Selected Project Reports, Fall 2005
Advanced OS & Distributed Systems (15-712)

edited by Garth A. Gibson and Adam G. Pennington

Rebecca Hutchinson, Neelakantan Krishnaswami, Ruy Ley-Wild, William Lovas
Jason Franklin, Mark Luk, Jonathan M. McCune
Mukund Gunti, Mark Pariente, Ting-Fang Yen, Stefan Zickler
Deepak Garg, Vaibhav Mehta, Shashank Pandit, Michael De Rosa

Dec 2005
CMU-CS-05-201

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This technical report contains four final project reports contributed by participants in CMU's Fall 2005 Advanced Operating Systems and Distributed Systems course (15-712) offered by professor Garth Gibson. This course examines the design and analysis of various aspects of operating systems and distributed systems through a series of background lectures, paper readings, and group projects. Projects were done in groups of three and four, required some kind of implementation and evaluation pertaining to the classroom material, but with the topic of these projects left up to each group. Final reports were held to the standard of a systems conference paper submission; a standard well met by the majority of completed projects. Some of the projects will be extended for future submissions to major system conferences.

The reports that follow cover a broad range of topics. While not all of these reports report definitely and positively, all are worth reading because they involve novelty in the systems explored and bring forth interesting research questions. These reports present methods for detecting the presence of a virtual machine monitor (VMM) by malware being analyzed using slight ways that a VMM differs from a real machine; the design and implementation of software transactional memory library for the O'Caml programming language which allows transactions to be used for synchronization, control flow, and shared variables; a virtual folder overlay to a filesystem combining context search techniques with traditional metadata information while also exploring indexing tradeoffs; and a proposed semantic file-system which would allow a user to search for and manipulate key-value pairs of attribute date on files using familiar file system primitives.

Contents

Detecting the Presence of a VMM through Side-Effect Analysis
15-712 Project Final Report

Jason Franklin Mark Luk Jonathan M. McCune

Abstract

Virtual machine monitors (VMMs) are a critical enabling technology for the security community. They allow for the secure deployment of high interaction honeypots and the dynamic analysis of malware such as worms and spyware. VMMs must remain undetectable if they are to be used for security applications since intruders or malware that are able to detect a VMM can modify their behavior to thwart analysis. Current botnet software like Agobot prevents dynamic analysis by running a simple check based on VMM implementation quirks. In this paper, we develop detection attacks which when run inside of a virtual machine monitor are capable of detecting the VMM without relying on VMM implementation details.

Our attacks are theoretically sound. They are based on the two exceptions to the equivalence property of a virtual machine monitor: timing dependencies and resource sharing [21]. We believe that these exceptions are inherent to any virtual machine monitor. In this paper, we describe the design and implementation of our attacks, their success detecting the Xen virtual machine monitor [3] without relying on specific software implementation features, and potential countermeasures.

1 Introduction

Because of the strong isolation properties provided by virtual machines monitors (VMMs), virtualization has become an important tool in the security community. Virtual machines (VMs) allow untrusted code to be executed and observed inside a constrained environment that is isolated from other virtual machines. Hence, individuals who wish to analyze malware are free to run potentially malicious code without the risk of compromise or in the case of a worm, without the risk of further propagation.

Dynamic analysis is the study of a program's behavior by monitoring its execution. Virtual machine monitors are a critical enabling technology for dynamic analysis. Security researchers have found dynamic analysis in conjunction with virtualization to be an effective tool to understand the behavior of malicious software such as worms and viruses [18, 31].

A critical assumption made in the dynamic analysis community is that the behavior of the code being analyzed is representative of the natural behavior of the code. Only if this assumption holds can the dynamic analysis community make valid inferences about program behavior through execution monitoring.

In this paper, we study the ability of a program to detect the presence of a virtual machine monitor. After detecting the presence of a VMM, a malicious program could modify its behavior in a number of ways to thwart dynamic analysis. For example, a malicious program could execute innocuous instructions instead of malicious ones, delete itself, execute instructions to denial of service (DoS) the VMM, obfuscate its behavior by adding spurious instructions to its execution, or generate false alarms in monitoring software.

We define the problem of virtual machine monitor detection, in which a program, called a detection attack, executes on a remote host in order to determine if a VMM is resident in memory. The main contribution of this paper is the development of a detection attack whose execution differs from the perspective of an external verifier when a VMM is resident in memory versus when it is executed directly on the underlying hardware. We describe the design and implementation of our attacks, their success detecting the Xen virtual machine monitor [3] without relying on specific software implementation features, and potential countermeasures.

Our developed attacks are based on two exceptions to the equivalence property [21] of a virtual machine monitor, timing dependencies and resource sharing. We believe that these exceptions are inherent to any virtual machine monitor. However, testing our attacks against VMMs other than Xen is a matter of future work.

In our scheme, an external verifier remotely exploits a target host and gains kernel privilege. After which, an instance of a detection attack is executed on the target host. When the detection attack finishes executing, it sends the result of a cryptographic computation back to an external verifier who records the time elapsed since transmission of the attack and verifies that the result is the expected value. If the result differs from the expected value by more than an experimentally determined threshold, a VMM is considered to be resident in memory and the detection attack returns success. If the target host does not execute the detection attack, the correct cryptographic response will not be returned to the external verifier with high probability and the detection attack defaults to success.

Most related work emphasizes software-dependent, hardware-independent detection attacks which are usually

possible to counter through modifications to the VMM implementation. Our detection attacks are software-independent and hardware-dependent. While the practicality and even ability of organizations who develop VMMs to modify VMM implementations to quickly counter software-dependent, hardware-independent attacks can be argued, we believe that organizations and individuals who use VMMs for security critical applications would likely modify the VMM implementation to thwart software-dependent attacks, especially when trivial software countermeasures exists. Our software-independent, hardware-dependent attacks should be more difficult to counter without hardware modification, a task which we assume is difficult for security conscious organizations who rely on commodity hardware.

This paper is organized as follows. Section 2 discusses background including virtual machine monitors, dynamic analysis, and honeypots. Section 3 contains a problem definition, our virtual machine monitor model, and our assumptions. In Section 4, we discuss the design, implementation, and evaluation of our attack. We describe future work including additional potential detection attack techniques in Section 5 and we discuss related work in Section 6. We conclude in Section 7.

2 Background

First, we describe the virtual machine monitor formalization of Popek and Goldberg [21]. We define dynamic analysis and explain its relation to virtual machine monitors, then we describe honeypots and their use of virtualization.

2.1 Virtual Machine Monitors

We follow Popek and Goldberg in defining a virtual machine monitor as an efficient, isolated duplicate of the underlying hardware. This definition encompasses the three primary properties that a control program must satisfy to be termed a virtual machine monitor: efficiency, resource control, and equivalence. In order to explain the three properties in detail, we must first introduce some terminology.

We classify the underlying instructions of a machine based on their behavior. An instruction is privileged if it can only be executed in the most priviledged ring, while executing it in any other ring would result in a trap. Hence, privileged instructions are characteristics of the underlying hardware and are invariant over a particular instruction set. One example of a privileged instruction on the x86 is the clear interrupts instruction (CLI). An instruction is sensitive if it can interfere with the VMM. A instruction is innocuous if it is not sensitive.

Informally, the **efficiency property** dictates that programs run in a virtualized environment show no more than minor decreases in speed. Since minor decreases in speed is difficult to quantify, a parallel requirement of the efficiency property is that a statistically dominant subset of the virtual processor's instructions be executed directly by the real processor. In terms of implementation, this typically requires that all innocuous instructions be executed by the hardware directly without intervention by the VMM.

The **resource control property** dictates that the VMM is in complete control of system resources. This requires that it be impossible for an arbitrary program to affect system resources not allocated to it, for example memory and peripherals.

The **equivalence property** dictates that the VMM provide an environment for programs which is essentially identical to the original machine. The following parallel formulation has also been put forward: any program P executing with a VMM resident in memory, with two possible exceptions, must perform in a manner *indistinguishable* from the case when the VMM did not exist and P had the freedom of access to privileged instructions that the programmer had intended. The two possible exceptions to the equivalence property result from timing dependencies and resource availability.

The **timing dependency exception** states that certain instruction sequences in a program may take longer to execute. Hence, assumptions about the length of time required for the execution of an instruction might lead to incorrect results. This exception results from the possibility of the VMM occasionally intervening in certain instruction sequences.

The **resource availability exception** states that it may be the case that a particular request for a resource can not be satisfied. As a result, a program may be unable to function in the same manner as it would if the resource were made available. This exception is made because the VMM shares the underlying hardware and hence consumes some amount of resources.

These exceptions allow for the theoretical possibility of detecting a virtual machine monitor. If these exceptions did not exist, a virtual machine monitor that perfectly satisfied the equivalence property would be impossible to detect. In order to eliminate any dependence on a particular virtual machine monitor implementation, we develop our attacks around an abstract idealization of a VMM which satisfies the equivalence property with the above mentioned exceptions. We term such a VMM an *indistinguishable VMM*.

One caveat is that there is no formal proof that these possible exceptions must exist across all architectures and VMM implementations. We give an informal argument for the existence of the timing dependency exception on our target architecture, the Intel Pentium. We assume that all instructions execute in non-zero time. A sensitive instruction is one that can affect the state of the VMM or VM.

In order to satisfy the resource control property, a VMM must prevent arbitrary programs from executing sensitive instructions without VMM intervention. If it was the case that all sensitive instructions were also privileged, then the sensitive instructions would be forced to trap and hence the VMM would be able to intervene in their execution. However, a previous analysis [24] of the Intel Pentium demonstrated that sensitive, unprivileged instructions exist. These instructions require a VMM to intervene and hence necessitate the existence of timing dependencies on the Intel Pentium, regardless of VMM implementation. Thus, for our target architecture, the timing dependence exception exists. Exploiting these exceptions to detect a VMM is a nontrivial task, the remainder of the paper explains our approach.

2.2 VMM Implementations

We are interested in two different virtual machine monitor implementation techniques [25]: full virtualization and paravirtualization. Both of these techniques are used to virtualize operating system instances rather than processes on one operating system; however, they differ in their approach to achieving this goal.

In full virtualization, the virtual replica of the underlying hardware exposed is functionally identical to the underlying machine. This allows operating systems and applications to run unmodified. In addition, these virtual machine monitors can be run recursively in virtual machines exposed by other virtual machine monitors – up to resource exhaustion. Full virtualization is typically implemented in one of two ways: (1) with full support from the underlying hardware, affording maximum efficiency; and (2) without full support from the underlying hardware, requiring sensitive instructions to be emulated in software.

A popular full system virtualization VMM is VMWare [33, 34]. VMWare runs inside of a host operating system – as opposed to running on the raw hardware – and exposes an accurate representation of the x86 architecture to guest operating systems. This causes VMWare to suffer a performance overhead during the execution of certain privileged instructions, since they must be emulated in software.

In paravirtualization, the virtual replica of the underlying hardware exposed is similar to the underlying machine, but it is not identical. This is done when the underlying machine architecture consists of sensitive instructions which are not privileged. Paravirtualized VMMs have the drawback that operating systems must be modified to run on them; however, they enable efficient virtualization to be performed even when hardware support for full virtualization is unavailable [3].

Xen is an open-source x86 virtual machine monitor developed by University of Cambridge, and is popular in the systems research community [3]. Xen uses paravirtualization to achieve high performance and presents a software interface to the guest OS that is not identical to the actual hardware. Therefore, the guest operating system needs to be modified before it can run on Xen.

Full virtualization on Xen can be accomplished with hardware support, e.g., Intel Vanderpool Technology (VT) [16] or AMD Pacifica [7]. VT support is already implemented in the development version of Xen. Because such machines are not yet widely available, even in the systems research community, we focus on detecting Xen on today's Intel Pentium 4 CPUs.

Note that paravirtualization is trivially detectable from within a guest OS, as certain features of the underlying hardware will be broken or missing. In the remainder of this paper, we do not consider such detection mechanisms. We are interested in detection attacks predicated on the theoretical timing and memory exceptions which will be present in all VMMs (recall Section 2.1). That is, we only include innocuous or privileged and sensitive instructions in our detection attack code.

2.3 Dynamic Analysis

Dynamic analysis is the study of a program's behavior by monitoring its execution. In contrast to static analysis, which is the study of a program's behavior by examining its source code, dynamic analysis is unsound. However, security researchers have found dynamic analysis to be an effective tool to understand the behavior of malicious software such as Internet worms, viruses, and spyware [18, 31].

A central assumption made in the dynamic analysis community is that the behavior of the malicious code being analyzed is representative of the natural behavior of the malicious code. Only if this assumption holds can the dynamic analysis community make valid inferences about program behavior through execution monitoring.

VMMs enable dynamic analysis through the detailed execution monitoring and control of executing software. For example, some VMMs can support rollback of file system changes by keeping an extensive logfile of the virtual machine. This allows an investigator to repeat an experiment multiple times with identical initial conditions. ReVirt [10] is an instantiation of one such system, where the VMM records all interrupts and inputs from the user.

2.4 Honeypots

Honeypots are specially constructed computer systems connected to the Internet with the intention of attracting malicious behavior. Once attacked, honeypots enable both static and dynamic analysis. We define different classes of honeypots, explain their relationship with VMM, and motivate the incentives for malware writers to detect the

existence of virtualized environments.

2.4.1 Definition

Spitzner defines a honeypot as *a computational resource whose value resides in being probed, attacked, or compromised by invaders* [29, 30]. A collection of honeypots is called a honeynet [14]. Typically, honeypots are computer systems that serve no legitimate purpose, so all traffic destined for honeypots can be considered suspicious.

2.4.2 Classes

Honeypots are classified as being either high- or low-interaction. Our work makes it possible to detect high-interaction honeypots which use system virtualization. We briefly describe both high and low-interaction honeypots.

Low-interaction Honeypots Low-interaction honeypots emulate well-known operating systems and services like web servers. These honeypots are less complex to deploy than high-interaction honeypots and represent less of a risk to the network where they are installed, however they capture less information about attackers. Perhaps the best-known low-interaction honeypot is Honeyd by Provos [22, 23]. Honeyd is a software package freely available under the GNU Public License which can emulate and gather information about attacks on many popular services (e.g., web servers, ftp servers).

High-interaction Honeypots In contrast to low-interaction honeypots which simply emulate services, high-interaction honeypots are real computer systems or virtualized systems running real applications and servers. They can capture far more information about attacks, but they also expose the network to greater risk. Carbone and de Geus present a mechanism for improving analysis of attacks by gathering digital evidence from the filesystem of high-interaction honeypots [8]. The basic premise of their mechanism is to intercept certain system calls and log the arguments for later analysis. Sebek is a tool which attempts to capture most of the attacker's activity once inside a honeypot [32]. Sebek is a client-server application, where the client is installed on the honeypot and sends data covertly (hopefully, so that the attacker does not realize it) to the server.

2.4.3 Use of VMMs

Honeypots can yield tremendous value for computer security personnel trying to understand attacks. Historically, high-interaction honeypots have incurred significant cost and risk because they consisted of a full computer system which if compromised could be used to attack other systems. The availability of virtual machine monitors for commodity computer systems has made it possible for a single physical machine to run a number of high-interaction honeypots. This not only reduces the financial burden of high-interaction honeypot deployment, but also reduces the risks inherent in the deployment of these honeypots because the VMM can isolate compromised virtual machines to prevent them from attacking other systems.

Vrable et al. describe a honeyfarm they constructed which makes heavy use of virtualization [35]. Their *Potemkin Virtual Honeyfarm* is built on top of the Xen VMM [3] and allows for the simultaneous execution of thousands of high-interaction honeypots on a single machine.

The use of virtualization to enable dynamic analysis of malware behavior as observed on high-interaction honeypots is a widely used and valuable tool for security personnel. Attackers have recognized this, and are beginning to develop malcode that can detect the presence of a VMM and alter its behavior to avoid detection or analysis [35].

Our project takes as its starting point the existence of high-interaction honeypots created for the purpose of performing dynamic analysis of malware, which leverage virtual machine monitors to reduce the requisite financial investment and operational risks involved in setting up a honeynet.

3 Problem Definition, VMM Model, and Assumptions

In this section, we define the problem of virtual machine monitor detection, discuss our assumptions, and introduce our virtual machine monitor model.

3.1 Problem Definition

We define the problem of virtual machine monitor detection, in which a program, called a detection attack, executes on a remote host in order to determine if a VMM is resident in memory.

We define a *detection attack* as any program D whose execution differs, from the perspective of an external verifier, when a VMM is resident in memory than when D is executed directly on the underlying hardware. In order to avoid trivial solutions, we require the additional stipulation that D work without the explicit cooperation of the VMM.

In our scheme, an external verifier sends an instance of a detection attack D to a remote host, hereafter refereed to as the target host. After transmission, the target host has complete control over the execution of D. It can tamper with the execution of the detection attack or prevent the attack from executing at all. For the purposes of this project, we assume the VMM faithfully executes D in an untampered execution environment. Designing our detection attacks so that tampering by way of dynamic analysis is noticeable to the external verifier is a matter of future work. In addition, resistance to tampering by way of static analysis can be achieved with code obfuscation techniques [19] and is also a matter of future work. When D

finishes executing, it sends the result of its cryptographic computation back to the external verifier who records the time elapsed since transmission of the attack and verifies that the result is the expected value. The external verifier checks the result by comparing the elapsed time and result of the computation against a predetermined range of values for the target host's hardware platform. If the results differ by more than this experimentally determined threshold, a VMM is considered to be resident in memory and the detection attack returns success. If the target host does not execute the detection attack, the correct cryptographic response will not be returned to the external verifier with high probability and the detection attack defaults to success. While this default detect behavior has the potential to cause false positives, we assume that non-virtualized hosts lack the ability to tamper with the execution of a detection attack since the attack exploits and takes control of the target host's operating system.

In order to provide a concrete mechanism to discuss detection attacks, we develop the following classification. We call a detection attack D software-independent if its probability of success is independent of VMM software implementation, more formally $P(S|I) = P(S)$ where S is successful detection and I is VMM implementation. D is software-dependent if its probability of success depends on a particular VMM implementation, so $P(S|I) > P(S)$. Similarly, a detection attack D is hardware-independent if its probability of success is independent of the hardware configuration of the target host, formally $P(S|H) = P(S)$ where H is the hardware configuration of the target host. D is hardware-dependent if $P(S|H) > P(S)$. These classifications can be combined in the obvious way. The detection attacks developed in this paper are software-independent and hardware-dependent.

In addition to defining attacks, we define a *countermeasure* as a defense mechanism incorporated into the VMM to mask detectable differences between $D's$ execution with and without a VMM resident in memory. The goal of a countermeasure is to render a given instance of a detection attack D ineffective.

3.2 Virtual Machine Monitor Model

In order to decrease the reliance of our attacks on an particular VMM software implementation, we base the development of our detection attacks around an idealized VMM. For our purposes, the VMM in question is an indistinguishable VMM which satisfies the resource control property. Hence, our idealized VMM model is an isolated duplicate of the underlying hardware which possesses the resource control and equivalence properties. We do not require that our abstract VMM be efficient, because a previous analysis [24] has shown that virtualization on our target architecture, the x86, can not satisfy the efficiency requirement. In practice, this exception does not effect our results.

3.3 Assumptions

We assume that we have root access to at least one VM running inside the VMM. This is not an unreasonable assumption, because remote root exploits are a key enabler of Internet worms. The detection attack executes at the highest privilege level with interrupts turned off after exploiting the OS.

We assume that the VMM will not tamper with the execution of our detection code. Tampering is defined to be modifying certain instructions in the execution stream, either prior to or during their execution. Thus, the VMM is required to faithfully execute all instructions in the detection code. Furthermore, we assume the VMM will only employ software-based countermeasures. This is a reasonable assumption because hardware changes are difficult to realize for organizations who lack the ability to develop their own platforms and hence must rely on commodity hardware.

In addition, we assume that the VMM is running on an unmodified x86 Pentium 4. The virtual machine monitor is assumed to be a full system virtual machine monitor. For the purposes of this paper, we are targeting Xen on legacy systems. Future work will address VMMs that do not require paravirtualization, such as Xen on VT and VMWare.

4 Attack Design, Implementation, and Evaluation

Recall that in Popek and Goldberg's formal requirements for virtualization [21], the equivalence property of a VMM has two exceptions: the timing dependency exception and the resource availability exception. These two exceptions highlight fundamental differences between running software natively and inside a virtualized environment. We leverage these two exceptions of a VMM in order to detect its existence, in conjunction with a remote verifier for reliable timing measurements. The significance of using these inherent properties of a VMM cannot be understated. Since most previous work in VMM detection focuses on a specific detail that is different from one particular software implementation to another, these attacks can be foiled by software patches to the VMM. Our attack, on the other hand, is based on intrinsic properties of a VMM, and thus is agnostic to the actual software implementation. Therefore, we conjecture that it should be difficult to create software-based countermeasure against our attack.

4.1 Attack Overview

We have two parties in our VMM detection protocol, an **external verifier** and a **target host**. The external verifier would like to determine whether the target host is running Xen by asking the target host to execute an instance of a

detection attack, refereed to as the **VMM Detection Code**, in a challenge-response protocol. The external verifier cannot trust the target host to execute correctly. However, it can trust itself to have an accurate clock, which is used to measure the runtime of VMM Detection Code on the target host. Because all timing measurements are made on the remote verifier, clock synchronization is not required between the machines.

Because of the inherent properties of a VMM, Xen should not be able to execute the detection code as fast as native Linux. The detection code is crafted specifically to target the timing and resource exceptions to the equivalence property of the VMM to ensure a noticeable runtime difference. In the following section, we will give a detailed description of our protocol and implementation.

4.1.1 Protocol

Our detection attack proceeds as in Figure 1. We employ a challenge-response protocol to determine the existence of Xen. The external verifier starts an internal timer and performs a remote root exploit against the target host. In other words, the external verifier exploits a piece of software that is running with root privilege on the target host. Since the attack inherits root privilege from the exploited software, it has the ability to install the detection code as a kernel module. Remotely acquiring kernel privilege by installing itself as a kernel module is a technique often found in malware and worm code, and has been proven to be easily achievable in the wild. After such an exploit, the detection code would run at the same privilege ring as the exploited OS. If the target host was running native Linux, the detection code would run in the highest priviledge ring, or ring 0. On the other hand, if the target host was running a VMM, the detection code would run in ring 1. The goal of the detection code is to determine this difference.

