A Performance Evaluation of Operating System Emulators

by

Joshua H. Shaffer

A Thesis

Presented to the Faculty of
Bucknell University
In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science with Honors in Computer Science
10 May 2004

Approved By: ________________________________
Luiz Felipe Perrone
Thesis Adviser

____
Gary Haggard
Chair, Department of Computer Science

Contents

List of Figures

Abstract

The concept of simulating one computer architecture on a system with a completely different architecture has existed for nearly as long as computers themselves. This technique, which has come to be known as emulation, is quickly changing the landscape of the world of computing. Technologies such as the Java programming language have brought emulation into the main stream. If this trend is to continue, a better understanding of the factors that affect the performance of emulated systems must be developed. This information will provide the basis on which future systems will be built.

In order to understand the performance characteristics of various emulator implementations, this thesis presents a comparison of the interpretive and dynamic recompiling microprocessor emulation techniques. These emulation technologies are studied through the examination of two implementations of an emulator designed to run binary applications written for Microsoft's Windows operating system on Apple Computer's Mac OS X operating system. Using tests which focus on three main areas of emulator architecture, a detailed study of the factors which have the greatest effect on overall emulator performance is presented. The goal of this study is to identify the main bottlenecks in these current emulators, thus providing a focus for the improvement of future implementations.

Emulated systems are highly complex, and many of the components from which they are built rely heavily on the performance of other components. As a result, it is often possible to dramatically improve the overall performance of an emulator through the optimization of a single component. The data presented in this text clearly show that the performance of every aspect of the emulators being studied can be directly linked to the speed and efficiency of the microprocessor emulator.

By optimizing the emulation of the microprocessor, significant improvements in the performance of the entire emulator can be achieved. After identifying the aspects of microprocessor emulators which have the potential to be optimized, we examine the techniques that may be used to improve their performance. The methods discussed include code caching, register emulation, and the minimization of data loads and stores. The results of these examinations can be used to enhance the performance of new emulators, ensuring that emulation continues to have an impact on the future of the world of computing.

Chapter 1

Introduction

The development of computer systems progresses at such a rapid pace that designs considered current one year may be outdated by the next. As systems change, it is often necessary to run legacy software on newer computers. More importantly, it may also be desirable to run software designed for one type of modern computer system on another with a completely different architecture. In order to accomplish either of these goals, two major problems must be overcome.

First, the instructions within an application are only compatible with a single type of microprocessor. Applications are composed of sequences of binary instructions that are interpreted by a microprocessor, which then performs the requested actions. The code comprising each application binary must be targeted at a specific family of microprocessors. If the application is written in a portable high-level language, it is possible to create binary executable files for multiple microprocessor families, as long as the source code can be compiled and linked on each platform. This is often not the case, yet it may be desirable to run the software on a system with an incompatible processor.

The second problem is that software must be constructed around a specific set of Application Programming Interfaces (APIs) and libraries, which are defined by the underlying operating system. For example, software written for Microsoft's Win32 API will run only on Windows, while software written for Apple's Cocoa frameworks will run only on Mac OS X. An application which overcomes these obstacles would allow a single application binary to be executed on systems with different processors and APIs [1].

One approach that may be taken to solve these compatibility issues involves the creation of a virtual machine. A virtual machine is a program that produces an artificial environment which *appears to be* a complete physical computer system to applications running within it. Virtual machines are often used to allow multiple instances of an entire operating system to run concurrently on a single physical computer. This scenario assumes that the underlying system is based on the same physical processor for which this operating system was written. It is also possible to create a software implementation of the target processor, through which all of the instructions executed on the virtual machine are processed. This technique is known as *emulation*, and solves the first of the two problems by removing an application binary's reliance on the existence of the processor for which it was initially written. This is not a perfect solution, as there is a significant amount of overhead inherent in emulation. Furthermore, as systems become more complex this overhead continues to increase [13]. To address this problem, various approaches to processor emulation have been developed which reduce the performance penalty inherent in running emulated systems.

There are two possible solutions to the second problem, based on different ways of providing API and framework compatibility. The first method is to simply ensure that the operating system for which the software being run was designed is available on the host computer. Running this operating system within a virtual machine is a simple way to accomplish this task. The second solution is to provide custom implementations of all of the interfaces that the application expects to find. This solution requires more work at the API and framework level, but removes the requirement that the target operating system be available. These two solutions correspond directly to the two possible types of system emulators that will be examined.

1.1 Types of System Emulators

The traditional solution to emulation is to implement in software every major component of the target computer's hardware. This approach allows a copy of the target operating system to be run within the emulator, and thus for applications designed for that system to be run in their native environment. Because a full copy of the target operating system is run on the emulator, a high level of compatibility can generally be achieved. However, this compatibility comes at the expense of speed, which is limited by the requirement that many components which already exist on the host machine be emulated rather than used directly. For example, software emulation of hard disk controllers and graphics cards is necessary, adding significant overhead to the already complex task of emulating the target processor. Such emulators, which will be referred to as *full-system emulators*, are the most versatile, as they allow any operating system written for the target architecture to be run within the emulated environment. Such versatility does not come for free, as they sacrifice speed for compatibility and flexibility [16].

The higher level approach to emulation, which is the focus of this study, removes the necessity for redundant systems by mapping operating system API calls directly to the frameworks that exist on the host system. A virtual machine of this type still requires the emulation of the target microprocessor when it differs from the host's processor, but other systems need not be emulated. Although this approach may at first appear easier to implement than a full system emulator, the need for a detailed knowledge of the target API leads to significant complexity. If the compatibility and flexibility provided by full system emulators is not necessary, this approach, which from here on will be called *partial-system emulation*, has the potential to provide a much more efficient method for running the target system's binaries.

An example of a full-system emulator is Microsoft's Virtual PC. Virtual PC emulates a full Intel Pentium-based PC, thus allowing a variety of operating systems to be executed within an instance of the emulated environment. To accomplish this, Virtual PC emulates many complex devices, including PCI bus interfaces, an IDE controller, an Ethernet controller, a sound card and a video card. The overhead involved in the emulation of all of these devices is significant, and the fact that a complete copy of the guest operating system must be installed and executed further increases the computational load on the host system. Furthermore, once the emulated operating system is running, the level of integration with applications running on the host is limited. In the case of Virtual PC, this integration includes support for copy-and-paste and drag-and-drop operations between environments [16].

A contrasting example to Virtual PC is Champagne, a partial-system emulator currently being developed by the author of this thesis [12]. Champagne takes the higher level approach to emulation, and so emulates only selected components of the hardware implemented by Virtual PC. When Windows applications executing within Champagne request services from the operating system, fully optimized native code is invoked instead of the emulated Windows API code executed by Virtual PC. The user interface and other high-level functionality is provided by native PowerPC implementations of various Windows dynamic link libraries (DLLs).

Not only does this reduce the amount of code that must be executed by the emulated microprocessor, but it also allows emulated applications to better integrate with other programs designed for the host operating system. For example, a common clipboard is shared between native and emulated applications, and the user interface in emulated applications is presented using native widgets. Speed is also improved, as the only chip in the system that is emulated is the Pentium microprocessor, and only application-specific code is executed under this emulation. Although partial system emulators such as Champagne are currently less prevalent, they have the potential to surpass full system emulators like Virtual PC in both speed and functionality

Figure 1.1 illustrates the architectural differences between Champagne and Virtual PC. Although both run on top of Mac OS X on PowerPC-based Macintosh computers, they have little else in common. Virtual PC implements its emulation of all of the components in a full Pentium-based PC in native PowerPC code, directly on top of the operating system. Champagne, on the other hand, provides only a microprocessor emulation core, and a set of native implementations of many Windows DLLs. By providing operating system services for the emulated environment at this level, all of the code can be implemented as optimized native code. Above this, any code being executed is run under emulation, resulting in performance that is, in general, significantly slower than that of native code.

Another important point to note is that Virtual PC requires a full, licensed

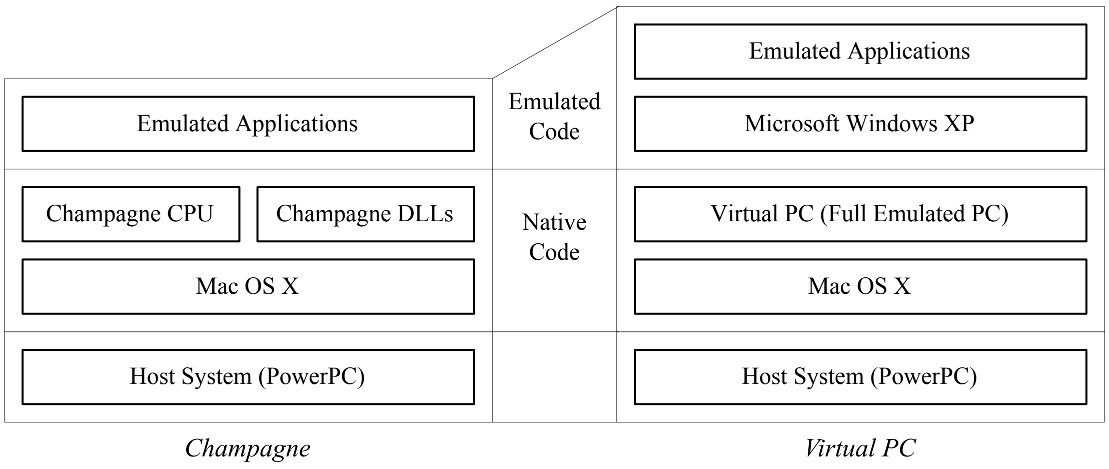


Figure 1.1: Comparison of the Virtual PC and Champagne architectures.

copy of Microsoft Windows to be run on top of its hardware emulation, on which the emulated applications are run. Champagne, on the other hand, runs emulated applications directly on top of the emulated microprocessor and the DLL implementation layer. This implies that the full functionality of the operating system can be provided by code native to the host system, significantly reducing the overhead involved in executing the emulated programs. The added overhead that Virtual PC incurs by emulating the complete operating system substantially reduces the potential performance of applications running in this environment.

A point which full- and partial-system emulators have in common is that they both must provide support for the execution of programs using an instruction set that is different from that supported on the host system. This support is provided by the emulation of the target system's microprocessor. Therefore, an efficient microprocessor emulation is critical to the performance of both full- and partial-system emulators. As the only emulated component of the target system in a partial-system emulator, its speed directly determines the overall performance of the emulator. For full-system emulators it is one of many emulated components, and as such achieving the highest possible performance may be even more critical. Without an efficient emulation of the target microprocessor, neither approach to building a complete emulator can provide acceptable performance. It is important to understand the alternative approaches to microprocessor emulation that exist so that the most appropriate one can be chosen for each particular system emulation project.

1.2 Types of Processor Emulators

Although the technology behind processor emulation has improved greatly since its inception, as we will discover, it remains a major bottleneck in overall emulator performance. Various types of processor emulators have been devised, from basic emulators written in high-level or assembly languages, to hand-optimized assembly language implementations of dynamic recompiling emulators. It is important to have a thorough understanding of the characteristics of each of these types of emulators, as the performance of an emulated system is tied directly to the performance of its microprocessor emulation.

