk. This issue stems from the X86's segmentation register behavior of having a selector that points to an entry in the global or local descriptor table and an inaccessible cached value of the entry in the descriptor. When a value is stored in the segment register, both the value in the descriptor table entry and the value in the cache will match. However, it is legal to write a new value in the descriptor table entry which is not reflected in the descriptor cache. This is expected behavior, and therefore is not a problem when the value in the GDT and the cached value are different. It becomes a problem once control of the machine is transferred away from the guest and the value of the register is overwritten. When that occurs, the value of the descriptor cache cannot be correctly restored when context is given back to the guest. The solution employed is to shadow the descriptor caches of the guest and translate the guest's segment instructions so that the segment selector value points to the shadowed entry. When context switch forces a segment register to be saved, PinOS is able to restore it by simply pointing the selector value to the shadowed location where the original value is relevant and correct.

One of the problems with implementing a machine code cache is what to do when a program modifies its own code. For application level caches, the only worry is self modifying code which is easily detected by write protecting the memory. For system level caches, the same problem is present but the cache must also be aware that multiple virtual addresses may be mapped to the single physical page that is being altered. PinOS uses a mapping of physical pages to all virtual pages that map to it. If the OS remaps pages in memory, any generated code in the cache will automatically be invalidated when used because the entry into the trace verifies the physical and virtual address ID. This ID will no longer match which will result in the code being invalidated and regenerated.

2.2.3 Advanced Machine Code Optimization

2.2.3.1 Andreas Gal’s Thesis

While this paper utilizes LLVM to perform the majority of the optimization that are applicable from a process level VM, it is beneficial to look at current research. This builds understanding of the optimization used in LLVM as well as the possible improvements to LLVM or even possible implementations for replacement of LLVM's JIT. Andreas Gal's research into improving the current JIT optimizations focuses on static single assignment and how it can be simplified.

Static single assignment (SSA) form is a well known transformation step as part of the compiler's intermediate representation (IR). This form greatly enhances the capabilities of many optimization passes to produce a higher performing output. Unfortunately, as noted by Gal, the cost to do this is expensive for basic blocks because the block may be entered from multiple locations, or control flow merges. This occurs despite the fact that at runtime many of the merge edges are never executed. All of these entries into the basic block result in the introduction of costly φ-instructions.

Due to the nature of a JIT, the exact usage of the code is known. This makes simplifying SSA form only into what is needed at runtime a desirable goal. Gal presents a simplifying variant by eliminating such unnecessary factors and eliminating most control flow merges to reduce the majority of the cost of conversion to SSA form. It is done through lazy updates of basic block merge points where only the edges actually used at runtime are considered. This reduces both memory and time costs involved in compiling the code at runtime. In the event that LLVM is inefficient in its optimizations and code generation, applying Gal’s work to the project will be useful at many stages of the compiler’s development.

Chapter 3 – Approach

This study is based upon readily available tools that when combined are able to evaluate the effectiveness of runtime machine code modification in a virtual machine. The effectiveness will be measured through benchmarks in order to identify the design choices and optimizations that are most successful. In this chapter we will look at (1) an in depth description of the architecture and present the system design from all levels, (2) a description of the benchmarks used to analyze the performance of some of the design choices and optimizations, and (3) the limitations of this design and of the tools employed will be noted.

3.1 Architecture

The code rewriting system consists of a few high level components executing on top of the computer system's hardware (Figure 3.1). The focus of this paper is on the code rewriter portion, but a general understanding of the system as a whole will hopefully answer questions regarding why it is useful.

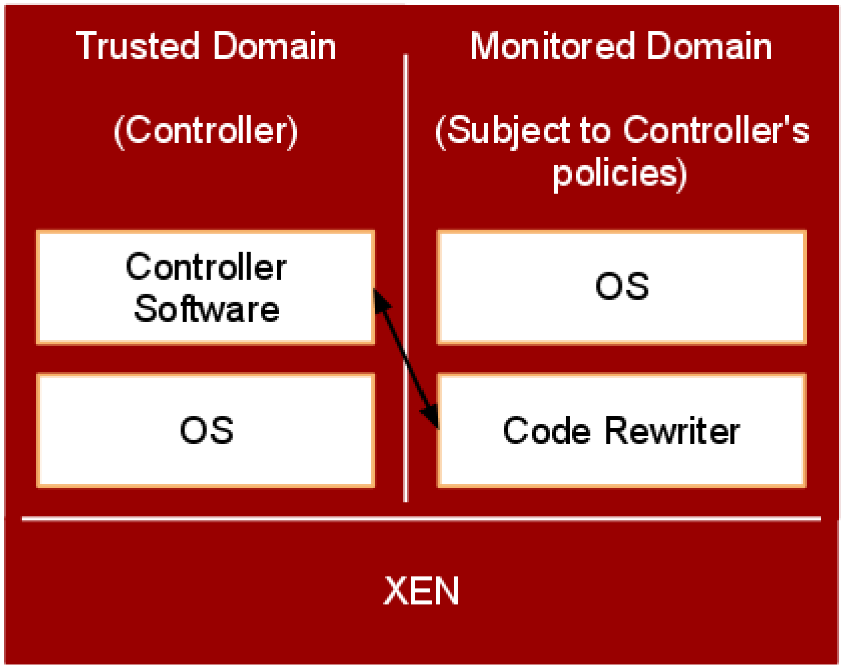

Figure 3.1 - Basic architecture of the code rewriting system

The first layer is Xen and it is the main component that everything else is built upon. Xen has some useful features that will be taken advantage of in order to simplify the role of the code rewriter, such as secure separation between the trusted domain and its OS and the domains being monitored. The trusted domain provides a safe location for the policies that create instrumentation, control and transformation to the monitored domain. The monitored domain is where the code rewriter exists and through the use of Xen features is able to hide itself from the OS the code rewriter will work on. All the output from the policies that control the code rewriter is returned to the trusted domain. This ensures the monitored domain never has access to the policies nor their results. Not all of these components have been implemented so far, but this chapter will in part explore their expected use.

Xen is a system level virtual machine monitor used by this project in order to provide security through logical separation of machines and to logically separate the code rewriter from the software it operates on. The first and main component is the hypervisor that maintains control over the computer system. The second component is the trusted domain that effectively controls the monitored domain. The final component is the monitored domain that executes the code rewriter which in turn executes the OS.

The Xen hypervisor is a thin layer between the physical hardware and all the software that executes on the machine. The purpose of this layer is to abstract away the details of the physical machine in order to allow multiple operating systems to execute on a single machine. This means it has control over the processor, and therefore the scheduling of the virtual machines, as well as the distribution of memory, and therefore logical separation of the virtual machines. However, the hypervisor gives the trusted domain the power to control nearly everything unrelated to CPU scheduling and memory distribution.

The first virtual machine executed by Xen is the trusted domain VM. It mostly plays the same role as that of an OS executing directly on the hardware. However, it is modified and fully aware it is executing on the hypervisor in order to share most of the I/O hardware. In this project the trusted domain, or domain 0, will be the manager of the code rewriter's policy which controls the transformations made to software that executes on the rewriter. A policy may be as simple as a count of all the memory writes to specific range of addresses or as complicated as a system that actively monitors for security breaches in order to deploy silent responses that halt the intrusion.

Unfortunately the definition of the policies is outside the scope of this paper and will not be covered in detail.

All other virtual machines are unprivileged monitored domains also known as domain U. These VMs are not able to access hardware directly and are either modified OSes that are aware they are paravirtualized on Xen, or are unmodified but executing as hardware assisted virtual machines. In this paper we focus on paravirtualized software. All I/O communication will pass through the hypervisor to the trusted domain where the action, such as sending data on the network, takes place. However, since the code rewriter is the first layer of software and not the OS, we must take special care to hide it as the OS assumes it has full control of the machine. This is done in the same way that PinOS used Xen to steal memory. A special chunk of physical memory will be reserved exclusively for use by the code rewriter.

Finally, the last piece of Xen used is its mini-OS. This is a sample kernel Xen provides and uses newlib for its C library. Mini-OS is designed specifically to demonstrate how a kernel interacts with the hypervisor so that other OSes can reference its usage. However, it does provide a minimal set of features that can be used to execute additional services or applications. For the standard ANSI C library, Mini-OS uses Redhat's newlib. This implementation is intended for embedded systems and lacks POSIX compliance and GNU extensions found in glibc. Therefore, any software ported from glibc to run on Mini-OS may require additional changes. Specifically, all the basic dependencies LLVM has must be ported to link with newlib.

