Efficient Software-Based Fault Isolation

Robert Wahbe Steven Lucco Thomas E. Anderson Susan L. Graham

Computer Science Division
University of California
Berkeley, CA 94720

Abstract

One way to provide fault isolation among cooperating software modules is to place each in its own address space. However, for tightly-coupled modules, this solution incurs prohibitive context switch overhead. In this paper, we present a software approach to implementing fault isolation within a single address space. Our approach has two parts. First, we load the code and data for a distrusted module into its own *fault domain*, a logically separate portion of the application's address space. Second, we modify the object code of a distrusted module to prevent it from writing or jumping to an address outside its fault domain. Both these software operations are portable and programming language independent.

Our approach poses a tradeoff relative to hardware fault isolation: substantially faster communication between fault domains, at a cost of slightly increased execution time for distrusted modules. We demonstrate that for frequently communicating modules, implementing fault isolation in software rather than hardware can substantially improve end-to-end application performance.

This work was supported in part by the National Science Foundation (CDA-8722788), Defense Advanced Research Projects Agency (DARPA) under grant MDA972-92-J-1028 and contracts DABT63-92-C-0026 and N00600-93-C-2481, the Digital Equipment Corporation (the Systems Research Center and the External Research Program), and the AT&T Foundation. Anderson was also supported by a National Science Foundation Young Investigator Award. The content of the paper does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

To appear in the Proceedings of the Symposium on Operating System Principles, 1993.

1 Introduction

Application programs often achieve extensibility by incorporating independently developed software modules. However, faults in extension code can render a software system unreliable, or even dangerous, since such faults could corrupt permanent data. To increase the reliability of these applications, an operating system can provide services that prevent faults in distrusted modules from corrupting application data. Such *fault isolation* services also facilitate software development by helping to identify sources of system failure.

For example, the POSTGRES database manager includes an extensible type system [Sto87]. Using this facility, POSTGRES queries can refer to general-purpose code that defines constructors, destructors, and predicates for user-defined data types such as geometric objects. Without fault isolation, any query that uses extension code could interfere with an unrelated query or corrupt the database.

Similarly, recent operating system research has focused on making it easier for third party vendors to enhance parts of the operating system. An example is micro-kernel design; parts of the operating system are implemented as user-level servers that can be easily modified or replaced. More generally, several systems have added extension code into the operating system, for example, the BSD network packet filter [MRA87, MJ93], application-specific virtual memory management [HC92], and Active Messages [vCGS92]. Among industry systems, Microsoft's Object Linking and Embedding system [Cla92] can link together independently developed software modules. Also, the Quark Xpress desktop publishing system [Dys92] is structured to support incorporation of

Email: {rwahbe, lucco, tea, graham}@cs.berkeley.edu

general-purpose third party code. As with POSTGRES, faults in extension modules can render any of these systems unreliable.

One way to provide fault isolation among cooperating software modules is to place each in its own address space. Using Remote Procedure Call (RPC) [BN84], modules in separate address spaces can call into each other through a normal procedure call interface. Hardware page tables prevent the code in one address space from corrupting the contents of another.

Unfortunately, there is a high performance cost to providing fault isolation through separate address spaces. Transferring control across protection boundaries is expensive, and does not necessarily scale with improvements in a processor's integer performance [ALBL91]. A cross-address-space RPC requires at least: a trap into the operating system kernel, copying each argument from the caller to the callee, saving and restoring registers, switching hardware address spaces (on many machines, flushing the translation lookaside buffer), and a trap back to user level. These operations must be repeated upon RPC return. The execution time overhead of an RPC, even with a highly optimized implementation, will often be two to three orders of magnitude greater than the execution time overhead of a normal procedure call [BALL90, ALBL91].

The goal of our work is to make fault isolation cheap enough that system developers can ignore its performance effect in choosing which modules to place in separate fault domains. In many cases where fault isolation would be useful, cross-domain procedure calls are frequent yet involve only a moderate amount of computation per call. In this situation it is impractical to isolate each logically separate module within its own address space, because of the cost of crossing hardware protection boundaries.