The accuracy of a particular microprocessor emulator can be classified according to five different categories [15]. Although it may initially seem unusual to strive for anything but perfect accuracy, it may be acceptable to trade off accuracy for speed in certain circumstances. The need for fast emulation versus the need for high accuracy determines which class of emulator is best suited for a particular project.

Data-path Accuracy is the highest possible level of accuracy. Emulators which provide data-path accuracy model the underlying physical structure of the emulated processor, down to the data-path level. This type of emulator, often referred to as a simulator, provides an exact duplicate of the target processor. The low-level modeling of the microprocessor results in an emulator which runs very slowly. For this reason, data-path accuracy is most often seen in testing frameworks, and is generally unsuitable for emulators designed to run real programs for end users.

Cycle Accuracy ensures that instructions complete in the same amount of time relative to one another as they would on real hardware. Although cycle accuracy is more efficient than data-path accuracy, it is only necessary in systems running applications which depend on the relative execution speed of instructions to determine timing information. For this reason, it is most often seen in emulators for embedded systems, as well as in emulators for some old game consoles. Unlike data-path accuracy, cycle accuracy requires only that the Instruction Set Architecture (ISA) of the target processor be modeled, not the structure of the underlying processor. Microprocessor components such as the pipeline and functional units need not be emulated.

Instruction-level Accuracy involves the modeling of individual instructions, but without the strict timing constraints placed on each instruction's execution by cycle-accurate emulators. Emulators falling in this category are fairly common, as they can provide relatively high accuracy with generally acceptable speed. If an emulation of processor interrupts accurate to the individual instruction is needed, this is the least accurate method that can be used.

Basic-block Accuracy is accomplished by replacing individual blocks of code with native code producing the same effect. Emulators falling in this category offer extremely fast execution speed, but provide less timing accuracy for applications which rely on the performance characteristics of the actual hardware being emulated.

High-Level or Very-High-Level (HLE) Accuracy is nearly identical to basic-block accuracy, except that high-level emulators replace larger blocks of code. The goal of this type of emulator is to identify structures in the machine language code which correspond to known high-level functionality, such as memory copies. Once such a structure has been identified in the emulated instruction stream, it can be replaced with optimized native code.

There are multiple classes of microprocessor emulators, each of which may be able to achieve more than one level of accuracy. The simplest type of processor emulator is the basic *interpretive emulator*. This type of emulator fetches, decodes and executes instructions one at a time from the instruction stream, and is capable of providing accuracy at the data-path, cycle or instruction level. In order to provide such high levels of accuracy, this type of implementation involves a significant amount of overhead for individual instruction emulation. When written in a high-level language, interpretive emulators are often easily portable, with all emulated instructions and processor state handled in the high-level code. Although interpretive emulators can be written in assembly language to improve performance, there is little benefit to doing so. Modern compilers generate highly optimized machine code, the performance of which is often better than what can be achieved by a human. Even if it is possible to improve disassembly and emulation speed through hand optimization, the resulting emulator would still execute a very large number of instructions for each emulated instruction.

Most interpretive emulators rely on jump-tables based on instruction opcodes to optimize disassembly and emulation, with each entry containing the address of a function or block of code used to implement a different instruction. This approach to instruction disassembly is depicted in Figure 1.2. A mask is applied to

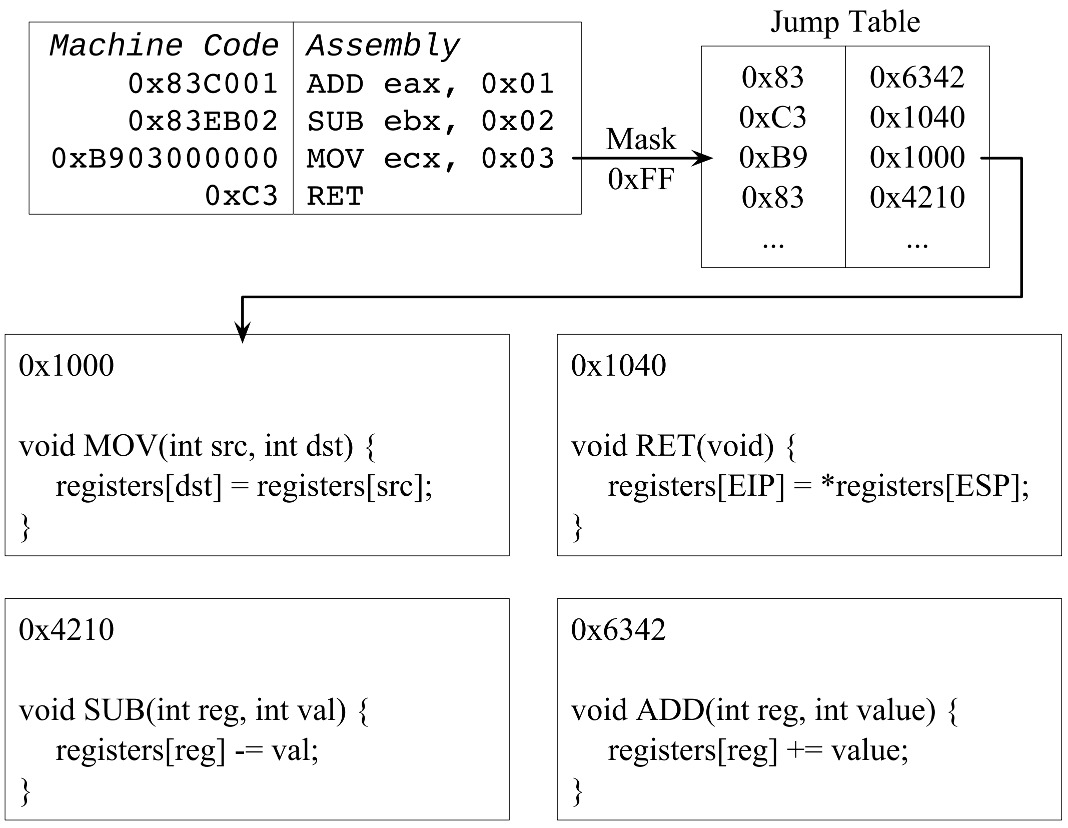


Figure 1.2: Using a jump table to optimize instruction disassembly.

an instruction's machine code to determine its opcode. The opcode is then looked up in the jump table, which is directly mapped to the address of a native function used to emulate instructions with the given opcode. In situations where opcodes overlap, for example the `ADD` and `SUB` opcodes in this figure, a second, smaller table may be needed in addition to the first. Although this technique can improve execution speed, interpretive emulators are inherently slow, and even optimized implementations will not provide high levels of performance. In situations where accuracy at the instruction level or lower is not as important as overall speed, interpretive emulators do not offer the best possible solution.

A significant improvement upon the interpretive emulator can be achieved with the use of a technique known as *dynamic recompilation*. Rather than emulating each instruction individually, a dynamic recompiling emulator examines a block of code to be emulated and produces a block of native code that performs equivalent tasks. In doing so, the emulator acts much like a compiler, and can manipulate the code it produces to provide benefits such as efficient instruction scheduling on the target microprocessor and minimized data loads and stores. Because it does not necessarily preserve the order of instructions as they appear in the emulated code, dynamic recompilation can provide only basic-block or HLE accuracy.

Dynamic recompilation, often referred to as *just-in-time compilation*, is how most modern Java Virtual Machines execute applications compiled to the Java byte code. Java virtual machines originally included interpretive cores, but this proved to be too slow when running large applications. As a result of the move to dynamic recompiling cores, the speed of Java applications increased by a factor of anywhere from 5 to 20 times, proving that the technique is highly efficient [7].

Even though basic dynamic recompiling emulators can produce very efficient code, further performance improvements can be obtained by caching the translated blocks for future use. A depiction of this process can be seen in Figures 1.3 and 1.4, which compare the execution of a repetitive function on an interpretive

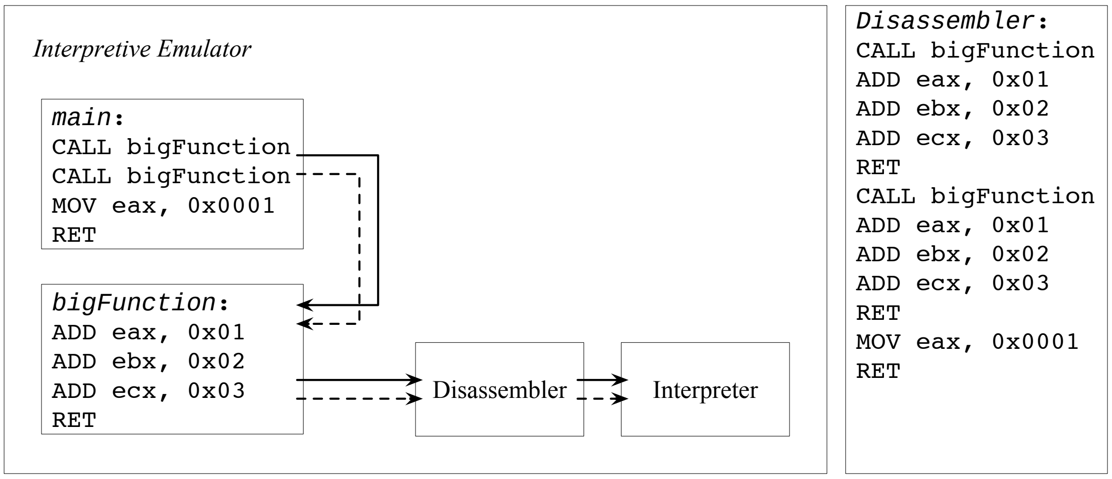


Figure 1.3: A trace of the execution of code on an interpretive emulator.

emulator to its execution on a dynamic recompiling emulator with code caching. The boxes at the right side of the figures show the instruction stream as processed by the disassembler for both implementations. The groupings in the listing for the dynamic recompiling emulator indicate blocks which are translated as a single unit.

The interpretive emulator, shown in Figure 1.3, first executes a call to `bigFunction()`, at which point it disassembles and interprets each instruction in the function. The process is then repeated, with every instruction in `bigFunction()` being disassembled and interpreted a second time. Even though the function is called two times in a row, all of the work done on the first pass is repeated the second time through.