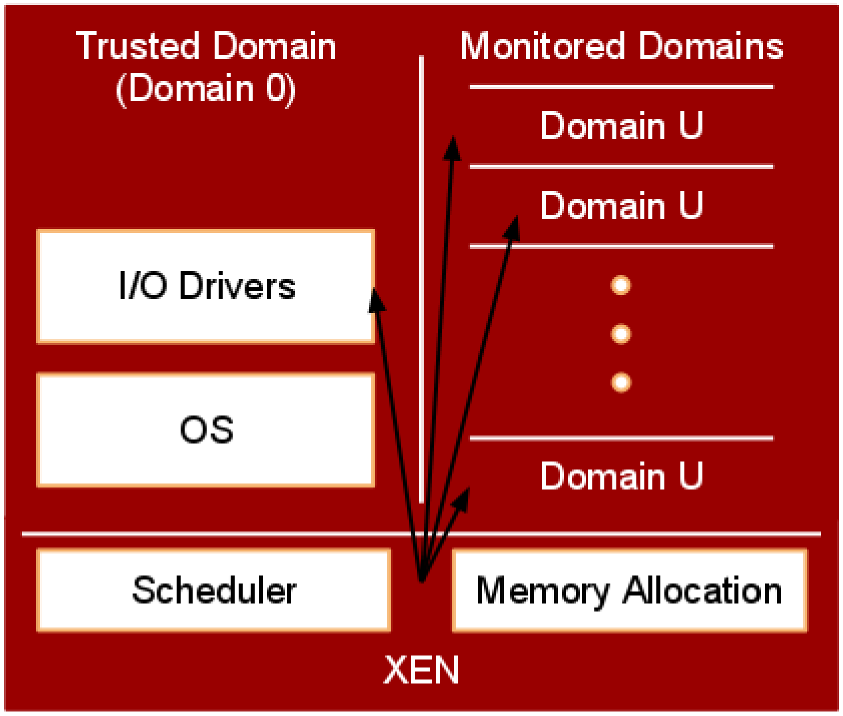

Figure 3.2 - The general architecture of Xen

The code rewriter is the main focus of this project. It combines a disassembler, a code generator, a virtualization of an X86 machine, an emulation of an X86 machine instructions, a JIT compiler and a generated X86 code cache. Figure 3.3 shows the basic layout of the rewriter and the flow of data through it. The exact operation will be furthered described in the design and implementation section of this chapter.

Each piece of the code rewriter is built on top of or uses the Low Level Virtual Machine (LLVM) tool chain. For example, the disassembler uses a C++ API provided by LLVM to take an instruction stream and decipher its bytes into their operation names and operands. This is a convenient tool because it provides most of the behavior needed in order to refactor the machine code into something new. The emulated instructions can even be precompiled into LLVM's IR format for easy code generation.

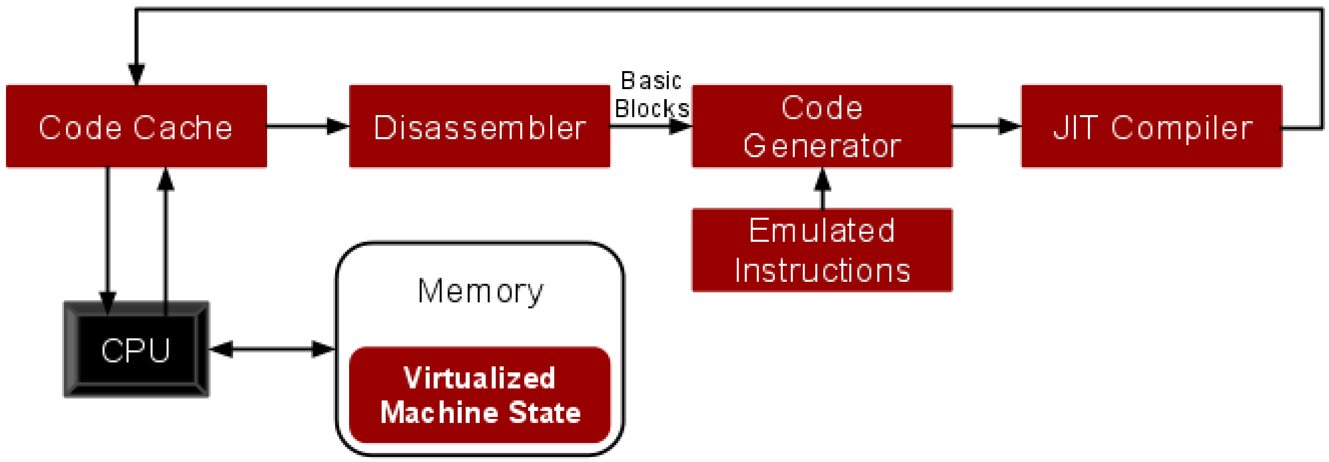

Figure 3.3 - The structure of the code rewriter. The code cache begins execution, gives control to the disassembler if it cannot execute the given address, the disassembler passes a block of instructions to the code generator, the code generator uses the implementation of the emulated instructions to stitch the code back together, and finally the JIT compiler generates new machine code that is given back to the code cache before it is executed on the CPU. The executed code will adjust the state of the virtualized machine code in memory that is hidden from the actual OS.

The final piece of the architecture for this project is the software under execution by the code rewriter. This is expected to be an operating system and furthermore is expected to be Linux. The kernel version must be known in order for the policies applied to the system to be effective. For example, a policy that monitors the execution of a specific application must be able to instrument certain parts of the system in order for it to determine when and where the application is loaded to memory.

3.2 Design and Implementation

This section will cover the design and implementation of the code rewriter only. The design and implementation of the policy component and LLVM's integration with newlib standard library will not be discussed. The approach will be bottom up beginning with the first code that needs to be rewritten up to the point where the rewritten code is executed on the machine.

The current state of the project does not use a virtual machine in order to execute the code. Although that is the desired goal, the current design is similar to a process virtual machine as it executes as a Linux user space process. However, this does not limit it to executing user space applications. A significant amount of testing has been done on the L4 microkernel's [12] boot loader, called Kickstart, to introduce more of the instructions used in loading an OS into memory. To be able to execute Kickstart, or some other application, its executable file format must be decoded so that it can be placed in memory at the correct locations and then executed from its predefined starting location. This is the first component of the code rewriter.

3.2.1 Loading Binaries into Memory

The executable file format supported by the code rewriter is the Executable and Linkable Format (ELF). ELF is used by many operating systems and most importantly by the OS used to evaluate this project. Its format is well documented and there are many tools and libraries for reading and writing these files. In this project we use Red Hat's elfutil library for parsing ELF files due to its simplicity and common usage across many Linux distributions.

The loader begins by reading the file's ELF header. Once all the data structures are filled in by elfutil, we begin by fetching the entry point address which is where the first instruction to execute is located. Next the file's memory segments defined in the ELF are loaded with their respective data. A simple executable's file segments are named .text, .bss, .rodata, etc.. Finally, the machine type is defined in the ELF header and it is used by the code rewriter to initialize LLVM with the proper configuration for disassembling the machine code. Figure 3.7 displays the basic flow of the ELF loader.

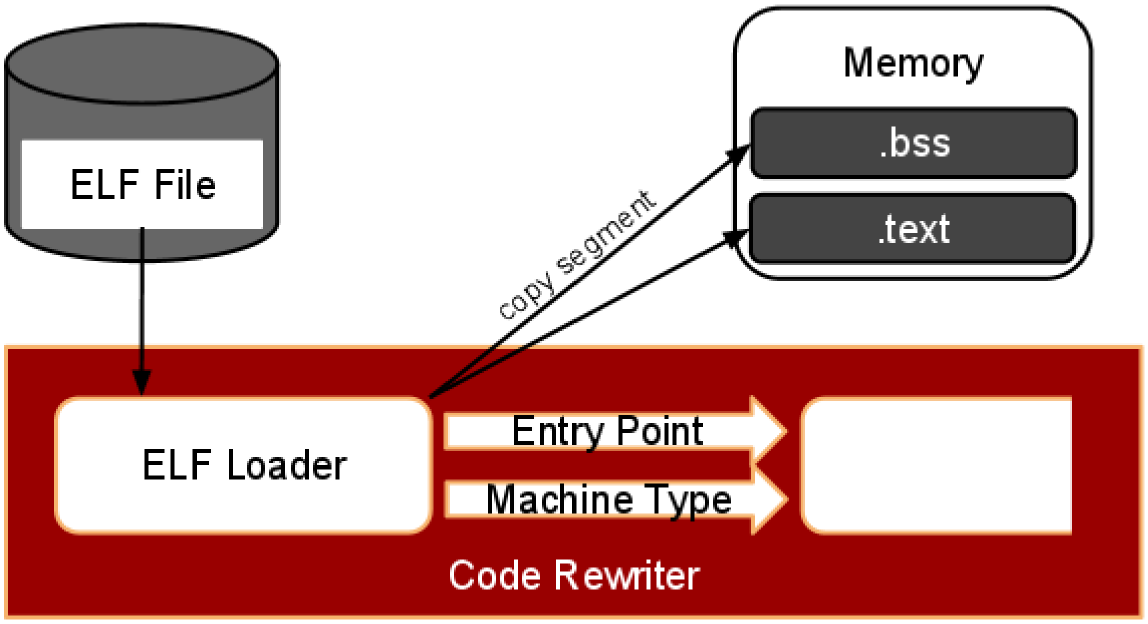

Figure 3.4 - The ELF loader is the first component of the code rewriter. It takes as input an ELF file and performs the following actions: (1) gets the entry point address, (2) loads segments into memory and (3) gets the machine type definition of the ELF file.