We propose a software approach to implementing fault isolation within a single address space. Our approach has two parts. First, we load the code and data for a distrusted module into its own *fault domain*, a logically separate portion of the application's address space. A fault domain, in addition to comprising a contiguous region of memory within an address space, has a unique identifier which is used to control its access to process resources such as file descriptors. Second, we modify the object code of a distrusted module to prevent it from writing or jumping to an address outside its fault domain. Program modules isolated in separate software-enforced fault domains can not modify each other's data or execute each other's code except through an explicit cross-fault-domain RPC interface.

We have identified several programming-language-independent transformation strategies that can render object code unable to escape its own code and data segments. In this paper, we concentrate on a simple transformation technique, called *sandboxing*, that only slightly increases the execution time of the modified object code. We also investigate techniques that provide more debugging information but which incur greater execution time overhead.

Our approach poses a tradeoff relative to hardware-based fault isolation. Because we eliminate the need to cross hardware boundaries, we can offer substantially lower-cost RPC between fault domains. A safe RPC in our prototype implementation takes roughly $1.1\mu s$ on a DECstation 5000/240 and roughly $0.8\mu s$ on a DEC Alpha 400, more than an order of magnitude faster than any existing RPC system. This reduction in RPC time comes at a cost of slightly increased distrusted module execution time. On a test suite including the the C SPEC92 benchmarks, sandboxing incurs an average of 4% execution time overhead on both the DECstation and the Alpha.

Software-enforced fault isolation may seem to be counter-intuitive: we are slowing down the common case (normal execution) to speed up the uncommon case (cross-domain communication). But for frequently communicating fault domains, our approach can offer substantially better end-to-end performance. To demonstrate this, we applied software-enforced fault isolation to the POSTGRES database system running the Sequoia 2000 benchmark. The benchmark makes use of the POSTGRES extensible data type system to define geometric operators. For this benchmark, the software approach reduced fault isolation overhead by more than a factor of three on a DECstation 5000/240.

A software approach also provides a tradeoff between performance and level of distrust. If some modules in a program are trusted while others are distrusted (as may be the case with extension code), only the distrusted modules incur any execution time overhead. Code in trusted domains can run at full speed. Similarly, it is possible to use our techniques to implement full security, preventing distrusted code from even *reading* data outside of its domain, at a cost of higher execution time overhead. We quantify this effect in Section 5.

The remainder of the paper is organized as follows. Section 2 provides some examples of systems that require frequent communication between fault domains. Section 3 outlines how we modify object code to prevent it from generating illegal addresses. Section 4 describes how we implement low latency cross-fault-domain RPC. Section 5 presents performance results for our prototype, and finally Section 6 discusses some related work.

2 Background

In this section, we characterize in more detail the type of application that can benefit from software-enforced fault isolation. We defer further description of the POSTGRES extensible type system until Section 5, which gives performance measurements for this application.

The operating systems community has focused considerable attention on supporting kernel extensibility. For example, the UNIX vnode interface is designed to make it easy to add a new file system into UNIX [Kle86]. Unfortunately, it is too expensive to forward every file system operation to user level, so typically new file system implementations are added directly into the kernel. (The Andrew file system is largely implemented at user level, but it maintains a kernel cache for performance [HKM+88].) Epoch's tertiary storage file system [Web93] is one example of operating system kernel code developed by a third party vendor.

Another example is user-programmable high performance I/O systems. If data is arriving on an I/O channel at a high enough rate, performance will be degraded substantially if control has to be transferred to user level to manipulate the incoming data [FP93]. Similarly, Active Messages provide high performance message handling in distributed-memory multiprocessors [vCGS92]. Typically, the message handlers are application-specific, but unless the network controller can be accessed from user level [Thi92], the message handlers must be compiled into the kernel for reasonable performance.

A user-level example is the Quark Xpress desktop publishing system. One can purchase third party software that will extend this system to perform functions unforeseen by its original designers [Dys92]. At the same time, this extensibility has caused Quark a number of problems. Because of the lack of efficient fault domains on the personal computers where Quark Xpress runs, extension modules can corrupt Quark's internal data structures. Hence, bugs in third party code can make the Quark system appear unreliable, because end-users do not distinguish among sources of system failure.