Figure 1.4 represents the work of the dynamic recompiling emulator. On the first call to `bigFunction`, it is disassembled and translated to native code, which

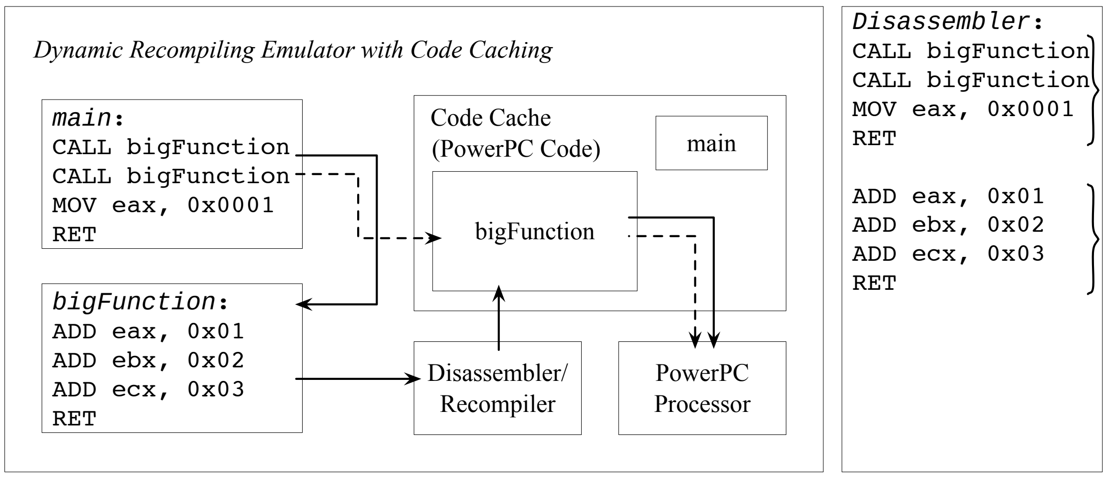


Figure 1.4: A trace of the execution of code on a dynamic recompiling emulator.

is stored in the code cache. This native code is then executed directly on the host processor. On the second call to `bigFunction` the emulator finds the function's code in the cache and proceeds to execute it directly. This removes the second translation of the code which was incurred by the interpretive emulator. Unless the cache replacement algorithm removes `bigFunction()` from the cache, all subsequent calls to it proceed directly to the native code in the cache.

By saving translated blocks in the cache, the overhead incurred during translation can be amortized across multiple executions of the same block of code. This allows complex algorithmic analysis of the native code block to be performed at translation time, resulting in a further increase in execution speed. Multiple levels of optimization may also be implemented, such that the more often a block of code is executed, the more optimization it receives. This ensures that excessive time is not spent optimizing code which is only rarely used, while emphasis is placed on the optimization of highly repetitive sections of an application.

Depending on the target and host systems, a dynamic recompiling emulator may also be able to map registers from the emulated microprocessor to registers on the host processor. Such an approach may only be possible if the host has sufficient registers to contain all those of the emulated processor. This technique is particularly relevant in this study, as the Pentium, with its eight general purpose registers (GPRs), can be emulated in this fashion on a PowerPC, which has thirty-two GPRs.

Although the techniques in use in modern emulators have evolved from approaches used in earlier systems, progress was only possible because the previous generation of emulators were well understood. In order to facilitate the development of new emulation technologies, we must study the approaches taken by the current generation. To this end, this thesis presents a detailed evaluation and comparison of the performance of two versions of Champagne, one with an interpretive emulation core and the other with a dynamic recompiling core.

We now present an overview of the structure of this thesis. Chapter 2 describes the motivation for this study, and the goals that we set out to achieve. Chapter 3 explains the design of the tests which comprise our performance evaluation framework. The model used in the design of the performance tests is outlined, and issues relating to the implementation of this model are discussed. An examination of how well it portrays the performance of the three system components being tested is also provided. The specific implementations of the tests are presented in Chapter 4, with a brief description of what each is designed to accomplish. Having described each test, Chapter 5 presents the results of the tests, along with an analysis of the final outcomes. Finally, the results of the entire study are presented in Chapter 6, along with a vision of the impact that emulation may have on the future of the world of computing.

Chapter 2

Motivation

Evaluating the performance of an emulator is a complex task, as the speed of an individual component of the system may have a large impact on the performance of all other components. Although real systems are also affected by this issue, the effect is exaggerated in emulated systems where the host's microprocessor is responsible for providing the functionality of the entire guest platform. For example, it must provide all of the services normally handled by hardware such as graphics cards and network interfaces. Interpreting the results of a performance analysis requires a thorough understanding of the architecture of the emulator being studied. Without this understanding, it is impossible to identify the components which have the greatest affect on the overall performance of the emulator.

Many issues specific to emulated systems are not taken into consideration during the development of benchmarks for normal systems. As a result, existing tests may provide useless results when run in an emulated environment. Depending on the level of accuracy achieved by the emulator being studied, many of the benchmarks generally used to measure system performance, including clock speed, cycles per instruction and clock cycle time, have little or no meaning in an emulated system. For example, in an emulator providing only basic-block accuracy there is no concept of an emulated clock, let alone a clock cycle. As a result, all three of these measurements are useless in characterizing its performance. Clearly, the evaluation of emulators must be approached differently than that of real computer systems [9].

Although current benchmarks are not designed to accurately portray the performance of emulators, this is not an indication that such performance is not worth studying. Over the last decade, the study of emulation technology has led to the development of new emulation techniques. These techniques have increased the utility of emulators, allowing more powerful systems to be emulated with greater performance than was previously possible. This has also led to an increase in the usefulness and efficiency of emulators, thus helping to solidify their position as valid computational tools. For example, when Java virtual machines began to make the transition from interpretive to dynamic recompiling cores, the result was an expansion in the usability of the Java platform. Just-in-time compilation, as the technique came to be known, turned what had traditionally been considered inefficient into a widely used and accepted technology.

Although much research has already been done in the area of just-in-time compilation during the development of the Java Virtual Machine (JVM) concept, it is only indirectly related to emulation in general. The JVM is entirely stack-based, and so has no registers to map to physical registers at execution time. This allows individual implementations of the virtual machine to deal with byte code execution in a way that is best suited to the underlying physical hardware. The entire Java platform, including its byte code, was designed to be run on physical systems that do not support the byte code directly. The study of emulation in general must take into consideration the fact that the machine code being emulated was designed to be run directly on a physical processor. The processors that the code being emulated were written for are almost always more complex than is the JVM, and so emulation of these systems is more complex as well [17].

By applying some of the same principles that Java Virtual Machines use to execute Java byte code to the emulation of binaries targeted at a real processor, the walls which have traditionally separated different hardware platforms can be torn down. This has already occurred in other areas, such as computer networks. Thanks to modern networking technology and infrastructure, systems with drastically different architectures and operating systems are able to communicate and share information. The next step, which could ultimately help to unify the world of computing, is to allow applications written for any platform to be run anywhere. As a specific example, the dynamic recompiling processor core in Champagne, coupled with its native implementations of specific operating system-level functions, could achieve this goal for Windows applications on Mac OS X.

Clearly, the study of emulator performance has the potential to effect broad change throughout the computing world. More important than the ability to simply run software on multiple platforms is the desire to integrate emulated software within the host's environment. By improving the usability and responsiveness of emulators, and by better integrating them within host systems, it is possible to make them nearly invisible to the user.

As a historical example, Apple Computer made use of an emulator for Motorola's 68000 series of processors during their transition to the PowerPC architecture in the mid 1990s. The initial implementation was based on an interpretive emulation core, which was rewritten as a dynamic recompiling 68LC040 emulator shortly after its introduction, resulting in a dramatic increase in performance. Designed to transparently execute legacy code, including significant portions of the operating system itself, the system gave the end user no indication as to whether the computer was executing PowerPC or 68000 code. The task of coordinating the context switches, called *mode switches* in this system, was handled transparently by the Mixed Mode Manager, which performed all of the tasks necessary to switch between the two architectures [2, 4].

With a history such as this, emulation has clearly changed the course of computing, and it holds the promise of continuing to do so in the future. Bridging the gap between different architectures and operating systems would allow software development resources which have traditionally been dedicated to the support of multiple platforms to be reallocated, potentially resulting in an increase in the quality of available software. Furthermore, using emulation to remove the distinction between different operating systems and architectures could bring about a revolution in modern computing. Although emulation has not traditionally been considered of major importance to mainstream computing, it has the potential to change the way in which systems are designed and used.

If this vision of the future is to be realized, a better understanding of the emulation techniques currently in use is necessary. With this, more advanced systems can then be developed. By characterizing current emulation techniques, the performance measurements obtained in this study will be used to determine the factors which have the largest impact on overall system performance. The result of this determination will be a list of techniques which can be further examined to improve the performance of future emulators. By identifying and optimizing the aspects of an emulator's implementation that are key to its performance, significant progress can be made in the design of highly efficient emulators.

Chapter 3

Design of the Evaluation Framework

The creation of a framework for measuring the performance of multiple emulators is a significant challenge. There are many factors which are specific to each individual emulator being tested that must be taken into consideration. Because of the wide array of techniques used in the design of emulators, certain aspects of the framework developed herein may need to be modified for use on alternate systems.

One of the most important tasks in the creation of any performance evaluation framework is the development of a method for producing precise timing measurements. Modern processors often include support for estimating various performance metrics in real time, and so developers of benchmarking applications can make use of hardware in the production of their results. However, achieving accurate results in an emulated environment presents a unique challenge, as the hardware performance counters that are available on the real microprocessors do not exist in emulators. Implementing the performance counters would require the emulation of the processor at a much lower level than is possible if usable performance is to be achieved. Without hardware support, then, the measurements must be made using software-based techniques. Because such approaches depend on each individual emulator's underlying implementation, running a single test binary on two different emulators can potentially result in inconsistent results and large errors.

Figure 3.1 demonstrates how a single function may have drastically different implementations on various types of emulators. When compiling a test application in Metrowerks CodeWarrior targeting Windows, the Metrowerks Standard Library compiles the C function `clock()` to a call to the Win32 API function `GetTickCount()` [8]. If this binary is then executed under Microsoft's Virtual PC, the actual Windows implementation of this function will be used. Because the source code to the the Windows implementation is not publicly accessible, it is considered to be *closed-source*. The way in which this function arrives at the current time is hidden within the closed-source library Kernel32.dll. It likely queries a clock in a real PC, in which case it is up to Virtual PC to determine how best to emulate this hardware clock. This leads to the problem that real-time hardware

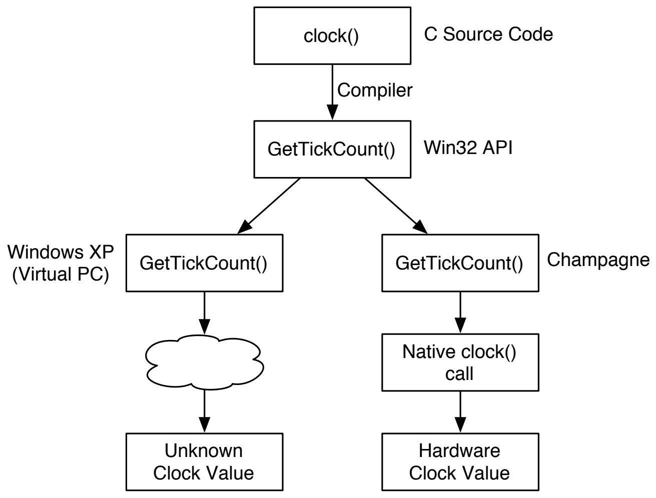


Figure 3.1: Comparison of the implementation of `clock()` in Virtual PC and Champagne.

---

clocks are extremely difficult to emulate, as they can potentially reduce the performance of other parts of the emulator to unacceptable levels. For this reason, they may be updated less frequently than would a real hardware clock, resulting in incorrect timing information. Champagne implements the same function as a single native call to the `clock()` function, ensuring that the value returned will be as accurate as the system can provide. Because this call will query a true hardware clock for its value, the result is guaranteed to be identical to the timing information that a native application would receive. Other emulators may have implementations which differ from both Champagne and Virtual PC. If each implementation returns a value that can not be directly related to that returned by the others, the results will be useless for comparison purposes.

