AUTOMATED CONTROL OF HARDWARE AND SOFTWARE-BASED FAULT ISOLATION

by

EDWARD L. CASHIN

(Under the direction of David K. Lowenthal)

ABSTRACT

Components are commonly used to rapidly create large software systems. In many such systems, composition of components is done between different groups or vendors. In these cases, reliability of the entire system is threatened by individual components that may not themselves be reliable or trustworthy.

In this thesis we present two techniques for protecting trusted applications from the components that extend them, while still maintaining efficient execution. One technique, Memory Access Control Contexts, is our own novel approach to hardware-based fault isolation. We contrast our new technique with a traditional solution, Software Fault Isolation. Recognizing these techniques as complementary, we present an experimental design for the automated selection of the most efficient protection mechanism. Our approach makes the appropriate choice close to 90% of the time on our microbenchmarks.

INDEX WORDS: Fault isolation, Page protection, Memory management, Security, Automated software engineering
AUTOMATED CONTROL OF HARDWARE AND SOFTWARE-BASED FAULT ISOLATION

by

EDWARD L. CASHIN

A.B., The University of Georgia, 1993

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2004
AUTOMATED CONTROL OF HARDWARE AND SOFTWARE-BASED FAULT ISOLATION

by

EDWARD L. CASHIN

Approved:

Major Professor: David K. Lowenthal

Committee: Kang Li
Liming Cai

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
August 2004
Many thanks go to those who helped in the research described in this thesis, including David Lowenthal, Barry Rountree, Scott Watterson, Andrew Seila, Kang Li, Rupa Krishnan, and Haijin Yan.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledgments</td>
<td>iv</td>
</tr>
<tr>
<td><strong>Chapter</strong></td>
<td></td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2. Overview</td>
<td>4</td>
</tr>
<tr>
<td>2.1 Extension Components</td>
<td>4</td>
</tr>
<tr>
<td>2.2 Hardware-based Fault Isolation</td>
<td>5</td>
</tr>
<tr>
<td>2.3 Software-based Fault Isolation</td>
<td>7</td>
</tr>
<tr>
<td>2.4 Memory Access Control Contexts</td>
<td>7</td>
</tr>
<tr>
<td>3. Implementation</td>
<td>10</td>
</tr>
<tr>
<td>3.1 MACCs</td>
<td>10</td>
</tr>
<tr>
<td>3.2 SFI Implementation</td>
<td>13</td>
</tr>
<tr>
<td>3.3 A Dynamic Strategy for Fault-Isolation</td>
<td>14</td>
</tr>
<tr>
<td>4. Performance</td>
<td>17</td>
</tr>
<tr>
<td>5. Discussion of Results</td>
<td>22</td>
</tr>
<tr>
<td>6. Conclusion</td>
<td>23</td>
</tr>
<tr>
<td>Bibliography</td>
<td>24</td>
</tr>
<tr>
<td><strong>Appendix</strong></td>
<td></td>
</tr>
<tr>
<td>A. MACCs on the Intel Architecture</td>
<td>26</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Component-based programming is well recognized for its ability to help programmers rapidly create stable software [9]. Under a component-based model, applications are created by combining components and are extensible at runtime by loading new components on the fly. This thesis focuses on the tradeoff between the efficiency and safety of extension components. In particular, programmers often chose to use object code native to the machine running the application in order to achieve high efficiency. However, executing native code leaves the application as a whole vulnerable to the problems extension components may introduce.

A buggy or malicious extension component may render useless an otherwise reliable application. Furthermore, if an extension fails, we would like to determine precisely that the failure was due to the extension, as opposed to a bug in the existing application. One way to do this is to use interpreters or virtual machines (e.g., the Java Virtual Machine); unfortunately, doing so adds possibly unacceptable overhead to the entire program.

One traditional solution is to use separate processes, one for the application and one for each extension component. While this avoids the overhead of interpreting the program, the interprocess communication (IPC) and scheduling overheads of this approach may be unacceptable. The key result in this area is software fault isolation, which inserts extra instructions into an extension’s code [12]. These instructions effectively wrap loads, stores, and jumps such that they are “sandboxed,” or confined to a safe area of memory. SFI works at the cost of runtime overhead added in proportion to the number of memory accesses made in the extension components.
We have developed a new protection mechanism that takes advantage of hardware memory protection features in a portable way, avoiding the per-memory-access overhead of SFI, while also avoiding the scheduling and IPC overheads that come with the use of separate processes. We call our protection domains Memory Access Control Contexts (MACCs), which realize a new abstraction different from processes. Just as different program threads are independent instances of the same program, distinguished by CPU state, MACCs are different views of the same address space, distinguished by protection settings. Using MACCs, we can easily accommodate modern multi-threaded, multi-component applications while maintaining control over the memory accesses of untrusted extension components.