3.2.2 Deciphering Machine Code

Now that the starting address and the machine's type have been identified, the next step is to begin deciphering the instruction stream of the program's code. At this point LLVM's MCDisassembler (MC stands for machine code) interface is used to determine the instruction’s operation and operands. The stream of instructions is disassembled until an instruction is encountered that may alter the instruction pointer register. Alternatively it could ignore conditional jumps and continue disassembling until an instruction is encountered that guarantees a change in the IP register or some other kind of optimizations. Once a block has been disassembled it is immediately sent to the code generating for emulation.

3.2.3 Generating Code

The code generator takes as input a set of emulated functions, a pointer to the virtualized CPU state and a block of disassembled X86 instructions. The code generator outputs an LLVM IR function. The function can then be called to execute the behavior of the block given the state of the virtualized machine. This flow is outlined in figure 3.5. This is the most important piece of the code rewriter as it requires the majority of analysis in order to determine the most effective optimizations.

The code generator first begins by creating an LLVM IR function that will eventually be generated to X86 code and called just like any other C function. One of the major focuses of this function is to define as much of the data used by it as constant data. This way the JIT compiler can optimize as much as possible in order to strip away unreachable blocks and use constant folding to eliminate unnecessary runtime computation. However, not all the input data can be eliminated. For example, the state of the virtualized machine cannot be known at the point in time when the code is generated. Therefore, any operations that use a register value or data from an address in memory must be evaluated at runtime. There are likely exceptions to this, such as writing to a register that is never read before it is overwritten again, but we hope those have already been found by the original optimizer of the binary.

The block of X86 instructions generated by the disassembler must be transformed so that the LLVM optimizers can eliminate as much of the emulation overhead as possible. The block is first separated into its individual instructions so that each is emulated by their associated and already compiled C function (see section 3.2.4 Emulation for a complete description of this component). Before the emulated instruction function is called, the possible operands are defined as constant values in the LLVM IR function. For example, a register operand is a constant integer that points to a register in the machine; a memory operand is composed of a set of registers and immediate values for calculating its targeted address; an immediate is simply a constant integer. The call into the emulated function is finally made and the process is repeated until all instructions in the block are generated. One final note is that the IP register must be incremented appropriately, either after each instruction or after each block, in order for relative jump instructions to resolve the correct destination. Alternatively, the appropriate compensation could be calculated directly to all relative branch instructions in the block.

Up to this point the JIT compiler is responsible for deciding if any of the emulated instruction function calls should be inlined or not. Due to reuse, the compiler may not choose to inline all of them to reduce the memory footprint. Therefore, each call is added to LLVM's list of functions that must be inlined.

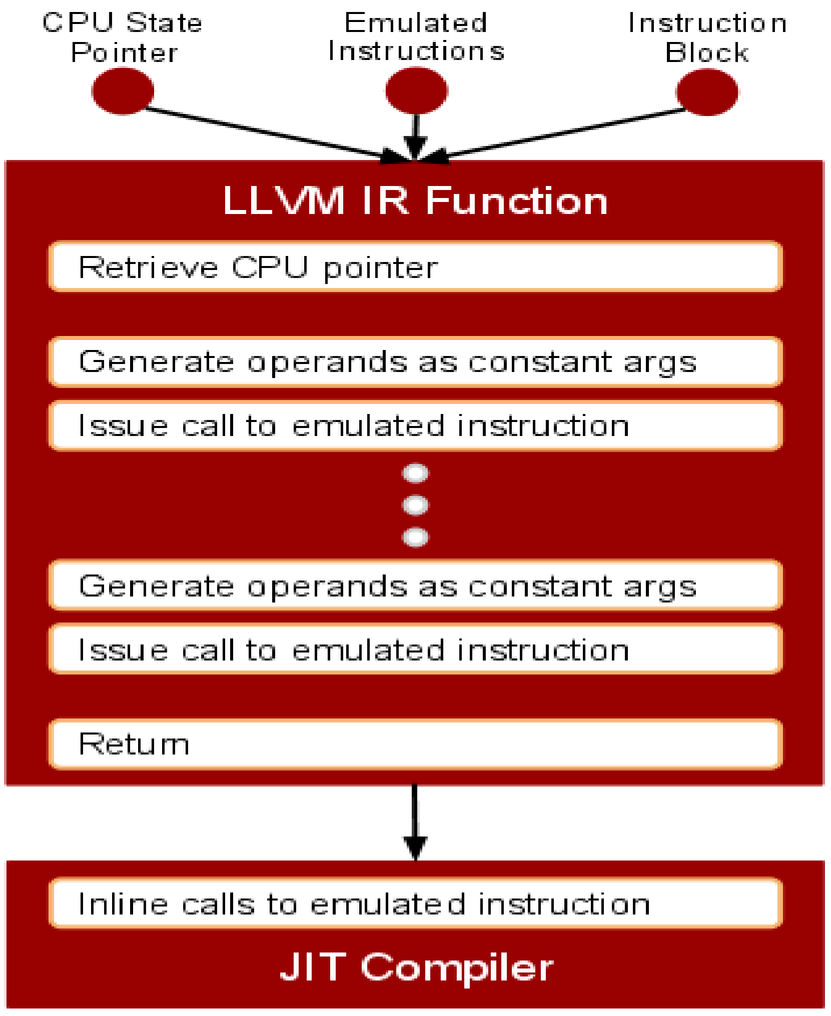
CPU State Pointer
Emulated Instructions
Instruction Block
LLVM IR Function
Retrieve CPU pointer
Generate operands as constant args
Issue call to emulated instruction
Generate operands as constant args
Issue call to emulated instruction
Return
Inline calls to emulated instruction
JIT Compiler

Figure 3.5 - How a block of instructions is generated into LLVM IR code that can then be optimized by a JIT compiler.

3.2.4 Emulating X86 Instructions

Each X86 instruction must be fully supported in order to run an arbitrary program. However, due to time constraints this project has limited the instructions to a subset of instructions necessary to run specific benchmarks and other applications. Each emulated instruction is a single function that performs the expected behavior. For example, an add instruction may add the contents of register RAX to the immediate value 2, write the result back to RAX and finally set the appropriate flags if necessary. However, there are a variety of issues ranging from simple ones like implementation choices discussed below, to complex ones such as how to handle self modifying code. Since this project remains too simple to process arbitrary programs, the resolution to the most complex issues will be left for future enhancements.

The decision to produce all the varieties of a single instruction verses producing one or two variants must be made. This is a typical trade-off between code reuse plus slightly increased complexity and code expansion with significantly less complexity per function. Due to the nature of X86 code containing 3 types per operand with up to 4 memory sizes and each basic instruction type containing multiple variants, the number of possible emulation functions for a single type of instruction quickly expands. Therefore, we typically used a single emulation function for each type of instruction in order to keep all of the instruction's logic in a single place. This places most of the work on the optimizers in order to eliminate the increased complexity in the dead code. However, we did evaluate if using specialized emulation functions based on the operation size has any advantage over using a single generalize emulation function for that operation.

It is important to keep the implementation of these functions as straight forward as possible in order to allow them to easily be optimized. This means any code that distinguishes between different types of operands must be separated so that the optimizer can eliminate dead code branches. For example, if the implementation checks if the value of an operand is a register, memory or immediate value then all but one of those branches can be eliminated. Then the optimizer can effectively generate code that is a single assignment just as if we had written a specialized version of the instruction. Figure 3.6 illustrates this principle. This is all possible due to the use of constants and inlining during the code generation stage.

Each call to an emulated instruction requires a pointer to the memory location of a data structure containing registers, flags, memory mappings and any other appropriate information. Since this is a common data structure used by all the generated code it cannot optimize any of the values away. However, employing the right optimizer may be able to eliminate reading and writing from one of these virtualized registers or flags. At this time only eliminating unnecessary writes to flags is supported. Otherwise this task is left solely to the JIT compiler and its optimizers.

Memory access is unfortunately insufficiently supported in the current implementation. For each memory address calculated by the original code, the real address where the data is stored must be obtained through translation. However, once the system is appropriately integrated with Xen this extra level of indirection will hopefully be eliminated.

```
cpu_t *cpu = ...;


const operant_t val =
       { .type = immediate, .imm = 2 };
const operant_t dst =
       { .type = register, .reg = RAX };
ADD(cpu, &dst, &val);


---------------------


void ADD(cpu_t *cpu, const operand_t *dst,
const operand_t *src) {
    uint64_t op0, op1, result;

    if (dst->type == register) {
        op0 = cpu->reg[dst->reg];
    }
    else if ... (other op0 assignments)

    if (src->type == immediate) {
        op1 = src->imm;
    }
    else if ... (other op1 assignments)

    if (dst->type == register) {
        cpu->reg[dst->reg] =
           op0 + op1;
    }
    else if ... (other state changes)
}
```

```
cpu_t *cpu = ...;

cpu->reg[RAX] += 2;
```

Figure 3.6 - The optimizer will be able to eliminate the ADD call and its multiple assignments and branches (left side) and turn it into the equivalent of a one line addition (right side in bold). This is possible by the use of constants during code generation as well as constant folding, inlining and dead code elimination during the optimization stage. (This example ignores the complexity of modifying the flags register.)