The scope of this evaluation was limited by the difficulty in determining the way in which functions such as `GetTickCount()` are implemented in closed-source emulators. In addition to the comparisons made between the various implementations of Champagne, test results for Microsoft's Virtual PC were also to be included. However, achieving accurate timing under this emulator proved to be extremely difficult. Due to the lack of access to the source code of the software involved, it is difficult to determine where the emulator obtains the timing information that it provides to applications running on top of it. What is clear is that the information that it provides is flawed, as the timing values returned were generally so erroneous as to be useless. Simply timing a test run of an application using a stopwatch showed the run times returned by the emulator to be impossibly low. The resulting times consistently appeared to exceed the performance of optimized native code, a situation which would certainly not occur. A significant effort was made to explore alternative methods for timing algorithms running under Virtual PC, but regrettably no suitable solution was found.

3.1 Methods for Measuring Performance

Two approaches to performance measurement were evaluated for use in this study, one of which is time-based and the other repetition-based. The repetition-based method was chosen because it provided more accurate results than the time-based method. Both approaches were designed to work with actual run-times, defined as the time elapsed from the start of a test to its end, rather than processor time, which would be a measurement only of the time that the host processor spent executing code. This is the most useful of the two metrics, as it can be directly correlated to the real-world usability of an emulator [9]. All timing methods suffered from the fact that high resolution timers may be inaccurate in certain emulators, and are likely to provide inconsistent results across different emulator implementations. For this reason, it is best to run tests for extended periods of time to ensure that measurement errors will not make an appreciable difference in the results.

In the time-based approach, which was not used, performance was measured by executing an algorithm for a pre-determined length of time. After this interval. the amount of work that the emulator accomplished was to be measured. This method benefits from the fact that the runtime of the tests can be regulated, ensuring that tests do not continue for unacceptable lengths of time. Although this appears to be a useful method, it is actually difficult to implement because simply timing the loop changes its performance characteristics. For example, an operating system call is needed to determine the current time after each iteration of the loop, causing the emulator to delay execution of the test to obtain the current time. This delay can be accounted for, but it makes performance measurement unnecessarily complex, and so we opted for an alternate method.

The repetition-based approach, which was ultimately the one used for all of the performance measurements in this study, obtains its results by executing an algorithm with a predetermined input for a fixed number of iterations. By measuring the time taken to repeat the given number of iterations, accurate comparisons can be made between different emulators. This method has the advantage that only the actual application code being tested will be run while timing is occurring, thus preventing unnecessary delays caused by the performance measuring system. A drawback resulting from this method is that it is difficult to limit the execution times of the tests to reasonably short periods.

The length of the execution of a test must be long enough to be captured by the granularity of the clock. For example, on systems providing accurate clock values to the hundredth of a second, run-times must be significantly longer than this. In order to ensure that fast emulators run for a measurable period of time, the number of repetitions of the test must be high. Unfortunately, this may result in slow emulators taking a very long time to complete the same test. By carefully choosing repetition counts which provide measurable run-times on a fast emulator without causing a slower one to run for an unnecessarily long time, accurate results can be achieved in a reasonable time.

3.2 Testing of Specific Emulated Components

This evaluation focuses on three components of computer architecture that are particularly relevant when examining emulator performance: the microprocessor emulation, the memory system and the file system. Although obtaining microprocessor performance measurements is fairly straightforward, doing so for other components of the emulator, especially the main memory and secondary storage, introduces a few complications.

Many standard benchmarks are available to quantify the performance of physical microprocessors and computer systems. These existing benchmarks do not meet the needs of this framework, which is designed to study emulated systems. One important consideration in the design of the performance tests was the fact that the microprocessor emulators on which the tests were run were incomplete, implementing only a subset of the Intel x86 instruction set. For this reason, many of the applications used in standard benchmark suites would not have run on the emulators being tested. More importantly, the goal of this framework is to determine the components of an emulated system which have the greatest effect on overall performance. To accomplish this, it was necessary to design tests that were biased in some way, such as toward register-based or memory-based computations. Contrary to this approach, standardized benchmark suites focus on providing a broad view of the overall performance of a system. These are fundamentally different goals, and providing a more focused study could only be accomplished with the design of a new set of benchmarks.

As an example, the widely-used benchmarks available from the Standard Performance Evaluation Corporation (SPEC) would not have tested the desired aspects of the emulated systems. All of the benchmark comprising this suite are geared toward determining the real-world performance that can be expected from a particular architecture. While this is certainly a large part of the study of emulator performance, for the purposes of this study it is more useful to obtain results describing individual components of the system. Furthermore, benchmarks such as SPEC CPU2000 are not freely available, and so are less desirable for use in an academic study [3].

For the purpose of measuring processor performance, this study consists of both tests composed of compact loops of a few instructions and tests using real-world algorithms. This mix is designed to provide a broad view of microprocessor performance which can be applied to various situations. Because of the significantly higher performance of the dynamic recompiling emulator relative to the interpretive one, each test is timed over a large number of repetitions, thus providing a more accurate evaluation of the performance of both implementations. The combination of the microprocessor tests provides an understanding of the performance characteristics of both interpretive and dynamic recompiling emulators in various situations.

Measuring the performance of the memory and storage components of a system proves to be a challenge that is uniquely difficult under emulation. When the performance of actual memory and storage devices is measured, individual components can be tested in the same physical system, ensuring a relatively unbiased comparison. In emulated systems, components can not be swapped in and out, and so all are dependent on the performance of the rest of the emulator. The main goal in the design of the memory and storage tests was originally to minimize the effect of the rest of the emulator on the results. Upon further consideration it became apparent that isolating these components from the other parts of the emulator would not provide useful statistics. For this reason, the tests do not attempt to isolate the memory and disk components. Instead, the tests are geared toward analyzing the effects that each has on the performance of different types of emulators.

In our experience, almost all applications copy memory using similar algorithms. Such memory copying algorithms are heavily dependent on the microprocessor, as most 32-bit systems accomplish memory copies with a loop of four-byte word copies. In an emulated system, determining the speed of main memory becomes difficult when the performance of the processor emulation overshadows it. The fact that all real applications utilize similar algorithms, however, makes this unnecessary, as the results provide an accurate reflection of real-world performance.

The performance of memory is also affected by another factor. Memory management in an application is often handled by allocating a large buffer at startup, and utilizing pieces of this block as needed during execution. Applications which utilize this type of custom memory allocation routine, such as those using the `malloc()` function in the C standard library, are heavily dependent on application-level code for memory management. Alternatively, applications which rely solely on operating system calls to manage memory will generate many more system calls, but are less dependent on application-level code. Consequently, the performance of applications which allocate and deallocate large amounts of memory is dependent on how well the emulator on which they are running supports the type of memory management they use.

The actual tests which are used in the performance measurements were designed with all of these facts in mind. Because they do not attempt to isolate any particular components, the results can not be guaranteed to demonstrate the performance of the component that was originally intended to be tested. This turns out to be beneficial, as it allows the results to demonstrate more accurately the performance that can be expected from the emulators in real-world situations.

Chapter 4

Tests for Selected System Components

Tests focusing on the microprocessor, the memory and the disk are presented for use in the evaluation of the emulators. All of these tests were designed with the interpretive and dynamic recompiling versions of Champagne in mind. They are tailored to their specific implementations, with the goal of providing accurate results for both systems without running for an unreasonably long period of time.

4.1 Descriptions of the Microprocessor Tests

Three tests are used to analyze microprocessor performance. The versions of Champagne being tested do not contain emulated floating point units, as the most important portions of the microprocessor had been implemented first. Therefore, the tests are limited to integer calculations, as care had to be taken during their development to ensure that they run within the emulators that they were executed on.

Two basic tests are included for the purpose of demonstrating the performance of the emulators when executing small looping portions of code. The first is an integer square root calculation algorithm (Figure 4.1). It first sets the most significant bit to a 1 and squares the resulting value. If the square of the value exceeds the input, it clears the bit. This is repeated for each remaining bit, resulting in the largest integer which, when squared, does not exceed the input value. The majority of the execution time is spent within this loop, which compiles to the code listed in Figure 4.2. The two local variables are designated as “register” variables to ensure that their values are kept within processor registers, thus reducing memory accesses. Although these instructions do not represent a broad cross section of the instruction set, they will serve to demonstrate the performance of the emulators in a tight loop.

The second basic test is a root finding algorithm which uses bisection to locate an integer root. By iteratively halving the domain over which the root is being searched for, and discarding the half in which a root does not lie, the algorithm can provide a close approximation to a root of a function. Because it will be run solely using integer arithmetic, it will only search for integer roots for the function. This

```c
unsigned long SquareRoot( unsigned long value )
{
    register unsigned long        curBit = 0x8000;
    register unsigned long        root = 0;

    do
    {
        root ^= curBit;

        if ( root * root > value )
            root ^= curBit;
    }
    while( curBit >>= 1 );

    return root;
}
```

Figure 4.1: C source code for the Square Root Function used in the microprocessor test

algorithm serves a purpose similar to that of the square root algorithm, except that it is slightly more complex and uses different instructions, and so gives a different view of the performance of the microprocessor emulation.

In order to provide a better analysis of the real-world performance characteristics of each type of emulator, the final test is an implementation of the Blowfish encryption algorithm [10]. Designed by Bruce Schneier and used in many applications, Blowfish is a block cipher algorithm utilizing 64-bit blocks, with a variable key length of anywhere from 32 to 448 bits. The algorithm is not patented and is freely available as source code or as a reference implementation [11]. This per-

```
Address             Machine Code     Intel x86 Disassembly

0x0150108E          31d0             xor eax, edx
0x01501090          89c3             mov ebx, eax
0x01501092          0fafd8           imul ebx, eax
0x01501095          3b5d08           cmp [ebp + 0x08], ebx
0x01501098          7602             jbe 0x0150109C
0x0150109A          31d0             xor eax, edx
0x0150109C          d1ea             shr edx, 1
0x0150109E          83fa00           cmp edx, 0x00
0x015010A1          75eb             jne 0x0150108E
```

Figure 4.2: Assembly code for the Square Root Function used in the microprocessor test

formance test will consist of one loop to encrypt a block of text, and a second to decrypt it. Encryption is a common, often used service, and so the performance of the emulated microprocessor can be tested and analyzed under simulated conditions similar to those seen in the real-world.

4.2 Descriptions of the Memory Tests

Two tests will be used to measure the performance of memory operations under the emulated systems. The first test measures memory allocation and deallocation performance by repeatedly allocating and releasing a block of memory. This test will be run with a block size of 1 byte to test small allocations, and a block size of 1 megabyte to test large allocations. The test is written in standard C, and so the memory is allocated using the `malloc()` function and released with the `free()` function. The performance of this test is therefore highly dependent on the particulars of the implementations of these two functions. The speed of the `malloc()` function is determined almost entirely by the speed of the processor emulation.