While MACCs are a useful abstraction with performance that is often superior to SFI, there are some cases where they are not better. This is because while MACCs have less overhead per memory access, they have a higher domain switching overhead and consume more system memory. In order to provide an effective and efficient way to isolate trusted applications from the faults of untrusted extension components, we have also designed an automated system that can select the appropriate isolation mechanism at runtime. It uses memory and workload information to make predictions about the relative performance of SFI and MACCs, choosing the mechanism that will perform best given the current runtime conditions. Results show that our prediction methodology is correct nearly 90% of the time.

Our contribution is twofold. First, we provide MACCs, a new portable way of taking advantage of hardware-based fault isolation while avoiding the overhead of IPC. Second, we have designed an experimental mechanism for choosing between MACCs and SFI in a running extensible application.

The rest of this thesis is organized as follows. Chapter 2 gives an overview of the problem we are addressing and discusses related work. Chapter 3 discusses the implementations of SFI, MACCs, and our system for automated selection of the most efficient
fault isolation technique. Next, Chapter 4 analyzes the relative performance of SFI and MACCs and evaluates our dynamic strategy. Finally, Chapter 5 discusses the results and Chapter 6 concludes.
CHAPTER 2

OVERVIEW

This chapter begins with a discussion of how fault isolation relates to component-based programming. We then present existing solutions and our new fault isolation technique, MACCS.

2.1 EXTENSION COMPONENTS

Most modern operating systems allow programs to be extended at runtime by loading native code from dynamically linked library files (dll’s). These runtime-loadable libraries are often referred to as “plugins” when considered from the perspective of a particular application, e.g., Netscape. An application uses plugins to provide features that may not have been envisioned by the designers of the application itself. Note that in this thesis we assume that multiple repeated calls to plugins have similar behavior, as is typically the case. The following pseudocode shows how an application may load and use a plugin named flashplayer. Assume that function work is called from the plugin.

```c
plugin = dlopen("flash.so");
result = plugin.work();
```

Components can be used together when their properties are well defined. To use plugins as components that have at least some degree of sharing with the rest of the (main) application (see Figure 2.1(a)), it is helpful to guarantee that the sharing will actually fit the advertised properties of the extension component. Through fault isolation techniques,
an application can ensure that components conform to the memory access behaviors that they specify. We next discuss in turn the two major classes of fault isolation mechanisms: hardware-based and software-based fault isolation.

2.2 HARDWARE-BASED FAULT ISOLATION

Perhaps the most straightforward technique for enforcing limits on sharing is to use separate processes to create separate address spaces (see Figure 2.1(b)). Below we show pseudocode from the same example as shown above, except that different processes are used.

```c
who = fork();
if (who == child process)
    plugin = dlopen("flash.so");
    result = plugin.work();
else if (who == parent process)
    // main application
```
For communication, the processes can use any of the standard IPC mechanisms, such as shared memory, pipes, message queues, or even an external Database Management System.

Putting the extensions in separate processes implicitly provides safety for the main application. This is because each process has a distinct one-to-one mapping from a series of virtual addresses to a series of \( \langle \text{page, offset} \rangle \) pairs. The page of data may be in physical memory or on disk; the critical point is that for different processes, the same virtual address does not map to the same physical page.\(^1\)

The advantage of using a separate process for an extension component is that the hardware enforces memory protection at no cost. However, there are two primary disadvantages. First, it is expensive to switch between different processes. Second, the ease of data sharing between the components is greatly reduced.

Numerous approaches to enhancing the efficiency of inter-process data sharing have been investigated with some success [2, 5]. It is worth noting, however, that extensible multi-threaded single-process programs are dominant among the most popular user applications today.

Recent work has made use of more sophisticated hardware-based protection. Examples include Palladium and Xen, which make use of the Intel architecture’s four-level security rings feature [3, 1]. These techniques provide excellent performance and a high degree of control. However, they are specific to the the x86 architecture and so are difficult to employ on new or varied architectures. In general, it is difficult to make use of hardware fault isolation in a portable way. For this reason, researchers have also investigated ways to isolate faults in software.

\(^{1}\)Features do exist in UNIX that allow the user to explicitly request that specific virtual addresses map to certain pages.
2.3 **Software-based Fault Isolation**

A different approach to fault isolation is to catch all faults via software checks. This was popularized by the fundamental result in this field, software fault isolation [12]. The basic idea is to allocate a region of memory for the extension code to use and then to insert extra protection instructions around every memory access and every jump instruction in the extension. From the point of view of the user, the interface looks exactly the same as the example code fragment given in Section 2.1. IPC is typically done using a lightweight domain crossing strategy such as lightweight remote procedure call (LRPC) [2].

The extra instructions of SFI force the extension code’s accesses to fall within the extension itself. The most efficient of the SFI techniques is called sandboxing; the idea is to simply set the most significant bits of the virtual address being used to the most significant bits of all virtual addresses of the safe area.

While the SFI creators argue that the technique could be used on pre-compiled object code, their implementation inserts instructions into source code that must be assembled into object files. Because their implementation works on reduced instruction set computers (RISC architecture), it is not difficult to identify places in the code where sandboxing is necessary.