3.2.5 JIT Compiling Generated Code

The JIT compiler component has very little notable implementation features as it is dependent on LLVM. The main features are that it operates on single LLVM module where all the generated code is located and specific optimizations are defined or will be defined here. Essentially this component configures LLVM's JIT in order to be as effective as possible for optimizing individual functions and the functions they call. Therefore, this is a location for possible improvement depending on what optimizations may be introduced to improve the generated code.

3.2.6 Caching Compiled Code

The code cache component, just like the JIT Compiler, can be mostly dependent on LLVM. Since LLVM is able to manage all of the generated code in an LLVM Module, then we simply need to be able to track the starting address of a code block, which is the value in the IP register, and use this to look up the corresponding code. Unfortunately LLVM uses a string as the identifier for the lookup which will be slower than using the address of the block as a key. Therefore, the code cache uses its own data structure to perform look ups of addresses that return LLVM Function objects.

Chapter 4 – Evaluation Methods

The evaluation of the code rewriter system is based on executing a program and then measuring how long it takes the program to complete its task. Programs that evaluate this system are expected to be benchmarks used by the industry, but will be limited just the Dhrystone benchmark due to the reduced X86 instruction set supported by the code rewriter. Each experiment will focus on one type of modification to the code rewriter so that a number of configurations of the implementation may be suggested.

Each experiment describes a configuration decision of the code rewriter. The majority of these configurations focus on pieces of the code generator and emulated instruction components since those directly influence the effectiveness of the optimizers in generating efficient X86 code.

4.1 Experiment 1 – JIT Optimization of Operands

This experiment focuses on evaluating the machine code generation of the JIT compiler, specifically when passing operands to an emulated instruction function. In order to simplify LLVM code generation, every emulated instruction function has an identical set of parameters. This means that whether an instruction uses 0 or more operands, the call to its emulated function will always pass the maximum number of arguments even if the arguments are unused. The style in which these arguments are passed through to the emulated function is important because the compiler must be able to apply optimizations on their constant values. While these changes are focused on the code generator, the actual component being evaluated is LLVM's JIT compiler. This means a change in the implementation or configuration of the JIT may alter the results.

Figure 4.1 is the input used to evaluate the results of this experiment and figure 4.2 is the basic implementation of the emulated call instruction function. The input is a relative call instruction where the offset is an immediate value. Each operand passed to this function will be known at compile time and therefore a constant folder and dead branch eliminator optimizer should be able to cull a large amount of code. The instruction will be evaluated in isolation as the only instruction in a block. Since each block requires a small amount of initialization there will be additional instructions generated that are not directly related to the call instruction. The generated machine code is retrieved through GDB's disassemble command.

```
callq <offset to main>
```

Figure 4.1 - The input sample to use to test the effectiveness of the JIT in optimizing the operand passing configurations. This specific instruction represents a call to the program's main function.

```
void CALL(cpu_t *cpu, <parameters>)
{
    reg_t source;

    // push return address (cpu->reg[RIP])
    ...

    switch (type) {
        case MEM:
            source= ...
        case REG:
            source= ...
        case IMM:
            source= ...
    }

    if (type == IMM) {
        cpu->reg[RIP] += source;
    }
    else {
        cpu->reg[RIP] = source;
    }
}
```

Figure 4.2 - The emulated call instruction function implementation. The parameters vary depending on the configuration. Additionally, some lines were replaced with `...' for brevity.

Experiment 1.1 – Structured Configuration

Every operand is passed to the function as a well defined structure of the possible type of operand and its data. Figure 4.3 displays the pseudo code for one possible layout. The argument is defined as a constant structure value by the code generator in the basic block's function. Whether a pointer to the operand is given to the emulated function or it is passed by value is likely irrelevant because the optimizer should inline the emulated instruction function call into the basic block's function.

```
struct {
    operand_type_t type; // {MEM, REG, IMM}
    reg_t bits;
    union operand_data_t {
        struct {
            reg_t base_reg_idx;
            reg_t multiplier_size;
            reg_t multiplier_offset_reg_idx;
            reg_t offset;
        } mem;
        reg_t reg_idx;
        reg_t immediate;
    } u;
} operand_t;
```

Figure 4.3 - A structured view of an instruction operand. The operand has: (1) a type, such as register, (2) a size in bits and (3) the operand's data, such as an index into a table of virtualized CPU registers.

Experiment 1.2 - Flat Configuration

The flat structure is simply a flattening of the operand_t structure seen in Figure 4.3 so that each field is passed as a unique argument to the emulated function. This means 8 arguments per operand are passed which drastically increases the prototype length of the emulated function. Just as with the structured argument, any inefficiencies produced by passing many arguments should be made irrelevant by inlining the call. Figure 4.4 illustrates a sample prototype for the emulated call instruction.

```
void CALL(...
          const operand_type_t src_type,
          const reg_t src_bits,
          const reg_t src_mem_base,
          const reg_t src_mem_multiplier_size,
          const reg_t src_mem_multiplier_offset,
          const reg_t src_mem_offset,
          const reg_t src_reg,
          const reg_t src_imm,
          ...
          );
```

Figure 4.4 - The flat view of a source operands (in bold) as an argument to the emulated CALL instruction function.

4.2 Experiment 2 – Instruction Operation Size

This experiment focuses on evaluating the machine code generation of the JIT compiler, specifically the effects of specializing an instruction based on operation size. Most X86 instructions have multiple operation sizes where the result of the operation will be stored in an 8, 16, 32 or 64-bit register or memory. This means that for each instruction that has multiple operation sizes there are 4 different implementations. However, 4 implementations of nearly the same behavior will increase the memory foot print of this project and create maintenance issues. Therefore, an evaluation of the machine code generation using generalization versus specialization is helpful to evaluate the cost of generated code size versus maintainability.

```
add $0x10, %ax
add $0x1, %rcx
```

Figure 4.5 - The input sample used to evaluate the machine code generation of generalizing versus specializing the implementation of an instruction operation based on its size.

Experiment 2.1 - Generalization Configuration

To better understand why generalization may not generate smaller code, consider the add instruction. A 64-bit add to a register requires no special handling for an overflow. This is possible because we calculate the result using 64-bit variables in the implementation. However, an 8-bit add to a register must be careful not to alter to the values of the other 56 bits when the result is written to its register. For example, consider the effects of an overflow in `add $0x1, %al` where the value of RAX is 0x1FF. The expected output on RAX for this instruction is 0x100 and not 0x200. Therefore, the result of the add instruction must be masked and the write to the RAX register must only alter the intended bits of the register. However, this rule only applies to 8 and 16-bit X86 operations as 32-bit operations simply clear the upper 32 bits of the register.

```
void ADD(cpu_t *cpu, const reg_t op_bits,
         ...
         const reg_t src_imm,
         ...
         const reg_t dst_reg_idx,
         ...) {
    const reg_t mask = (((reg_t) (1 << op_bits)) - 1);
    const reg_t result = cpu->reg[dst_reg_idx] + src_imm) & mask;
    if (op_bits >= 32)
        cpu->reg[dst_reg_idx] = result;
    else
        cpu->reg[dst_reg_idx] = cpu->reg[dst_reg_idx] & ~mask) | result;
}
```

Figure 4.6 - A generalized implementation of an add of an immediate to a register. (This example ignores the complexity of modifying the flags register and does not support any other form, such as immediate to memory, of the add instruction.)

Experiment 2.2 - Specialization Configuration

Each version of the ADD function is specific to its operation size and therefore its implementation is less complex in C. Figure 4.7 contains the implementations being evaluated for a 16-bit add and a 64-bit add.

```
void ADD16(cpu_t *cpu, const reg_t op_bits,
           ...
           const reg_t src_imm,
           ...
           const reg_t dst_reg_idx,
           ...) {
    uint16_t *dst = (uint16_t*) &cpu->reg[dst_reg_idx];
    *dst += src_imm;
}
void ADD64(cpu_t *cpu, const reg_t op_bits,
           ...
           const reg_t src_imm,
           ...
           const reg_t dst_reg_idx,
           ...) {
    cpu->reg[dst_reg_idx] += src_imm;
}
```

Figure 4.7 - A specialized implementation of a 16 and 64-bit add instruction of an immediate to a register. (This example ignores the complexity of modifying the flags register and does not support any other form of the add instruction.)

4.3 Experiment 3 – Flag Liveness Analysis

This experiment focuses on evaluating the machine code generation of the JIT compiler with regards to flag liveness in the virtual machine. Many X86 instructions read and/or modify the flags register. A single flag result set by one instruction could be optimized away if the same flag is written to before it is read. This experiment aims at comparing the code generated for an identical block of machine code when no flags are set (flags are ignored), 1 flag is set, and when explicit liveness analysis is used.

```
add $0x40,%rcx
add $0x40,%rcx
jmp 400192
```

Figure 4.8 - The sample code block that evaluates the machine code generation of the JIT compiler with regards to flag liveness.