The breakdown of this test into two separate tests utilizing different block sizes ensures that the two main behaviors of `malloc()` are tested. At application launch, a block of memory is allocated as a buffer, which is distributed as small allocation requests are made. The 1 byte test serves to evaluate this behavior. The 1 megabyte test provides an understanding of application performance during the alternate behavior, in which allocation requests are passed to the operating system. Because 1 megabyte exceeds the block size that the algorithm can provide from pre-allocated storage, the memory must be requested directly from the system.

The second memory test measures the speed of a copy operation on a large block of data. Two blocks of equal size are allocated, after which the uninitialized contents of one block is copied into the other block. The memory copy is performed with a single call to the `memcpy()` function, and so is completely dependent on the library's implementation. The main part of the function, as compiled from the Metrowerks Standard Library, is a single `REP:MOVSD` instruction [8]. This instruction copies an arbitrarily large amount of data between two locations in memory in four byte blocks. Therefore, an efficient implementation of this one instruction will result in an emulator with optimal memory copy speeds.

4.3 Description of the File System Test

We measure the file system performance using a single test: the duplication of a 64 megabyte file consisting of a repeating pattern of digits. To copy the file, a fixed-size buffer is allocated into which the source file is read in chunks, and from which the data is written out to the destination file. It is difficult to accurately measure file system performance in an emulator, as the speed of the file copy will always be dependent on other systems, especially the microprocessor. If the copy buffer is not sufficiently large, the speed of the copy operation will be dominated by the speed of the microprocessor emulation, because many more loops will be necessary to complete the copy. However, if the buffer is too large, the runtime may be lengthened by swapping in the virtual memory system. For this reason, a one megabyte buffer is used for this test. This limits the loop count to 64 repetitions, but also ensures that memory will not be swapped out during the copy operation.

Chapter 5

Test Results

When taken together, the tests provide an understanding of the components of an emulator that have the greatest effect on performance. As a result, all of the analyses presented with the test results are geared toward explaining the effect that the particular component being tested had on the overall performance of the emulator. As the microprocessor emulation is the most complex component, it is presented first, followed by the memory and file system analyses.

All quoted performance numbers are the mean of the results of three runs of each test. In general three samples would not provide acceptable accuracy, but the run times of the tests were so close to each other that three will be sufficient. The narrow 95% confidence intervals computed for all of the test results demonstrate that this is the case. Results for each test are listed in Appendix A. The intervals were calculated with $t_{n-1,\alpha/2} = t_{2,.025} = 4.30$ using the following formula [14]:

$$\bar{x} - t_{n-1,\alpha/2}\frac{S}{\sqrt{n}} \leq \mu \leq \bar{x} + t_{n-1,\alpha/2}\frac{S}{\sqrt{n}}$$

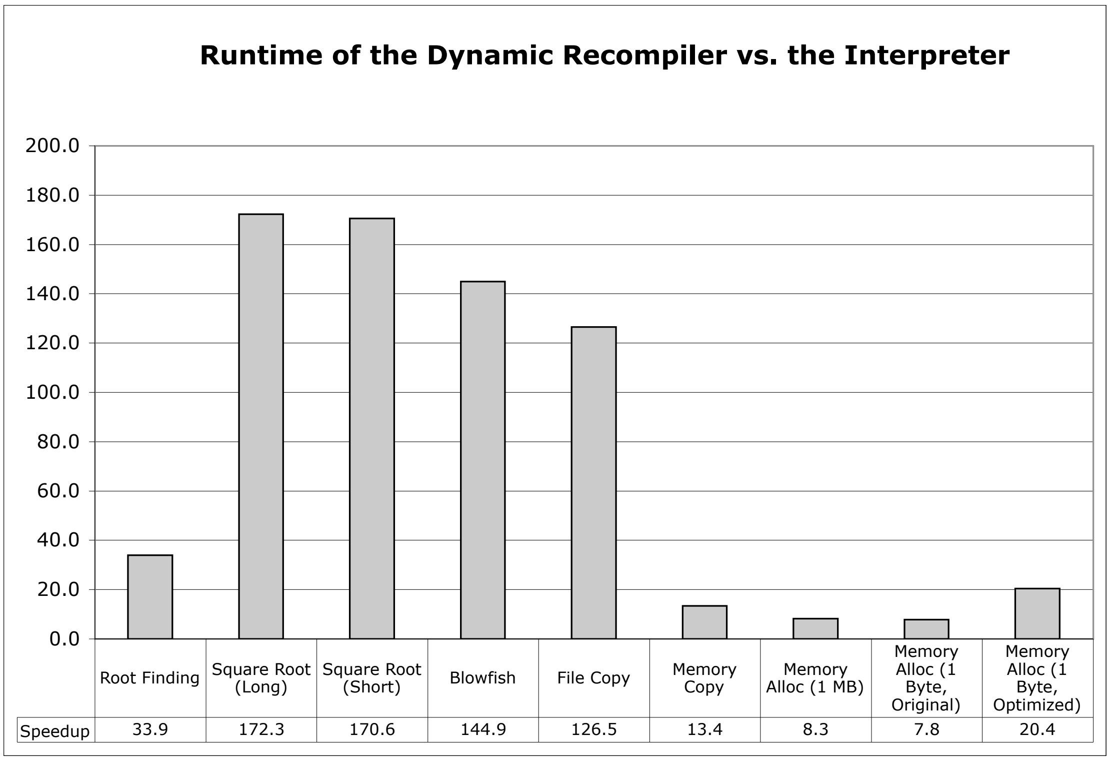
All of the performance comparisons made between the dynamic recompiling emulator and interpretive emulator use speedup as the metric. In this case, speedup represents the ratio of the run time of a test on the interpretive emulator versus its run time on the dynamic recompiling emulator.

5.1 Microprocessor

As would be expected, the results for all of the microprocessor tests show the dynamic recompiling emulator performing significantly better than the interpretive emulator. Figure 5.1 shows that the dynamic recompiling emulator often performs more than 100 times faster than the interpretive emulator.

5.1.1 Square Root

Although the data from the two runs of this test show only a very small difference, the gap in the performance of the two types of emulators widens as the number of code repetitions increases. While the interpretive emulator must disassemble and execute every instruction in the loop during each pass, the dynamic recompiling emulator needs only to disassemble the instructions in the loop once. After the first pass, the equivalent PowerPC code can be run on each subsequent execution. Demonstrating this is difficult due to the low resolution of the timing mechanisms



	Root Finding	Square Root (Long)	Square Root (Short)	Blowfish	File Copy	Memory Copy	Memory Alloc (1 MB)	Memory Alloc (1 Byte, Original)	Memory Alloc (1 Byte, Optimized)
Speedup	33.9	172.3	170.6	144.9	126.5	13.4	8.3	7.8	20.4

Figure 5.1: Ratio of the execution speeds of the interpretive and dynamic recompiling emulators for all tests.

used. Accurate to only one hundredth of a second, it is nearly impossible to measure the time that the dynamic recompiling emulator takes to execute small numbers of repetitions of any of the tests.

In order to demonstrate this, the square root test was run twice, once for approximately one million repetitions, and again for approximately seventeen million repetitions. These values were chosen to ensure that the dynamic recompiling emulator would run for a measurable time, while keeping the run time for the interpretive emulator at a reasonably low level. For the shorter test run, the dynamic recompiling emulator ran approximately 171 time faster than the interpretive emulator. For the longer test, this increased to 172 times. Although this increase is very small, it indicates that as the repetition count increased, the dynamic recompiling emulator became more efficient. It is difficult to demonstrate the efficiency gains in this manner, because a single repetition of any test runs too fast to be accurately timed. The 95% confidence intervals (see Appendix A) for the square root tests overlap, so these results are not conclusive. A better example will be seen in the root finding tests.

5.1.2 Root Finding (Bisection)

The results of the test for 13,568 repetitions indicate that the dynamic recompiling emulator ran approximately 34 times faster than the interpretive emulator. Because of the small numbers of repetitions used, the time intervals being measured are extremely short. This proves to be an issue for the dynamic recompiling emulator, where the interval is only about four tenths of a second. For this small number of repetitions, the initial investment in time for the recompilation of the code was not amortized across sufficient executions of the loop. Although the deviation from the results of the other microprocessor tests was significant, the anomalous results do not represent an error in measurement. The 95% confidence interval puts the true result within a range of values that does not match up with the results of the square root test.

When the algorithm was timed over 1 million repetitions, the speedup increased to 364, a level that is much more consistent with the square root tests. This provides a clear example of the benefits seen by the dynamic recompiling emulator over a large number of loop iterations. By the end of the longer test, the cost of recompilation had been amortized over a very large number of repetitions, resulting in the dynamic recompiling emulator becoming faster relative to the interpretive emulator.

5.1.3 Blowfish

Performance for the Blowfish test showed the dynamic recompiling emulator to have a slightly smaller benefit over the interpretive emulator than did the square root test. This is easily explained by the fact that each test has a different mix of instructions. The square root test, shown in Figure 4.1, is quite short and makes use of only six x86 instructions. The Blowfish test is more complex, with the `Blowfish_encipher()` algorithm consisting of fourteen different instructions, and as such provides a better overview of the performance to be expected in real-world situations. The assembly code for this algorithm is available in Appendix B.2.

This test also demonstrates that the operand type of each instruction has a significant affect on the performance of the emulated code. Intel's IA-32 architecture defines a highly complex addressing model, with many different addressing modes.

Almost every instruction defined in the architecture can have an operand which references memory. In the PowerPC architecture only load, store and branch instructions can reference memory, and only six addressing modes are defined for these instructions. Of these modes, three are specific to load and store instructions and three to branch instructions. When IA-32 instructions are emulated on a PowerPC processor, the effective addresses generated by the emulated addressing modes must be calculated using native instructions. A single emulated instruction may require up to five PowerPC instructions just to determine the address of a memory reference. For this reason, code which performs the majority of its computation within the eight available x86 registers will perform significantly better under emulation than will code which accesses memory heavily [5, 6].

Furthermore, an x86 instruction with operands consisting solely of registers can often be translated into a single PowerPC instruction. Instructions that require memory accesses are often translated into more than five PowerPC instructions, thus degrading the performance. Because of the small number of registers that the IA-32 architecture defines, most real-world applications include significantly more memory accesses when compiled for the Intel architecture than they would if they were compiled to PowerPC code.