All these examples share two characteristics. First, using hardware fault isolation would result in a significant portion of the overall execution time being spent in operating system context switch code. Second, only a small amount of code is distrusted; most of the execution time is spent in trusted code. In this situation, software fault isolation is likely to be more efficient than hardware fault isolation because it sharply reduces the time spent crossing fault domain boundaries, while only slightly increasing the time spent executing the distrusted part of the application. Section 5 quantifies this trade-off between domain-crossing overhead and application execution time overhead, and demonstrates that even if domain-crossing overhead represents a modest proportion of the total application execution time, software-enforced fault isolation is cost effective.

3 Software-Enforced Fault Isolation

In this section, we outline several *software encapsulation* techniques for transforming a distrusted module so that it can not escape its fault domain. We first describe a technique that allows users to pinpoint the location of faults within a software module. Next, we introduce a technique, called *sandboxing*, that can isolate a distrusted module while only slightly increasing its execution time. Section 5 provides a performance analysis of this techinique. Finally, we present a software encapsulation technique that allows cooperating fault domains to share memory. The remainder of this discussion assumes we are operating on a RISC load/store architecture, although our techniques could be extended to handle CISCs. Section 4 describes how we implement safe and efficient cross-fault-domain RPC.

We divide an application's virtual address space into segments, aligned so that all virtual addresses within a segment share a unique pattern of upper bits, called the *segment identifier*. A fault domain consists of two segments, one for a distrusted module's code, the other for its static data, heap and stack. The specific segment addresses are determined at load time.

Software encapsulation transforms a distrusted module's object code so that it can jump only to targets in its code segment, and write only to addresses within its data segment. Hence, all legal jump targets in the distrusted module have the same upper bit pattern (segment identifier); similarly, all legal data addresses generated by the distrusted module share the same segment identifier. Separate code and data segments are necessary to prevent a module from modifying its code segment[1]. It is possible for an address with the correct segment identifier to be illegal, for instance if it refers to an unmapped page. This is caught by the normal operating system page fault mechanism.

3.1 Segment Matching

An *unsafe instruction* is any instruction that jumps to or stores to an address that can not be statically ver-

[1] Our system supports dynamic linking through a special interface.

ified to be within the correct segment. Most control transfer instructions, such as program-counter-relative branches, can be statically verified. Stores to static variables often use an immediate addressing mode and can be statically verified. However, jumps through registers, most commonly used to implement procedure returns, and stores that use a register to hold their target address, can not be statically verified.

A straightforward approach to preventing the use of illegal addresses is to insert checking code before every unsafe instruction. The checking code determines whether the unsafe instruction's target address has the correct segment identifier. If the check fails, the inserted code will trap to a system error routine outside the distrusted module's fault domain. We call this software encapsulation technique *segment matching*.

On typical RISC architectures, segment matching requires four instructions. Figure 1 lists a pseudo-code fragment for segment matching. The first instruction in this fragment moves the store target address into a *dedicated register*. Dedicated registers are used only by inserted code and are never modified by code in the distrusted module. They are necessary because code elsewhere in the distrusted module may arrange to jump directly to the unsafe store instruction, bypassing the inserted check. Hence, we transform all unsafe store and jump instructions to use a dedicated register.

All the software encapsulation techniques presented in this paper require dedicated registers[2]. Segment matching requires four dedicated registers: one to hold addresses in the code segment, one to hold addresses in the data segment, one to hold the segment shift amount, and one to hold the segment identifier.

Using dedicated registers may have an impact on the execution time of the distrusted module. However, since most modern RISC architectures, including the MIPS and Alpha, have at least 32 registers, we can retarget the compiler to use a smaller register set with minimal performance impact. For example, Section 5 shows that, on the DECstation 5000/240, reducing by five registers the register set available to a C compiler (gcc) did not have a significant effect on the average execution time of the SPEC92 benchmarks.