Experiment 3.1 - No Flags Set

This experiment shows what the JIT compiler could be output if this machine code block ignored setting any flags. In a proper implementation the add instruction would set the overflow, sign, zero, carry, parity, and adjust bits. This exists simply as a base case for the other experiments.

Experiment 3.2 - 1 Flag Set

This experiment shows what the JIT compiler outputs when only a single flag is set. In this experiment only the zero flag is used to make the generated machine code easier to read.

Experiment 3.3 - 1 Flag Set with Explicit Liveness Analysis

This experiment shows what the JIT compiler outputs when only a single flag is set and liveness analysis is not dependent on the JIT compiler. In this case, a piece of logic external to the JIT optimizers is used to evaluate when a flag must be set and when it can be ignored. Effectively this culls the calculations that can be ignored. In this experiment only the zero flag is used to make the generated machine code easier to read.

4.4 Experiment 4 – Optimizations

This experiment checks if the available optimizations provided by LLVM affect the time performance of the Dhrystone benchmark [16]. There are two possible places for optimization to occur. The first is applied to the LLVM IR code where optimization passes transform the generated IR prior to it being generated to native machine code. The second optimization is applied by the JIT compiler when it generates native code. Each optimization applies to a block of machine instructions and no more. An evaluation of the effects of the optimization passes to the IR and JIT compiler optimization level may be beneficial in determining which optimizations are useful and which are too costly. Table 4.1 contains a listing of the optimization experiments.

Experiment Name	**LLVM IR Optimization**	**JIT Optimization**
No Optimization	● None	None
LLVM IR Optimization	● Alias Analysis ● Instruction Combining ● Re-association ● Global Value Numbering (eliminate redundant expressions and dead loads) ● Control Flow Graph Simplification	None
JIT Default Optimization	● None	Default
JIT Aggressive Optimization	● None	Aggressive
LLVM IR plus JIT Aggressive Optimization	● Alias Analysis ● Instruction Combining ● Re-association ● Global Value Numbering (eliminate redundant expressions and dead loads) ● Control Flow Graph Simplification	Aggressive

Table 4.1 - Listing of the optimizations to evaluate. LLVM is evaluated solely based on the time performance of the generated machine code. However, since optimizations can be done on the LLVM IR or by the JIT compiler, both areas are evaluated to see if any gains a particular advantage.

4.5 Experiment 5 – Instruction Level Instrumentation

The final experiment looks at one way to add instrumentations and modifications to the machine code (see figure 4.8). The input program is the same Dhrystone benchmark used to evaluate the system's performance. Instead of simply letting the program execute normally, the VM uses a special instruction emulation for all executions of the call instruction. The modification replaces all calls to a rudimentary printf function with a call to stdlib's implementation of printf as seen in figure 4.9. Essentially this requires knowing the address of the original printf function in the running program's memory so that any call to printf will be replaced with the alternative printf's code. Additionally, a basic instrumentation will be used that prints the addresses of all call instructions executed at runtime. One example use case of this instrumentation is to determine which function calls are most frequently executed at runtime. The Dhrystone benchmark is then executed with and without the modification to evaluate possible performance impacts.

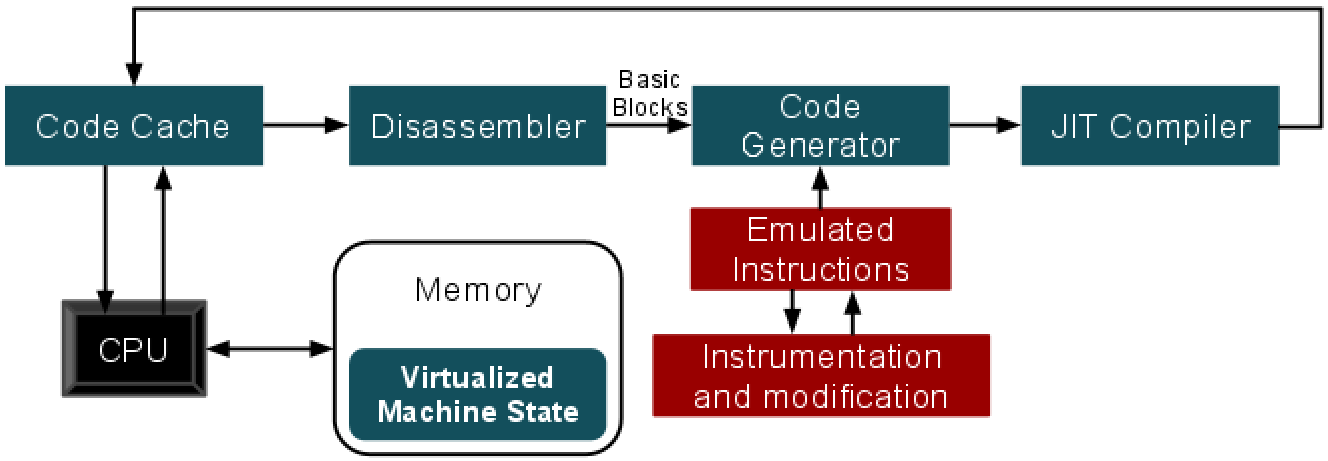

Figure 4.8 - The structure of the code rewriter with the addition of an instrumentation and modification block. The red blocks represent a simple instrumentation engine for introducing code modifications at an instruction level. Unaltered blocks are blue.

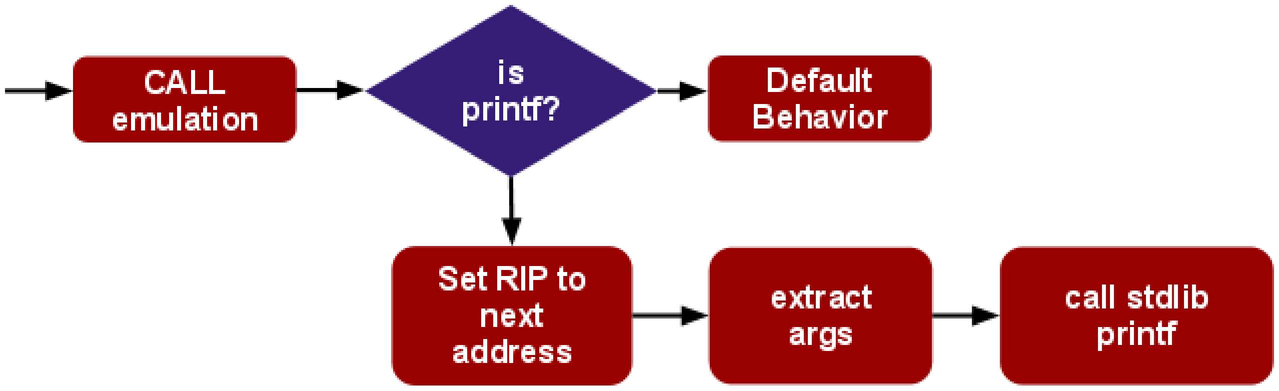

Figure 4.9 - This shows the basic execution flow of a code modification that replaces a dump printf function call in the original machine code with a call to the standard library's implementation. (The dump printf does not handle any conversion specifiers.) This is done by checking each call instruction if its destination address targets the original application's printf. If it is then the modification code performs the following steps: (1) set the PC register to the instruction following the call instruction (since the real call is skipped), (2) extracts the arguments so they can be passed on to the next step, and (3) call's stdlib's implementation of printf which is feature complete.