The composition of the square root test's instructions is heavily biased toward register rather than memory accesses, as shown in Figure 5.2a. For the `Blowfish_encipher()` function, however, more than 95% of the instructions re-

Instruction	Count
cmp	2
jcc	2
xor	2
imul	1
mov	1
shr	1

Access Type	Count	Percentage
Memory	1	11.11%
Register	8	88.89%
Total	9	

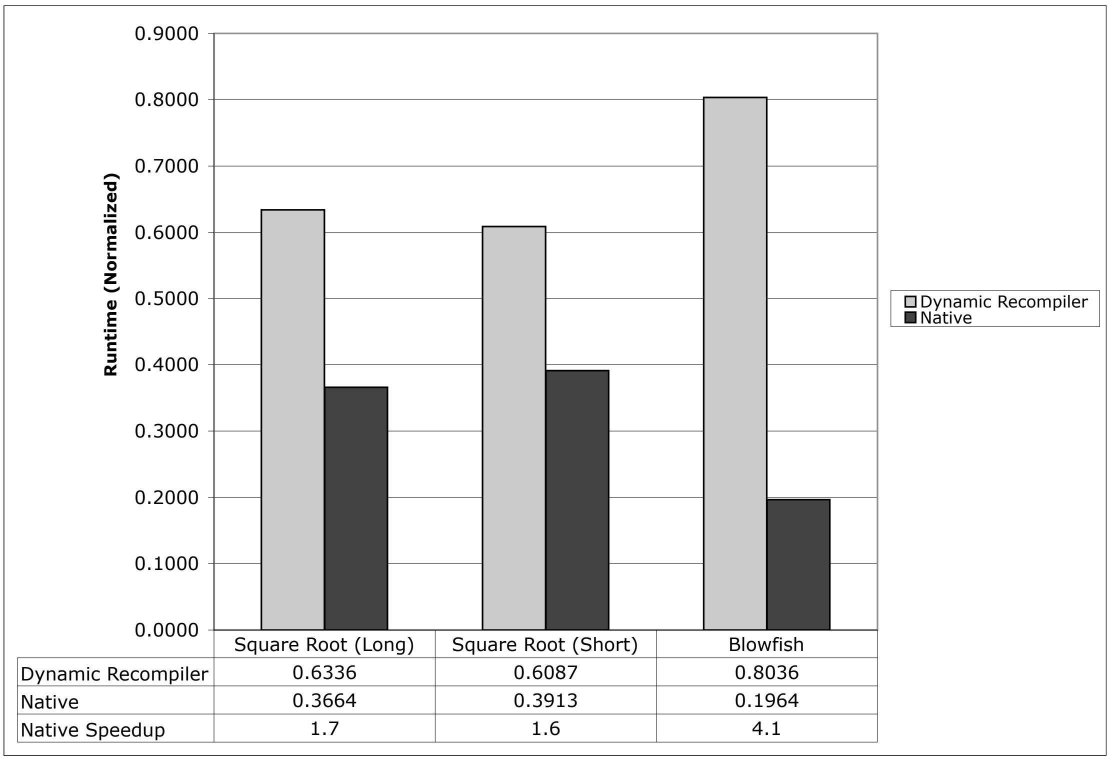
a) Instruction breakdown for the square root function.

Instruction	Count
mov	34
pop	5
push	5
stosd	4
xor	4
lea	2
call	1
cmp	1
inc	1
jcc	1
jmp	1
movsx	1
ret	1
sub	1

Access Type	Count	Percentage
Memory	59	95.16%
Register	3	4.84%
Total	62	

b) Instruction breakdown for the Blowfish_encipher() *function.*

Figure 5.2: Square root and Blowfish instruction breakdown and composition.



	Square Root (Long)	Square Root (Short)	Blowfish
Dynamic Recompiler	0.6336	0.6087	0.8036
Native	0.3664	0.3913	0.1964
Native Speedup	1.7	1.6	4.1

Figure 5.3: Performance of the dynamic recompiling emulator versus native code for the square root and Blowfish tests.

sult in one or more memory accesses (Figure 5.2b). When the performance of these tests on the dynamic recompiling emulator is compared with their native performance, as seen in Figure 5.3, it becomes clear that memory accesses come with a significant performance penalty. The native version of the square root tests ran less than twice as fast as the emulated version, demonstrating the respectable performance of the emulated code. The native version of the Blowfish algorithm ran more than four times faster than the emulator. This significantly better speedup

Original

```
xor eax, [ebp - 0x0C]
mov [ebp - 0x0C], eax
mov eax, [ebp - 0x08]
mov [ebp - 0x10], eax
mov eax, [ebp - 0x0C]
mov [ebp - 0x08], eax
mov eax, [ebp - 0x10]
mov [ebp - 0x0C], eax
```

Optimized

```
mov rTemp, [ebp - 0x0C]
xor eax, rTemp
mov rTemp, eax
mov eax, [ebp - 0x08]
mov [ebp - 0x10], eax
mov eax, rTemp
mov [ebp - 0x08], eax
mov eax, [ebp - 0x10]
mov [ebp - 0x0C], eax
```

Figure 5.4: Optimization of a portion of the Blowfish_encipher() function.

for the Blowfish test test is due almost entirely to the fact that it performs all of its computation in memory, which penalizes the dynamic recompiling emulator.

These results lead naturally to the conclusion that it is beneficial for the dynamic recompiling emulator to optimize memory accesses in recompiled code. By converting repeated references to a single address into register accesses, and by loading or storing the value to memory only once, performance can be improved significantly. The example in Figure 5.4 shows a portion of the `Blowfish_encipher()` function where four accesses to the effective address `[ebp - 0x0C]` have been reduced to two using this technique. Although the optimized x86 code contains more instructions than the original, when translated to PowerPC code the resulting block is more compact due to the reduced memory access count. With thirty-two general purpose registers available on PowerPC processors, only some of which are used as emulated x86 registers, such an approach is quite feasible, and can potentially lead to significant speed improvements.

A full listing of the C source code for the Blowfish encryption algorithm is available in Appendix B.1. The code used for this test is derived from a C implementation developed by Bruce Schneier which is available from the official Blowfish website [11]. The timing of the test includes one call each to `BlowfishStartup()` and `InitializeBlowfish()`, followed by many calls to `Blowfish_encipher()` and `Blowfish_deciper()` to repeatedly encrypt and decrypt a string of text. The majority of the test time is therefore spent within the encipher and decipher algorithms, which are nearly identical. Therefore, focusing this analysis on the `Blowfish_encipher()` function is reasonable, as it provides a good approximation of the overall behavior of the test.

5.2 Memory

The results of the memory tests are not nearly as biased toward the dynamic recompiling emulator as are the tests of microprocessor performance. There are a variety of possible explanations for the slower relative execution of the memory tests, but much of the reason is the simple fact that memory transfer speeds are significantly slower than processor speeds. Because of this, even code which is not well optimized, such as that generated by the dynamic recompiling emulator, can execute at the same speed as highly tuned code. The interpretive emulator continues to perform poorly in these tests, as the overhead involved in its processor emulation is significantly larger than the memory latency.

5.2.1 Memory Allocation and Deallocation

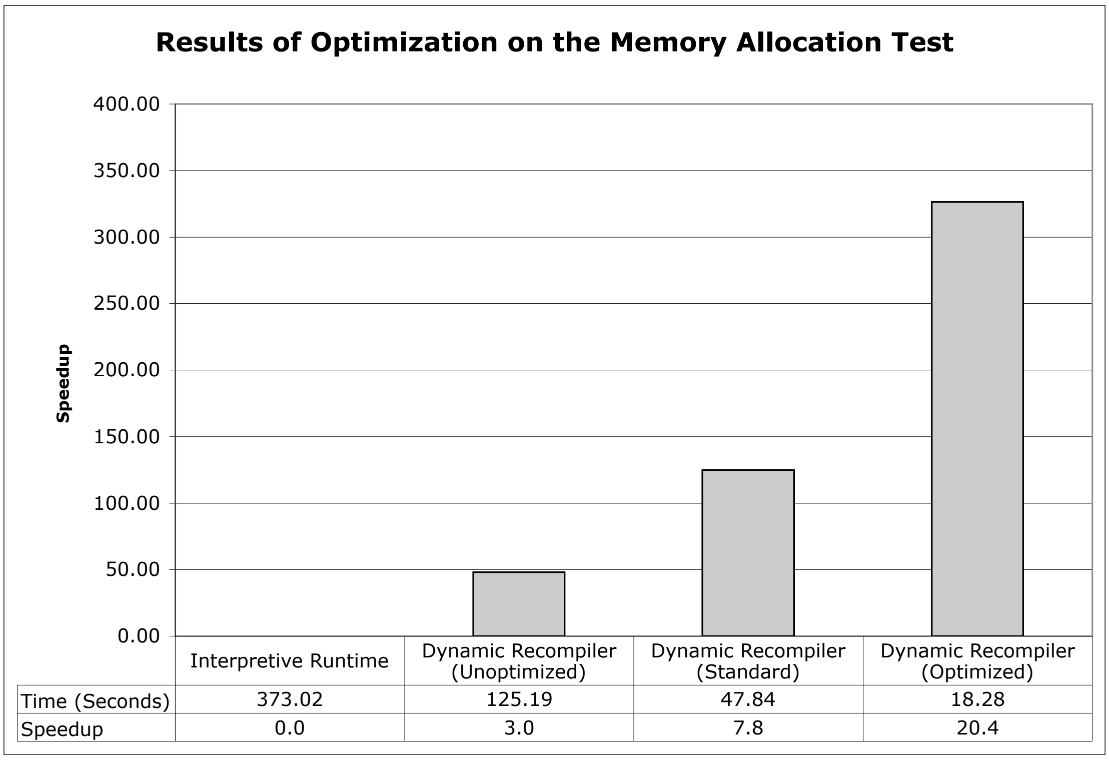
Of all of the tests, the memory allocation and deallocation test had the smallest performance gap between the interpretive and dynamic recompiling emulators. In order to determine the cause of this discrepancy, an understanding of the behavior of the code is needed. For the 1 byte allocation test, tracing its execution shows that no calls are made to the operating system to request that memory be allocated. Each call to `malloc()` does invoke two calls to functions located within Kernel32.dll, one each to `EnterCriticalSection()` and `LeaveCriticalSection()`. For the 1 megabyte allocation test, each repetition results in a call to `GlobalAlloc()` and `GlobalFree()`, in addition to the two critical section calls. In the current implementation the critical section functions should have nearly zero overhead, and the memory allocation and deallocation calls should be similarly inexpensive. Nevertheless, the tests which performed significantly better on the dynamic recompiling emulator do not make any calls to functions located in external DLLs. Although all of the execution time should theoretically be spent within the emulated code, this turns out not to be the case. This discrepancy merits a closer examination of these calls.

Calls to functions residing in DLLs are handled differently by the interpretive and dynamic recompiling emulators. The two implementations store the emulated processor's state in different ways, and so can not handle calls between native and emulated code in the same manner. The interpretive emulator maintains the x86 processor's *state*, that is, the values in its eight registers, in corresponding variables in memory. Access to the emulated registers is then accomplished through memory reads and writes. Although there is a significant performance cost due to the slower speed of memory relative to processor-based registers, this method ensures that the guest processor's state is completely independent of the host's state. The dynamic recompiling emulator, on the other hand, maps the guest processor's registers to real registers on the host microprocessor. Although this allows for much faster access to the registers, it creates a separate context which must be saved and restored each time the emulated code is exited or entered. This context switch is expensive, as it requires the reading and writing of all of the emulated registers, as well as the condition registers. This overhead is, unfortunately, in many cases unavoidable.

The dynamic recompiling emulator generates two context switches for each call to a function imported from a DLL. If these context switches were the cause of the bad performance of the memory allocation tests, then it would be reasonable to assume that more DLL calls would equate to worse performance. However, the 1 byte test, with only two DLL calls per allocation, performs *worse* than the 1 megabyte test, with four DLL calls per allocation. The cause of the performance penalty actually turns out to be a different source of context switches, which by chance only occurred in the memory allocation tests.