The VMM Detection Code is structured as follows. It has initial setup code that exploits the target machine and installs itself as a kernel module. Next, it turns off interrupts and executes a benchmarking loop that will experience detectable performance differences depending on the execution environment. Note that the benchmarking loop expects to run in an untampered environment inside the target host. The benchmarking loop is the crux of our detection attack and will be explained in detail. Finally, the result of the benchmarking loop will be sent back to the external verifier and interrupts re-enabled.

The verifier times the entire execution of the VMM detection code to determine whether or not it ran natively or inside a virtualized environment. If the execution time exceeds a certain threshold based on empirical measurements, the verifier would suspect that the target host is running a VMM, and would consequently command the detection program to self-destruct or alter its behavior. Else, if the execution time is as expected for running on native Linux, the verifier would command the detection code to proceed.

4.1.2 Benchmarking Loop

Since the external verifier determines the existence of a VMM by timing the execution of the benchmarking loop, we need this loop to have strict timing properties. Intuitively, this benchmarking loop needs to *verifiably* experience *longer runtime* if it is executed inside a virtual machine. A detailed explanation of its desired properties follows.

A *longer runtime* is defined by the following two statements. First, for the same processor, executing the benchmarking loop in a native OS should be faster than running the benchmarking loop in an OS executing on top of a VMM. Second, we would like the benchmarking loop running on top of a VMM on the fastest available Pentium 4 to execute slower than the benchmarking loop running in a native OS on the slowest available Pentium 4. Note that the second statement implies the first.

According to the above definition, the benchmarking loop needs to experience runtime such that one can differentiate between a virtualized environment and a native OS without knowledge of the actual processor speed. We assume the absence of such information because it is not clear how one can obtain this knowledge prior to executing the benchmarking loop.

Since an arbitrarily fast processor can simulate an arbitrarily slow processor, we need a realistic bound on the underlying machine's possible configurations. For the VMM detection attack, we chose the Intel Pentium 4 architecture, which features processor speeds ranging from 2.0 GHz to 3.8 GHz.[1] Thus, our benchmarking loop needs to execute faster on a 2.0 GHz non-virtualized machine than on a 3.8 GHz virtualized machine. We show how we satisfy this claim in the following section, where we describe a benchmarking loop that experiences a factor of four slowdown when executed in a virtualized environment.

The second property required for the benchmarking loop is one of *verifiable execution*. Intuitively, this means that the external verifier needs a strong guarantee that the benchmarking loop has finished execution on the target host. In other words, the external verifier needs to verify that a sequence of instructions X had been executed on a target host.

Achieving the verifiable execution property is a challenging research problem. Pioneer [28] has a checksum function that can achieve this verifiable execution property. However, Pioneer requires knowledge of exact hardware specifications, while we only assume a particular proces-

[1] `http://www.intel.com/products/processor/pentium4`

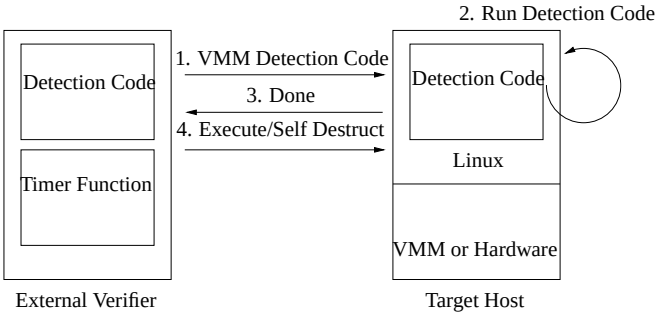


Figure 1: Overview of VMM Detection Protocol.

sor family.

Client puzzles [20] are tests based on cryptographic primitives that require a machine to spend a certain amount of cycles to brute-force an answer. This computation is a function of a fresh challenge generated by the external verifier. The fact that the target host is able to respond with the correct value implies that the target host has performed the needed computation. However, client puzzles do not guarantee that the target host actually executed the intended program X. The target host could have executed a program Y that returns the same result.

Achieving the verifiable execution property remains an open question. For this paper, we assume an untampered execution environment. Therefore, the external verifier can assume that the benchmarking loop executesd as intended on the target host.

4.2 Trace Cache based Side Effects

In this section we detail the way in which we exploit the properties of the *trace cache* – a consistent feature across all of Intel's currently-released Pentium 4 processors – to detect the presence of a VMM. Our detection strategy is based on the presence of a small number of sensitive, privileged instructions inside a loop of thousands of arithmetic instructions. In the remainder of this section, we refer to sensitive instructions when in fact we mean sensitive, privileged instructions.

4.2.1 Architecture of the Pentium 4

All of Intel's Pentium 4 CPUs include a Level 1 instruction *trace cache*. A trace cache caches *traces* of the dynamic instruction stream, so instructions that are noncontiguous in a traditional cache appear contiguous [26]. A trace is a sequence of at most n instructions and at most m basic blocks[2] starting at any point in the dynamic instruction stream. An entry in the trace cache is specified by a starting address and a sequence of up to $m-1$ branch outcomes, which describe the path followed. Intel has published the size of the trace cache in the Pentium 4 CPU family – 12K μops. However, the parameters m and n, as well as the number of μops into which x86 instructions decode, have not published.[3]

A line in the trace cache is allocated and a trace is created as instructions are fetched from the instruction cache. If the same trace is encountered again (specified by a starting address and a set of predicted branch outcomes), it will be available in the trace cache and fed directly to the execution engine. If a trace is not available, instruction fetching proceeds normally from the instruction cache.

The Netburst Microarchitecture of the Intel Pentium 4 family includes a trace cache with consistent specifications across all currently-produced Pentium 4 CPUs. While a full description of this architecture is beyond the scope of this paper, it is necessary to summarize some relevant concepts. Elements of the microarchitecture include an execution trace cache, an out-of-order core, and a rapid execution engine [4].

The trace cache stores instructions in the form of decoded μops rather than in the form of raw bytes which are stored in more conventional instruction caches. This facilitates removal of the instruction decode logic from the main execution loop, enabling the out-of-order core to schedule multiple μops to the rapid execution engine in a single clock cycle. In the case of arithmetic instructions without data hazards, it is possible to retire three μops every clock cycle. Register-to-register x86 arithmetic instructions (e.g., `add`, `sub`, `and`, `or`, `xor`, `mov`) decode into a single μop. Thus, it is possible to attain a Cycles-Per-Instruction (CPI) rate of $\frac{1}{3}$.

Other x86 instructions – especially sensitive instructions – decode to hundreds or even thousands of μops. These expensive instructions are decoded via lookup in a *Microcode ROM*; the trace cache does not fully support caching of these expensive operations.

4.2.2 Detailed Design

Our basic VMM detection strategy is to construct code that yields predictable behavior by the trace cache. Context switches will alter the content of the trace cache, which is detectable based on the resulting performance impact. Note that since the trace cache operates on decoded μops, there is no way for the VMM to efficiently simulate the trace cache. It would have to emulate every instruction, even register-to-register arithmetic instructions.

To detect the presence of a VMM, however, it is necessary that we understand the behavior of the trace cache for some subset of the CPU's full instruction set. This way,

[2] A *basic block* is a sequence of instructions without any jumps.

[3] Intel's primary motivation for keeping this information secret is not to inhibit competitors; rather, it is to prevent developers from writing software which is dependent on the existence of the trace cache. Ideologically, this is similar to information hiding in an object-oriented programming paradigm. It is interesting that this project, where we emulate the behavior of a malware writer, gives us cause to write trace cache-dependent code.

we can make statements about the expected behavior of a piece of code constructed from the subset of instructions.

Before executing the block of arithmetic instructions, we disable interrupts so that the trace cache will not be polluted by spurious interrupts. If our code is executing on the bare hardware, interrupts are truly disabled (except non-maskable interrupts and exceptions, which are either rare and can be detected, or can be avoided by careful coding), and the arithmetic instructions can execute with the full benefit of the trace cache. If the block of instructions are running on top of a VMM, then the VMM will disable interrupts only for the VM which requested them to be disabled. Thus, the trace cache will be polluted when the VMM context switches between other VMs and itself.

More significantly, sensitive instructions which trap to the VMM for emulation will incur a very large performance overhead. We have written code and run experiments which thoroughly explore these conditions. Our implementation and evaluation are presented in the coming sections.

4.2.3 Implementation

Our implementation proceeded through several phases: (1) determine the expected behavior of the trace cache with large basic blocks of arithmetic instructions; (2) determine the effects of adding small numbers of sensitive instructions to the code blocks; and (3) determine how these effects differ when the experiment is run on top of a VMM which must execute a significant number of instructions to emulate the behavior of the sensitive instruction.

Figure 2(a) shows a simplified version of the instruction sequences we executed to explore the trace cache behavior. Essentially, we take a reading from the performance counter (`rdtsc`: Read Time-Stamp Counter), execute a large basic block of arithmetic instructions, and then take another reading from the performance counter. The timestamp counter is incremented every clock cycle, but it is not a serializing instruction. That is, it does not necessarily wait until all previous instructions have finished executing before reading the counter. Similarly, instructions following the `rdtsc` may begin execution before the read operation is performed. For this reason, we iterate through the loop of arithmetic instructions 131072 (2^{17}) times, and divide by 131072 to get an actual cycle reading (the y-axis in Figure 2(b) and subsequent figures).

Figure 2(b) shows the elapsed CPU cycles for the execution of varying numbers of arithmetic instructions on both Xen and Linux. Since the instructions are not sensitive, and do not require a trap to the VMM, the performance on Xen and vanilla Linux is comparable. The most significant feature in Figure 2(b) is the existence of the knee just before 12K instructions. Prior to 12K instructions, the arithmetic instructions (which decode to one μop each) inside the loop fit entirely inside the trace cache. Note that the graph shows a CPI of $\frac{1}{3}$ before the knee, which is inline with the specifications of the Pentium 4 CPU (execution of three arithmetic μops per clock cycle in the absence of data hazards). Above 12K instructions, performance degrades significantly, and the code executes with a CPI of 1. This is because the trace cache is not designed for basic blocks this large and becomes entirely useless.

In the next section, we will show how the performance on Xen and vanilla Linux diverge significantly upon the introduction of sensitive instructions.

4.2.4 Evaluation

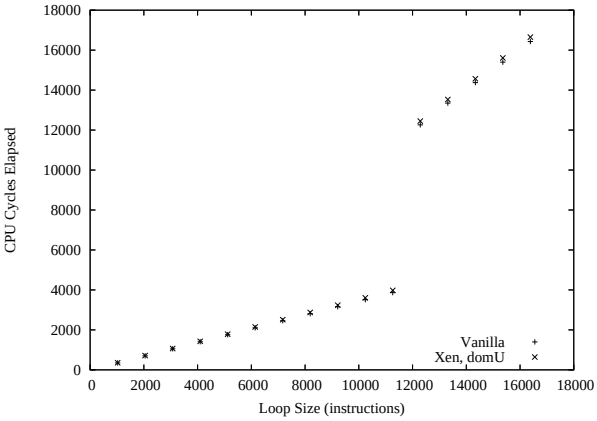
Our experiments focus on the presence of 1, 2, 4, 8, and 16 CLI (Clear Interrupt Flag – disable interrupts) instructions in a loop of thousands of arithmetic instructions. The CLI instruction is a sensitive instruction which results in a trap to the VMM if it is executed from within a VM. Figure 3(a) shows the impact of adding just a single CLI instruction at the beginning of a loop of arithmetic instructions. Already, the performance impact on Xen is considerable. Compare this with Figure 2(b), where the performance difference between Xen and vanilla Linux is negligible. We emphasize that the presence of a single sensitive instruction is sufficient to cause the disparity. The next four graphs in Figure 3 show the impact of 2, 4, 8, and 16 CLI instructions in the loop of arithmetic instructions. These CLI instructions are inserted evenly spaced throughout the loop of arithmetic instructions. For example, with two CLI instructions in a loop of 10240 arithmetic instructions, instructions 1 and 5120 will be CLI instructions.

In our experiments, we concentrated on the CLI instruction as the sensitive instruction in our experiments. There are other sensitive instructions which will produce similar behavior running on top of a VMM. Figure **??** shows the impact of a single `mov %cr0, %esi` instruction, which reads the current value of Processor Control Register 0 into general-purpose register `esi`. Note that Figure **??** is very similar to 3(a). As the attacker, we can take our pick from among the sensitive instructions to construct a code sequence that will be very difficult for the VMM to tamper with or otherwise modify in an attempt to fool our VMM detection.

One interesting artifact in Figures 3(a) and **??** is the two vanilla Linux data points which lie on the slope where the CPI is 1, when the trace cache behavior model we have presented thus far suggests that they fall on the slope where the CPI is $\frac{1}{3}$. These outliers occur with exactly 10784 and 11168 arithmetic instructions with one sensitive instruction, regardless of whether that instruction is a CLI or MOVCR0. Similar outlying points occur with greater frequency as the number of sensitive instructions in the loop of arithmetic instructions is increased. Intel has not

```
        rdtsc                ;; get start time
        mov $131072, %edi    ;; n = 131072
loop:   xorl %eax, %eax      ;; begin special
        addl %ebx, %ebx      ;; instr. seq.
        movl %ecx, %ecx
        orl %edx, %edx
        ...                  ;; 1K – 16K instr.
        sub $1, %edi         ;; n = n - 1
        jnz loop             ;; until n = 0
        rdtsc                ;; get end time
```

(a) Simplified assembly code used to fill trace cache with arithmetic instruction sequences requiring 1 μop per instruction. Note that the inclusion of 1 or more sensitive instructions occurs between varying numbers of blocks of the 4 arithmetic instructions.



(b) Average CPU cycles elapsed on a 2.0 GHz P4 for each iteration of a loop of varying numbers of arithmetic instructions on vanilla Linux and Linux on top of Xen 3.0.0. Arithmetic instruction block sizes run from 1024 to 16384 in increments of 1024. The 12 KB trace cache is plainly visible running vanilla Linux or Xen.

Figure 2: ASM code example and resulting execution graphs for increasing-sized blocks of arithmetic instructions on the Intel Pentium 4. The arithmetic instructions decode to a single μop on the Intel Pentium 4 Netburst Microarchitecture [4].

publicly released a detailed specification of the trace cache characteristics inside the Pentium 4 CPU, but we have a hypothesis for this behavior. As discussed in Section 4.2.1, the trace cache is composed of multiple cache lines, which can be strung together based on the length of a trace with $m - 1$ or fewer branches. However, the trace cache lines are of a finite size, and certain numbers of instructions inside the loop may not fit cleanly into the trace cache lines. Since the traces we create are unusually long, this misalignment results in the loop not fitting in the trace cache, despite the total number of μops being less than the absolute capacity of the trace cache. We believe that additional experimentation and reverse-engineering effort will make it possible to exactly determine the trace cache parameters, and we plan to pursue this as part of our future work.

Figure 4 shows the impact of the position of a single CLI instruction in the loop of arithmetic instructions. This figure shows the effect of the CLI placed in the middle (Figure 4(a)) and end (Figure 4(b)) of the loop of arithmetic instructions. Figure 3(a) shows the case where the CLI is at the beginning of the loop. The fewest outliers occur when the CLI instruction is at the end of the loop. Our hypothesis for this behavior is that the CLI instructions cannot be placed into the trace cache, as they decode into too many μops (on the order of hundreds of μops). Thus, the negatively impact the trace cache alignment when placed at the beginning and middle of the loop.

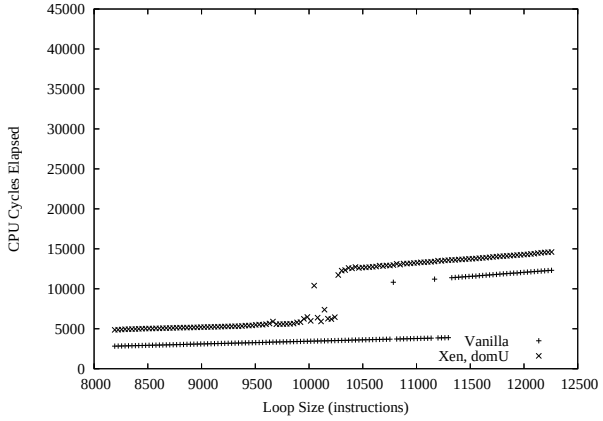
Figure 5 shows the results of our code when run on other architectures (64-bit Pentium 4 – Figure 5(a), and 32-bit Pentium M – Figure 5(b)). These results are consistent with our expectations. That is, even 64-bit Pentium 4 CPUs are built with a trace cache, and its behavior is consistent with the trace cache in its 32-bit counterparts. The periodic increase in CPI provides further evidence that alignment inside the trace cache impacts our results. The Pentium M does not have a trace cache at all, and so the instructions specially crafted to bring out trace cache behavior do not look interesting on the Pentium M.

4.3 Countermeasures

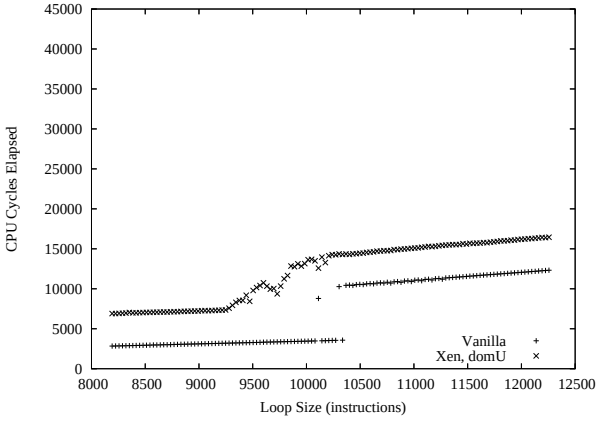
We have shown in the previous sections that it is possible to craft code which has pathological performance on a VMM, while still executing efficiently on the native hardware. This discrepancy is a threat, as it provides an avenue through which malware writers can make their code more resistant to dynamic analysis. In this section, we briefly describe possible countermeasures that may thwart our detection attack.

If a VMM is able to detect the detection attack, either before or during its execution, a number of countermeasures are possible. If the VMM realizes it is under a detection attack prior to the execution of the benchmarking loop, it may be able to use static analysis and binary rewriting to prevent the detection attack from executing correctly, perhaps returning a valid response in the correct amount of time for a nonvirtualized host. Similiarly, if the VMM recognizes the attack during the execution of the benchmarking loop, it may be able to employ dynamic analysis to predict the expected result of the detection attack. Although these countermeasures are currently beyond the scope of our detection attack because of the untampered execution assumption, we plan on extending our work to remove such an assumption. Thus, these countermeasures present possible hurdles that we need to overcome in our future work.

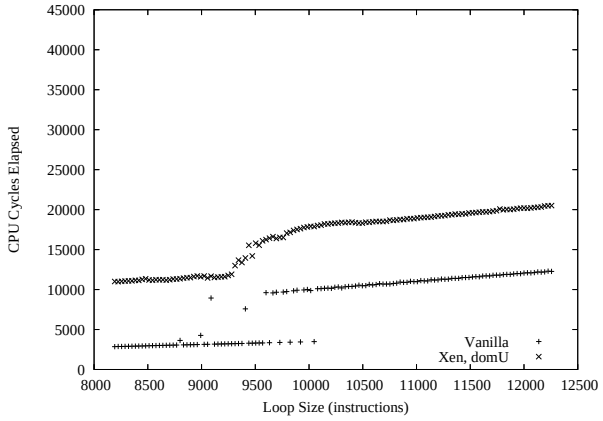
There are a number of difficulties inherent in imple-



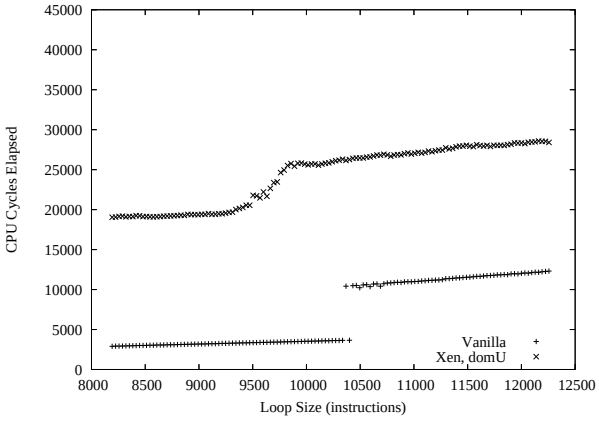
(a) One `cli` (Clear Interrupt Flag) instruction.



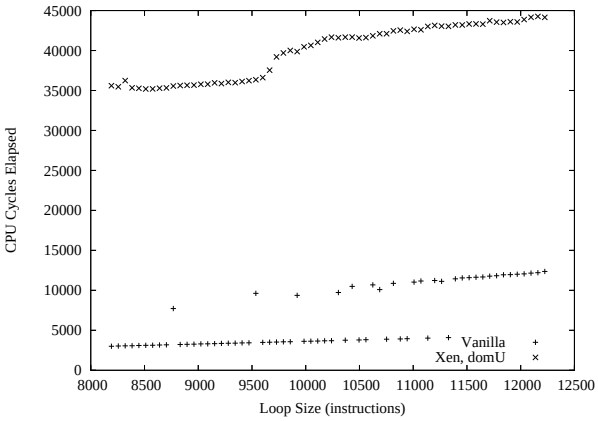
(b) Two `cli` instructions.



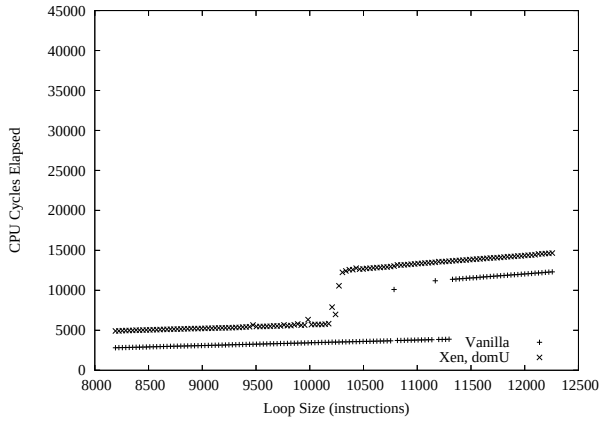
(c) Four `cli` instructions.



(d) Eight `cli` instructions.