Chapter 5 – Results

5.1 Experiment 1 - JIT Optimization of Operands

Experiment 1 focuses on evaluating the machine code generation of the JIT compiler by comparing two ways to pass the operation's operands to the emulation function. The block consists of a callq instruction that uses a single operand representing an offset from this instruction's address. LLVM's JIT compiler was used to generate the X86 code which is then disassembled into human readable assembler using GDB. Each instruction is placed on its own numbered line. Figure 5.1 is the disassembled machine code when using structured operand arguments. Figure 5.2 is the disassembled machine code when using flattened operand arguments.

```
1   movabs $0x7ffff7e3d010,%rax
2   mov  (%rax),%rax
3   movq   $0x400255,0x40(%rax)
4   movl   $0x2,-0x60(%rsp)
5   movabs $0x617461642e,%rcx
6   mov  %rcx,-0x58(%rsp)
7   movq   $0xffffffffffffffcf,-0x50(%rsp)
8   mov  -0x48(%rsp),%rcx
9   mov  -0x40(%rsp),%rdx
10  mov  -0x38(%rsp),%rsi
11  mov  %rsi,-0x8(%rsp)
12  mov  %rdx,-0x10(%rsp)
13  mov  %rcx,-0x18(%rsp)
14  mov  -0x60(%rsp),%rcx
15  mov  -0x58(%rsp),%rdx
16  mov  %rdx,-0x28(%rsp)
17  mov  %rcx,-0x30(%rsp)
18  movq   $0xffffffffffffffcf,-0x20(%rsp)
19  mov  -0x30(%rsp),%ecx
20  cmp  $0x2,%ecx
21  je   0x7ffff7e4d1c8 <block_400250+328>
22  cmp  $0x1,%ecx
23  je   0x7ffff7e4d1ba <block_400250+314>
24  test   %ecx,%ecx
25  jne  0x7ffff7e4d1ac <block_400250+300>
26  xor  %edx,%edx
27  mov  -0x20(%rsp),%rsi
28  cmp  $0xffffffffffffffff,%rsi
29  mov  %rdx,%rdi
```

```
je   0x7ffff7e4d126 <block_400250+166>
mov  (%rax,%rsi,8),%rdi
mov  -0x10(%rsp),%rsi
cmp  $0xffffffffffffffff,%rsi
je   0x7ffff7e4d139 <block_400250+185>
mov  (%rax,%rsi,8),%rdx
imul   -0x18(%rsp),%rdx
add  %rdi,%rdx
add  -0x8(%rsp),%rdx
mov  0x118(%rax),%rsi
mov  (%rsi),%rsi
mov  $0x10,%edi
mov  $0xffffffffffffffff,%r8
mov  %rdi,%r9
inc  %r8
cmp  %rsi,%r8
jae  0x7ffff7e4d19c <block_400250+284>
lea  0x18(%r9),%rdi
mov  0x110(%rax),%r10
mov  -0x8(%r10,%r9,1),%r11
cmp  %rdx,%r11
ja   0x7ffff7e4d15d <block_400250+221>
cmp  %rdx,(%r10,%r9,1)
jb   0x7ffff7e4d15d <block_400250+221>
sub  %r11,%rdx
add  -0x10(%r10,%r9,1),%rdx
jmpq   0x7ffff7e4d1a9 <block_400250+297>
movl   $0xdead002b,0x0
xor  %edx,%edx
mov  (%rdx),%rdx
cmp  $0x2,%ecx
je   0x7ffff7e4d1cd <block_400250+333>
jmpq   0x7ffff7e4d1d4 <block_400250+340>
mov  -0x20(%rsp),%rdx
mov  (%rax,%rdx,8),%rdx
mov  %rdx,0x40(%rax)
retq
mov  -0x20(%rsp),%rdx
add  $0x400255,%rdx
mov  %rdx,0x40(%rax)
retq
```
Figure 5.1 - Experiment 1.1 - Structured Configuration

```
movabs $0x7ffff7e3d010,%rax
mov    (%rax),%rax
movq   $0x400255,0x40(%rax)
mov    0x38(%rax),%rcx
mov    0x40(%rax),%rdx
mov    %rdx,(%rcx)
addq   $0xffffffffffffffcf,0x40(%rax)
retq
```
Figure 5.2 - Experiment 1.2 - Flat Configuration

5.2 Experiment 2 - Instruction Operation Size

Experiment 2 compares the JIT's code generation by determining if specialization is a beneficial characteristic of instruction emulation. Specifically, this evaluates if separating add operations into their 8, 16, 32 and 64-bit sizes produces smaller code. Figures 5.3 and 5.4 represent 16 and 64-bit add operations that are generated from a single function that generalizes the emulation of the add instruction. Figures 5.5 and 5.6 represent 16 and 64-bit add operations that are generated from two separate implementations of an emulated add function.

```
movabs $0x7ffff7e3d010,%rax
mov    (%rax),%rax
movq   $0x400193,0x40(%rax)
mov    (%rax),%ecx
add    $0x10,%ecx
mov    %cx,(%rax)
retq
```

Figure 5.3 - Experiment 2.1 - Generalization Configuration (16-bit add)

```
movabs $0x7ffff7e3d010,%rax
mov    (%rax),%rax
movq   $0x40019e,0x40(%rax)
addq   $0x40,0x10(%rax)
retq
```

Figure 5.4 - Experiment 2.1 - Generalization Configuration (64-bit add)

```
movabs $0x7ffff7e3d010,%rax
mov    (%rax),%rax
movq   $0x400193,0x40(%rax)
movzwl (%rax),%ecx
add    $0x10,%ecx
mov    %cx,(%rax)
retq
```

Figure 5.5 - Experiment 2.2 - Specialization Configuration (16-bit add)

```
movabs $0x7ffff7e3d010,%rax
mov    (%rax),%rax
movq   $0x40019e,0x40(%rax)
addq   $0x40,0x10(%rax)
retq
```

Figure 5.6 - Experiment 2.2 - Specialization Configuration (64-bit add)

5.3 Experiment 3 - Flag Liveness Analysis

Experiment 3 lists the results of code generated when the block of code contains instructions that alter the flags register. These experiments indicate how much additional code is generated due to side effects related to setting and reading from the flags register. Figure 5.7 shows the result of the code block when flag side effects are not implemented in the emulated add instructions. Figure 5.8 shows the result of the code block when flag side effects are left to the JIT's optimizers to eliminate. Figure 5.9 shows the result of the code block when flag side effects are manually optimized when generating LLVM IR.

```
movabs $0x7ffff7e3d010,%rax
mov    (%rax),%rax
movq   $0x4001a6,0x40(%rax)
addq   $0x40,0x10(%rax)
addq   $0x40,0x10(%rax)
addq   $0xffffffffffffffec,0x40(%rax)
retq
```

Figure 5.7 - Experiment 3.1 - No Flags Set

```
movabs $0x7ffff7e3d010,%rax
mov    (%rax),%rax
movq   $0x4001a6,0x40(%rax)

mov    0x10(%rax),%rcx
add    $0x40,%rcx
mov    %rcx,0x10(%rax)
test   %rcx,%rcx
sete   %cl
movzbl %cl,%ecx
mov    %rcx,0x88(%rax)

mov    0x10(%rax),%rcx
add    $0x40,%rcx
mov    %rcx,0x10(%rax)
test   %rcx,%rcx
sete   %cl
movzbl %cl,%ecx
mov    %rcx,0x88(%rax)

addq   $0xffffffffffffffec,0x40(%rax)
retq
```

Figure 5.8 - Experiment 4.2 - 1 Flag Set

```
movabs $0x7ffff7e3d010,%rax
mov    (%rax),%rax
movq   $0x4001a6,0x40(%rax)

mov    0x10(%rax),%rcx
add    $0x40,%rcx
mov    %rcx,0x10(%rax)

mov    0x10(%rax),%rcx
add    $0x40,%rcx
mov    %rcx,0x10(%rax)
test   %rcx,%rcx
sete   %cl
movzbl %cl,%ecx
mov    %rcx,0x88(%rax)

addq   $0xffffffffffffffec,0x40(%rax)
retq
```
Figure 5.9 - Experiment 4.3 - 1 Flag Set with Explicit Liveness Analysis

5.4 Experiment 4 – Optimizations

Experiment 4 graphs the results of a variety of optimizations performed on a benchmark tool in order to improve the runtime performance. There are 3 areas where optimizations are analyzed. The first area is focused on applying LLVM IR optimizations, the second on JIT optimization settings and the third on manual optimizations performed when generating the LLVM IR. Additionally, the overhead of the entire system outside of the code cache and dispatch loop is eliminated to determine the theoretical speed of the benchmark in this system.

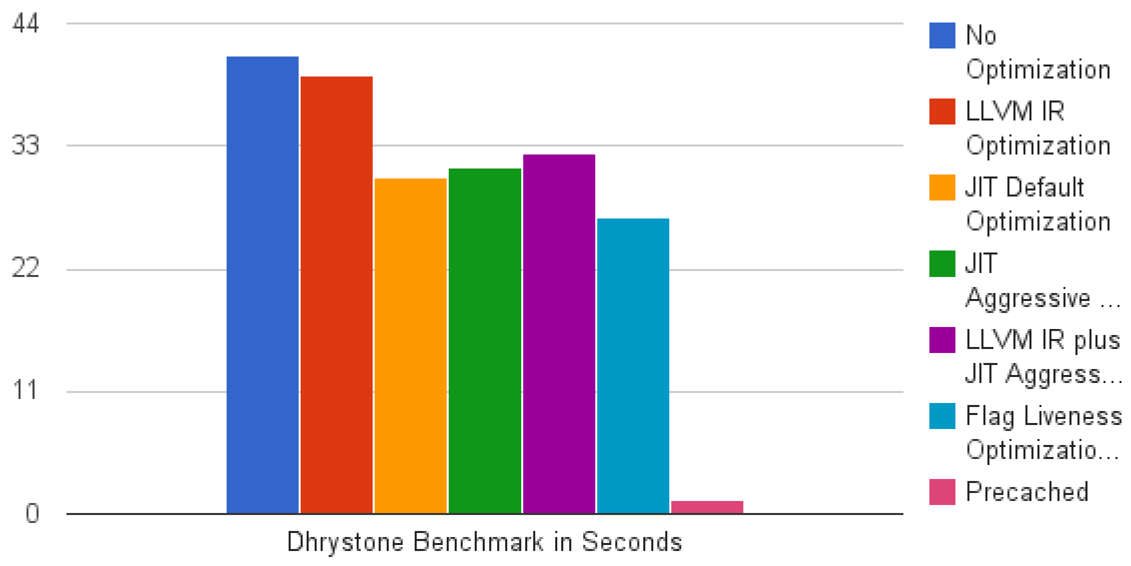

Figure 5.10 - Experiment 4 - Optimization results graph. This displays the results of running the Dhrystone benchmark on a variety of LLVM optimizations, JIT compiler optimization levels, manual optimizations, etc..