There exist systems that use SFI on a CISC (complex instruction set computer) architecture [8]. Whereas on a RISC SFI must consider only loads, stores, and jumps, SFI on a CISC must consider many more instructions that could potentially lead to memory accesses. This added complexity makes the task of determining which instructions may be unsafe more difficult and more susceptible to error.

2.4 **Memory Access Control Contexts**

A MACC is conceptually just a protection domain. Two MACCs share the same address space—they have the same one-to-one mapping of virtual addresses to ⟨page, offset⟩ pairs.
The “X” on the right-hand MACC shown in Figure 2.1(c) signifies that the MACC on the right has fewer access privileges than the one on the left. The key distinction between MACCs is their protection settings, which may be set independently.

For example, consider a virtual address $v$ that corresponds to a physical address $w$ on page $p$. The trusted components of an application can run in one MACC where there is both read and write access to $p$ when accessing memory at address $v$. A second MACC may be created for untrusted components, and in this second MACC page $p$ may be read only. If the untrusted components then attempt a write to virtual address $v$ an exception is generated (via a segmentation fault), and a trusted component can regain control.

MACCs provide a single new feature that extends the multi-threaded, single-address-space programming model in a natural way, by allowing different parts of an application to have different access rights. Programmers do not have to adopt an unfamiliar programming model to use MACCs.

Essentially, MACCs provide a solution to both of the problems associated with using separate processes: much of the overhead in process switching is eliminated (specifically, process scheduling overhead), and the natural sharing model is used. As the modifications are at the OS level, it is a portable solution not tied to any one specific architecture. Note that while the MACC switching overhead is reduced compared to processes, there are times when this overhead is enough to make SFI a better choice for high performance. We will recognize such a case at runtime and use SFI as the isolation mechanism. Section 3.3 contains the full details on this automated decision-making process.

The MACC implementation discussed in Section 3.1.2 is similar to the nooks [10] system in that the replication of page tables is a core strategy in both implementations. However, MACCs are designed to create protection domains only in user-space, whereas nooks creates protection domains only inside the kernel. MACCs can be used to protect an application against buggy or malicious components, whereas nooks is only limited to providing protection against buggy kernel extensions.
The Millipage implementation bears superficial resemblance to MACCs. Whereas multiple MACCs allow different protection settings for a given virtual page, millipage creates multiple virtual page mappings to the same physical page. This is because Millipage is concerned only with preventing thrashing in distributed shared memory systems [7]. The application using Millipage retains full access to every virtual page. On the other hand, MACCs are used for inter-component protection purposes.
CHAPTER 3

IMPLEMENTATION

We have implemented Memory Access Control Contexts by extending the Linux kernel. In this chapter we first describe the MACC interface and implementation. Next, we discuss the SFI implementation. Finally, we describe how we determine which isolation technique to use at run time.

3.1 MACCs

The trusted components in an application make use of our new MACC-specific kernel features by using four new system calls and a user-level gateway function. We discuss the interface and implementation in turn below.

3.1.1 MACC INTERFACE

The MACC interface consists primarily of five interface routines: macc_create, macc_switch, macc_call, protect_regions, and load_untrusted. These are described below.

1. macc_create

1There exist i386 and sparc64 ports of our MACC extensions to Linux. Because our SFI implementation is sparc64-specific, we used sparc64 exclusively as the experimental platform for the work presented in this thesis; therefore, we reserve discussion of the i386 MACC implementation for Appendix A.
An application creates a new protection domain with a call to `macc_create`. On the first such call, a *macc family* is created so that all the MACCs in a process can be easily found.

2. **macc_switch**
   Once a process contains more than one MACC, trusted code can switch between different MACCs using the `macc_switch` system call.

3. **macc_call**
   The `macc_call` function is the gateway between the trusted and untrusted protection domains. It simply calls `macc_switch` and sets up the extension function’s program stack.

4. **protect_regions**
   A call to `protect_regions` allows the application to configure the protection settings for a MACC in just one system call. The `protect_regions` system call implementation is similar to that of the POSIX system call, `mprotect`, except that it can handle disjoint regions.

5. **load_untrusted**
   The trusted part of the application uses `load_untrusted` to bring untrusted code into its address space.

   The pseudocode below shows how an application using MACCs loads an untrusted extension component, using the same example as in the previous chapter.

   ```c
   plugin = load_untrusted("flash.so");
   plugin_macc = macc_create();
   result = macc_call(plugin_macc, work);
   ```
3.1.2 MACC Implementation

The `macc_create` system call is implemented by a copy of the caller’s in-kernel data memory management structures. In Linux these data structures are (1) the page tables, (2) the virtual memory area descriptors, and (3) the memory management descriptor that contains pointers to these data structures as well as accounting information. On a `macc_switch` system call, the kernel simply associates the calling thread with another memory management descriptor, namely the one associated with the requested MACC. The `protect_regions` system call is implemented by calling `mprotect` iteratively without leaving the kernel. The `load_untrusted` call causes the kernel to record that a region of code is not allowed to escape its protection domain, e.g., by calling `macc_switch, macc_create, protect_regions, or mprotect`.