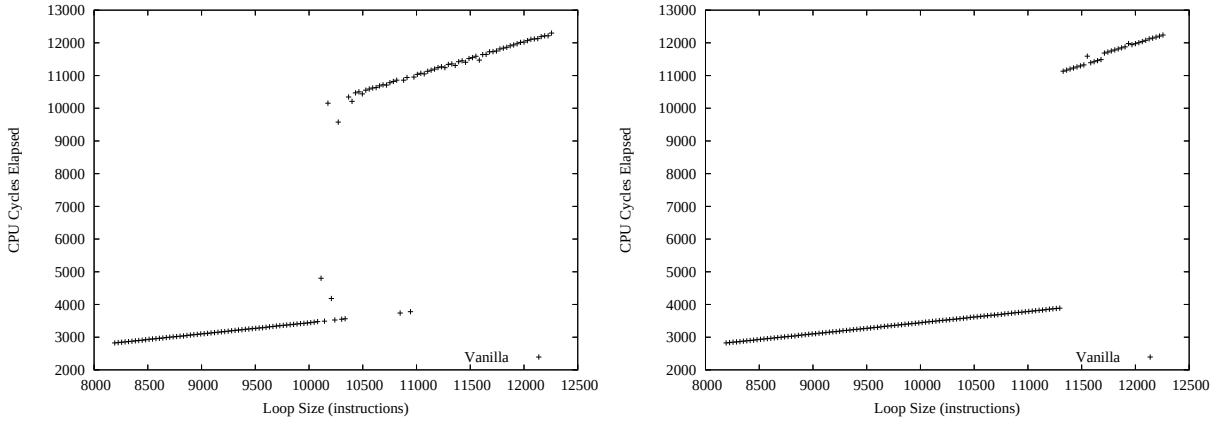


(e) Sixteen `cli` instructions.



(f) One `mov %cr0, %esi` (read Processor Control Register 0) instruction. Note the similarity with Figure 3(a).

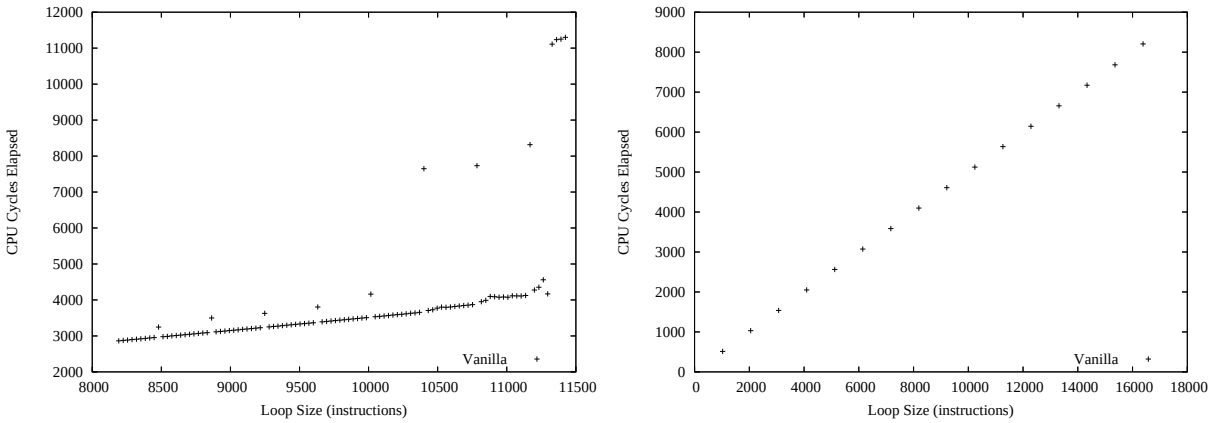
Figure 3: Xen vs. Vanilla Linux for varying numbers of sensitive instructions. The graphs show average CPU cycles elapsed for each iteration of a loop of varying numbers of arithmetic instructions with a small number of sensitive instructions. Each subfigure represents a different number of sensitive instructions inside the loop. The first sensitive instruction occurs at the beginning of the loop, and subsequent sensitive instructions (where applicable) are evenly spaced in the remainder of the loop.



(a) 1 CLI instruction in the middle of the ASM loop.

(b) 1 CLI instruction at the end of the ASM loop.

Figure 4: Effects of CLI instruction placement in the ASM loop on Vanilla Linux.



(a) The trace cache on the 64 bit Pentium 4 is visible.

(b) The complete lack of a trace cache on the Pentium M is visible.

Figure 5: Other popular Intel architectures.

menting these countermeasures. First, the VMM will need to employ sophisticated intrusion detection system (IDS) algorithms to determine if a particular piece of code is an instance of a detection attack. These IDS algorithms will likely have false positives which could result in the DoS of benign applications. Even if a VMM is able to detect a detection attack, static analysis can be prevented with code obfuscation techniques [19]. In addition, our detection attacks could be designed so that tampering by way of dynamic analysis is noticeable to the external verifier [28].

Finally, these countermeasures will need to be efficient and have a minimal impact on the execution of software within a VMM. Companies who develop VMMs for general use would likely not be willing to take a significant performance hit during the execution of all applications in order to satisfy the concerns of the security community. Much like the example of the VMM DoS-ing a benign application, there are a number of risks and complexities inherent in deploying countermeasures inside a VMM. These risks and complexities are likely to be a major hurdle for companies who develop VMMs to develop and deploy VMM detection attack countermeasures.

5 Future Work

5.1 Cryptographic Proofs of Work

The following section details a hypothesis we formulated early in this work, which we believe still has potential. It may be possible to detect the presence of a VMM using cryptographic proofs-of-work [17] to profile machines.

In a proof-of-work protocol, a client demonstrates to a server that it has invested a certain amount of resources by providing a short certificate whose generation necessitated the expenditure of resources. One example of a CPU-bound proof-of-work is finding hash collisions for truncated cryptographic hash functions.

We postulate that using proofs-of-work to profile a system can be done in a fashion that makes it difficult for a

VMM to mask the differences in timing and memory that exist between the virtualized and nonvirtualized architectures. A naive strategy for hiding the existence of a VMM is using a VMM running on a fast machine to present the appearance of a slower machine. However, accurately representing the timing and memory behavior (recall Section 2) of a particular machine configuration is non-trivial. Accurate behavior emulation will require significant slowdown, which will violate the efficiency property from Section 2. Thus, we believe proof-of-work schemes have promise.

Proof-of-work protocols are partitioned into different classes based on their required workloads. Three interesting classes are CPU-bound, memory-bound [1, 5], and disk-bound proofs. CPU-bound proofs exercise the capabilities of the CPU, drawing instructions primarily from caches (as opposed to frequent memory accesses). In memory-bound proofs, the server constructs a workload that requires near constant memory accesses. This in turn makes the resulting computation time dominated by the time spent accessing memory. Analogously, disk-bound proofs require near constant access to disk.

Memory- and disk-bound proofs force different workloads on the client as a way to marginalize the disparity between client capabilities. For detecting the existence of a VMM, memory- and disk-bound workloads serves as a control mechanism to minimize the profile variance between devices. A proof-of-work which is strictly based on the CPU speed of a client will have an exponentially large difference between machines as predicted by Moore's Law, but disk or memory bound workloads will have significantly smaller variance (e.g., memory and disk *capacity* has been growing in accordance with Moore's Law, but memory and disk *performance* has not).

By timing the execution of a proof-of-work benchmark, we may be able to determine if the benchmark was performed by a resource-constrained virtual machine rather than a nonvirtualized host.

There are a number of obstacles which must be overcome for a proof-of-work protocol to serve as a useful VMM detection attack. The first of these obstacles is the construction of proofs-of-work whose runtimes allow a remote party to accurately profile a host. This problem may be solvable by testing a number of host configurations and evaluating differences between them. Another potential obstacle is the construction of a threshold metric that allows one to differentiate between virtual machines and nonvirtualized hosts simply by comparing their benchmarks.

The resource costs incurred by proof-of-work schemes and required by a change in the number of virtual machines which can run on a single hardware platform can serve as a useful metric. This metric will allow us to analyze the monetary costs of defending against detection attacks. A cost-benefit model which utilizes this economic metric may allow us to develop virtual machine detection attacks whose defense poses a substantial monetary cost.

5.2 Detection of OS & VMM Scheduler Interference

One VMM detection strategy we considered developing during the early stages of this work exploits the existence of multiple scheduler quanta. A VMM multiplexes operating system instances with some time quantum for each, and each operating system instance multiplexes application threads with some time quantum for each. The selection of appropriate time quanta is known to have a significant impact on overall system performance. For example, Hensbergen shows that virtualization can have a negative impact on system performance by increasing the severity of OS scheduler interference [12]. If we can exploit unintended interactions between VMM and OS quanta to detect the existence of the VMM, defenses may have a severe penalty in terms of system performance.

The scheduler interference strategy shares many common ideas with research on covert channels. For example, Gligor explored the use of CPU scheduling to transmit data between two processes [11]. The sender of information varies the nonzero CPU time, which it uses during each quantum allocated to it, to send different symbols. For 0 and 1 transmissions, the sender picks two nonzero values for the CPU time used during a quantum, one representing a 0 and the other a 1. Huskamp refers to this method of communication as the *quantum-time channel* [15].

The work of Huskamp [15] and Gligor [11] provides detailed information about using shared hardware for timing channels. We believe it is feasible to use these same concepts to pass a message between different VMs running on the same VMM. If a VM receives a message, it can echo the received message and the sending VM will know that it is running on virtualized rather than real hardware.

The primary hypothesis for the interference-based approach is that differences in the execution pattern of the quanta app on VMM and non-VMM systems will be evident with the appropriate filtering. It will be necessary to select an appropriate filtering scheme, run it on the data output by an application designed to interact with scheduler quanta (the *quanta app*), and compare the results across VMM and non-VMM systems. It is likely that a detection scheme such as this will have false positive and false negative rates, which will need to be analyzed.

The default scheduler quanta for the Linux kernel changes infrequently. We hypothesize that we can accurately profile the expected behavior of the quanta app on a non-VMM system. The quanta app shall be equipped with this knowledge when it is deployed to hosts suspected of

running a VMM, so that it is better able to make decisions about the existence of a VMM.

We hypothesize that the actual detection criteria for the quanta app will be non-trivial. Signal processing techniques that have been applied in the domain of intrusion detection may be able to help identify suspicious interlocking module behavior. For example, Barford et al. have applied wavelet analysis to network traffic anomalies [2].

6 Related Work

Most related work emphasizes software-dependent, hardware-independent detection attacks which are usually possible to counter through modifications to the VMM implementation. Our detection attacks are software-independent and hardware-dependent. While the practicality and even ability of organizations who develop VMMs modifying VMM implementations to quickly counter software-dependent, hardware-independent attacks can be argued, we believe that organizations and individuals who use VMMs for security critical applications would likely modify the VMM implementation to thwart software-dependent attacks, especially when trivial countermeasures exists. Our software-independent, hardware-dependent attacks should be more difficult to counter without hardware modification, a task which we assume is difficult for security conscious organizations who rely on commodity hardware.

Holz and Raynal describe some heuristics for detecting honeypots and other suspicious environments from within code executing in said environment [13]. Dornseif et al. study mechanisms [9] designed specifically to detect the Sebek high-interaction honeypot. While many of these software-dependent heuristics suffice today, trivial countermeasures exist. The mechanisms we have constructed are not based upon specific software implementations.

Execution path analysis (EPA) [27] was first proposed in Phrack 59 by Jan Rutkowski as an attempt to determine the presence of kernel rootkits by analyzing the number of certain system calls. Although the main idea can also apply to detect VMMs, EPA has several severe drawbacks. The main drawback is that it requires significant modification to the system (debug registers, debug exception handler) that could be easily detected and consequently forged by the underlying VMM.

Delalleau proposed a scheme to detect the existence of a VMM [6] by using timing analysis. The proposed scheme requires a program to first time its own execution on a VMM-free machine in a learning phase. Then, when the program infects a suspect host, its execution time could be compared against the results from the learning phase. However, because the result of the learning phase is dependent on a particular machine configuration and the scheme is not designed to prevent tampering, it is unclear how practical it is to deploy such a detection attack in the wild.

Pioneer [28] is a primitive which enables verifiable code execution on remote machines. As part of the inherent challenge of verifiable code execution, Pioneer needs to determine whether or not it is running inside a VMM. The solution in Pioneer is to time the runtime of a certain function that also reads in the interrupt enable bit in the EFLAGS register. This function is pushed into the kernel and is expected to run with interrupts turned off. However, if it was running inside a VMM, the output of the EFLAGS register would be different than expected. Although promising, Pioneer assumes that the external verifier knows the exact hardware configuration of the target host. We eliminate this assumption and rely on machine interrogation and benchmarking to discover and verify the target host's hardware configuration.

Vrable et al. touch briefly on non-trivial mechanisms for detecting execution within a VMM [35]. They allude to the fact that although a honeynet maybe be able to perfectly virtualize all hardware, an attacker may be able to infer that it is executing inside a VMM by certain side channels.

There are a number of previously developed software-dependent techniques from the blackhat community, two of which we describe next.

Redpill Redpill[4] is an example detection attack developed to detect the VMWare virtual machine monitor. Redpill operates by reading the address of the Interrupt Descriptor Table (IDT) with the `SIDT` instruction and checking if it has been moved to certain locations known to be used by VMWare. This attack can be easily countered by moving the IDT away from these "known" locations. Hence, we consider Redpill to be a software-dependent attack (it only applies to VMWare) with a trivial countermeasure (the IDT can be relocated at load-time).

VMWare's Back Similar to Redpill, VMWare's Back[5] is a software-dependent detection attack which uses the existence of a special I/O port, called the VMWare backdoor. This I/O port is specific to the VMWare virtual machine and hence can be used to detect VMWare. In order to demonstrate the simplicity of the attack we have included the sequence of commands[6] used to call the backdoor function:

```
MOV EAX, 564D5868h ; Magic Number
MOV EBX, COMMAND_SPECIFIC_PARAMETER
MOV ECX, BACKDOOR_COMMAND_NUMBER
MOV DX, 5658h ; Port Number
IN EAX, DX
```

Depending on the value of the `BACKDOOR_COMMAND_NUMBER`, this code is capa-

[4] `http://invisiblethings.org/redpill.html`
[5] `http://chitchat.at.infoseek.co.jp/vmware/`
[6] `http://chitchat.at.infoseek.co.jp/vmware/backdoor.html`

ble of retrieving VMWare's version number, device information, virtual machine configuration information, and the host's system time. In addition, the above code allows data to be transfered into the virtual machine by writing to EAX, EBX, and ECX which are used as input parameters to different backdoor functions.

7 Conclusion

In this paper, we studied the ability of a program to detect the presence of a virtual machine monitor. After detecting the presence of a VMM, a malicious program can modify its behavior in a number of ways to thwart dynamic analysis. We define the problem of virtual machine monitor detection, in which a program called a detection attack executes on a remote host to determine if a VMM is resident in memory. The main contribution of this paper is the development of a detection attack whose execution differs from the perspective of an external verifier when a VMM is resident in memory (versus when it is executed directly on the underlying hardware). Our developed attacks are based on two exceptions to the equivalence property of a virtual machine monitor: timing dependencies and resource sharing. We believe that these exceptions are inherent to any virtual machine monitor and that the strategy our attacks use should be capable of detecting any virtual machine monitor. We described the design and implementation of our attacks and their success detecting the Xen virtual machine monitor without relying on specific software implementation features. We also briefly treat potential countermeasures. Most related work emphasizes software-dependent, hardware-independent detection attacks which are usually possible to counter through modifications to the VMM implementation. Our detection attacks are software-independent and hardware-dependent. Our attacks should be more difficult to counter without hardware modification, a task which we assume is difficult for security conscious organizations who rely on commodity hardware.

Most related work emphasizes software-dependent, hardware-independent detection attacks which are usually possible to counter through modifications to the VMM implementation. Our detection attacks are software-independent and hardware-dependent. Our attacks should be more difficult to counter without hardware modification, a task which we assume is difficult for security conscious organizations who rely on commodity hardware.

8 Acknowledgments

We are grateful to Garth Gibson and Adam Pennington for their instruction and guidance throughout this project. We also wish to thank Arvind Seshadri, Adrian Perrig, Leendert van Doorn, and Michael Kozuch for their insightful comments and useful discussions.

References

[1] M. Abadi, M. Burrows, M. Manasse, and T. Wobber. Moderately hard, memory-bound functions. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium (NDSS)*, February 2003.

[2] Paul Barford, Jeffery Kline, David Plonka, and Amos Ron. A signal analysis of network traffic anomalies. In *Proceedings of the ACM SIGCOMM Internet Measurement Workshop*, 2002.

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, 2003.

[4] D. Boggs, A. Baktha, J. Hawkins, D.T. Marr, J. A. Miller, P. Roussel, Singhal R, B. Toll, and K. S. Venkatraman. The microarchitecture of the Intel Pentium 4 processor on 90nm technology. *Intel Technology Journal*, 8(1), February 2004.

[5] M. Naor C. Dwork, A. Goldberg. On memory-bound functions for fighting spam. In *Advances in Cryptology*, 1992.

[6] G. Delalleau. Mesure locale des temps d'execution: application au controle d'integrite et au fingerprinting. In *Proceedings of SSTIC*, 2004.

[7] Advanced Micro Devices. AMD64 virtualization: Secure virtual machine architecture reference manual. AMD Publication no. 33047 rev. 3.01, May 2005.

[8] Martim d'Orey Posser de Andrade Carbone and Paulo Licio de Geus. A mechanism for automatic digital evidence collection on high-interaction honeypots. In *Proceedings of the IEEE Workshop on Information Assurance and Security*, June 2004.

[9] Maximillian Dornseif, Thorsten Holz, and Christian Klein. Nosebreak - attacking honeynets. In *Proceedings of the 2004 IEEE Information Assurance Workshop*, June 2004.

[10] G. Dunlap, S. King, S. Cinar, M. Basrai, and P. Chen. Revirt: Enabling intrusion anlysis through virtual- machine logging and replay. In *ACM SIGOPS Operating Systems Review*, 2002.

[11] Virgil D. Gligor. A guide to understanding covert channel analysis of trusted systems. Technical Report NCSC-TG-030, National Computer Security Center, November 1993.

[12] Eric Van Hensbergen. The effect of virtualization on OS interference. In *Proceedings of the Workshop on Operating System Interference in High Performance Applications*, September 2005.

[13] Thorsten Holz and Frederic Raynal. Detecting honeypots and other suspicious environments. In *Proceedings of the IEEE Workshop on Information Assurance and Security*, June 2005.

[14] Honeynet Research Alliance. The honeynet project. Available at: `http://www.honeynet.org/`, October 2005.

[15] J. C. Huskamp. *Covert Communication Channels in Timesharing Systems*. PhD thesis, University of California, Berkeley, 1978. Technical Report UCB-CS-78-02.

[16] Intel Corporation. Intel virtualization technology. Available at: `http://www.intel.com/technology/computing/vptech/`, October 2005.

[17] Markus Jakobsson and Ari Juels. Proofs of work and bread pudding protocols. In *Proceedings of Communications and Multimedia Security*, 1999.

[18] Xuxian Jiang, Dongyan Xu, Helen J. Wang, and Eugene H. Spafford. Virtual playgrounds for worm behavior investigation. In *8th International Symposium on Recent Advances in Intrusion Detection (RAID '05)*, 2005.

[19] C. Linn and S. Debray. Obfuscation of executable code to imporove resistance to static disassembly. In *10th ACM Conference on Computer and Communications Security (CCS)*, 2003.

[20] R.C. Merkle. Secure communications over insecure channels. In *Communications of the ACM*, volume 21, April 1978.

[21] Gerald J. Popek and Robert P. Goldberg. Formal requirements for

virtualizable third generation architectures. *Communications of the ACM*, 17, July 1974.

[22] Niels Provos. A virtual honeypot framework. In *Proceedings of the USENIX Security Symposium*, August 2004.

[23] Niels Provos. Honeyd virtual honeypot. Available at: `http://www.honeyd.org/`, October 2005.

[24] J. S. Robin and C. E. Irvine. Analysis of the intel pentium's ability to support a secure virtual machine monitor. In *Proceedings of the USENIX Security Symposium*, 2000.

[25] Robert Rose. Survey of system virtualization techniques. Available at: `http://www.robertwrose.com/vita/rose-virtualization.pdf`, March 2004.

[26] E. Rotenberg, S. Bennett, and J. E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, November 1996.

[27] Jan Rutkowski. Execution path analysis: finding kernel rootkits. *Phrack*, 11(59), July 2002.

[28] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. VanDoorn, and P. Khosla. Pioneer: Verifying integrity and guaranteeing execution of code on legacy platforms. In *Proceedings of the Symposium on Operating Systems Principals (SOSP)*, 2005.

[29] L. Spitzner. *Honeypots: Tracking Hackers*. Addison-Wesley, Boston, MA, 2002.

[30] L. Spitzner. Honeypots: Definitions and values. Available at: `http://www.tracking-hackers.com/papers/honeypots.html`, October 2005.

[31] Stuart Staniford, Vern Paxson, and Nicholas Weaver. How to 0wn the internet in your spare time. In *Proceedings of the 11th USENIX Security Symposium (Security '02)*, 2002.

[32] The Honeynet Project. Sebek. Available at: `http://www.honeynet.org/tools/sebek/`, October 2005.

[33] G. Venkitachalam and B. Lim. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *USENIX Technical Conference*, 2001.

[34] VMWare. VMWare Workstation. Available at: `http://www.vmware.com/`, October 2005.

[35] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Scalability, fidelity and containment in the potemkin virtual honeyfarm. In *Proceedings of the Symposium on Operating Systems Principals (SOSP)*, 2005.

First-Class Transactions for O'Caml

Rebecca Hutchinson (rah@cs)
Neelakantan Krishnaswami (neelk@cs)
Ruy Ley-Wild (rleywild@cs) William Lovas (wlovas@cs)

December 15, 2005

Abstract

This paper describes 712Caml, a software transactional memory library for O'Caml. 712Caml provides first-class transactions as the only synchronization primitive, functions for control flow among transactions, and transactional shared variables. We evaluate the performance and scalability of 712Caml and contrast our design choices by comparing against AtomCaml [10].

1 Introduction

Software transactional memory (STM) systems provide transactions as a general-purpose synchronization primitive with the guarantee that shared variable modifications within a transaction appear as atomic updates to global state. They borrow from the database community the idea of a *transaction*: each critical region is executed speculatively, but if any contention occurs the transaction is rolled back and retried. Thus, critical regions appear to run atomically without interference from any concurrent threads.