During dynamic recompilation, the target address of instructions which change the flow of execution can not always be determined at compile time. For this reason, it is often necessary to compile x86 instructions such as `JMP` and `CALL` such that they invoke a native function to look up the target, recompile it, and jump to its starting address. This will be the case any time that the instructions at the target address are not already available in recompiled form. If the emulator follows this model, there will be two context switches for every execution of this type of instruction, which will certainly have an adverse affect on performance.

Profiling the dynamic recompiling emulator's execution using a performance monitoring tool demonstrated that the code that switches between the emulated x86 and native PowerPC contexts accounted for a majority of the execution time. This result is consistent with the performance of Champagne with various levels of optimization, as shown in Figure 5.5. The standard configuration is the one used for all of the performance measurements of the dynamic recompiling emulator. The unoptimized configuration requres a context switch during the execution of x86 conditional jump instructions (`Jcc`), which is not needed in the standard configuration. Finally, the fully optimized version also removes the need for a context switch during the execution of an unconditional jump instruction (`JMP`). Simply removing these two sources of context switches decreases the runtime by



	Interpretive Runtime	Dynamic Recompiler (Unoptimized)	Dynamic Recompiler (Standard)	Dynamic Recompiler (Optimized)
Time (Seconds)	373.02	125.19	47.84	18.28
Speedup	0.0	3.0	7.8	20.4

Figure 5.5: Results for the memory allocation test on four implementations of Champagne.

more than 85%, and increases the speedup of the dynamic recompiling emulator over the interpretive emulator from approximately 3.0 to 20.4. Although this is still lower than the speedup obtained for the microprocessor tests, it is a significant improvement. The remaining discrepancy is likely the result of the two critical section calls; the context switches for those calls are unavoidable with the current implementation.

5.2.2 Memory Copy

For efficient code, the speed of the memory copy test is highly dependent on the speed of the physical memory in the system. In this test, the dynamic recompiling emulator saw a speedup of approximately 13 over the interpretive emulator. This is the second smallest speedup observed between the two emulators, only to be topped by the memory allocation tests. In order for this to be the case, the dynamic recompiling emulator must spend a significant amount of time waiting for the memory system to catch up with the processor.

The code which performs the memory copy is extremely simple, with almost all of the work done in a single `REP:MOVSD` instruction. This instruction copies the four byte double word referenced by the address stored in the `ESI` register to the memory location referenced by the address in the `EDI` register. After doing so, it either increments or decrements the addresses being referenced, and decrements a repetition counter. The repeated `MOVSD` is a fairly complex instruction, and its implementation in the dynamic recompiling emulator is not optimized. Even in its non-optimized state, the speed that it achieves is nearly identical to the speed that can be achieved by optimized native code.

It can be inferred from this result that, above a certain threshold, the level of optimization of the recompiled code is irrelevant. For portions of an application which access memory heavily, the processor will waste a significant number of cycles waiting for data from memory. Because of this, even poorly optimized code will not have a negative impact on the overall performance of those segments of the program.

Although this conclusion is not specific to emulated code, it is an important observation due to the difficulty of optimizing recompiled code in real time. The time spent on recompilation must be amortized across multiple execution of the emulated code in order for the emulator to be efficient. If a portion of a program is not used very often, or if its execution speed will not affect the overall performance of the emulated application, less time can be spent on optimization. An advanced dynamic recompiling emulator could include the ability to perform simple profiling of the code being emulated. Using this information, it could perform optimizations appropriate for the type of code being emulated. For instance, for as long as memory speeds remain much lower than processor speeds, optimizing segments of code dominated by instructions such as `MOVSD` would provide little to no benefit. By opting to leave such code in an unoptimized state, the delay before the execution of the first iteration of the loop can be reduced without decreasing the overall performance of the emulated application.

5.3 File Copy

The performance of the file copy test is bounded by the performance of the processor emulation, as can be seen in the results in Figure 5.1. The speedup of the dynamic recompiling emulator relative to the interpretive emulator is in line with that of the square root and Blowfish microprocessor tests. The slightly smaller speedup in this test can be explained by the fact that the file copy requires exactly 64 invocations of the Win32 API function `ReadFile()`. Calls to this function, located in the library Kernel32.dll, require two context switches in the dynamic recompiling emulator. This is another indication of the effect that context switches have on the overall performance of the emulator.

The physical disk should theoretically be the bottleneck in this test. The runtime of the test on the dynamic recompiling emulator is significantly greater than the runtime of the test compiled to native code, indicating that this is not the case. Further optimization of the dynamic recompiling emulator to reduce the number of context switches, or to decrease the amount of time that they take, could potentially improve the performance of this test.

Chapter 6

Conclusions and Future Work

Emulators for computer systems have existed for almost as long as computers themselves. The technology behind emulation, however, has only recently begun to mature to the point where modern systems are able to emulate other modern systems with acceptable performance. If the techniques used in system emulation are to be further improved, a thorough understanding of their behavior will prove to be invaluable. Although the results of this study are specific to partial system emulators, an evaluation of full system emulators should produce similar overall results for the performance of microprocessor-intensive tasks.

This study presents a close comparison of the performance of two types of microprocessor emulators with the intent of developing a better understanding of their performance characteristics. In doing so, the goal is to lay the groundwork for the creation of a new generation of emulators that will be even more capable than the current generation. To this end, the performance of interpretive and dynamic recompiling emulators is compared, and the components that have the greatest impact on their overall performance are identified.

It is clear from the results of this study that the speed of the microprocessor emulation has a significant impact on the performance of every aspect of the emulator. In particular, the use of code caching, and the requirement that context switches be performed, have a dramatic affect on overall performance. By ensuring that commonly used functions are never translated more than once, an efficient code caching implementation can increase performance significantly. Even if the time required for recompilation is large, it can be amortized across many executions of the translated block. Coupled with a dynamic recompiling core that is capable of producing optimized code for the host architecture, this technique has the potential to increase the performance of emulated applications nearly to the level of native code. Further performance enhancements can be achieved by implementing techniques to reduce memory accesses. For example, by temporarily storing data from commonly referenced locations in memory in unused registers on the PowerPC microprocessor, memory accesses can be reduced.

The context switches introduced by dynamic recompilation, on the other hand, tend to have an adverse affect on the performance of such emulators. These context switches may be required either to invoke native code to perform a service for the emulated application, or as a result of the target of a jump or call instruction not being found in the code cache. Their occurrence as a result of such cache misses results in the recompilation of a new block of code, a process that requires a significant amount of time. It is this type of context switch which is most useful to avoid, as it incurs a significant performance penalty. Although repetitive context switches could potentially reduce the performance of dynamic recompiling emulators to the level of interpretive emulators, careful optimization of the recompiling microprocessor core can minimize the number of switches that must be performed, resulting in increased performance.

All of the performance measurements obtained in this study are targeted exclusively at integer-based computations. This limitation was imposed by the fact that the microprocessor emulators being tested did not yet emulate other portions of the x86 instruction set. The author of this study intends to extend the evaluation framework to include the measurement of Floating Point and Single Instruction, Multiple Data (FPU and SIMD) units in the future.

The performance of applications executing under emulation is beginning to reach an acceptable level. By further increasing the performance of emulated systems, it will become impossible for the user to determine if an application is running natively or on top of an emulation layer. Although not yet widely implemented, emulators that are integrated within operating systems hold the promise to broaden the library of applications available for all platforms. By removing the distinction between hardware platforms and operating systems, such emulators may eventually unify the world of computing. The result of such a unification would be the ability for applications to be executed on any system, thus increasing the quality and quantity of software available on all platforms. It is this possibility that holds the potential to fundamentally alter the landscape of the computing environment.

Appendix A

Tables of Results

Square Root

16,777,216 Repetitions	**Mean**	**Standard Deviation**	**95% Confidence Interval**
Interpretive Runtime (s)	2302.97	10.41	[2,277.13, 2,328.80]
Dynamic Recompiler (s)	13.37	0.01	[13.34, 13.40]
Native (s)	7.73	0.07	[7.56, 7.90]
Speedup (Int/DR)	172	0.7217	[171, 174]

1,048,576 Repetitions	**Mean**	**Standard Deviation**	**95% Confidence Interval**
Interpretive Runtime (s)	143.89	0.20	[143.41, 144.37]
Dynamic Recompiler (s)	0.84	0.01	[0.83, 0.86]
Native (s)	0.54	0.01	[0.53, 0.56]
Speedup (Int/DR)	171	1.3636	[167, 174]

Root Finding

13,568 Repetitions	**Mean**	**Standard Deviation**	**95% Confidence Interval**
Interpretive Runtime (s)	14.80	0.07	[14.64, 14.97]
Dynamic Recompiler (s)	0.44	0.01	[0.42, 0.45]
Native (s)	0.03	0.01	[0.02, 0.05]
Speedup (Int/DR)	34	0.4085	[33, 35]

1,048,576 Repetitions	**Mean**	**Standard Deviation**	**95% Confidence Interval**
Interpretive Runtime (s)	1152.99	6.93	[1,135.78, 1,170.20]
Dynamic Recompiler (s)	3.17	0.01	[3.14, 3.20]
Native (s)	0.03	0.01	[0.02, 0.05]
Speedup (Int/DR)	364	3.3790	[356, 372]

Blowfish

55.5 Megabytes	**Mean**	**Standard Deviation**	**95% Confidence Interval**
Interpretive Runtime (s)	521.24	0.36	[520.33, 522.14]
Dynamic Recompiler (s)	3.60	0.01	[3.58, 3.61]
Native (s)	0.88	0.02	[0.84, 0.92]
Speedup (Int/DR)	145	0.1734	[144, 145]

Memory Allocation/Deallocation

1 Byte, 1,048,576 Repetitions	**Mean**	**Standard Deviation**	**95% Confidence Interval**
Interpretive Runtime	373.02	1.32	[369.74, 376.30]
Dynamic Recompiler (Unoptimized)	125.19	0.63	[123.64, 126.74]
Dynamic Recompiler (Standard)	47.84	0.32	[47.05, 48.62]
Dynamic Recompiler (Optimized)	18.28	0.08	[18.09, 18.47]
Native	0.46	0.01	[0.45, 0.48]
Speedup (Int/DR)	8	0.0630	[8, 8]
Optimized Speedup (Int/DR)	20	0.1420	[20, 21]

1 MB, 1,048,576 Repetitions	**Mean**	**Standard Deviation**	**95% Confidence Interval**
Interpretive Runtime	1080.30	9.18	[1,057.51, 1,103.08]
Dynamic Recompiler (Standard)	130.69	1.94	[125.87, 135.52]
Native	11.42	0.18	[10.96, 11.87]
Speedup (Int/DR)	8	0.1234	[8, 9]

Memory Copy

768 Megabytes	**Mean**	**Standard Deviation**	**95% Confidence Interval**
Interpretive Runtime (s)	124.79	0.09	[124.57, 125.01]
Dynamic Recompiler (s)	9.35	0.11	[9.07, 9.62]
Native	7.99	0.01	[7.97, 8.00]
Speedup (Int/DR)	13	0.1651	[13, 14]