Our kernel extensions manage all of the MACCs associated with a process. A normal process has only one MACC. For each page of data that the process uses, there is a page table entry (PTE) describing the location of the page (physical memory or secondary storage). The page table entry also specifies whether the page is readable, writable, and executable.

When a page is accessed its PTE may be absent or the page may not be in physical memory. At such a time, the hardware lets the operating system take control and handle the page fault. The operating system moves the page into physical memory if the page is in secondary storage and the access is permitted. Otherwise, the access is not legitimate, and the kernel terminates the process.

On a page fault for a legitimate access, our system recognizes a process that has a family of MACCs and can easily tell in which MACC the faulting thread is running. It looks at all the “sibling” MACCs belonging to the process, looking for a usable PTE for the faulting page. If a sibling MACC has a PTE, the page permissions are adjusted according
to the protection settings of the current MACC, and the adjusted PTE is inserted into the page tables of the current MACC so that the memory access may proceed.

In addition to the page tables, each MACC has a series of virtual memory area descriptors (VMAs). These descriptors contain information about ranges of virtual memory. A file that is memory mapped will have its own VMA, as will a range of memory that has distinct permissions.

While changes to page table entries may be propagated lazily by putting off the propagation until a page fault occurs, our system must eagerly propagate changes to these VMAs as soon as they occur, most notably on the mmap and brk system calls. Otherwise, when one thread in MACC A expands the heap via malloc, another thread in MACC B could be (incorrectly) terminated upon trying to use the newly-allocated heap memory.

### 3.2 SFI Implementation

The SFI implementation we used is the more efficient of two techniques described in Wahbe et al., namely sandboxing [12]. This technique entails allocating a specific region of memory that an untrusted extension component may use. The extension component itself is then modified. Its load, store, and jump instructions are sandboxed, or “wrapped.” Each wrapped instruction is preaced by an arithmetic operation that confines the resulting address to the safe region.

As an example, consider the case where we have set aside a region of memory from virtual address (in hex) 401000 to virtual address 401fff. Our SFI implementation needs to force all memory accesses to fall within that area. If a store instruction like the one below is seen, it is possible that register r1, contains an address that is outside of the untrusted component’s designated area. Therefore, it is necessary to sandbox it as follows.

```
and r1, 0x000fff --> r1
or  r1, 0x401000 --> r1
store r2 --> [r1]
```
Line one clears the most significant bits from the address. Line two sets the most significant bits so that the address falls inside the allocated memory region. When the store occurs on line three, the memory access is safe.

3.3 A Dynamic Strategy for Fault-Isolation

As will be shown in Chapter 4, the MACC and SFI techniques excel in different circumstances. Therefore, our goal is to, given a plugin whose runtime behavior we know nothing about in advance, choose the technique that provides better performance. Note that the runtime conditions cannot in general be determined statically. This is because a static approach, even if possible, would depend on precisely determining how much work an extension does as well as its working set, which are both extremely difficult problems for an optimizing compiler. (Furthermore, the plugin is likely separately compiled.)

In this chapter we discuss the complementary nature of MACCs and SFI, pointing out the circumstances in which each is preferable. Then, we propose a dynamic strategy, one that makes use of both MACCs and SFI to provide the most efficient solution for a particular application, extension component, and runtime environment.

3.3.1 Overheads of MACCs and SFI

MACCs provide a highly flexible foundation for applications that require predictable cooperation between components. However, they contain very different overheads from those of SFI.

In particular, SFI incurs a per-memory instruction overhead. That is, in any untrusted component, every load, store, or jump instruction is potentially dangerous and must be wrapped. Every such instruction requires execution of a number of extra “wrapper” instructions, typically two. The overhead incurred by these instructions is most significant when memory accesses hit in the CPU cache. This is because a cache miss itself is several
orders of magnitude larger than that of a cache hit. The cost of the extra sandboxing
instructions is more than the time for a cache lookup on our primary experimental plat-
form, sparc64—but it is far less than the time for an access to main memory. Note that
SFI also potentially incurs extra instruction cache misses because of an increase in the
number of instructions [12], but we did not find this effect to be significant.

On the other hand, MACCs add overheads that are not per memory access, but rather
primarily per MACC switch. On each MACC switch, there is a fixed overhead for the
round-trip into the kernel through the macc_switch system call (which we added to Linux).
In addition, threads running in different MACCs cannot share page table entries that are
cached in the translation lookaside buffer (TLB). Consequently, a MACC switch incurs a
small number of TLB misses. Third, increasing the frequency of MACC switches (as well
as the number of MACCs) means that more space is consumed by in-kernel, process-level
memory management data structures, as well as additional overhead to keep them con-
sistent. In particular, the kernel must make sure that the shared address space remains
consistent while enforcing the distinct protection configurations of the different MACCs.
This overhead is incurred, for example, the first time a thread in a MACC uses a page, or
when a call to a memory allocator must call on the kernel to expand the heap area.