712Caml is a software transactional memory library for O'Caml that provides first-class transactions, functions for manipulating control flow between transactions, and transactional shared variables. In contrast to AtomCaml's [10] compiler support for implicit transactions, 712Caml follows STM Haskell's [5] design as a library that provides transactions as first-class data objects. The key differences and tradeoffs between the design choices of 712Caml and AtomCaml include transaction interface, assumptions, levels of optimism, and strength of atomicity.

AtomCaml is a modified O'Caml compiler that makes transactions implicit by exposing a single function `atomic` that runs a piece of code atomically; 712Caml is an O'Caml library that makes transactions explicit as first-class data objects and provides functions for manipulating transactions. This has deep implications in terms of expressivity and implementation. First-class transactions are easier to manipulate and track because transactional regions are clearly identified, whereas AtomCaml has to generates two versions of each function:

one ordinary version and one that can be run in an atomic context; the version used is chosen at runtime depending on whether execution occurs inside a transaction.

AtomCaml operates under the assumption that atomic blocks are always short and guarantees atomicity by requiring transactions to complete in one scheduler quantum. The problem with this assumption is that the definition "short" varies with the choice of computer system, operating system and thread library scheduler, since different configurations may have different scheduler quanta. This assumption precludes the need to consider the amount of contention, since interleaved execution of concurrent threads is necessary for there to be any contention at all. In contrast, 712Caml does not make any assumptions about the duration of transactions, but instead optimistically assumes that there is low contention for access to transactional shared memory among threads. Contention is dealt with by retrying the transaction if it is unable to commit.

The two systems also differ in their levels of optimism. AtomCaml pessimistically rolls back any atomic thread whose execution is preempted, since this might lead to a violation of atomicity guarantees. On the other hand, 712Caml optimistically permits interleaved execution of transactions and only rolls back when contention is detected. An important tradeoff is precisely when to detect contention. The "lazy" approach we take is waiting until a transaction is ready to commit and then checking that the transaction's original view of memory is consistent with the current state. One drawback is that long-running transactions may waste processor time if they conflict with other threads and have to perform large rollbacks. Like AtomCaml, we alleviate this problem by adjusting the scheduler quantum for long-running transactions that repeatedly fail to complete. Alternatively, we could look for contention on every transactional memory access, paying a small up-front price often with the hope that we might prevent large rollbacks early.

A fourth dimension in which AtomCaml differs from 712Caml is in strength of the atomicity guarantee [1]. Weak atomicity means that transactions can overlap with non-atomic threads. Strong atomicity means that transactions must appear truly atomic: they cannot overlap even with other non-atomic threads. Strong atomicity can lead to deadlock, since transactional code has no way of synchronizing with non-transactional code. 712Caml provides weak atomicity, which allows for more flexibility in program design. AtomCaml implements strong atomicity by construction, since it disallows interleaved execution of transactions.

The rest of this paper is organized as follows. In Section 2, we describe our system design. Section 3 discusses the benchmarks we used to evaluate 712Caml. In Section 4 we discuss our hypotheses, benchmarks, and results. Section 5 describes related work, and Section 6 concludes.

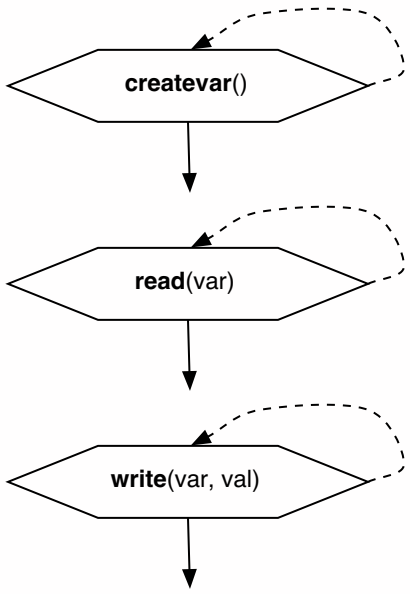


Figure 1: Unitary transactions

2 System Design

2.1 Interface

712Caml applies Kung and Robinson's [8] proposal for database transactions to O'Caml by providing a library for software transactional memory with minimal modifications to the runtime system. The library provides facilities for executing transactions atomically, combining transactions and handling shared variables. The runtime needs to be modified to support a custom scheduling policy to provide fairness for concurrent transactions.

Instead of demarcating transactions with `tbegin` and `tend` like Kung, we build transactions incrementally using a set of transaction combinators. There are unitary transactions that create, read from, and write to shared variables (Figure 1). Larger transactions can be made either by sequencing two transactions, where the second may depend upon the result of the first (the `bind` operation, Figure 2), or by composing two transactions in alternation, running the second transaction only if the first fails (the `orelse` operation, Figure 3). We also provide a transaction for explicit failure (the `retry` operation, Figure 4), which forces a transaction to rollback and attempt to re-execute later, in case some precondition was not met, for example.

Once created, transactions can be executed atomically with exactly-once semantics using the `atomic` operation. Moreover, in order to facilitate compositionality of transactional code, we permit nested transactions but handle their

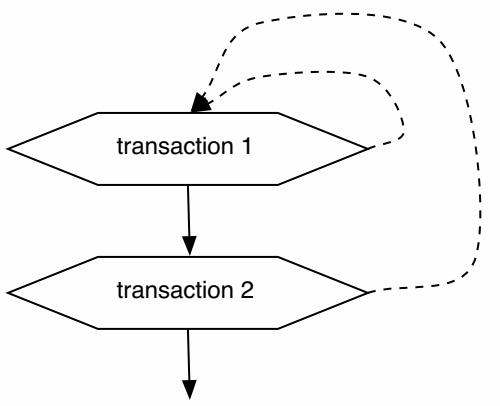


Figure 2: Transaction sequencing

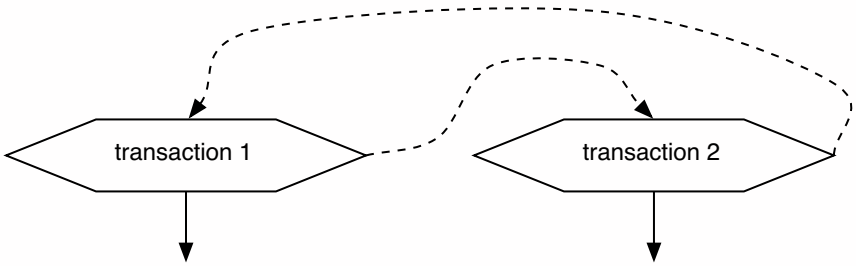


Figure 3: Transaction alternation

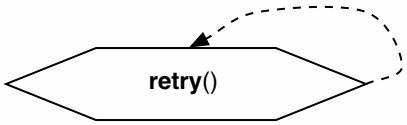


Figure 4: Explicit failure

completion according to their depth. A nested (inner) transaction completes by merging its log with its parent's log, while an unnested (outer) transaction completes by committing its write set globally provided its read set is consistent with the global state at completion time.

Similar to Kung's transactional object management facilities, we provide transactional shared memory for arbitrary data types, with the programming convention that using non-transactional shared memory with transactional shared memory may void the serializability guarantee for concurrent transactions. Although transactional variables are the only transactional shared memory primitive we provide, other transactional data structures can easily be built on top of this foundation. For example, transactional arrays can be implemented as immutable vectors of transactional variables. Since O'Caml is a garbage-collected language, we do not need to handle memory deallocation explicitly.

2.2 Implementation

712Caml is divided into three primary modules: the transaction library, transactional variables, and a transaction log. In addition, our system relies on a few small thread scheduler modifications. We describe each component of the system below.

2.2.1 Transaction Log

Each transaction maintains a log recording its changes locally, to be committed later. The log is a mapping from transactional variables accessed to the values they had when first accessed together with the new values with which they should be updated if the transaction succeeds.

Currently, our implementation uses a purely functional map based on a balanced binary search tree. We experimented briefly with an implementation based on an imperative hash table, but initial results indicated that an imperative log decreased performance on account of copying and allocation. (The purely functional log enjoys the benefit of persistence—many conceptual copies of the log may share data in memory; the imperative log would require each conceptual copy to be an actual copy in memory, which can lead to poor cache performance due to sacrificed locality.)

Before committing, the old value of each transactional variable in the log is compared with the actual value in memory to ensure a consistent view of the world. The values stored in transactional variables are double-indirected, so this is a pointer equality comparison. If another transaction has written to a variable, the variable's value will be pointer-unequal to the recorded value.

In order to rollback, the log is discarded, and no permanent changes take place. In order to read a variable, the log must be consulted to see if it has changed.

2.2.2 Transaction Library

Transactions are implemented as functions from a log to a result. The `atomic` function simply applies the transaction to an initially empty log and returns the value returned by the transaction, after performing the consistency checks and commit protocol described above.

The `retry` transaction is implemented as a function that raises an exception; `atomic` handles this exception by yielding to the scheduler and later reattempting atomic execution of the transaction that called `retry`.

2.2.3 Transactional Variables

Transactional variables are implemented as doubly-indirected values—pointers to pointers to values. The first level of indirection permits us to modify the value while the second level allows us to determine memory consistency via the protocol described above.

2.2.4 Scheduler Modifications

712Caml also requires minor modifications to the scheduler in order to account for the possibility of starvation if a long-running highly contentious thread keeps getting rolled back. The modified thread scheduler provides exponential compensation by doubling the time slice given to a thread each time a transaction running in that thread rolls back, ensuring that eventually any transaction will be able to run to completion. In order to ensure fair scheduling, threads with larger time slices are scheduled proportionally less often. When a transaction completes successfully, its thread loses any special status and gets scheduled normally.

3 Evaluation

We evaluated 712Caml with a comparative analysis against AtomCaml using a set of microbenchmarks, and portions of the AtomCaml and SXM benchmarks.

3.1 Microbenchmarks

We gauged the performance and scalability of transactions with tests consisting of threads with different instruction mixes. We varied the number and duration of transactions and the proportion of transactional and idle computation to test scalability, the degree of read and write contention and the distribution of transactional computation throughout the transaction to test scheduling and conflict detection, and the number of transactional read, write, and read/write variables to test the impact on performance. We generated the test cases with a function parameterized in the number of threads, the number of transactional variables and their use by each thread, and the duration and distribution of idle computation.

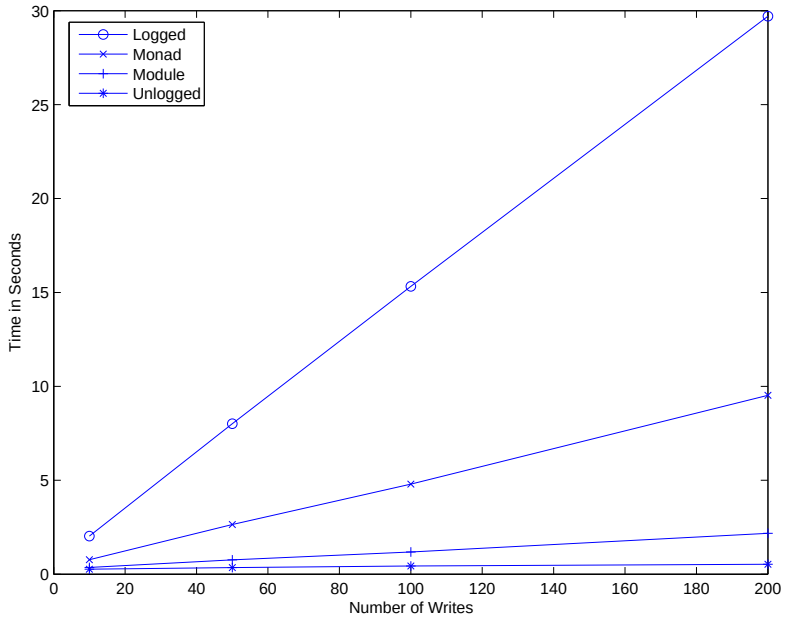


Figure 5: Nanobenchmarks: overhead of transactions

We used a homogeneous set of threads ($1, 2, 4, 8, 16, 32$ threads) executing a single transaction atomically 16 times with varying idle computation overhead (looping 0, 10, 100, $1,000$, $10,000$ times) between accesses to transactional variables. In order to test the overhead of transactions in the absence contention, we used a homogeneous transaction that performed the same operation (read or write) 32 times on 32 shared transactional variables in random order. In order to test the impact of contention on performance, we used a homogeneous transaction that alternated 32 reads and 32 writes on 32 shared transactional variables in random order.

We also simulated contention with a heterogeneous set of reader threads executing a read transaction ($10, 20, 50, 100$ reads per transaction) and writer threads executing a write transaction ($100, 50, 20, 10$ writes per transaction), each thread executed its transaction 16 times with varying idle computation overhead (looping 0, 10, 100, $1,000$, $10,000$ times) and sharing a buffer ($1,000$ entries).

3.2 Nanobenchmarks

In order to measure the overhead incurred by transactional execution, we created a set of "nanobenchmarks", so called because they are smaller in scope than our microbenchmarks. These benchmarks performed some number n of writes to a mutable variable in a loop executed 100,000 times. We tested $n = 10$, 50, 100,

and 200, under four different conditions:

- writing to a mutable transactional variable in a logged transaction (*Logged*),
- writing to an ordinary reference cell in an unlogged code fragment (*Unlogged*),
- writing to a reference cell through an abstraction boundary in an unlogged code fragment (*Module*), and
- writing to a reference cell via a monadic library with an interface similar to ours (*Monad*).

By comparing the logged execution against the unlogged executions, we can estimate the overhead our transactional library incurs. A graph showing this data appears in Figure 5.

According to the graph, 712Caml's logged execution is 120 times slower than equivalent code using ordinary O'Caml references (slope(*Logged*)/ slope(*Unlogged*) ≈ 120). Much of this overhead is likely due to code crossing module boundaries, though, since the O'Caml compiler does not perform inlining optimizations across modules. As the graph shows, we only perform about 15 times worse than equivalent reference updating code that lives in a module behind an abstraction boundary (slope(*Logged*)/ slope(*Module*) ≈ 15).

An even more likely source of overhead is our underlying representation of transactions. Since every transaction is represented as a closure and large transactions are built from many smaller ones, large transactions must build many, many closures which are eventually applied away. To determine how much of our runtime overhead is due to closure-building, we wrapped the reference writing code with a monadic interface that builds roughly the same number of closures as our transactional library, but without any logging. We perform only 3 times slower than the monadic code (slope(*Logged*)/ slope(*Monad*) ≈ 3).

The rest of our overhead is due to logging. A more clever log implementation coupled with a more clever transaction representation has the potential to increase our library's performance by several orders of magnitude.

3.3 Testing Environment

Our tests were conducted on a machine with a 3 GHz Pentium 4 processor with 512 KB of L2 cache and 1 GB of RAM, running Linux 2.4.25.

4 Results

4.1 Scalability of Rollbacks

Since 712Caml does not roll back transactions until the commit point but AtomCaml rolls back on thread preemption, we predicted that 712Caml's roll back behavior would scale more gracefully in the presence of long-running transactions.

This hypothesis was confirmed by the homogeneous read-only and write-only transactions: 712Caml incurred no rollbacks while AtomCaml started incurring hundreds of rollbacks when the busy work went from $1,000$ to $1,0000$ idle iterations between each shared variable access. This means that the duration of transactions with busy work of $10,000$ exceeded the scheduler quantum. Figures 6 and 7 show AtomCaml's number of rollbacks in contention-free transactions as the number of threads and amount of busy work varies. The number of retries is linearly proportional to the number of threads and logarithmically proportional to the amount of busy work. The logarithmic scaling is explained by AtomCaml's scheduler policy of exponential compensation. There are no figures for 712Caml because by design it does not roll back contention-free transactions.

In the presence of contention, although 712Caml incurred more roll backs early on, the number of rollbacks scaled more smoothly than AtomCaml as the amount of busy work increased; see Figures 8 and 9. 712Caml appears to scale logarithmically presumably due to exponential scheduler compensation. AtomCaml's performance on the same test exhibited a much sharper performance degradation. Since AtomCaml's performance on contention-free transactions scaled logarithmically, the best it can hope to do on transactions with contention is also logarithmic. However, even if AtomCaml achieves this lower bound, its performance degradation is still much steeper than 712Caml's, so much so that we were unable to collect data for AtomCaml with busy work of $100,000$.

4.2 Scalability of Running-Time

We tested the running-time performance of both systems with homogeneous instruction mixes. For contention-free transactions, both systems scaled linearly with the number of threads and amount of busy work. 712Caml outperformed AtomCaml by roughly a factor of between two and four in read-only and write-only transactions (Figures 10, 11, 12, and 13). However, for transactions with contention 712Caml performed only slightly better than AtomCaml (Figures 14 and 15). Some data points are absent because they took too long to collect.

4.3 Speculation: Native Code

One of the initial design considerations for 712Caml was that it would be easier to leverage the O'Caml native code compiler to produce fast standalone binaries by implementing a library instead of a compiler extension like AtomCaml. We were unable to fully explore this possibility because like AtomCaml, we modified the virtual machine thread scheduler to prevent starvation. The native code compiler uses the `pthread` library, which on our test platform is implemented using kernel threads.

In order to get a taste of the kind of performance gain we could expect from native code compilation, we ran a single comparative test of running time of an *a priori* starvation-free set of transactions. The test consisted of 16 threads performing a homogeneous read-only transaction with busy work

of $1,000$, varying the number of shared variables $(1, 2, 4, 8, 16, 32)$ each accessed 32. Although 712Caml bytecode performed better than AtomCaml bytecode, native code compilation outperformed both dramatically: its performance was approximately 35 times faster per operation than AtomCaml ($\text{slope}(\mathit{AtomCaml}) / \text{slope}(\mathit{Native}) \approx 35$). We didn't get running time for native code compilation and 32 shared variables due to a technical glitch in our testbed setup.

5 Related Work

Transactions were originally proposed as an alternative to lock-based synchronization for databases [8], but have found applications in dependable systems and as a general-purpose concurrency primitive [7]. The ACID (atomicity, consistency, isolation, durability) properties [3] of database transactions provide an abstraction for concurrency in which applications should preserve data structure consistency while databases deal with concurrent accesses and recovery from incomplete executions and system failures. In dependable systems, transactions are used as an abstraction for distributed systems in which participants may cooperate on a task or compete for resources [11]. Fault tolerance is achieved by confining the errors of a transaction with backward (rollback and retry) and forward error recovery (exception handling), although the all-or-nothing semantics may be relaxed to exactly-once or run-then-compensate.

Achieving correctness and efficiency with explicit locks is difficult because coarse-grained locks simplify correctness but limit scalability and parallelism, while fine-grained locks reduce contention but incur a greater locking overhead and may introduce data races and deadlocks. Transactions solve these difficulties by enabling optimistic concurrent execution without explicit locks. However, transactions are not strictly safer than lock-based synchronization and the semantics of atomicity between transaction and non-transaction code is subtle [1].

In addition to the AtomCaml compiler extension for O'Caml [10], there are various implementations of software transactional memory for other languages. The SXM transactional memory extension for C# [6], transactions as preemptible atomic regions for real-time Java [9], and the STM Haskell extension for Haskell [5]. Similar to 712Caml, SXM and STM Haskell feature first-class transactions and transaction combinators like `orelse`.

The 712Caml API is closely modeled after the STM Haskell API, with an abstract type denoting transactions as first-class objects and several functions for composing transactions. Unlike Haskell, we do not treat a transaction that raises an exception specially but instead as a legitimate way for a transaction to finish its computation, committing any changes upon exceptional return. This has the benefit of noticeably simplifying the semantics of transactions.