File Copy

64 Megabytes	**Mean**	**Standard Deviation**	**95% Confidence Interval**
Interpretive Runtime (s)	2644.42	4.29	[2,633.76, 2,655.08]
Dynamic Recompiler (s)	20.91	0.06	[20.77, 21.04]
Native (s)	1.22	0.03	[1.15, 1.29]
Speedup (Int/DR)	126	0.1293	[126, 127]

Appendix B

Blowfish Code Listing

B.1 Full C Source Code

```
#include <string.h>
#include <ctype.h>
#include "Blowfish.h"

#define N    16

unsigned long    P[N + 2];
unsigned long    S[4][256];
void             *blowfishData;

int BlowfishStartup( void ) {
    FILE    *temp;

    blowfishData = malloc( 5032 );

    temp = fopen( "Blowfish.dat", "r" );

    if ( !temp ) {
        fprintf( stdout, "Error opening Blowfish data file." );
        return 0;
    }
    else {
        fread( blowfishData, 1, 5006, temp );

        fclose( temp );

        return 1;
    }
}

unsigned long F( unsigned long x ) {
    unsigned short  a, b, c, d;
    unsigned long   y;

    d = x & 0x00FF;
    x >>= 8;
    c = x & 0x00FF;
    x >>= 8;
    b = x & 0x00FF;
    x >>= 8;
    a = x & 0x00FF;
    y = S[0][a] + S[1][b];
    y = y ^ S[2][c];
    y = y + S[3][d];

    return y;
}

void Blowfish_encipher(unsigned long *xl, unsigned long *xr) {
    unsigned long  Xl, Xr, temp;
    short          i;

    Xl = *xl;
    Xr = *xr;

    for (i = 0; i < N; ++i) {
        Xl = Xl ^ P[i];
        Xr = F(Xl) ^ Xr;

        temp = Xl;
        Xl = Xr;
        Xr = temp;
    }
    temp = Xl;
    Xl = Xr;
    Xr = temp;

    Xr = Xr ^ P[N];
    Xl = Xl ^ P[N + 1];

    *xl = Xl;
    *xr = Xr;
}

void Blowfish_decipher(unsigned long *xl, unsigned long *xr) {
    unsigned long  Xl, Xr, temp;
    short          i;

    Xl = *xl;
    Xr = *xr;

    for (i = N + 1; i > 1; --i) {
        Xl = Xl ^ P[i];
        Xr = F(Xl) ^ Xr;

        temp = Xl;
        Xl = Xr;
        Xr = temp;
    }

    temp = Xl;
    Xl = Xr;
    Xr = temp;

    Xr = Xr ^ P[1];
    Xl = Xl ^ P[0];

    *xl = Xl;
    *xr = Xr;
}

short InitializeBlowfish( char key[], short keybytes ) {
    short           i, j, k, l;
    unsigned long   data, datal, datar;

    if ( !blowfishData )
```

```
        return -1;

    for (i = 0; i < N + 2; ++i) {
        memcpy( &P[i], (void *)((int)blowfishData +
                ( i * sizeof( long ) )), sizeof( long ) );
    }

    l = i;

    for (i = 0; i < 4; ++i) {
        for (j = 0; j < 256; ++j) {
            memcpy( &S[i][j], (void *)((int)blowfishData +
                    ( i * 256 * sizeof( long ) ) +
                    ( j * sizeof( long ) )), sizeof( long ) );
        }
    }

    j = 0;
    for (i = 0; i < N + 2; ++i) {
        data = 0x00000000;
        for (k = 0; k < 4; ++k) {
            data = (data << 8) | key[j];
            j = j + 1;
            if (j >= keybytes) {
                j = 0;
            }
        }
        P[i] = P[i] ^ data;
    }

    datal = 0x00000000;
    datar = 0x00000000;

    for (i = 0; i < N + 2; i += 2) {
        Blowfish_encipher(&datal, &datar);

        P[i] = datal;
        P[i + 1] = datar;
    }

    for (i = 0; i < 4; ++i) {
        for (j = 0; j < 256; j += 2) {
            Blowfish_encipher(&datal, &datar);
```

```
                S[i][j] = datal;
                S[i][j + 1] = datar;
            }
        }

        return 0;
    }
```

B.2 `Blowfish_encipher()` Intel x86 Assembly Code

Offset	Machine Code	Intel x86 Disassembly
0x00000000	55	**push ebp**
0x00000001	89E5	**mov ebp**,**esp**
0x00000003	53	**push ebx**
0x00000004	83EC10	**sub esp**, 0x10
0x00000007	51	**push ecx**
0x00000008	57	**push edi**
0x00000009	8D7C2408	**lea edi**, [**esp** + 0x08]
0x0000000d	B8CCCCCCCC	**mov eax**, 0xCCCCCCCC
0x00000012	AB	**stosd**
0x00000013	AB	**stosd**
0x00000014	AB	**stosd**
0x00000015	AB	**stosd**
0x00000016	5F	**pop edi**
0x00000017	59	**pop ecx**
0x00000018	8B4508	**mov eax**, [**ebp** + 0x08]
0x0000001b	8B08	**mov ecx**,[**eax**]
0x0000001d	894DF8	**mov** [**ebp** − 0x08], **ecx**
0x00000020	8B450C	**mov eax**, [**ebp** + 0x0C]
0x00000023	8B08	**mov ecx**, [**eax**]
0x00000025	894DF4	**mov** [**ebp** − 0x0C], **ecx**
0x00000028	C745EC00000000	**mov** [**ebp** − 0x14], 0x0
0x0000002f	EB35	**jmp** ∗ + 0x37
0x00000031	0FBF5DEC	**movsx ebx**, [**ebp** − 0x14]
0x00000035	8B55F8	**mov edx**, [**ebp** − 0x08]
0x00000038	33149D00000000	**xor edx**, [**ebx** ∗ 4 + _P]
0x0000003f	8955F8	**mov** [**ebp** − 0x08], **edx**
0x00000042	FF75F8	**push** [**ebp** − 0x08]
0x00000045	E800000000	**call** _F
0x0000004a	59	**pop ecx**
0x0000004b	3345F4	**xor eax**, [**ebp** − 0x0C]
0x0000004e	8945F4	**mov** [**ebp** − 0x0C], **eax**
0x00000051	8B45F8	**mov eax**, [**ebp** − 0x08]
0x00000054	8945F0	**mov** [**ebp** − 0x10], **eax**
0x00000057	8B45F4	**mov eax**, [**ebp** − 0x0C]
0x0000005a	8945F8	**mov** [**ebp** − 0x08], **eax**
0x0000005d	8B45F0	**mov eax**, [**ebp** − 0x10]

```
0x00000060      8945F4          mov [ebp - 0x0C], eax
0x00000063      FF45EC          inc [ebp - 0x14]
0x00000066      66837DEC10      cmp [ebp - 0x14], 0x10
0x0000006b      7CC4            jl   * - 0x3A
0x0000006d      8B45F8          mov eax, [ebp - 0x08]
0x00000070      8945F0          mov [ebp - 0x10], eax
0x00000073      8B45F4          mov eax, [ebp - 0x0C]
0x00000076      8945F8          mov [ebp - 0x98], eax
0x00000079      8B45F0          mov eax, [ebp - 0x10]
0x0000007c      8945F4          mov [ebp - 0x0C], eax
0x0000007f      8B55F4          mov edx, [ebp - 0x0C]
0x00000082      331540000000    xor edx, _P + 0x40
0x00000088      8955F4          mov [ebp - 0x0C], edx
0x0000008b      8B55F8          mov edx, [ebp - 0x08]
0x0000008e      331544000000    xor edx, _P + 0x44
0x00000094      8955F8          mov [ebp - 0x08], edx
0x00000097      8B4508          mov eax, [ebp + 0x08]
0x0000009a      8B4DF8          mov ecx, [ebp - 0x08]
0x0000009d      8908            mov [eax], ecx
0x0000009f      8B450C          mov eax, [ebp + 0x0C]
0x000000a2      8B4DF4          mov ecx, [ebp - 0x0C]
0x000000a5      8908            mov [eax], ecx
0x000000a7      8D65FC          lea esp, [ebp - 0x04]
0x000000aa      5B              pop ebx
0x000000ab      5D              pop ebp
0x000000ac      C3              ret near
```

Bibliography

[1] Apple Computer, Inc. "Cocoa". Apple Developer Connection: Cocoa. 2004. Apple Computer, Inc. [http://developer.apple.com/cocoa/].

[2] Apple Computer, Inc. Inside Macintosh: PowerPC System Software. New York: Addison-Wesley Publishing Company, 1994. Chapter 2: Mixed Mode Manager.

[3] Henning, John L. "SPEC CPU2000: Measuring CPU Performance in the New Millennium." July 2000. [http://www.spec.org/cpu2000/papers/COMPUTER_200007.JLH.pdf].

[4] Hoskins, Jim. "The PowerPC Initiative: An Executive Summary." The Henry Samueli School of Engineering. 1995. [http://www.eng.uci.edu/comp.arch/processors/powerpc/PCPower.html].

[5] IBM. PowerPC Microprocessor Family: The Programming Environments for 32-Bit Microprocessors. [http://www.chips.ibm.com/products/powerpc]. 2000.

[6] Intel. IA-32 Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference. Order Number 245471-008. 2002.

[7] Krall, Andreas. "Efficient JavaVM just-in-time compilation," in Proceedings of International Conference on Parallel Architectures and Compilation Techniques, pages 205-212, October 1998.

[8] Metrowerks. CodeWarrior Development Tools MSL C Reference. [ftp://ftp.metrowerks.com/pub/docs/Technologies/MSL/MSL_C_Reference.pdf].

[9] Patterson, David A. and John L. Hennessy. Computer Organization & Design: The Hardware/Software Interface, Second Edition. San Francisco: Morgan Kaufmann Publishers, Inc., 1998.

[10] Schneier, Bruce. Applied Cryptography, Second Edition: Protocols, Algorithms, and Source Code in C. New York: John Wiley & Sons, Inc, 1996. Pages 336-339.

[11] Schneier, Bruce. "The Blowfish Encryption Algorithm." Schneier.com. [`http://www.schneier.com/blowfish.html`].

[12] Shaffer, Joshua H. (joshshaffer@mac.com). "Champagne, an emulator that runs Microsoft Windows applications on the Macintosh." [alpha version].

[13] Silberschatz, Abraham, Peter Galvin and Greg Gagne. Operating System Concepts with Java, Sixth Edition. Hoboken: John Wiley & Sons, Inc, 2004.

[14] Tamhane, Ajit C. and Dorothy D. Dunlop. Statistics and Data Analysis: from Elementary to Intermediate. Upper Saddle River: Prentice Hall. 2000.

[15] Tijms, Arjan. "Binary translation: Classification of emulators." 2000. [`http://www.liacs.nl/~atijms/bintrans.pdf`].

[16] Traut, Eric. "Building the Virtual PC: A software emulator shows that the PowerPC can emulate another computer, down to its very hardware." Byte. November 1997. Accessed 8 October 2003. [`http://www.byte.com/art/9711/sec4/art4.htm`].

[17] Venners, Bill. "Bytecode basics: A first look at the bytecodes of the Java virtual machine." JavaWorld. 1996. [`http://www.javaworld.com/javaworld/jw-09-1996/jw-09-bytecodes.html`].