Name	**Dhrystone Time in Seconds**
No Optimization	41.207
LLVM IR Optimization	39.349
JIT Default Optimization	30.252
JIT Aggressive Optimization	30.994
LLVM IR plus JIT Aggressive Optimization	32.292
Flag Liveness Optimization + JIT Default	26.600
Precached	1.270
Native Execution	0.110

Figure 5.11 - Experiment 4 - Raw optimization results.

5.5 Experiment 5 - Instruction Level Instrumentation

Experiment 5 displays the runtime performance impact on the Dhrystone benchmark when applying arbitrary code modifications and instrumentation. The code modification replaces a custom printf function call with stdlib's version. The code instrumentation logs all call instructions to a file on the host's file system.

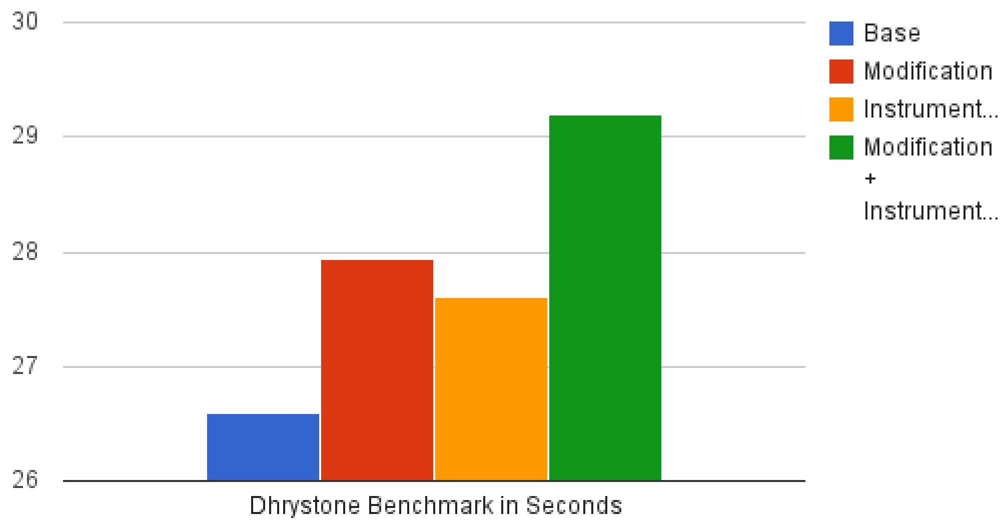

Figure 5.12 - Experiment 5 - Virtual machine performance comparison when using the example code modification and instrumentation from experiment 5.

Name	**Dhrystone Benchmark in Seconds**
Base	26.600
Modification	27.933
Instrumentation	27.611
Modification + Instrumentation	29.195

Figure 5.13 - Experiment 5 - Raw instrumentation and modification results.

Chapter 6 – Discussion

The main question of this paper is whether a virtual machine using a ready made JIT compiler is sufficiently powerful to optimize emulated machine code instructions. In order to evaluate this question we used a series of tests to evaluate how a general purpose JIT compiler, such as LLVM's JIT, generates code. This chapter will discuss the results and determine what changes are necessary to improve the generated machine code.

Before the results are analyzed a special note is required to explain certain aspects of the generated machine code. Every code block contains a few header and footer instructions. A block's header has two parts: (1) a global pointer to the virtual CPU machine state, instruction numbers 1 and 2, and (2) presetting the PC register to the next block's starting address (`0x400255` in the figure 6.1), instruction number 3. A block's footer only has one instruction which is just a return from the emulated block's function in the block's final instruction. Figure 6.1 shows the appropriate instructions for the header and footer.

```
1  movabs <global address>,%rax  <-- Virtual CPU machine state setup
2  mov    (%rax),%rax            <-- Virtual CPU machine state setup
3  movq   $0x400255,0x40(%rax)   <-- Set PC to the next block's address
...
n  retq
```

Figure 6.1 - Sample header (instructions 1-3) and footer (instruction n) that every emulated machine code block uses.

The first experiment is built upon the assumption that an emulated instruction function's parameters will be challenging for the JIT compiler to optimize. Experiment 1.1 uses a structured parameter for each possible operand used by the instruction. From a development perspective this is simple to maintain since any additions or deletions from the operand structure takes place in a single location. Figure 5.1 contains the JIT's generated machine code which is clearly far too complex for a simple call instruction. Despite initializing the structure with constant values and passing the argument as a constant, it is easy to see that little constant folding and no dead branch optimizations are used. Experiment 1.2 uses a series of single parameters to essentially flatten the structured configuration of experiment 1.1. While it provides an equivalent amount of data, using 8 arguments per operand makes an emulated instruction function's header extremely long and hard to read. However, the JIT compiler found this configuration much easier to optimize as it shrunk the entire block from 70 to only 8 instructions. Figure 5.2 lists the generated code. Only four instructions are directly related to the emulation of the call instruction: 4-6 are used to push the return address while 7 uses a relative offset to increment the PC register to the new function.

Experiment 2 seeks code generation improvements by creating an 8, 16, 32, and 64-bit version of an emulated instruction versus a single generic implementation. For simplicity this experiment only uses a 16 and a 64-bit bit add instruction and the flags register is left unmodified in order to simplify the generated code. (The modified flags are not influenced by the size of the operation.) Figures 5.3-5.6 list identical generated code when comparing the specialized versus generalized implementation for a given operation size. This means that any advantage that a size specialized implementation gains is made irrelevant by the effects of the optimizer. Clearly it is better to implement a single generic emulation of an instruction rather than multiple operation size based implementations.

Experiment 3 focuses on evaluating the machine code generation of the JIT compiler with regards to flag liveness in the virtual machine. Obviously the results from figure 5.7 are useless because they miss a key part of the instruction's behavior, that is the task of modifying the CPU's flags register. However, this serves as a base case to analyze how much additional code is generated to deal with just one of the many flags an instruction may modify. The base case uses just three instructions, plus the block's header and footer, for the original three instructions in the block. However, when just one flag, the zero flag, is introduced the generated code grows from 3 instructions to 15 instructions as seen in figure 5.8. This is obviously a dramatic increase in the instruction count and will undoubtedly have a dramatic impact on performance. Fortunately, if we perform a limited amount of preoptimization prior to generating the code, the number of wasted instructions dedicated to setting flags can be mitigated. Figure 5.9 shows that four instructions related to setting the zero flag have been removed from the first add instruction. Unfortunately, the basic add instruction has expanded from just one instruction in figure 5.7 to three instructions despite eliminating the zero flag calculation. The final add instruction still requires the instructions related to setting the zero flag. In total we have saved 4 instructions and dropped the total number of instructions to 11. It should be noted that a longer block may see even more significant savings by eliminating more writes that are never read from.

Experiment 4 focuses on analyzing how the system performs under Dhrystone, a synthetic benchmark. Experiment 4.1 uses no JIT optimization and no LLVM IR optimization other than inlining which is common to all. Unsurprisingly it is the worst performer of the group but establishes a base case from which to judge other results. Experiment 4.2 uses a series of function pass optimizers, listed in order in table 4.1, to optimize the generated LLVM IR code. While the performance is marginally better by 4.5%, they either add too much overhead in this example or they do very little to improve the final generated code. Experiment 4.3 shows a significant improvement over 4.1 and 4.2 by simply leaving all of the optimization to the JIT compiler. The result is 26.5% faster than the base case. Experiment 4.4 evaluates a more aggressive JIT. In this use case it is marginally slower than the default optimization. However, under other conditions, such as a real world application, the optimization could be more effective. Experiment 4.5 essentially evaluates how much of a positive effect is achieved by using the LLVM IR optimizations from 4.2. Unfortunately, it appears this actually hurts overall performance by increasing the runtime of 4.3 by 6.7% and 4.4 by 4.1%. Therefore, generically preoptimizing the generated code before it is run through the JIT compiler's optimizations is not ideal. However, specific optimizations, such as a function inline optimization pass, should not be ruled out simply because a broad and generic approach did not work. Finally, experiment 4.6 attempts to evaluate the benefits of preoptimizing instructions that modify specific flags in the flags register but whose modifications are never used by later instructions. As experiment 3 showed, relying on the JIT compiler to optimize away unused flag calculations is insufficient, but explicitly determining the liveness of the flag modifications can remove a significant portion of a code block's instruction count. The result shows a 12% reduction over the previously fastest time and is clearly a worthy source for improvement. It is unclear whether LLVM's optimization passes are capable of doing something like this on their own, but for now explicitly calculating a flag's liveness within a code block is an effective performance optimization. To evaluate the impact of the building up the code cache, the same benchmark was executed with the cache already filled. This was done by simply instructing the VM to restart the program without clearing the cache. This has a massive impact on the speed which clearly shows the bottleneck is generating the optimized blocks. The final value in figure 5.11 represents the same benchmark running natively on an X86 processor. Unfortunately, this value shows that this VM is currently several orders of magnitude slower than native performance and one order of magnitude slower than the precached version.