SXM also has transactional types, which are constructed using a metadata annotation on class definitions with the necessary machinery built at runtime using a proxy class which does run-time code generation. This is potentially

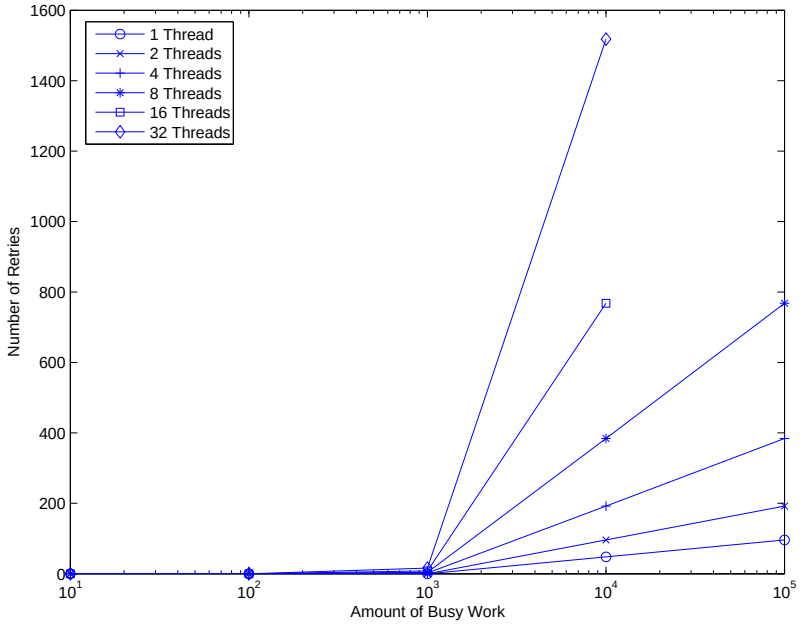


Figure 6: Number of Retries for Homogeneous Reads, AtomCaml

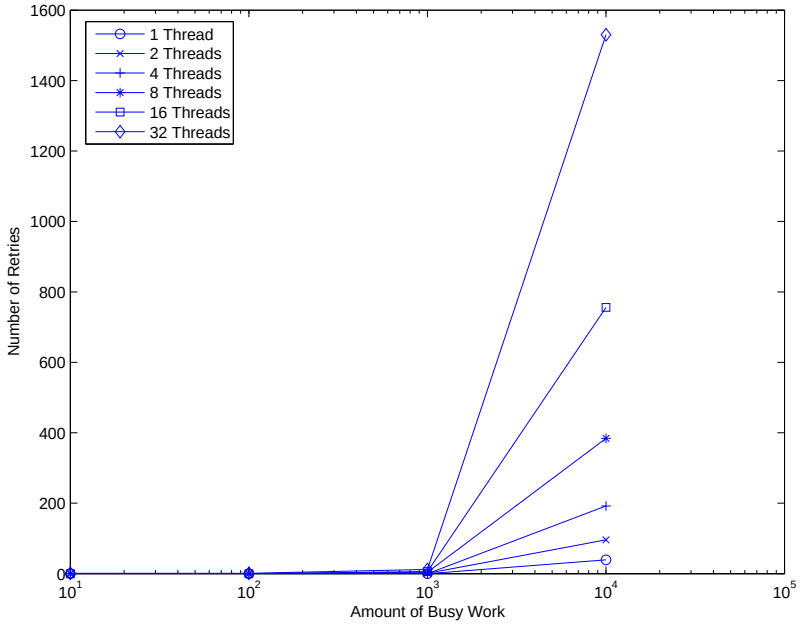


Figure 7: Number of Retries for Homogeneous Writes, AtomCaml

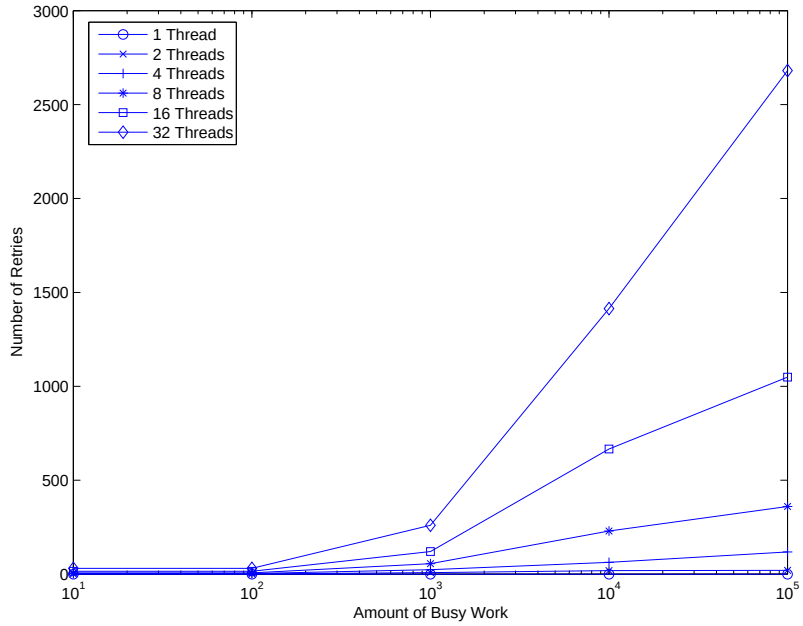


Figure 8: Number of Retries for Homogeneous Reads/Writes, 712Caml

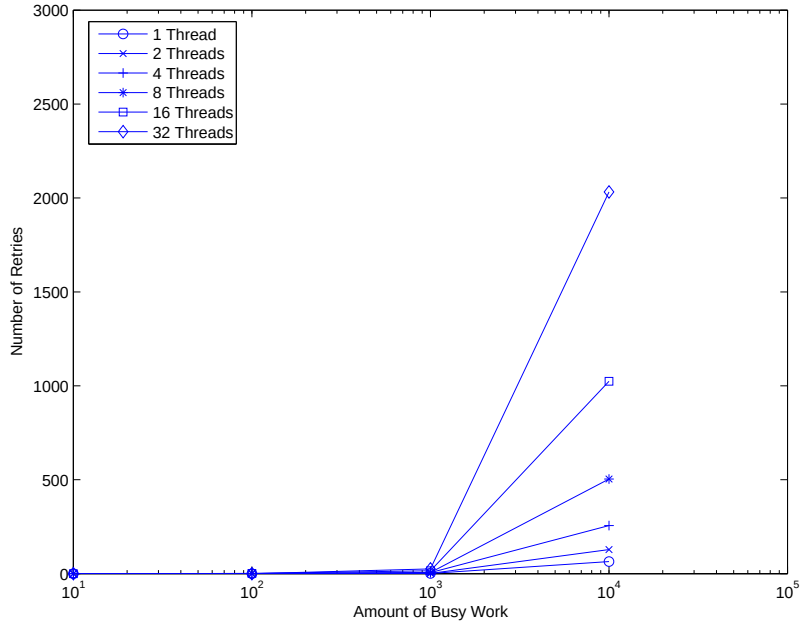


Figure 9: Number of Retries for Homogeneous Reads/Writes, AtomCaml

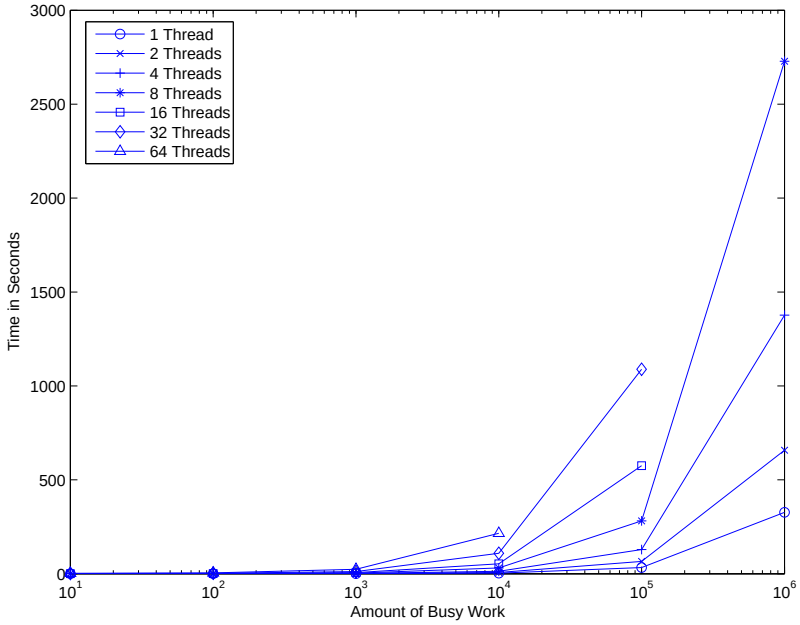


Figure 10: Running Time of Homogeneous Reads, 712Caml

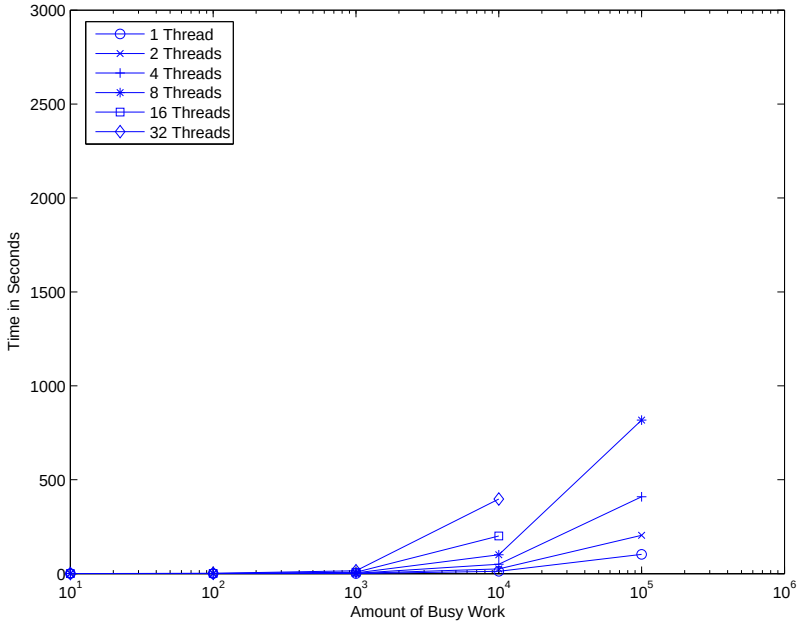


Figure 11: Running Time of Homogeneous Reads, AtomCaml

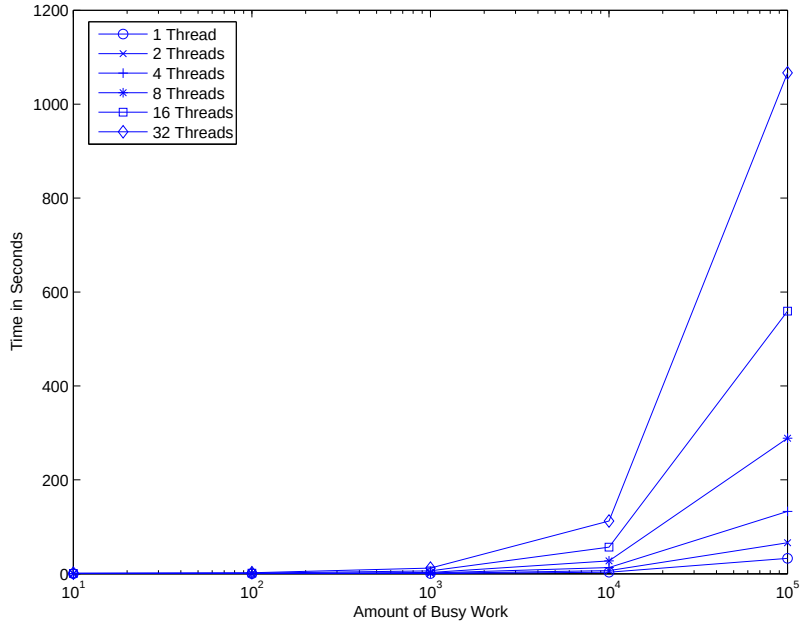


Figure 12: Running Time of Homogeneous Writes, 712Caml

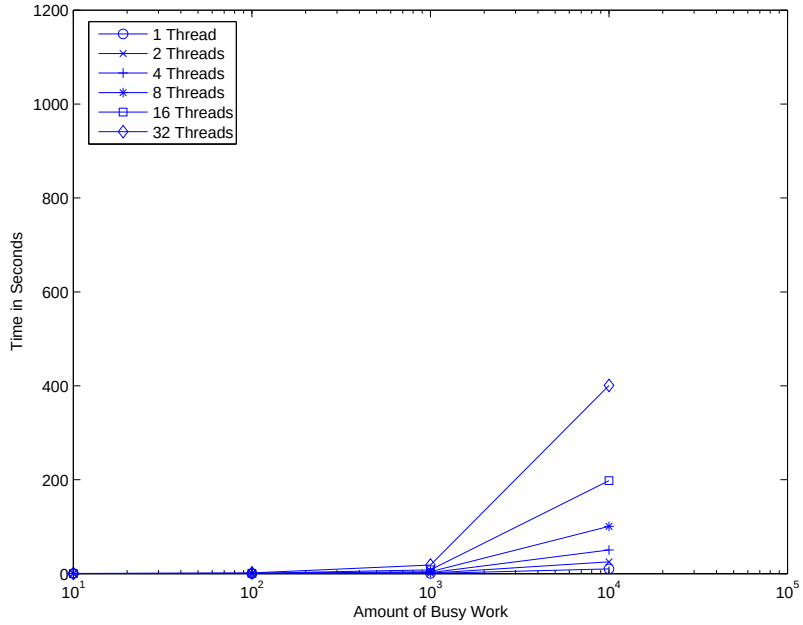


Figure 13: Running Time of Homogeneous Writes, AtomCaml

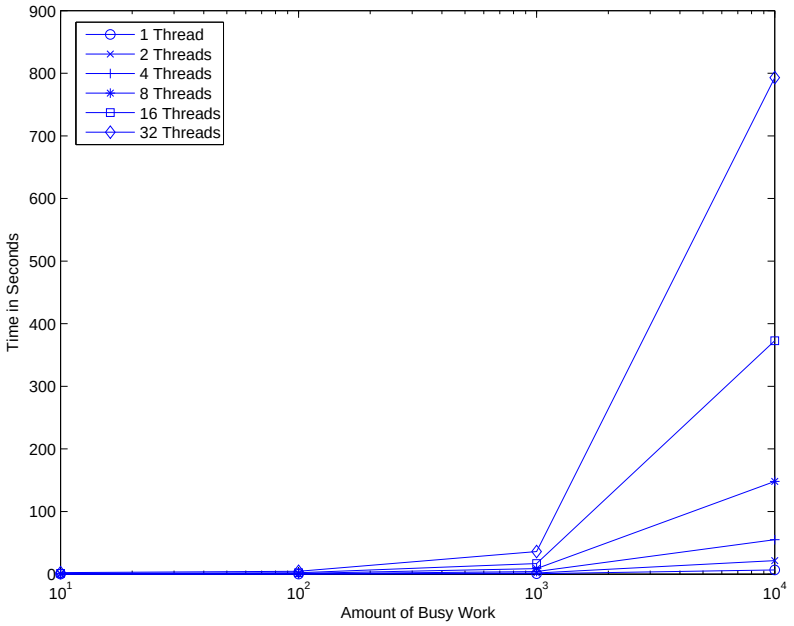


Figure 14: Running Time of Homogeneous Reads/Writes, 712Caml

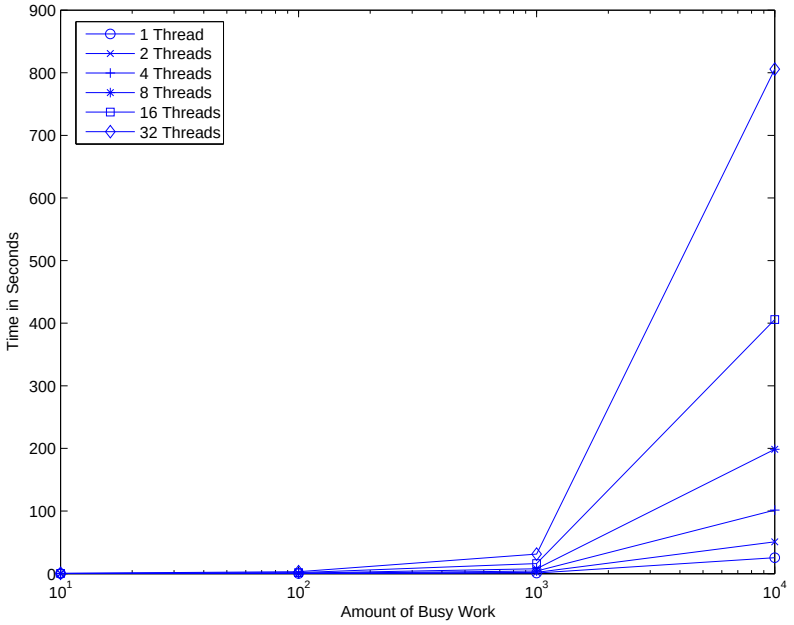


Figure 15: Running Time of Homogeneous Reads/Writes, AtomCaml

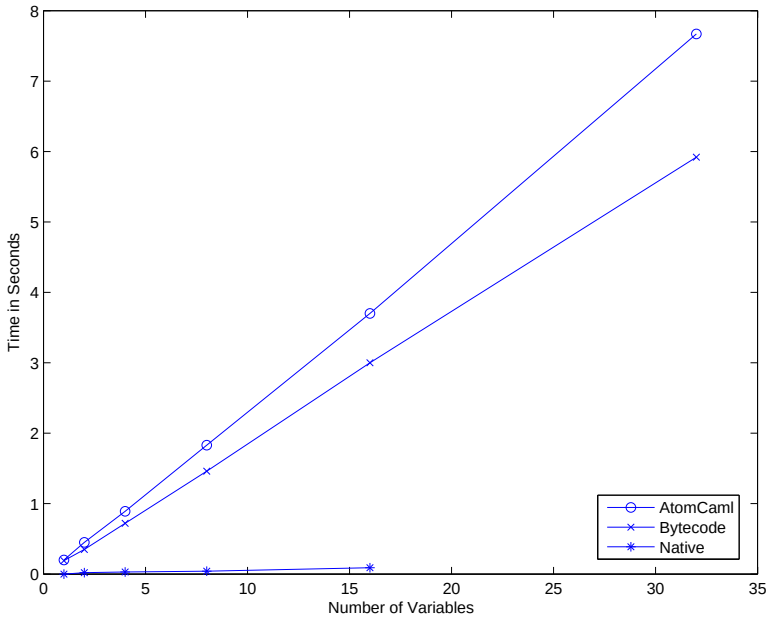


Figure 16: Code Generation, Execution Time

quite slow, but is done to permit the use of transactional types outside of transactions.

In spite of the differences between our work and AtomCaml, our designs do share many aspects. For example, our choice of exception semantics, mentioned above, follows theirs: exceptions are simply non-local transfers of control that cause an enclosing atomic block to commit its transaction log. Their rationale is that programmers already have the `yield` and `yield_r` primitives as a mechanism for indicating failure of an atomic block, while 712Caml's `retry` primitive has similar semantics. Harris [4] provides an overview of the consequences of different design decisions regarding exceptions in atomic blocks.

An older related work is that on "Tinkertoy Transactions" by Haines, et al [2], which aimed to add transactions to Standard ML, a strict, mostly functional language very similar to O'Caml. Their goal was primarily to separate transactionality into its constituent parts: persistence, undoability, threads, and locking. As a consequence, their system has a more complicated interface, forcing programmers to make explicit decisions about what to do in the case of failure. In contrast, our design only gives the programmer control of what *constitutes* failure (via the `retry` primitive) rather than what *to do* in case of failure. Their system does not remove much locking burden from the programmer, which is a primary goal of our design.

6 Conclusion and Future Work

We designed and implemented an alternative software transactional memory system for O'Caml and demonstrated that our design choices led to better scalability and in many cases better performance than the existing system AtomCaml. By optimistically assuming low contention instead of short duration, 712Caml was able to incur less rollback overhead than AtomCaml in many realistic cases.

One problem we faced in evaluating our hypotheses was how to measure the effects of a variety of non-independent factors. Since it was infeasible to explore the entire space of interactions between these factors, we had to run our microbenchmarks at points scattered throughout that space. While we chose the parameters for our microbenchmarks for both general breadth and to target interesting subspaces, we may have missed interesting interactions with this evaluation strategy, and one direction for future work would be to run a more extensive set of tests exploring this space more completely.

This work only begins to scratch the surface of possibilities for software transactional memory for O'Caml. There are many obvious opportunities for optimization; our nanobenchmarks showed that either a more efficient log data structure or a more clever transaction representation could easily lead to an order of magnitude improvement in performance. Moreover, a richer implementation could include more expressive primitives such as AtomCaml's `yield_r` that suspends execution until a particular transactional variable is updated or more sophisticated scheduling algorithms to prevent starvation by allowing different thread priorities. We could move towards native code compilation by implementing starvation-prevention for non-O'Caml VM threads, either by leveraging a user space thread library or by obtaining hooks into the kernel thread scheduler. The possibilities are endless.

References

[1] Colin Blundell, E Christopher Lewis, and Milo M. K. Martin. Deconstructing transactions: The subtleties of atomicity. In *Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking*. Jun 2005.

[2] Nicholas Haines, Darrell Kindred, J. Gregory Morrisett, Scott M. Nettles, and Jeanette M. Wing. Tinkertoy transactions. Technical Report CMU-CS-93-202, School of Computer Science, Carnegie Mellon University, December 1993.

[3] Theo Härder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, 1983.

[4] Tim Harris. Exceptions and side-effects in atomic blocks. In *Workshop on Concurrency and Synchronization in Java Programs*, pages 46–53, July 2004.

[5] Tim Harris, Maurice Herlihy, Simon Marlow, and Simon Peyton-Jones. Composable memory transactions. In *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming*, June 2005.

[6] Maurice Herlihy. SXM: C# software transactional memory. Retrieved from `http://www.cs.brown.edu/~mph/SXM/README.doc`, October 9, 2005.

[7] Cliff Jones, David Lomet, Alexander Romanovsky, Gerhard Weikum, Alan Fekete, Marie-Claude Gaudel, Henry F. Korth, Rogerio de Lemos, Eliot Moss, Ravi Rajwar, Krithi Ramamritham, Brian Randell, and Luis Rodrigues. The atomic manifesto: A story in four quarks. *SIGMOD Record*, 34(1):63–69, Mar 2005.

[8] H.T. Kung and John T. Robinson. On optimistic methods of concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, Jun 1981.

[9] Jeremy Manson, Jason Baker, Toni Cunei, Suresh Jagannathan, Marek Prochazka, Bin Xin, and Jan Vitek. Preemptible atomic regions for real-time java. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS05)*, 2005.

[10] Michael F. Ringenburg and Dan Grossman. AtomCaml: First-class atomicity via rollback. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming*, September 2005.

[11] Jie Xu, Brian Randell, Alexander B. Romanovsky, Cecilia M. F. Rubira, Robert J. Stroud, and Zhixue Wu. Fault tolerance in concurrent object-oriented software through coordinated error recovery. In *Symposium on Fault-Tolerant Computing*, pages 499–508, 1995.

File Organization and Search using Metadata, Labels, and Virtual Folders

Mukund Gunti, Mark Pariente, Ting-Fang Yen, Stefan Zickler
{mgunti,mparient,tyen,szickler}@andrew.cmu.edu

December 15, 2005

Abstract

We introduce a system that allows the user to combine metadata, user-defined labels, and Virtual Folders as primary means of file organization and localization. We not only adopt and extend the advantages of typical semantic file systems, but also preserve backward compatibility with current applications by running on top of hierarchical file-systems and by providing the user with a virtual view of our new organizational space. We show that there exist many typical organizational tasks for which our model allows a more efficient solution than hierarchical file systems. Additionally, we explore various trade-offs between functionality and performance; in particular we demonstrate how dynamic indexing strategies can significantly improve performance.

1 Introduction

File searching on current systems has mostly been made possible by separate applications that run on top of the operating system, such as Google Desktop Search, Beagle, and Spotlight.

While all of these tools provide search as an effective way of file localization, they do not significantly improve the way in which we browse and organize our data. In order to find a file, current search tools require the user to specify exact keywords (such as filenames or pathnames) and often necessitate that a new search be started each time, even though identical queries might have been issued in previous instances.

In this paper, we attempt to show that the interplay of search, semantic labels, and Virtual Folders can be used to significantly improve data organization and localization. We introduce our implementation of a user-space file indexing system that not only allows searching for contents and meta-attributes of files, but also includes mechanisms for semantic labeling and persistent query storage through Virtual Folders. We furthermore take a detailed look at several factors for increasing the system's overall indexing performance. We show that indexing overhead can be minimized by dynamic adjustment of indexing frequency, depending on the current system load, and also by coalescing file-system events to avoid redundant index processing. Finally, we introduce a set of highly expressive tools that allow the user to operate efficiently in our new organizational space.

For evaluation we examine organizational patterns in sample bash histories and measure system overhead by running the Andrew benchmark.