Overall, MACCs are a better choice when their switching overhead is outweighed by
SFI’s per-memory-access overhead. SFI is a better choice for extension components that
have few memory accesses and return control quickly to trusted code. (See Chapter 4.)

3.3.2 Selection Strategy

By using the CPU performance counters and information exported by the kernel, our
system collects information that reflects the behavior of the application and the state of the
runtime environment. To create a predictor that can estimate the performance of MACCs
and SFI, we use the formula below. (The terms of the formula are defined before the esti-
mation formula itself is presented.)
\[ s \equiv \text{switching rate} \]
\[ m \equiv \text{cache hits in extension} \]
\[ b \equiv \text{size of working set} \]
\[ \hat{T} \equiv \text{estimated running time} \]
\[ \hat{T} = \alpha + \beta_0 s + \beta_1 m + \beta_2 b \]

Each coefficient\( \beta_i \) and the constant term\( \alpha \) may be found mathematically using any number of statistical methods. The most straightforward method is to minimize the squares of the differences between estimated runtimes and observed runtimes.

Consider one data point, where there is an observed rate of switching between protection domains, called\( s \), an observed number of cache hits\( m \) per call to the untrusted extension component, an observed working set size\( b \), and coefficients\( \langle \alpha, \beta_0, \beta_1, \beta_2 \rangle \). The formula finds an estimated runtime\( \hat{T} \). The coefficients that minimize\( (T - \hat{T})^2 \) summed over all such data points are the ones that we can use for predicting runtime behavior. Finding coefficients in this way is a simple application of linear regression.

Offline, we apply this technique to data generated by programs running under identical conditions in both SFI and MACC isolation modes. The observations collected in SFI mode are\( \langle s_S, m_S, b_S \rangle \), where the subscripts denote the observation context, SFI. Likewise, from the MACC-mode run we have observations\( \langle s_M, m_M, b_M \rangle \). Our system must make predictions for SFI runtime,\( \hat{T}_S \), and for MACC runtime,\( \hat{T}_M \), using either MACC observations or SFI observations. In this way an application may make predictions about the performance of both modes using only observations collected in its current mode.

In the next chapter we quantify the effects of the terms in our predictor formula and examine the effectiveness of the predictors in selecting the most efficient isolation strategy.
To study the performance of MACCs and SFI, we conducted tests on a Sun Ultra 5 workstation with a 360MHz UltraSPARC III CPU and 128 Megabytes of physical memory as well as a Sun Ultra 60 with a 296 MHz UltraSPARC II CPU and 1024 Megabytes of physical memory. During the tests, the machines were unloaded with the exception of daemon processes.

To accurately observe the overheads of the two isolation strategies, MACCs and SFI, we created a synthetic benchmark program that could switch between protection domains at a specified rate, performing specified amounts of memory-intensive and cpu-intensive work. To increase the accuracy of our timing measurements we used the CPU’s “tick” register to measure time with high resolution and without the need for a system call.

While there are many possible runtime observations that may affect predictions, we chose to focus on three key indicators: protection-domain switches over time (the switching rate), the number of cache hits per call to the untrusted component, and the working set size. We consider only cached memory accesses because cache misses incur a latency that is equally large for MACCs and SFI, and that latency is large enough to overshadow the overhead of SFI’s extra sandboxing instructions.

Figures 4.1 and 4.2 show how these three observable quantities correspond to the most critical overheads for MACCs and SFI. We measure the overhead as the increase in runtime over the “no isolation” baseline run. Each figure shows four plots, with the working set size increasing as the plots are read from left to right and top to bottom. (The size of the
working sets, shown in pages, correspond to data areas of 1, 25, 625, and 15625 kilobytes in size.) Each data point on the plots represents the median from three redundant runs of our synthetic benchmark program mentioned above.

The plots in Figure 4.1 show clearly that MACC overhead is not sensitive to an increased amount of memory-intensive work. SFI overhead, in contrast, depends strongly on the amount of memory-intensive work performed. Recall that SFI adds extra sand-boxing instructions for every memory reference. We varied the working set size from small to large in order to observe whether the contrast was most pronounced when the working set was small and the TLB and CPU caches are effective. We found that to be the case. As the working set size increases, the number of TLB misses caused by
MACC switching becomes more significant. Nevertheless, even for large working set sizes MACCs are superior to SFI as the percentage of memory-intensive work grows large.

Figure 4.2 shows runtime overhead increasing as the rate of switching between protection domains increases. When switching rates are low, MACCs are superior. As the switching rate increases, the SFI overhead does not increase significantly, but MACCs incur a penalty for high rates of protection-domain switching. This effect is magnified when the working set size is large.

It is worth noting that these results are from tests where the performance of MACCs and SFI are not widely separated (the difference in overhead is under 20%), and the predictors are performing well. We have run tests where SFI overhead is as great as 98.90% and
MACC overhead is up to 73.22%, and the predictors continue to perform well, choosing the correct strategy 86.57% and 83.33% of the time based on MACC-context observations and SFI-context observations, respectively.