Ultimately these experiments show the capabilities of LLVM’s JIT compiler when optimization series of instruction emulation functions. While the optimizations are clearly improving performance, looking at some of the generated code indicates there are further possibilities for improvement. For example, looking at the results for experiment 3 show that the default optimizer is insufficient for determining when a calculation can be ignored. Instead, optimizations must be applied explicitly to reduce the amount of unnecessary machine code. Additionally, experiment 3 contains 2 identical add instructions but the optimizers were not able to recognize that the first result could be cached and is not required to be written back to the VM’s CPU register state. Undoubtedly many of optimizations are possible to improve this to a point where it is useable, but overcoming several orders of magnitude will require extensive work.

The secondary question of this paper is how to build a VM that is capable of introducing arbitrary instrumentation into any code executing on the system. In order to evaluate this question we provide two examples using the call instruction. The first example completely replaces all function calls to rudimentary printf with the standard library’s version. The second example shows a simple instrumentation that prints the destination address of all call instructions.

Modifying the executing software using a JIT compiler in the virtual machine turns out to be a trivial process if one has sufficient knowledge of the system. Consider the example printf modification. Since we know the address in the program where printf is called and we know the calling convention and number of arguments, it is easy to extract the arguments intended for the original call and supply them to another. The JIT takes care of the difficult work of gluing the new call into the old code and the code cache allows subsequent executions to keep the same behavior.

Instrumentation is just as easy as modification because they are effectively the same thing. However, instead of modifying the code with the intention of altering the expected behavior of the executing software, instrumentation simply analyzes and saves information regarding the runtime behavior. This information could then be used for later analysis by the administrator or it could be used to determine when a code modification should be activated. In this paper's instrumentation example it is used for the former case.

The overall impact of using both code modification and instrumentation on the Dhrystone benchmark is listed in figure 5.12. It is expected to find performance degradation, but it is surprising to see how much these minor changes influence the overall performance by nearly 10%. The printf modification may increase the overall execution time because it takes a rudimentary implementation that does not support expanding "%s" or any other conversion specifiers and replaces it with a full featured one that will naturally be slightly slower. The instrumentation experiment results show that 850088 call instructions are executed and it is expected that generating a log recording all the calls will impede the execution significantly. Despite a significant slowdown in runtime performance, the code modification and instrumentation performance is likely related to increasing the complexity of the code rather than any bottlenecks within the VM.

Chapter 7 – Conclusion

This paper has presented a solution to the lack of instrumentation in system level virtual machines. The core of the solution is one that involves taking a ready made disassembler and JIT compiler solution, LLVM, that is capable of optimizing the emulated instructions as well as inserting arbitrary code into the preexisting instruction stream. The solution was evaluated to determine if the optimization of the JIT compiler are able to decrease the execution time and cost of machine abstraction as well as to evaluate inserting instrumentation and modifications into the normal instruction stream.

Unfortunately the results obtained are far from overcoming the penalties of machine abstraction. If one assumes that the entire program is already compiled and placed into the code cache, then the cost of this VM is currently a 10x performance penalty. If the program must be compiled block by block, then the cost is a 100x performance penalty. Obviously neither of these results are desirable, but perhaps additional optimizations can eliminate a significant portion of the overhead. However, at this point it is clear that a general purpose JIT compiler like LLVM is either too slow to handle this specialized use case or it needs special optimizations for machine code emulation optimization that the general purpose optimization settings do not offer.

Machine code instrumentation is a promising outcome of these results. Assuming enough information is known about the system and the program currently executing, a developer of the instrumentation or modification can do anything to the running code. We showed this by completely replacing one printf function with another printf function. Likewise, low level instrumentation can be applied to the running code simply by adding hooks into the emulation code. If enough work is put into building up a framework for the instrumentation to go from the instruction level to higher source level access, then this could become a very powerful instrumentation utility.

While the overall performance results are disappointing, the hope of future optimization specific to this type of system could make the difference between an unusable system level VM and a competitively performing VM. Most of the related work researched for this paper relied on internal compilers to optimize the code rather than off the shelf tools like LLVM. The poor results found may indicate why readily available JIT compilers are insufficient for optimizing this type of code. However, the potential benefits of arbitrary instrumentation and code modification is attractive and potentially very useful.

7.1 Future Work

This type of system is likely to see its largest improvement through further research into JIT optimizations that target instruction emulation that is built into code blocks. When a code block is constructed it is made up of many individual instructions that are eventually replaced by emulations of those instructions. However, as the flags liveness experiment showed, a significant amount of inefficiency is introduced because calculations are made when there is no requirement for them. In other situations, a result does not need to be committed to a register in the virtualized CPU because it is used and modified by a subsequent instruction in the block. By preventing these unnecessary calculations or writes to memory, the emulated code generated for a blocks will approach the same complexity as the original block.

Many of the projects in the related works section offer other methods for improvement. One major cost reduction in Dynamo's optimized code generator is that traces eliminate the overhead of call instructions and other types of branches. This allows the optimizer to effectively inline subroutines which reduces the overhead costs of Dynamo's system. Dynamo provides a good example of a potential cost reduction by introducing traces in addition to simple code blocks as used in this project.

Many of the related projects start off like this one where each exit from a code block is returned to the dispatcher to either execute the next block from the cache or to compile the targeted code. If one connects code blocks that are already compiled and in the code cache then we could avoid going to the dispatcher. This is a potentially valuable opportunity for reducing the runtime cost when most of the code is in the code cache.

Interrupts and exceptions are problems common to any system that attempts to virtualize away the machine. Unfortunately, this problem was issue was never approached in this project and will need to be explored later. PinOS has an interesting solution to this problem and the lessons found there could also be applied to future iterations of this project. PinOS solves interrupts with a simple check at the beginning of each trace to query the virtualized interrupt controller to check if any have occurred and then handle them. All interrupts that do occur are queued in order to allow the currently executing code to be safely free from any instrumentation code.

Finally, a framework for building up instrumentation beyond the instruction level is needed. It is unlikely building up instrumentation from the instruction level will see popular usage. Therefore, a more advanced framework that perhaps resembles a typical source level debugger will be beneficial.

Chapter 8 – References

[1] A. Gal, Efficient Bytecode Verification and Compilation in a Virtual Machine Dissertation. PhD thesis, University Of California, Irvine, 2006.

[2] P.P. Bungale, C.-K. Luk, PinOS: a programmable framework for whole-system dynamic instrumentation. In VEE ’07: Proceedings of the 3rd International Conference on Virtual Execution Environments, pages 137–147, New York, NY, USA, 2007. ACM Press.

[3] F. Bellard, QEMU, a Fast and Portable Dynamic Translator. In Proc. 2005 USENIX Annual Technical Conference – FREENIX Track (2005), pp. 41–46.

[4] V. Bala, E. Duesterwald, and S. Banerjia, Dynamo: A transparent dynamic optimization system. In Proc. 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (June 2000), pp. 1–12.

[5] (Unattributed), Understanding Full Virtualization, Paravirtualization and Hardware Assist. Technical report, VMWare, Inc., 2007.

[6] L. van Doorn, Hardware virtualization trends. In Proceedingsof the 2nd international Conference on Virtual Execution Environments (Ottawa, Ontario, Canada, June 14 - 16, 2006). VEE ’06.ACM Press, New York, NY, 45-45.

[7] P. Dhawan T. Abels and B. Chandrasekaran, An Overview of Xen Virtualization. Dell Inc., Auguest 2005.

[8] C. Lattner, V. Adve, LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. pp.75, International Symposium on Code Generation and Optimization (CGO'04), 2004.

[9] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, Efficiently computing static single assignment form and the control dependence graph. Trans. Prog. Lang. and Sys., pages 13(4):451–490, October 1991.

[10] M.J. Anderson, M. Moe, C.I. Dalton, Towards trustworthy virtualisation environments: Xen library os security service infrastructure. Technical Report HPL-2007-69, Hewlett-Packard Laboratories, April 2007.

[11] Derek L. Bruening , Saman Amarasinghe, Efficient, transparent, and comprehensive runtime code manipulation, Massachusetts Institute of Technology, Cambridge, MA, 2004.

[12] Julian Seward, The design and implementation of Valgrind. http://valgrind.org/docs/manual/mc-tech-docs.html, March 2002.

[13] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox, Design of the Java HotSpotTM client compiler for Java 6. Technical report, 2007.

[14] K. Ishizaki, M. Kawahito, T. Yasue, M. Takeuchi, T. Ogasawara, T. Suganuma, T. Onodera, H. Komatsu, and T. Nakatani, Design, implementation, and evaluation of optimizations in a just-in-time compiler, in: Proceedings of the ACM 1999 Java Grande Conference, June 1999.

[15] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, Pin: Building customized program analysis tools with dynamic instrumentation. In Proceedings of PLDI 2005, pages 191–200, Chicago, Illinois, USA, June 2005.

[16] R. P. Weicker, “Dhrystone: A Synthetic Systems Programming Benchmark,” Communications of the ACM, vol. 27, pp. 1013-1030, 1984.