The rest of this paper is organized as follows: section 2 presents other related work in this area; section 3 discusses several design considerations we made; section 4 describes system design and implementation; section 5 presents the results of our experiments; conclusion and future work is given in section 6.

1.1 Virtual Folders and Labels

We introduce 'Virtual Folders', which act as abstract containers of search queries. Internally, a Virtual Folder can store a logical set of one or more queries that are connected by set-operations. To the user however, a Virtual Folder is represented as a single, folder-like data structure. A Virtual Folder could for example be called "recent pictures" and when being browsed it would internally execute a search-query for all ".jpg" files with a creation-date within the past 20 days. The term 'folder' might be slightly misleading because a

file is not physically contained in the folder, but only happens to match the folder's internal query. There is no direct way of 'moving' a file into the "recent pictures"-folder; the file either happens to be a recent picture or it does not.

This is where 'labels' come into play. A label is a special kind of semantic tag created by the user that allows him to group files together arbitrarily even though they may not share common file attributes. In our system there exists a central label-table which technically is nothing but a set of strings. A user can create as many meaningful labels as he needs, such as 'Work', 'Bills', 'Car', etc. The user then has the possibility to tag each of his files with any subset of these labels. In our system, each existing label will automatically be represented by a matching Virtual Folder which contains a query to search for all files carrying that label. More concretely, this means that if I open the 'Car' Virtual Folder, it will show me all the files carrying the label 'Car'. For these Label-Based Virtual Folders we can actually define a notion of 'moving' files and files being 'inside' a folder. Moving a file from the 'Car' folder to the 'Work' folder would simply mean removing the 'Car' label from the file and adding the 'Work' label to the file. A file can carry multiple labels at the same time, and thus can simultaneously exist in several Virtual Folders. This should be considered an advantage over hierarchical file-systems where users often tend to either copy or symbolically link files which fit two or more nodes of their existing directory-hierarchy.

When interacting with the system through our shell interface (as described later in detail), the user is presented a flattened appearance of the underlying hierarchical file-system. Browsing is performed by issuing queries and appending them to a 'query-stack'. The concept of the query-stack is that Search, Virtual folders, and Labels all represent equally valid queries and can be arbitrarily chained together through set-operations. By appending queries to the stack, the user can narrow down or expand the current search results. Furthermore, any query-chain can also be stored as a Virtual Folder.

2 Related Work

2.1 Desktop Search Applications

A number of desktop search applications exist today, but the market is dominated by Beagle on GNU/Linux, Google Desktop on Microsoft Windows systems, and Spotlight on Apple Mac OS.

2.1.1 Beagle

An existing implementation of content indexing for use in desktop search is Beagle [2]. Beagle is a desktop search utility that runs on the GNU/Linux platform. Along with content-indexing the file-system (including full text search, PDF text dumps, etc.), Beagle also indexes personal information such as e-mail, instant messaging logs, etc. It is extensible through content filters, which can be implemented to extract attributes from new data sources.

Beagle has two major components: A per-user indexing daemon, and tools to interact with the daemon (a GUI tool for querying, API's to use Beagle etc.). Unlike similar implementations, Beagle requires every user to run their own daemon which indexes files accessible to them. The Beagle daemon uses the Inotify interface provided by the Linux kernel to receive file modification events and do incremental updates. As such, the daemon includes a scheduler to handle various events such as queued Inotify updates and other jobs such as periodic optimization of indexes.

2.1.2 Google Desktop Search

Google Desktop Search [3] supports indexing for various file types, including email messages, chat messages, Microsoft Office files, PDF files, text files, C program files, etc. On installation, a full system index is constructed, which could take hours depending on the size of the file-system. Once the full index is constructed, later changes to the file-system are updated incrementally. In addition to searching the files available to the user, Google Desktop Search also indexes web history caches. Search results are displayed in a browser window and show excerpts from where the keywords used in the query match the file, just like in the web search. As a privacy mechanism, users can list directories that they do not want to be indexed and searched.

Although Google Desktop claims to do full-content indexing, it actually only indexes the first few kilobytes when processing large files. It also omits content indexing for file types with unknown extensions, such as a plain ASCII file. Email messages are indexed immediately upon reception, but for other files in the file-system, the index is only updated when the system is idle.

2.1.3 Spotlight

Spotlight runs on Apple Mac OS [1]. In addition to indexing file contents, it also extracts file meta-data such as file modification dates, ownership, access permissions, and other type-specific attributes like the EXIF header in JPEG files. Email messages, address book contacts, iCal calendars, and system preferences are also indexed and searched.

Spotlight allows the user to save search queries in "Smart Folders", which groups files together based on search criteria instead of physical location, similar to the virtual folders in our system. Files in Smart Folders are automatically updated as documents are added or removed in the file-system.

Because Spotlight maintains a comprehensive, constantly updated index of the file-system, although Apple claims that indexing is done transparently in the background, the actual overhead is quite high up to the point that the user can notice significant decrease in system performance.

2.2 Semantic File Systems

The concept of introducing semantics into the hierarchical file-system for content access was first proposed by Gifford et al. [6]. During indexing, file attributes are automatically extracted using "transducers" that work on specific file types. The indexed attributes are extracted from file headers or content keywords, such as the functions exported or imported in C program files, the article name in newspaper articles, or the sender and receiver's name in email messages. Whenever a search query is issued, a Virtual Folder having the same name as the query is created that stores symbolic links to the search results. Users can also implement their own transducers to extract file attributes that are of interest to them.

Although the semantic file system provides more advanced searches using file attributes and Virtual Folders as a new means of data organization, its main drawback is that users cannot actively organize their files using virtual directories. Also, whenever users want to find a file, they have to supply a specific query with exact keywords to locate that file.

A recent example of a Semantic File System is the planned commercial integration of Microsoft's WinFS (Future Storage) [8]. Like in [6], the system will automatically retrieve various meta-properties of files and store them in a so-called WinFS Store, which is a coupling of a database-server and the physical file-system. Similar to our approach, the system provides a query based API for interactions with applications. Virtual folders are implemented as XML-based search queries [10]. Microsoft is also aiming to provide Virtual Folders as a fully integrated GUI interface which might ultimately replace the users' dependence on purely hierarchical folder structures. Unlike our system, WinFS' planned solution does not fully reside in user space, but will instead be partially implemented on the filesystem level. This might increase performance, but will also hurt platform-independence.

Soules et al. [9] used contextual relationships between files to further enhance file system search. They examined the temporal locality of file accesses, constructing a relation-graph of files for a given time window. When the user searches for a keyword, the system first performs a content-only search, and the results are then fed into the relation-graph to locate additional hits through contextual relationships. This approach was able to improve both the recall (increasing the number of relevant hits) and precision (returning fewer false-positives) by around 10% compared to content-only search.

2.3 Flattened File-Space and Virtual Folders

Other work extended the semantic file-system concept to incorporate more flexible functionalities in Virtual Folder usage. The Nebula File System [5] implements files as sets of attributes. The traditional file naming is replaced by combining file attributes to uniquely identify a file with a query. This is similar to the process of refining searches to narrow down the results returned. In their example, the text version of a notes file for a project called "plan2" is identified by the query "format=text & project=plan2 & name=notes2.txt".

These attributes can be created by the user or automatically generated by the system. Nebula also defines a "view" of the file-system as a set of objects that match a particular query, similar to the concept of virtual directories. The views are arranged in a flat structure instead of the traditional file-system hierarchy. Files in a view must satisfy the particular query associated with the view, which disallows arbitrary adding or moving of files.

The Hierarchy And Content (HAC) file system [7] extended the virtual directory concept by allowing users to dynamically add or delete files under virtual directories. For new search results from future index updates, the HAC file system will only move files that are obtained by evaluating queries; anything explicitly added (or deleted) by users will stay in (or out) of that virtual directory. This provides more flexibility in the organization of virtual directories, where users have control over the search query results and can fine-tune the results manually. However, the HAC file system has the disadvantage that their virtual directories are still based on pre-defined file attributes. If users want to include other files that do not have those existing query attributes, they would have to be added to a particular Virtual Folder by hand. This could be a painful process if the size of the file-system is large, and would counterbalance the gain from having a searchable file-system.

Our project also adopts many concepts of a Semantic File System (SFS) proposed by Gifford et al [6]. In addition to plain meta-data attributes that can be extracted from the file header and content, we also allow full-text search and user-defined labels to act as equally valid organizational components. Chains including any of these queries can not only be executed, but also be stored as new Virtual Folders which allows us to achieve a much more powerful notion of data-organization than most previous approaches.

By using a combination of Virtual Folders and user-defined labels, we provide the same functionality as the HAC file system [7], and Nebula [5], with the addition of full-text indexing and arbitrary labels that users can apply to related files while achieving higher indexing performance.

3 Design Considerations

We decided to implement our system in user-space as opposed to kernel space. This is because it is easier to develop and debug user-space code, and because it keeps portability and installation simple. To interact with the user, we chose a shell interface rather than GUI interface since we feel that the shell is more expressive and more powerful, while with a GUI interface the operations that can be performed are restricted to those that are explicitly implemented.

4 System Design and Implementation

Figure 1 shows the overall view of our system structure, which is divided into three big components: the database server, the Inotify daemon, and the shell interface. Our system runs on the Linux platform. The entire server and shell are written in Java, while the Inotify daemon is implemented in C. A MySQL database is used as central storage for all file indexing information, Virtual Folders, and user-defined labels, which interfaces with the server through the JDBC (Java Database Connectivity) API.

4.1 Server

The server receives file-system change events from the Inotify daemon and interacts with the user through our shell interface. Its main function is to index and retrieve files. It automatically extracts basic file information, such as the filename, path, size, modified date, time of creation, etc. File types are also recognized by the filename extension, and they are passed to appropriate "transducers" for further metadata extraction and indexing. The server is able to recognize MP3 files and reads their ID3 headers, extracting information such as song title, song duration, genre, artist, sampling rate, etc.

We also provide full-text indexing on text files by leveraging a search engine library in Java from Apache called Lucene. Lucene provides a flexible API for updating, adding, and deleting indices. After the IndexWriter daemon is invoked to create indices for a given file, the content analyzer parses and filters out useless words and characters from the index according to the semantics and syntax of the English language.

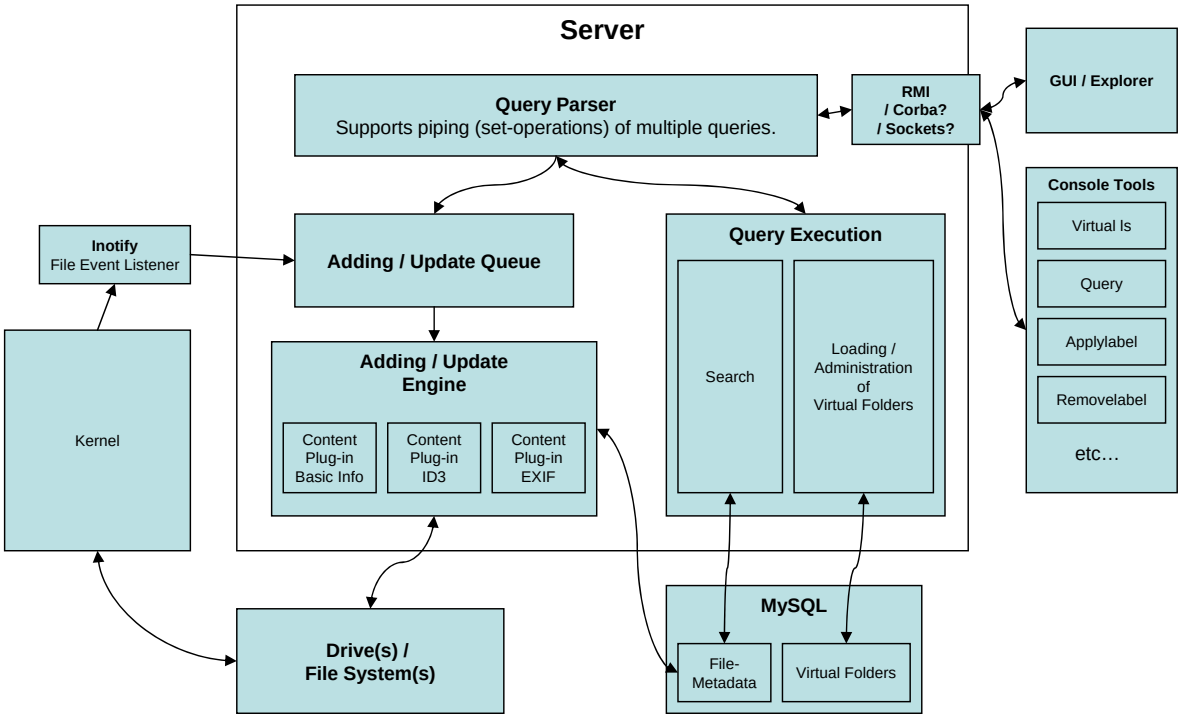


Figure 1: A schematic of our system's components

The server also manages Virtual Folders and user-defined labels. Users interact with the server through our shell interface to save queries as Virtual Folders, and to define their own labels on files and use them as organizational semantic tags.

When events are received from the Inotify daemon, the Inotify reader daemon in the server parses the output and appends a matching event to the server's Adding Queue, which is examined every 1000ms. If there are events on the queue, the Adding Engine then performs all the index updates, moving, or deleting files in the database that correspond to CREATE, MODIFY, and DELETE events. For ACCESS events (file reads), we store both the last time the file was accessed and an incremental counter of number of accesses to allow searching for recently used files.

4.2 Inotify Daemon

Inotify is a file-system event-monitoring mechanism in the Linux kernel that checks for file-system changes like file creation, deletion, modification, and renaming. Once a watch for a directory is registered, the kernel writes all event notifications for that directory to a file descriptor that is accessible to our Inotify Daemon. The Inotify Daemon catches the ACCESS, CREATE, MODIFY, and DELETE events. After performing some basic coalescing, the events are stored in a queue, and flushed out to the server at a frequency dependent on a load balancing algorithm.

To eliminate unnecessary event processing, such as indexing temporary files that only exist for a short time, or handling multiple updates to the same file, the Inotify Daemon first coalesces the received file-system events before storing them on the queue. Our coalescing scheme performs several checks: remove any CREATE and DELETE event pairs to the same file, merge CREATE and MODIFY events to the same file, and remove CREATE and MODIFY events that are followed by a DELETE to the same file.

Since the file indexing operations should not become a burden on the system, we implemented a load balancing algorithm, called the "Queue Size Driven, Load Feedback Controlled Throttle Engine", to control the frequency at which events are being flushed out to the server based on the current system load and the

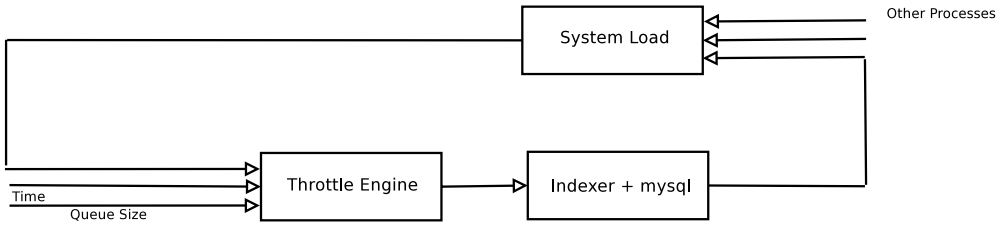


Figure 2: Simplified load balancing algorithm

state of the event queue. The load balancing algorithm is controlled by 3 parameters, listed in Appendix A.

The simplified flowchart for our algorithm is show in Figure 2. The throttle engine takes as input the current system load and the queue size, then dynamically determines how many events to flush out to the indexer.

The throttle engine algorithm is basically similar to the TCP (Transmission Control Protocol) congestion control. If the system load is under SAFE_LOAD value, the Inotify daemon doubles the number of events flushed; if the system load is above SAFE_LOAD, the number of events flushed is cut in half.

4.3 Shell Interface

In order for the user to interact efficiently with the system, we had to rethink the classical shell paradigm. Since folders are virtual, there is no longer a real notion of being inside of a directory. With Virtual Folders there no longer exists a clear, predefined hierarchy that we can descend on by using a 'cd' command. As a replacement of a path we introduce the idea of a query-stack. This is best explained by an example: The user starts out with an empty query-stack. This means that if we were to do a 'ls', we would see all files of the entire system. We then append the Virtual Folder 'Car' to the stack. A 'ls' whould now give us all car-related files (that is, all files carrying the label 'Car'). We can then append 'Bills' to the stack. In this case 'ls' would give us all the files which are both related to 'Car' and 'Bills'. By default, appending something to the stack means to intersect (logical AND) the results of the previous stack with the results of the appended query. However, the user is equally able to perform unions (logical OR) or subtraction (logical AND NOT). 'Car union Bills' would therefore give us all files which are related to either the 'Car' or 'Bills' (or both). 'Bills minus Car' would give us all Bills which are not related to 'Car'.

Virtual Folders, Labels, and Search Queries are treated as equal citizens in our shell. We could easily append the query 'filename contains ferrari' and the Virtual Folder 'Cars' to the stack, which will give us all files which contain 'ferrari' and also happen to carry the 'Cars'-label. If desired, the user can save the current query-stack as a new Virtual Folder (remember, Virtual Folders are nothing but saved queries). The shell is also the place where the user can define, apply, and remove labels. We introduce the 'apl' and 'rml' commands for applying and removing labels. A full listing of all available Shell commands can be found in Appendix B.

A big problem that we had to overcome was to combine classical hierarchical file-systems with our virtual approach. Enforcing a flat underlying file-system was considered an option, but this would render our approach incompatible with almost all of today's applications (e.g. how would we untar a source-tree in a flat file-system?). We decided to take a more natural approach by running on top of the hierarchical file-system and treat it virtually flattened within our space. This flattening process introduces at least two questions: The first one is how to handle the fact that multiple files in our flattened space could carry the same filename (because they could come from several physical directories). We decided to require the user to disambiguate these cases by presenting him the different underlying paths (or possibly other meta-attributes) of the affected ambiguous files. The second question is where in hierarchical file-space we should place newly created files. We decided that each user should have his own default-directory where all newly created files will be placed.

It should be noted that with our approach it is still possible to easily navigate through hierarchical file-space by appending search queries to the stack. For instance, to list all files of the directory '/var/log/', the user can simply append a query for 'path = /var/log/' to the query-stack. In other words, a file's path has

simply become one of the many meta-attributes of the file itself.

Two other interesting problems are how to treat copy and move operations of files. For this we need to distinguish two concepts: moving within our new virtual space and moving within the underlying hierarchical file-system. In our virtual space, moving a file from one label-based Virtual Folder to another one would be achieved by simply changing its semantic labels. To duplicate (copy) a file in our virtual space, we would perform a physical duplication operation and attribute all of the source file's labels to the new file as well. Copy and move operations within the physical file-system are still possible as our shell provides full access to the underlying 'cp' and 'mv' commands. However, it should be noted that an execution of the traditional 'cp' command will not be perceived as a copy operation by our system; it will merely be seen as the creation of another independent file (as Inotify will receive a CREATE and not a COPY event) and as such will not correctly copy the file's labels. The best method of fixing this would be to define our own version of the 'cp' command which will correctly preserve the file's attributes during copy operations. However, the traditional 'mv' command will execute correctly, as we will perceive it as a content-preserving move through Inotify and therefore keep all of the affected files' labels in a consistent manner.

A more complex problem is to intelligently move files between Virtual Folders that consist of more than a single label query. This is only achievable if the query carries the property of being non-ambiguous. For example, moving a file into a Virtual Folder with the query for the labels 'Car' AND 'Bills' is possible as we will simply apply both labels 'Car' and 'Bills'. If the Folder would contain the query 'Car' OR 'Bills' however, we would not really have a clear definition of which labels to apply (either 'Car', 'Bills', or maybe both?). Move operations are therefore only defined between logically non-ambiguous Virtual Folders.

5 Evaluation

There are several hypotheses we make about this project. The first is that Virtual Folders are able to save the amount of work users have to do to accomplish certain tasks, while making the shell more expressive; the second is that the file indexing frequency plays a major role on system performance; and the third is that our new file-system search imposes reasonable overhead on the system.

The first part of our experiments is for evaluating our shell and our new organizational paradigm. This is done by collecting sample bash histories and determining what sequence of commands can be replaced by more efficient operations from our introduced organizational concepts. The second part of our experiments is for measuring the overall system overhead by running the Andrew Benchmark and keeping a timer in the server to measure the execution time for various system components during indexing.

5.1 Virtual Folders

We collected bash histories from experienced Linux users and examined them to see which commands can be replaced by simple Virtual Folder operations. We try to categorize common file access patterns to see how we can save work for particular types of file-system operations.

From our observation, there are roughly three types of common file accesses that can be made more efficient with our shell:

- File search through directory traversal
- File search through keyword refining
- Recent file accesses

An example sequence of commands is:

```
cd ../../..
cd ../../
ls
cd root
cd cvsClaytronics/
cd WorldGen0.1/
./run cube.xml cube.dpr
```

In this example, the user is required to change between directories and manually scan through their contents to search for a particular file. With our system, we can simply issue the query

```
a path contains WorldGen0.1;
```

This will append a search for filenames containing the keyword "WorldGen0.1" to the query-stack. The user would not need to browse through the directory hierarchy to locate particular files. It is important to note that this approach is optimistic and might in fact require additional disambiguations as there might be several directories with the name "WorldGen0.1" on the filesystem that each contain the desired "cube.xml" and "cube.dpr" files. In these cases, the user is easily able to refine his search by pushing additional queries onto the stack, and our system will comfortably point out whenever such disambiguation is absolutely required.

Another example is locating files through the *slocate* command

```
slocate *.so
slocate *.so|more
slocate m*.so|more
slocate m*.so
slocate mozilla*.so
cd /usr/lib/mozilla
cd plugins/
ls
ls -l
```

Here the user searches multiple times and refines the query keyword until he or she finds the relevant files. In our shell, we can eliminate some of these redundant commands by first saving the first query result into a Virtual Folder, and then performing an AND operation to further fine-tune the search.

```
a filename contains *.so;
s2f so_file_label;
a filename contains mozilla;
```

By saving query results into Virtual Folders, we can also speed up file retrieval the next time similar files are requested, since the server does not need to fetch from the database again.