By running our synthetic benchmark program with a large number of combinations of settings for memory-intensive work, switching rate, and working set size, we collected observations that describe a surface in four-dimensional space relating these three variables to runtime. We use the statistics software package, R [6], to fit our formula to this surface. The resulting formulas are listed below, with subscripts M and S corresponding to MACCs and SFI, respectively. These predictors were used for the results presented in Table 4.1.

Predicting from MACC observations:
\[
\hat{T}_S = 4675 + 0.009565 m_M - 11.13 s_M + 0.00017 b_M
\]
\[
\hat{T}_M = 4649 + 0.009064 m_M - 5.795 s_M + 0.0001939 b_M
\]

Predicting from SFI observations:
\[
\hat{T}_S = 4647 + 0.009616 m_S - 85.37 s_S + 0.0001702 b_S
\]
\[
\hat{T}_M = 4641 + 0.009071 m_S - 40.554 s_S + 0.0001936 b_S
\]

To evaluate the success rate of our dynamic system for isolation technique selection, we apply our four fitted models to the data. For example, we have data from synthetic benchmark program runs using both SFI and MACCs. We use the observations from a MACC run to make predictions about the performance of MACCs and SFI, using the macc-mode predictors for \(\hat{T}_M\) and \(\hat{T}_S\), respectively. We choose the fault-isolation mechanism with the lowest predicted runtime. We also have synthetic benchmark program runs under the same conditions but using SFI for isolation, so it is easy to determine whether the predictions correctly identify the better option (MACCs or SFI).
Under conditions similar to expected application behavior, we find that our system chooses the correct option 87% of the time given MACC-mode observations and 89% of the time given observations collected in SFI mode. For choosing correctly, the average benefit was 10% of running time. For choosing incorrectly, the average cost was 1% of running time. We conducted two tests, called “low work” and “high work” according the amount of work done between protection domain switches. Each test consisted of several thousand runs of our synthetic benchmark program. Table 4.1 shows how models fitted to each data set perform when choosing between MACCs and SFI based on the information in each of the data sets.

These results are encouraging. Despite the complexity of the factors affecting performance, the predictors can choose correctly most of the time. An end user could gain a performance benefit about 83-90% of the times the predictor makes a decision, and with low risk—the benefit for a correct prediction is greater than the cost of a wrong prediction. We discuss the results further in the next chapter.

<table>
<thead>
<tr>
<th>observations</th>
<th>predicting from</th>
<th>% correct</th>
<th>cost when wrong</th>
<th>benefit when correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>high work</td>
<td>MACC</td>
<td>86.16</td>
<td>0.54</td>
<td>2.58</td>
</tr>
<tr>
<td>high work</td>
<td>SFI</td>
<td>85.74</td>
<td>0.85</td>
<td>2.58</td>
</tr>
<tr>
<td>low work</td>
<td>MACC</td>
<td>87.27</td>
<td>0.10</td>
<td>11.31</td>
</tr>
<tr>
<td>low work</td>
<td>SFI</td>
<td>92.56</td>
<td>0.66</td>
<td>10.82</td>
</tr>
<tr>
<td>all</td>
<td>MACC</td>
<td>86.78</td>
<td>0.73</td>
<td>4.72</td>
</tr>
<tr>
<td>all</td>
<td>SFI</td>
<td>89.53</td>
<td>0.77</td>
<td>4.29</td>
</tr>
</tbody>
</table>

Table 4.1: Predicting performance of isolation techniques
Our results show that the primary MACC and SFI overheads are complementary, occurring in mutually exclusive circumstances. Further, using simple linear models, we have been able to correctly predict the effects of these overheads. Finally, we have demonstrated that significant runtime savings may be achieved (10% on average in our tests) by selecting the most efficient fault isolation strategy.

We selected linear models as our runtime estimators because they have the advantages of simplicity and comprehensibility. There is no reason, however, to assume that our results are the best possible. Noting that some runtime-observable indicators (like competition for memory) result in a non-linear change in runtime, we expect that more powerful modeling methods may outperform our linear models. Promising techniques include generalized additive modeling and projection pursuit regression [11, 4].

Projection pursuit regression can accommodate variables that are not independent, making it a promising candidate for our dynamic strategy. By including a larger number of possibly inter-dependent variables, our system will be able to respond correctly to a greater range of environmental pressures.
Two complementary techniques that make extensible component-based applications more reliable are software fault isolation, a traditional technique based on extra sandboxing instructions, and Memory Access Control Contexts, our own hardware-based fault isolation technique. We have demonstrated that it is possible to make predictions about their relative performance based on observed runtime application and system characteristics, resulting in significant gains in performance. Our experimental results suggest that it is practical and efficient to automate the choice of isolation strategy at runtime.

Our performance predictions are based on linear models of the relationships between runtime-available information and application running time. They are accurate when extension behavior is stable and the relationships are roughly linear. We expect that as we include more runtime indicators and move to more powerful modeling techniques, the automated selection of isolation technique will continue to be made more general and more accurate.