An example of the third type of file accesses is

```
cd /usr/src/linux
ls
cd ..
cd linux_vanilla_2.6.12-rc4/
ls
vi fs/nfs/layout_driver.c
...
cd /home/apalekar/pnfs/
...
vi/usr/src/linux_vanilla_2.6.12-rc4/fs/nfs/layout_driver.c
```

The user modifies a file, then after working on something else later, wants to access that file again. Without our metadata indexing, the user must change back and forth between directories if they want to retrieve previously accessed files. With our system, such type of access can be done in one command:

```
a access.timestamp.last > 5min
```

which will return queries containing recently accessed files.

Our observation is that around 20% to 40% of the commands are *cd* and *ls*, and the probability that a file will be accessed again in the near future is high. This common file usage pattern makes our Virtual Folder organizational feature especially useful. We can reduce the number of steps that the user spends locating a particular file and descending into that directory, and enable fast retrieval to frequently accessed files by grouping them together using labels.

	Average Execution Time per File
SQL	17.08ms (18%)
Metadata Extraction	0.5ms (1%)
Lucene	78.30ms (81%)
Total	96.52ms

Table 1: Average execution time spent in each indexing component.

5.2 Indexing Overhead

Since our system presents a new organizational paradigm, there is not really an adequate standardized performance benchmark that would provide a typical usage pattern involving our new organizational features. However, it needs to be remembered that our system runs on top of classical file-systems and is therefore likely to encounter traditional application workloads. We chose to run traditional benchmarks as we believe them to cover an important subset of our performance space and because they offer a good basis of comparison with existing systems.

5.2.1 System Timer

To measure our general indexing performance, we implemented a timer in the central server that measures the execution of various components in the indexing process. It allows us to inspect how much execution time is spent in the SQL database, in extracting file attributes and metadata, and in performing full-text indexing. The following experiment data is the result of indexing a directory of over 30 HTML files, each between 4 and 20 kilobytes in size. The experiment was performed on an initially empty-database. The results are shown in Table 1. From Table 1 we see that most of the indexing overhead comes from the fulltext-indexing in the Lucene library. However, expensive full-text indexing is only performed on recognized text files, so we expect the overhead for average execution to be smaller than this.

5.2.2 Standardized Benchmarks

At first we experimented with the Postmark Benchmark [4], which simulates the workload of a large Internet electronic mail server. Postmark creates a large number of random text files, accesses the files with reads and appends, and then deletes them. It turned out however, that our optimization in coalescing file-system events successfully removed all pairs of CREATES and DELETES to avoid redundant indexing, so that the total overhead measurement from the default Postmark-setup was in fact zero.

Thus we chose to go with the Andrew Benchmark instead, as it is not only well-known, but as it also provides a workload which we believe to be more typical of a user-task than what Postmark does.

We ran the Andrew Benchmark on a machine with a Intel(R) Pentium(R) M processor 1600MHz and 768 MB of RAM. The benchmark consists of 5 phases: Phase I: recursively create files and directories; Phase II: copy files; Phase III: recursively stat directories; Phase IV: scan files. For Phase V of the benchmark, we ran two sets of experiments. The first first was a Linux kernel untar and compile, while the second is compiling and installing Openssh-2.5.2p2.

For now, the load balancing parameters are set as follows:

```
CRITICAL_LOAD = 1.00
SAFE_LOAD = 0.75
NEGLIGIBLE_LOAD = 0.10
```

The event queue is flushed every 4 seconds, and the maximum number of events that can be flushed at a time is 100.

To explore the tradeoff between performance and the indexing frequency options, we ran the Andrew Benchmark with three different options: base-line execution without indexing, indexing with load balancing and event throttling, and indexing without load balancing and event throttling.

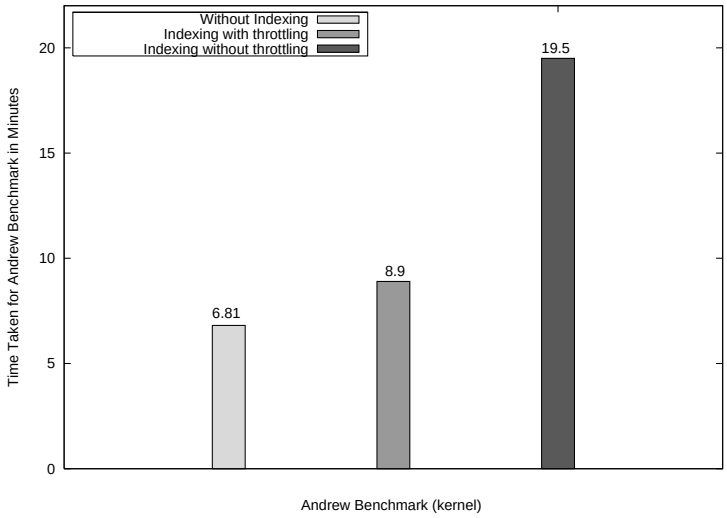


Figure 3: Time Taken to Run (Phase V being kernel untar and compile)

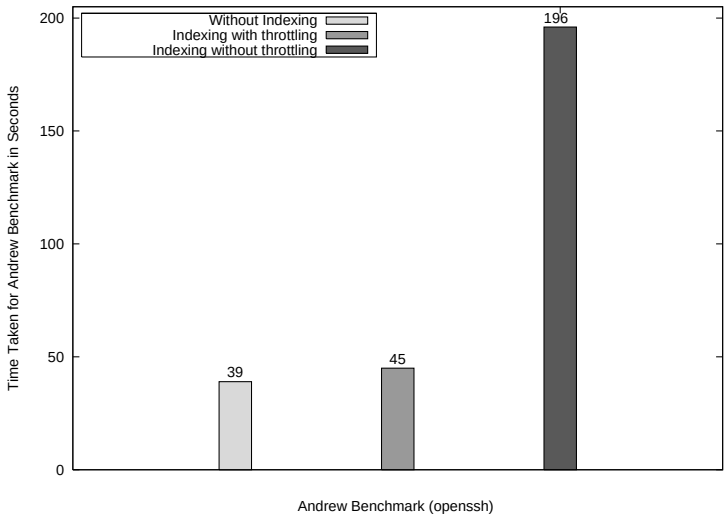


Figure 4: Time Taken to Run (Phase V being compiling and installing Openssh-2.5.2p2)

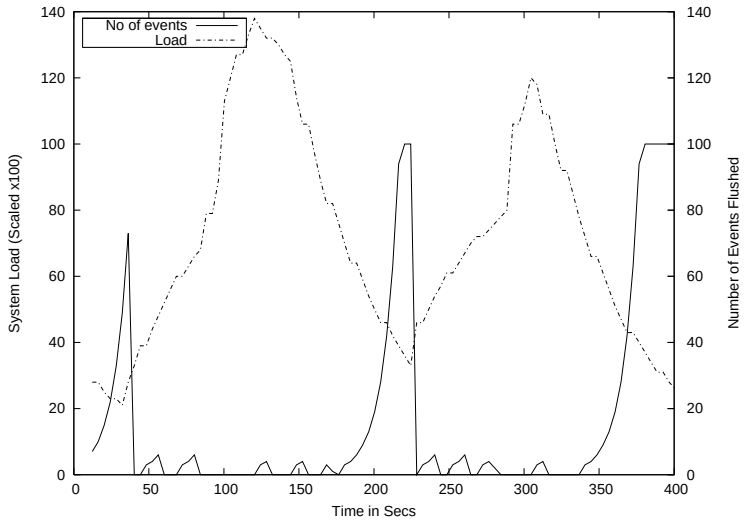


Figure 5: Number of Events Flushed vs. System Load (Phase V being kernel untar and compile)

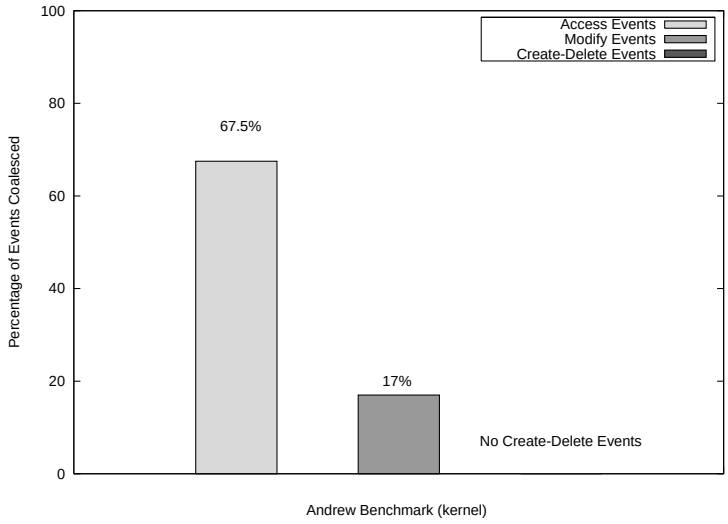


Figure 6: Percentage of Events Coalesced (Phase V being kernel untar and compile)

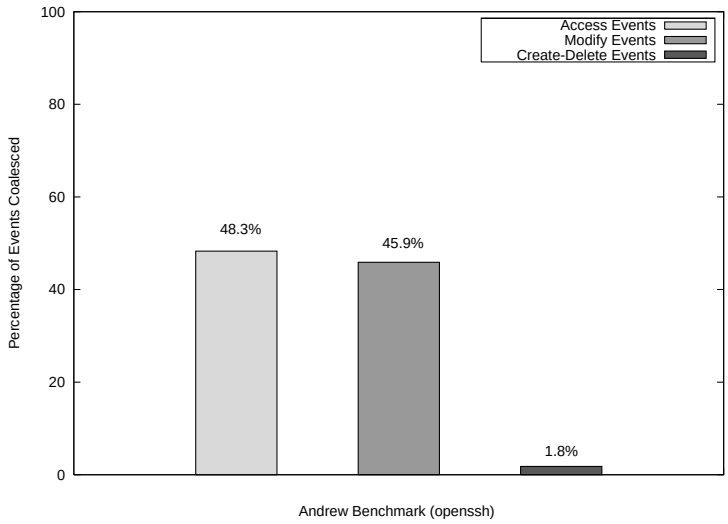


Figure 7: Percentage of Events Coalesced (Phase V being compiling and installing Openssh-2.5.2p2)

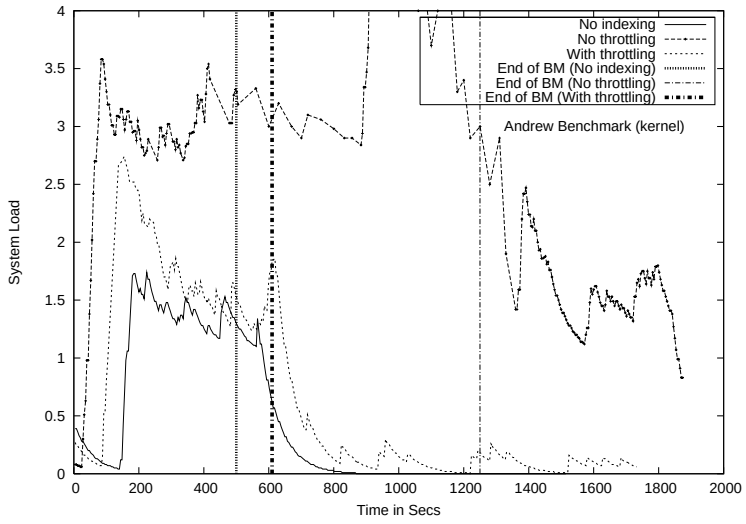


Figure 8: Load Comparison with Throttling (Phase V being kernel untar and compile)

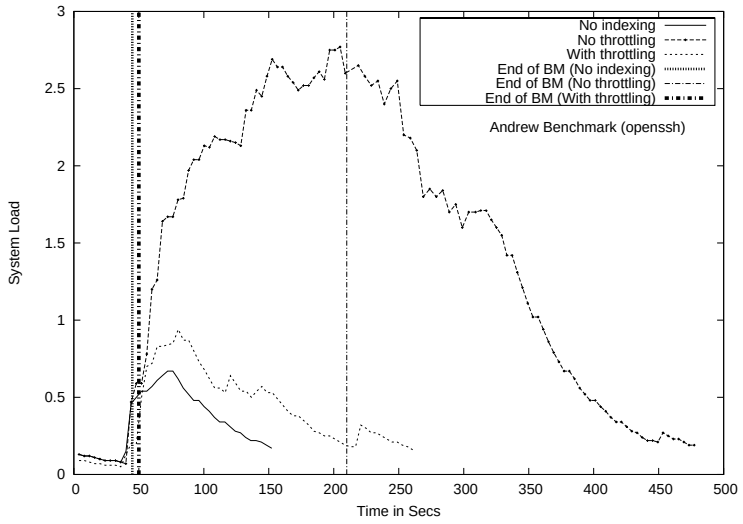


Figure 9: Load Comparison with Throttling (Phase V being compiling and installing Openssh-2.5.2p2)

Figure 3 and Figure 4 shows the result of the total time to run the benchmark for different set of Phase V. When indexing is performed with the load balancing algorithm enabled to throttle the number of events flushed each time, there is approximately a 30% and 15% overhead for indexing time, respectively. But without the load balancing algorithm, we get an indexing time overhead of 186% to 400%.

Figure 5 shows how the number of events flushed is dynamically adjusted by the load balancing algorithm to depend on the current system load. When the system load is in the range between SAFE_LOAD and NEGLIGIBLE_LOAD, the number of events flushed increases exponentially; while when the system load is between SAFE_LOAD and CRITICAL_LOAD, the number of events flushed decreases exponentially. This allows the indexing operations to be performed without becoming a burden on the system.

Figure 6 and Figure 7 shows the percentage of events coalesced by the Inotify daemon before they are flushed so unnecessary event processing can be eliminated. For the case with kernel untar and compile, 67.5% of the ACCESS events are coalesced, which are redundant file read that correspond to the same file and so can be merged together. 17% of the MODIFY events are coalesced. This is a result of coalescing CREATE-MODIFY event pairs into one CREATE event, or coalesing MODIFY-DELETE event pairs into one DELETE event. Since the benchmark does not measure the operation of deleting all the temporary working files after the benchmark terminates, no CREATE-DELETE event pairs are coalesced.

Figure 8 and Figure 9 shows how the system load varies with time when running the benchmark. The vertical lines show the time when the benchmark terminates for each of the three scenarios. For the case with kernel untar and compile, the benchmark shows a 20% slowdown when indexing with load balancing and throttling is enabled as compared to the base-line case. However, when indexing is performed without load balancing and throttling, this slowdown becomes 150%, and the extra system load imposed becomes unacceptable. From the figures we can also see that the indexing operations continue in the background after the benchmark terminates, but with load balancing and throttling, the extra system load imposed by indexing is minimized.

From these results, we show that indexing frequency has a major impact on system performance. By dynamically adjusting the indexing frequency based on the current system load, we can greatly reduce the overhead our system imposes.

6 Conclusion and Future Work

Our system successfully allows the user to effectively combine metadata, user-defined labels, and Virtual Folders as primary means of file organization and localization while preserving full backward compatibility with today's systems. There are many instances where search, labels and Virtual Folders can in fact signifi-

cantly improve the efficiency of our daily file-system tasks. Through optimizations in coalescing file-system events and controlling indexing frequency by system load balancing, our experiments show that indexing performance can be greatly increased while imposing little overhead on system load.

In order for our approach to become even more effective in the future, it would be desirable to modify user-space applications to automatically provide additional labels and meta-data. When saving a file, applications could for example ask the user to apply any relevant labels. Generally, it would be interesting to explore the possibilities in which our system could be integrated into existing graphical user-interfaces as a classical file-tree view is not enough to fully represent our organizational space.

Another interesting extension to our system would be network-based querying between hosts. Locating and organizing files among several machines using distributed indexing-servers is a problem that could be approached in many different ways and should be considered a promising research project.

7 Appendix A: List of Indexing Frequency Option Parameters

```
CRITICAL_LOAD - the system load value above which no indexing should be
performed, but instead wait until the system load reduces
SAFE_LOAD - if system load is between SAFE_LOAD and CRITICAL_LOAD, reduce
the number of events indexed; if system load is between SAFE_LOAD and
NEGLIGIBLE_LOAD, increase the number of events indexed
NEGLIGIBLE_LOAD - the system load below which we can perform indexing at
maximum frequency
```

8 Appendix B: List of Shell Commands

8.1 Stack Commands

```
a   [VFOLDER/LABEL/SEARCH] - append a query as intersection ("and")
a+  [VFOLDER/LABEL/SEARCH] - append a query as union ("or")
a-  [VFOLDER/LABEL/SEARCH] - append a query as subtraction ("and not")
a .. - remove the last item from the query-stack
a /  - clear the query-stack
```

8.2 Operational Commands

```
apl [LABEL] [filename(s)] - will apply a label to a filename
rml [LABEL] [filename(s)] - will remove a label from a filename
s2f [VFOLDER] - create a new Virtual Folder, made of the current stack
f2s [VFOLDER] - deconstruct a Virtual Folder onto the stack
mklabel [LABEL] - create a new label
rmlabel [LABEL] - delete a label from the system (and remove it from all files)
rmfolder [VFOLDER] - delete a Virtual Folder
folders - list all available Virtual Folders and labels
fields - list all availabe meta-fields that we can search for
cd [(partial) physical path] - will append a search for this path to the stack ("and")
refresh - refresh search results by running the stack against the server again
check [physical_folder] - tell the server to (re-)index the given physical folder
```

References

[1] Apple - mac os x - spotlight. http://www.apple.com/macosx/features/spotlight/.

[2] The beagle project. http://beaglewiki.org/Main_Page.

[3] Google desktop. http://desktop.google.com/.

[4] Network Appliance. Postmark: a new file system benchmark. Technical Report TR3022.

[5] C. Mic Bowman, Chanda Dharap, Mrinal Baruah, Bill Camargo, and Sunil Potti. A File System for Information Management. In *Proceedings of the ISMM International Conference on Intelligent Information Management Systems*, March 1994.

[6] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. O'Toole Jr. Semantic file systems. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 16–25. Association for Computing Machinery SIGOPS, 1991.

[7] Burra Gopal and Udi Manber. Integrating content-based access mechanisms with hierarchical file systems. In *Operating Systems Design and Implementation*, pages 265–278, 1999.

[8] Richard Grimes. Code name winfs - revolutionary file storage system lets users search and manage files based on content. http://msdn.microsoft.com/msdnmag/issues/04/01/WinFS/, January 2004.

[9] Craig A. N. Soules and Gregory R. Ganger. Connections: Using context to enhance file search. In *Proceedings of the 20th ACM symposium on operating system principles*, pages 119–132, 2005.

[10] Paul Thurrott. Virtual folders: A windows vista technology showcase. http://www.winsupersite.com/showcase/winvista_virtualfolders.asp, July 2005.

A Writable Semantic File System

Deepak Garg, Vaibhav Mehta,
Shashank Pandit, Michael De Rosa

Abstract

Traditional hierarchical file systems present a single organizational view of the files. This makes them cumbersome in situations where multiple views of a single file system are required. Semantic file systems attempt to address this problem by extending the space of file names to include searchable metadata in the form of key-value pairs. However, there is no unified semantics for search and modify primitives. Search is traditionally implemented through virtual directories, whereas modification is done by direct assignment to key-value pairs. Also, the set of keys available to the user is fixed when the system is designed. We address both these issues by defining a unified semantics for accessing metadata and permitting the user to define arbitrary key-value pairs.

Every file in our file system is tagged by an arbitrary number of key-value pairs. Abstractly, each key is assumed to define a different dimension in an attribute space. Each value that a particular key can take is a value along the corresponding dimension. Under this abstract view, each point in the attribute space is inhabited by at most one file, and each file has a unique location in this space. All shell commands like `cp, mv, rm, ls` work on sets of points in the attribute space rather than individual files or directories.

1 Introduction

Traditional hierarchical filesystems provide only a single schema for organizing files on a disk. For many applications, the logical view of the filesystem expected by the user differs largely from the hierarchical organization present on disk. As a result, the files become disorganized over a period of time. Semantic filesystems were introduced in 1991 by Gifford et al. as a solution to this problem. In a semantic file system, files can be tagged with metadata. This metadata can be queried, allowing for arbitrarily complex organization schemes.

Unfortunately, most descriptions of semantic file systems to date are biased towards querying as opposed to updating metadata. Most of them do not permit users to define their own metadata tags. Instead, they work with a fixed set of system generated metadata like modification date, file type, summary of contents, etc. Additionally, no existing implementation provides a uniform semantics for querying and updating metadata. They specify the meaning of file operations in an ad-hoc manner.

In this paper, we propose a writable semantic file system (WSFS), where users can manipulate file metadata in the form of arbitrary key-value pairs. As a simple example, the file representing this paper can be given the attributes $\{\texttt{project} : \texttt{wsfs}, \texttt{year} : 2005, \texttt{filetype} : \texttt{pdf}\}$. The semantics of our filesystem allows users to remove, add and modify such metadata tags using common filesystem operations like copy and move. Further, the semantics of our filesystem is uniform, in the sense that it is based on a single view of metadata as a multi-dimensional space. The example below illustrates how the common shell command `mv` can be used to modify and add metadata in our filesystem.

```
mv /local/(filetype:pdf + project:os)/foo.pdf  /usr/(filetype:ps)/
```

We use `+` for disjunction. The above command affects all files that are in the directory `/local` and either have `filetype:pdf`, or the `project:os` attribute. The command alters the directory of all these files to `/usr` and changes the value associated with the key `filetype` for each of these files to `ps`. If this key does not exist in the metadata set of some of these files, it is created.

An important aspect of the above command is that its source is a set of files matching a logical query. All common file operations in our system follow this pattern. Their arguments are logical queries, and it is from this that our system derives a tremendous expressive power.

The rest of this paper is organized as follows. In section 2, we present details of the abstract semantics of common file operations like move, copy, delete, create etc. In section 3, we describe a partial implementation of a concrete subset of these semantics. We use a database backend to manage our filesystem metadata. Section 4 describes quantitative and qualitative evaluation of our proposed system. In section 5 we compare WSFS to existing semantic file systems. Section 6 concludes the paper with some directions for future work.