The Memory Access Control Contexts (MACCs) technique allows the creation of multiple trust domains within a process. The technique is easily portable to various architectures and (with somewhat more work) operating systems. Because our implementation of SFI was targeted to the UltraSPARC architecture, this thesis focuses on the MACC implementation for the UltraSPARC. The first implementation of MACC support for the Linux kernel was not targeted to the UltraSPARC but to the popular Intel i386 family of processors (x86).

For completeness we discuss the x86 implementation below. Rather than simply listing architectural differences between the UltraSPARC and the x86 platforms, we instead list and describe the major issues associated with the transition from the x86 implementation of MACCs to the UltraSPARC architecture. In this way we aim to show that MACCs are easily portable while touching on the differences between the UltraSPARC and x86 implementations.

A.1 Page Tables

A major difference between the UltraSPARC and the x86 architectures is the structure of the page tables, the data structures that define the mappings from virtual pages to the corresponding physical pages. Each entry in the page table maps a single virtual page to a physical page. The physical page of data may be in physical memory or on disk. The page table entry (PTE) also contains information about the protection settings for the page.
The page tables are indexed for quick lookup in a 1024-ary tree, with actual page tables at the leaves. The name “page table” is commonly overloaded and can refer to either the tree or to a leaf of the tree. In Linux parlance, the top level “page global directory” has pointers to middle level directories, which have pointers to the page tables themselves, which hold PTEs. In the Linux source code, the tree appears to have three levels on all architectures, but on x86, the middle level is illusory—the actual data structure has only two levels. On the UltraSPARC the page tables are stored in a true three-level 1024-ary tree.

While knowledge of the two-level page table of the x86 architecture was embedded in some debugging code, most of the MACC kernel extensions followed the Linux convention of using the illusory middle-level page table directories. This architectural difference did not, therefore, lead to any great difficulties in creating the UltraSPARC port.

A.2 Word Size

The x86 architecture has a 32-bit word. It usually goes without saying that the word size is the same for kernel mode and user mode alike. The UltraSPARC, however, has a word size of 64 bits in kernel mode and 32 bits in user mode. On x86 it was easier to move data between the kernel and user memory areas. Greater care was required on UltraSPARC, and the explicitly-sized data types Linux provides (like uint32_t for an unsigned 32-bit integer) helped provide consistency.

A.3 Virtual Memory Layout

On the UltraSPARC the kernel has a completely different address space from the user-space applications. A virtual address is just a number, and the same number means completely different things depending on whether it is used as a virtual address inside or outside the kernel. A virtual address that points to valid data in user-space may be nonsensical inside
the kernel, e.g., pointing to an unmapped region of virtual memory. This separation of address spaces is similar to that between user-space processes.

On x86, in contrast, a given process always has the same mapping from virtual to physical pages, whether it is running in kernel mode or user mode. The virtual address 0xA0000000, for example, maps to the same page in the kernel as it does in user space. This shared mapping makes it easy for kernel code to mistakenly use pointers that have come from user space.

For example, if a system call like `read` provides a pointer to a buffer, it is a bug for the kernel to casually dereference the pointer, writing to the memory at that virtual address. The page of data may not even be in physical memory, so the kernel must use special functions like `copy_to_user` instead of directly using pointers from user space.

Bugs latent in the x86 implementation of the MACC extensions manifested on the UltraSPARC. Whereas user-space pointers could be safely used most of the time on x86, they were nonsense on the UltraSPARC.

### A.4 TLB Miss Handling

The UltraSPARC hardware, like most modern architectures that support virtual memory, has a small cache (realized as several small caches) for page table entries called the translation lookaside buffer (TLB). When a virtual address is used, the physical address of the page is needed for access to the data in physical memory. At that time, the TLB is consulted. If the needed PTE is not already in the TLB, the hardware looks in the page tables to find the PTE, installing it in the TLB. This event is called a “TLB miss.”

Unlike the x86, the UltraSPARC uses software routines for handling TLB misses. The handlers are installed in special locations in memory. These software routines have many restrictions on what they can do and on their size. Compared to the hardware TLB miss
handling of x86, there is a potential for greater flexibility with the software TLB miss handling of the UltraSPARC. However, the tight constraints on the code that does the handling limits that flexibility in practice. Therefore this architectural difference was not an issue for MACCs.
Appendix B

COW Marking

The above issues lead to several puzzles and challenges during the process of porting MACC support from x86 to the UltraSPARC. The COW marking feature described in this appendix was abandoned for two major reasons: 1) contrary to expectations, switching between different trust domains was fastest by simply switching between existing MACCs, not by changing page protections through protect_regions; and 2) the UltraSPARC port was created under severe time constraints, and it saved time to eliminate code rather than port it. Nonetheless, COW marking is worth documenting if only for what the discussion reveals about the inner workings of virtual memory management in Linux.