2 Semantics of WSFS operations

The simplest way to view our metadata is as a multi-dimensional space where each key forms a different dimension. The data points along a dimension are the various data values that the particular key can take. For example, the key `filetype` can take the data values ps, pdf, cc, txt, etc. For uniformity, we treat name of a file as a metadata attribute having the key `filename`. This particular dimension of the metadata space can take any valid filename as the data value. Similarly, the physical path of a particular file is treated as a metadata attribute having the special key `pathname`. In principle, the metadata space has infinite dimensionality, and many dimensions have an infinite number of data points, but for a given file system at a particular time, the number of dimensions and the number of data points along any dimension are finite. For each key, we assume a special data point called `null` which is the value associated with the key for files that actually do not have that particular key in their attribute set.

We now impose two consistency constraints on our metadata space. These are maintained by all operations that we define.

1. (Uniformity) For any file, there is exactly one data value associated with every key. This is only a theoretical concept. In reality, some files may not have any data associated with particular keys. Implicitly, our semantics assume that in such cases the data `null` is associated with these keys.
2. (Uniqueness) At any point in the metadata space, there is at most one file having the given metadata. Realistically this means that given data values for all keys present in a file system (including `filename` and `pathname`), at most one file has exactly those key-value pairs.

Observe that the consistency constraints defined above allow two distinct files having the same name and same physical path to co-exist if they differ in the value associated with some other key. This kind of flexibility gives our file system tremendous expressive power. We now describe how common operations like copying, moving, deletion, etc. are defined for our file system. Perhaps the most novel feature of our semantics is that these operations are defined on *sets* of files rather than single files. Performing these operations on a single file is a special case of the general semantics where the sets under consideration are singletons.

The first important idea governing our operations is that the notion of a path (or current directory) is replaced by a structured query that selects a subset $\mathcal{F}$ of the files present in the file system. These queries (q) have the form:

$$q \ ::= \ \texttt{key:value} \mid q_1 \vee q_2 \mid q_1 \wedge q_2$$

For simplicity, we consider basic queries of the form `key:value` only but this can be generalized to allow queries like `key > value` or wildcard matching for keys where such operations make sense. Given the set of all files $\mathcal{F}$ in a file system, we define the *meaning* $S(q)$ of a query q to be a subset of files as follows:

$$\begin{aligned} S(\texttt{k:v}) &= \{f \in \mathcal{F} | f \text{ has value } v \text{ associated with key } k\} \\ S(q_1 \vee q_2) &= S(q_1) \cup S(q_2) \\ S(q_1 \wedge q_2) &= S(q_1) \cap S(q_2) \end{aligned}$$

The equivalent of the current working directory in our file system is a structured query which we call the *current query* and denote by q_0. We now define the equivalents of some common file operations in our file system.

1. (`cd`) The equivalent of a change directory operation in our semantics is a *refine query* operation, which takes a structured query q as an argument. The semantics of this operation is to replace the current query q_0 by $q_0 \wedge q$ if q is a relative query, and to replace q_0 by q if the latter is an absolute query. We distinguish absolute and relative queries much in the same way that the shell distinguishes absolute pathnames from relative pathnames – absolute pathnames start with a '/'.
2. (`rm`) The remove operation takes as argument a query q and deletes every file in $S(q_0 \wedge q)$.
3. (`ls`) This command should take a query q and list the entire set $S(q_0 \wedge q)$. In order to make the listing user friendly and readable, our implementation extracts the physical path from the query and lists all files having that physical path.
4. (`mv`) The file move command in our file system takes two queries q and q' as arguments. The essential restriction here is that q' be a *subspace* of the multi-dimensional metadata space. This is easily enforced by prohibiting the operator $\vee$ in q'. The semantics of the move command is an orthogonal projection of the set $S(q_0 \wedge q)$ onto the subspace q' in the metadata space. More concretely, this amounts to changing some attributes of all files in $S(q_0 \wedge q)$. For example, projecting the set $S(\texttt{pathname:/usr} \vee \texttt{filetype:pdf})$ onto the subspace `filetype:ps` amounts to changing the `filetype` attribute of every file that is either in the /`usr` directory, *or* has the attribute `filetype:pdf` to `ps`.

   While taking the orthogonal projection, it is possible that many files get mapped to the same destination file. This is an error condition because it violates the uniqueness invariant stated above, and is reported to the user by our implementation. The user can then refine the source query q and reissue the command. Similarly, in some cases, the projection may try to overwrite an existing file (this is equivalent to moving onto an existing file in a traditional file system). This is also an error condition.
5. (`cp`) This command takes exactly two queries like the `mv` command, the second of which must be a subspace. It behaves like the `mv` command, except that instead of changing file attributes in place, files are duplicated and the new files are given fresh attributes.
6. (`create`) The create operation takes a query q as an argument. First, it checks that both the current query q_0 and q are free of disjunctions ($\vee$). Next, it ensures that the `filename` and `pathname` attributes are both present in $q_0 \wedge q$. If both these conditions are met, it creates a new file containing every key:value pair that occurs in $q_0 \wedge q$, and setting all other attributes to null. If there is already a file with exactly these attributes, the operation returns an error.

3 Implementation

The implementation of WSFS consists of two main components. There is a user level NFSv2 server frontend, and a backend comprising a query parser, and a database that manages the metadata. Presently, the implementation of the backend is complete. We have not yet implemented the frontend completely. On the whole, the system works as follows. NFS requests generated by the client are passed to the server, which provides a cache and an interface to the backend. The server makes appropriate calls to the parser, the database and the filesystem as required. The client side of NFS remains unmodified, as does the system shell. However, the meaning of commands like `cp, mv, rm` etc. changes in accordance with our semantics.

As mentioned in section 2, the file name and path are represented by metadata tags. The physical files stored on disk are named using random strings, which we call nonces. A new nonce is generated for each new file that is created. These files are physically stored in a single, flat directory by the NFS server. The database maintains mappings between the nonces and the attributes of the files they represent (including file name and path).

3.1 NFS Frontend

The frontend of our implementation is an NFS server. At present, we do not have a complete implementation of this part. However, we have been able to use this partial implementation to create traces of the exact database and parser calls that are generated during normal operation. We used these traces to benchmark our parser and database backend. On the whole, we needed to modify only five functions of a standard NFS server: `lookup, remove, rename, readdir` and `create`.

3.2 Backend: Parser and Database

Our parser takes structured string queries, and parses them to an internal representation used by the database backend. The database is implemented using SQLite. It is organized as follows. For each metadata key, we have a separate database table, that associates nonces (files) present in our system with their corresponding values for the key. There are two distinguished tables `filename` and `pathname`, that map nonces to the actual names and paths of the files they represent. Parsed queries are mapped directly to SQL queries which are run on the database. At present, all the tables are stored in a single file on disk.

4 Experiments/Evaluation

We performed three types of evaluation on WSFS. First, we performed a number of microbenchmarks on common file system operations. These microbenchmarks show the general overhead associated with tracking metadata during various single operations. Second, we ran a modified Andrew benchmark at three different scales (NFS server, openSSH and linux kernel 2.6) to determine how the system scales, as well as the overhead associated with quasi-representative workloads. Both these benchmarks were run on a P4 3.0 GHz, 512KB L2 cache and 1GB RAM with a 80GB hard drive, running Linux kernel 2.4.25. Third, in order to evaluate the expressiveness of our system, we describe the semantics of a revision control system (RCS) using the operations provided by our filesystem.

4.1 Microbenchmarks

We microbenchmarked four common filesystem operations: create, delete, list directory and move. We compared the overhead incurred in the parser and database backend to the time

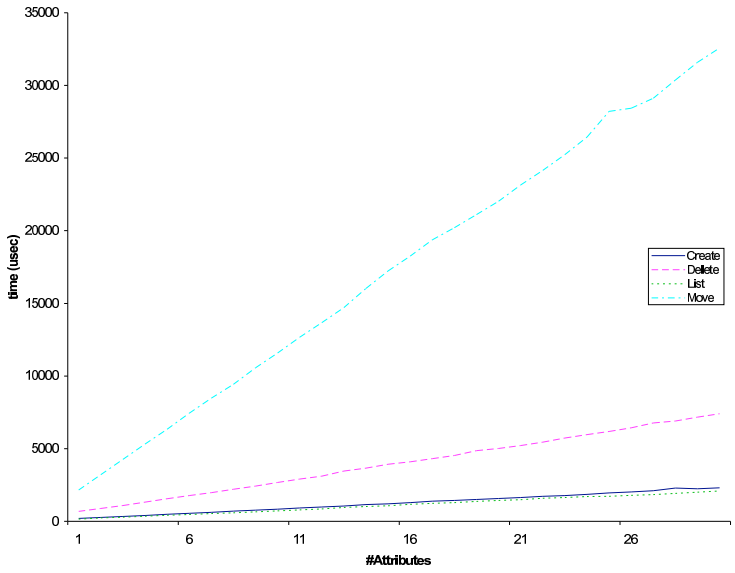


Figure 1: Variation in backend overhead with number of attributes involved

taken by the basic linux filesystem operation. The table below compares the average time taken by the linux filesystem to our backend's overhead. The average is taken over 60,000 operations. The number of attributes involved in each operation varied from 1 to 30 and the number of files involved varied from 1 to 20.

Operation	Ext2fs (ms)	Backend (ms)	% overhead
Create	2.00	1.25	62.5%
Delete	19.1	3.99	20.8%
List	2.49	1.11	44.6%
Move	19.2	17.44	90.7%

Move has a huge overhead, because in our semantics a single move operation may move many files, each with many attributes. This results in a large number of database operations. The graph in figure 1 shows how the backend time for each operation varies with the number of attributes involved. The linearity of the graph is a direct consequence of the fact that each attribute is stored in a separate database table.

4.2 Andrew benchmarks

In order to test how the performance of our proposal scales, we ran a modified Andrew benchmark. We made two significant changes to the benchmark. First, we replaced the small payload of the benchmark with heavier payloads. The payloads we used are (1) a standard client side NFSv2 server (110 files), (2) openSSH version 2.5.2 (303 files) and (3) Linux kernel 2.6.14.4 (20,877 files). The second modification we made is that we added a phase that removes all intermediary files (like *.o files) from any previous run of the benchmark. This phase was added so that we could test our system on file remove commands.

We ran the benchmark on a mounted NFS system and collected call traces in the NFS server. Then we ran a simulation of the query parse and database calls that this trace would generate

in our semantics and timed them on our backend. All traces were collected on a cache that had been warmed up by running the benchmark once before. During this simulation, we used only two keys for file attributes: filename and physical path. The results of this experiment are shown below.

Benchmark	Trace length	Time on ext2 (s)	Backend (s)	% overhead
NFSv2 server	799	5.63	0.48	8.52%
OpenSSH 2.5.2	8064	24.48	9.64	39.37%
Kernel 2.6.14.4	254,418	660.33	2574.92	389.94%

The second column above shows the number of `lookup, rename, create, delete` and `readdir` NFS calls made by the benchmark. All other NFS commands do not require any database/parser access and hence were not traced. The third column is the time taken by the benchmark on the standard linux filesystem. The fourth column is our estimate of the extra time needed by the backend to run the same traces. The last column shows the percentage overhead induced by our file system.

The table indicates a super-linear relationship between the size of the benchmark and the overhead of WSFS. While this is discouraging, we should note two facts. First, our implementation is non-optimized, and uses a standard database. In real practice, ACID guarantees are not required to track filesystem metadata, so the implementation of the backend can be made considerably faster. Second, we use a single file database for the entire filesystem. This slows down searches when there are a large number of files.

4.3 RCS semantics

In order to evaluate the expressiveness of our semantics, we describe below a basic set of operations for an RCS server implemented using only basic shell functionality on a WSFS file system. We use four attributes to track a file's RCS status. The attribute `HEAD=1` indicates that a file in the RCS is the most recent version. The attribute `REVISION` indicates a revision number (monotonically increasing). `VERSION` is a repository wide epoch number that indicates a stable state. `WORKING=1` indicates a working copy in the local image of the repository. We posit a simple function `getattr()` which given a key and a file returns the value associated with the key for that file. We note that all operations can be performed with very simple shell operations with the exception of update, which requires additional logic to prevent a new version from overwriting a locally modified copy of a file.

Checkout

```
cp -r /rcsroot/HEAD=1/ /sandbox/WORKING=1/
```

Checkin

```
R = getattr(REVISION, /rcsroot/HEAD=1/file.cpp) + 1
mv /rcsroot/HEAD=1/file.cpp /rcsroot/HEAD=0/file.cpp
cp /sandbox/file.cpp /rcsroot/HEAD=1/REVISION=$R/file.cpp
```

Add

```
cp -i /sandbox/file.cpp /rcsroot/REVISION=1/file.cpp
mv /rcsroot/REVISION=1/file.cpp HEAD=1/
```

Tag Branch

```
cp -r /rcsroot/HEAD=1/ /rcsroot/HEAD=0/VERSION=2/
```

Rollback

```
cp -rf /rcsroot/VERSION=2/ /sandbox/WORKING=1/
```

Update

```
RA = getattr(REVISION,/sandbox/WORKING=1/file.cpp)
RB = getattr(REVISION,/rcsroot/HEAD=1/file.cpp)
if [$RB > $RA]
   if [/sandbox/WORKING=1/file.cpp -nt /rcsroot/REVISION=$RA/file.cpp]
       mv /sandbox/WORKING=1/file.cpp WORKING=0/
   fi
   cp -f /rcsroot/HEAD=1/file.cpp /sandbox/WORKING=1/
fi
```

Similar semantics can be used in any application that involves manipulation of large logically defined sets of files. Examples of such applications include bibliographic management, photographic workflow, implementation of access control lists for filesystems like AFS, and file sharing across multiple project domains.

5 Related Work

SFS [GJSJ91] originated the concept of a semantic file system. The proposed semantics use virtual directories to browse the attribute space. Queries can be formulated only on exact matches, and only with conjunctive operators. Additionally, the attribute space is read only. Metadata is created and updated only through the use of automated extractors, giving the user no control over the key/value pairs. In a subsequent note [OG92], the authors argue the usefulness of semantic attributes, based on their experience with the semantic file system. They highlight the effectiveness of semantic attributes in organizing and searching vast and disparate collections of files such as those found in the USENET newsgroups.

LISFS [PR03] presents a logic based file system for providing uniform associative and navigational access. File metadata is in the form of boolean properties. These are used to simulate key-value pairs, but only with a significant space and time overhead. LISFS restricts the source of `mv`/`touch` to conjunctions of attributes. In contrast, we allow arbitrary groups of files as sources of `mv`/`touch`. LISFS uses a complicated mechanism involving logical entailment to provide hierarchical attributes. This is non-intuitive. We prefer to use the '/' operator for providing hierarchical attributes.

HAC [GM99] provides content based access capabilities on top of a hierarchical file system. Files can either be accessed by navigating through the hierarchy or by specifying a query. A query in the HAC file system consists of two parts: a set of keywords, and a path in the hierarchy which limits the scope of the search. The response to a query is defined as the set of all files whose content matches the given keywords and are contained in the scope defined by the path. Given a query, HAC creates a new semantic directory with symbolic links to all the query results. In contrast to our approach, semantic directories are physically created on the disk. These semantic directories are writable, and the user is free to add or remove files from the query results. However, modifying physically existing semantic directories can lead to an inconsistent file system (e.g., previously out-of-scope files that have been moved into the scope of a query must show up in the corresponding directory). Maintaining consistency is a major overhead in the HAC approach. Further, the results for a single query can be large in number and since no further organization of results is provided, it can be cumbersome to navigate through them. Query refinement leads to the creation of a new subdirectory with replicated symbolic links. Also, HAC limits associative access to the content of a file, and a file can not be assigned arbitrary associative attributes.

[SM92] provides navigational support by enforcing hierarchical constraints over the associative attributes. It explains a multi-structured naming system which tries to blend traditional hierarchical or graph structured naming (e.g., the UNIX file system) with flat attribute or set based naming (e.g., SFS [GJSJ91]). It attempts to combine the "sense of place" present in graph-based naming with the ability of set-based naming to retrieve files using any combination of information about them. Rules can be defined over attributes which determine context or scope in which the attributes are valid. Using these rules, the user can define ancestor-descendant relationships on labels, and selectively loosen the relationships so that users can name files by path names containing labels, queries or a combination of both. However, attributes are boolean and queries return virtual directories which are not modifiable.

Essence [HS93] is an indexing tool that extends the idea of automatic extraction of file attributes. Essence differs from the semantic file system in that it extracts metadata from files depending on file types and produces summaries of files as well. It is primarily a search tool, and provides no navigational access in the form of virtual directories.

Nebula [BDB+94] treats a file as an object consisting of a set of attributes. File objects have well-specified types, which define the kind of attributes that they can possess (e.g., a file with type `C-source` needs to have an attribute `includes`). Wherever possible, Nebula infers such attributes automatically, but a file object can also have arbitrary user-specified attributes. The user can define *views* (i.e., queries) as arbitrary boolean expressions involving file attributes and browse through them. Nebula thus focuses on providing read-only associative access.

SynFS [BJ96] is similar to Nebula, and aims to provide a uniform interface to access information in a heterogeneous environment. It builds a logical abstraction on top of the heterogeneous data sources through a structure called a *synopsis*. A synopsis is a textual summary of the file it represents, and consists of a list of key-value pairs. Abstractly, it is a typed object that defines associated member data values and methods. Synopses help to hide away the heterogeneity of the underlying information sources. A uniform interface for manipulating synopses is then built on top of this layer. This interface has operations for addition, deletion, automatic generation, search and display of the synopsis' attributes. Again, associative access is read-only and no navigational support is provided. However, the basic approach of abstracting heterogeneous data sources into a semantic object can still be integrated with our work.

[XKTK03] presents a high-level design of an extensible schema-based semantic file system. The schema are defined in RDF/XML, and can contain complex relationships, including one-to-many groupings and inheritance. The system provides a default schema, and users are expected to override or inherit this with their own constructions. Attributes are again managed only by the operating system, and no consideration is given to implementation issues, or the formulation of query semantics.

Another context in which associative access has been used is archiving, i.e., being able to access older "versions" of a given file. None of these approaches try to integrate associative access with navigational access. The Inversion File System [Ols93] (and the similar Cedar system [MTX03]) aim to provide users with a rich set of archival services to manage a large data store by leveraging the power of relational databases. Inversion File System is built on top of the POSTGRES database and uses its no-overwrite storage feature to provide "time travel" through different versions of files. In addition, it supports ad hoc search on various file metadata such as owner, file type, etc. However, providing search is more a consequence of using a database backend rather than being the motivation behind building the file system. Further, the database schema predefines the set of attribute names that can be associated with a file, and the user is not allowed to assign arbitrary metadata.

Deviating from the traditional way of organizing files on disk, several file systems have been built on top of a relational database system. Some examples are BFS [Gia99], Oracle's iFS, Microsoft's WinFS and Inversion File System [Ols93]. At the lowest level, the main operations that our file system needs to perform are store, index, retrieve and intersect. The use of a

database is therefore extremely appealing in our context. Given the prevalent use of databases in academic as well as commercially available file systems, we feel that the use of a database is the right choice for us.

In summary, past work on file systems providing associative access suffers from one (or more) of the following drawbacks:

1. Associative access is effectively "read-only", i.e., the user can only view the results of a query, but he can not pro-actively add files to the set of results so that they can be retrieved if the same query is issued later.
2. No provision for the user to define metadata attributes of his/her choice.
3. No uniform semantics for query and update of metadata.

We improve upon these limitations by proposing a unified semantics for query and update of filesystem metadata. The semantics are intuitive to understand and easy to use.

6 Conclusion and Future Work

We have shown that semantic file systems can be extended with uniform write operations. This enables a very rich file operation semantics, which can directly manipulate logically defined sets of files with single operations. Such a filesystem can be used for several different applications. The challenging part about this design is implementation. In particular, a database backend (or its equivalent) is needed to maintain file metadata. Such a backend works well for small sets of files, but with our naive implementation performance scales poorly as the number of files is increased. We expect that this degradation in performance can be avoided through a carefully designed metadata store. This is a promising avenue for future research. Another direction of future work is a complete implementation of a filesystem that includes a front end. Once this is done, it will be interesting to verify the utility of our semantics via a user study. Finally, our file operation semantics do not incorporate concurrent access by multiple users. Such extensions would be an obvious choice for future work.

7 Relationship to course material

The course focuses on design and analysis of selected aspects of operating systems and distributed systems. It examines design of various filesystems (e.g., LFS, DAFS, NFS, etc.) which are a core component of any operating system. Our project aims to provide an intuitive and useful design of the filesystem as a multi-dimensional semantic file store. As part of the project, we have implemented our design's backend and analyzed it using a number of evaluation strategies like microbenchmarking and emulation of a revision control system. Our project incorporates work on filesystems, databases and RPC(NFS) which are major concepts discussed in the course.

References

[BDB+94] C. Mic Bowman, Chanda Dharap, Mrinal Baruah, Bill Camargo, and Sunil Potti. A File System for Information Management. In *Proceedings of the ISMM International Conference on Intelligent Information Management Systems*, March 1994.

[BJ96] M. Bowman and R. John. The Synopsis File System: From Files to File Objects. In *Workshop on Distributed Object and Mobile Code*, 1996.

[Gia99] D. Giampaolo. *Practical file system design with the Be file system*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 1999.

[GJSJ91] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. O'Toole Jr. Semantic file systems. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 16–25. Association for Computing Machinery SIGOPS, 1991.

[GM99] Burra Gopal and Udi Manber. Integrating content-based access mechanisms with hierarchical file systems. In *Operating Systems Design and Implementation*, pages 265–278, 1999.

[HS93] Darren R. Hardy and Michael F. Schwartz. Essence: A resource discovery system based on semantic file indexing. In *USENIX Winter*, pages 361–374, 1993.

[MTX03] M. Mahalingam, C. Tang, and Z. Xu. Towards a semantic, deep archival file system. In *The 9th International Workshop on Future Trends of Distributed Computing Systems (FTDCS)*, 2003.

[OG92] James W. O'Toole and David K. Gifford. Names should mean what, not where, 1992.

[Ols93] M. A. Olson. The Design and Implementation of the Inversion File System. In *Proceedings of the USENIX Winter 1993 Technical Conference*, pages 205–217, San Diego, CA, USA, 1993.

[PR03] Y. Padioleau and O. Ridoux. A logic file system. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 99–112, San Antonio, TX, June 2003.

[SM92] Sechrest and M. McClennen. Blending hierarchical and attribute-based file naming. In *Proc. of 12th International conference on Distributed Computer Systems, Yokohama, Japan*, 1992.

[XKTK03] Z. Xu, M. Karlsson, C. Tang, and C. Karamanolis. Towards a semantic-aware file store, 2003.