Most Linux programmers are not aware that it is expensive to make an area of stack or heap memory writable again after previously limiting write access. In fact, all private pages of memory are expensive to change back and forth between writable and non-writable via the mprotect system call. Every first write to such a newly-writable page incurs a minor fault and a “COW break”—copy-on-write is explained below—when the page is copied to a newly allocated physical page. This expense occurs on Linux and FreeBSD, and is likely to occur in other operating systems. In the course of researching the technology that became MACCs, a method for avoiding the overhead of protection switching was discovered. This “COW marking” extension works by recording extra state (but without consuming extra memory) in the kernel when a process uses our new system call, mark_cow, as described below.

We were interested in the fast switching of page protection settings because we were investigating ways of splitting programs into trusted and untrusted parts, allowing the
trusted code security against being disrupted by malicious or buggy untrusted code. On a single processor, the page protection settings might be simply adjusted to more restrictive settings upon entering untrusted code, and then reset upon returning to the trusted code. The Linux support for the copy-on-write (COW) feature, which normally improves performance, had the opposite effect in this case, degrading performance by introducing the needless copying of data in memory.

B.1 COPY ON WRITE

Copy-on-write (COW) is the technique of postponing the copying of data by initially duplicating just a reference to the data [13]. If one of the references is subsequently used to modify the data, the “write” is intercepted and the data is finally copied. In a paged environment, the virtual-to-physical page mapping is the reference and the physical page is the data.

When a process forks, for example, its address space is duplicated to create a new address space for the child process. All the pages of the parent are duplicated copy-on-write, so that there is only one set of physical pages. When any page is written to, the COW break then occurs with an actual copying of the page to a newly-allocated page frame.

B.2 IMPLEMENTATION OF COW IN LINUX

Linux accomplishes COW by adjusting the protections in the PTE of each COW page to unwritable. The virtual memory area that the page falls in has a descriptor in the kernel, and that descriptor (called the VMA) records whether the region’s pages are writable or not. If the VMA says that the pages in the region are writable, but the PTE of a particular page says that the page is not writable, then the kernel knows that this is a COW page. This case is recognized on a page fault following a write, at which time the COW is broken by
allocating a new page and copying data to it. The PTE for the faulting process is updated to point to the newly allocated page.

B.3 Ambiguity after Protection

Consider the region of memory associated with the process stack. After a fork, the child has the same stack as the parent, but all the pages are set up for COW. Note that the VMA says the memory is writable but the PTES say unwritable. If the child then uses mprotect to set part of the stack to no-access, the VMA says the protected region is unwritable and so do the PTES. Note that now there is no information at all that the kernel might use to determine whether the pages in the protected region were COW before they were made to be unwritable.

Without any way to tell whether the pages are supposed to be COW or not, the kernel makes the safest choice, treating them as if they were. In the current example, that is the right choice. If the child makes its whole stack writable again, and the kernel simply modifies the VMA and PTES so that the pages are writable, then the child can write to the stack and modify the stack of the parent! That would be a disaster.

On the other hand, making pages COW when they don’t have to be is not a disaster at all. It just involves the extra work of copying data and allocating pages.

The solution we chose is to enable the kernel to distinguish pages that used to be COW from pages that used to be fully writable and owned only by the current process itself, even when the pages are made unwritable.

B.4 Marking COW Pages

A new system call was added to the Linux kernel that allows a program in user space to request that all COW pages in its address space be marked as such. When the system call occurs, the kernel sets a special (otherwise unused) bit in the PTE of each page that is set
for COW. It then records in each of the process’ VMAs that the COW marking has been performed.

With this extra bit in the PTEs and the VMAs, the kernel now has the state it needs for distinguishing unwritable pages that used to be COW from unwritable pages that used to be writable. The page fault handler was updated to make use of this information and avoid unnecessary page copying when the VMA shows that COW pages have been marked and the PTE for the faulting page has not been marked.

This extension allows protection settings to be changed freely between restrictive (no-access) and loose (read-write access) without incurring the heavy penalty of repeated page allocation and copying. Without using the COW marking feature, it takes twice as long to protect one page of data as read-only, make it read-write, and then write to the page. With more pages, the savings is even greater.

B.5 COW MARKING: DISCUSSION

Although the COW marking technique was successful, it turned out that the mechanics of adjusting page protections in the kernel, with possible VMA splitting and with page table traversal and modification, is more expensive than the penalty associated with changing between two MACCs. The latter penalty is mostly due to an increase in TLB misses. Because MACC switching is cheap, COW marking was not ported to the UltraSPARC.

Still, creating COW marking required a thorough understanding of copy on write and the shortcuts that the Linux kernel takes. The technique may be useful in another circumstance.

The rest of the port to UltraSPARC showed that some architectural differences did not matter much to the portability of the MACC support extensions to Linux. Among these less significant differences are the number of levels in the page tables and the method for TLB miss handling. What mattered most was the relationship between kernel space and
user space. On the UltraSPARC kernel and user space are worlds apart, with different word sizes and completely independent virtual address mappings.