3.2 Address Sandboxing

The segment matching technique has the advantage that it can pinpoint the offending instruction. This capability is useful during software development. We can reduce runtime overhead still further, at the cost of providing no information about the source of faults.

[2]For architectures with limited register sets, such as the 80386 [Int86], it is possible to encapsulate a module using no reserved registers by restricting control flow within a fault domain.

```
dedicated-reg ⇐ target address
    Move target address into dedicated register.
scratch-reg ⇐ (dedicated-reg>>shift-reg)
    Right-shift address to get segment identifier.
    scratch-reg is not a dedicated register.
    shift-reg is a dedicated register.
compare scratch-reg and segment-reg
    segment-reg is a dedicated register.
trap if not equal
    Trap if store address is outside of segment.
store instruction uses dedicated-reg
```

Figure 1: Assembly pseudo code for segment matching.

```
dedicated-reg ⇐ target-reg&and-mask-reg
    Use dedicated register and-mask-reg
    to clear segment identifier bits.
dedicated-reg ⇐ dedicated-reg|segment-reg
    Use dedicated register segment-reg
    to set segment identifier bits.
store instruction uses dedicated-reg
```

Figure 2: Assembly pseudo code to sandbox address in `target-reg`.

Before each unsafe instruction we simply insert code that *sets* the upper bits of the target address to the correct segment identifier. We call this *sandboxing* the address. Sandboxing does not catch illegal addresses; it merely prevents them from affecting any fault domain other than the one generating the address.

Address sandboxing requires insertion of two arithmetic instructions before each unsafe store or jump instruction. The first inserted instruction clears the segment identifier bits and stores the result in a dedicated register. The second instruction sets the segment identifier to the correct value. Figure 2 lists the pseudo-code to perform this operation. As with segment matching, we modify the unsafe store or jump instruction to use the dedicated register. Since we are using a dedicated register, the distrusted module code can not produce an illegal address even by jumping to the second instruction in the sandboxing sequence; since the upper bits of the dedicated register will already contain the correct segment identifier, this second instruction will have no effect. Section 3.6 presents a simple algorithm that can verify that an object code module has been correctly sandboxed.

Address sandboxing requires five dedicated registers. One register is used to hold the segment mask, two registers are used to hold the code and data segment

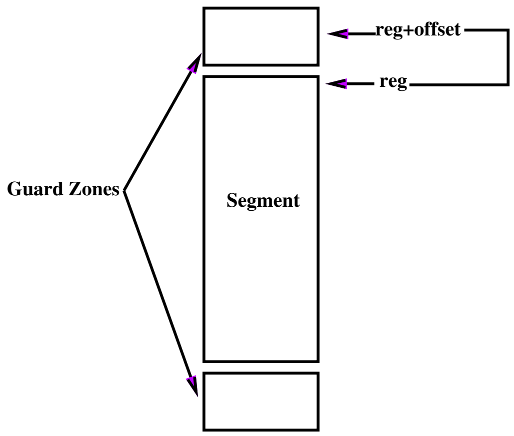

Figure 3: A segment with guard zones. The size of the guard zones covers the range of possible immediate offsets in register-plus-offset addressing modes.

identifiers, and two are used to hold the sandboxed code and data addresses.

3.3 Optimizations

The overhead of software encapsulation can be reduced by using conventional compiler optimizations. Our current prototype applies loop invariant code motion and instruction scheduling optimizations [ASU86, ACD74]. In addition to these conventional techniques, we employ a number of optimizations specialized to software encapsulation.

We can reduce the overhead of software encapsulation mechanisms by avoiding arithmetic that computes target addresses. For example, many RISC architectures include a register-plus-offset instruction mode, where the offset is an immediate constant in some limited range. On the MIPS architecture such offsets are limited to the range -64K to +64K. Consider the store instruction `store value,offset(reg)`, whose address `offset(reg)` uses the register-plus-offset addressing mode. Sandboxing this instruction requires three inserted instructions: one to sum `reg+offset` into the dedicated register, and two sandboxing instructions to set the segment identifier of the dedicated register.

Our prototype optimizes this case by sandboxing only the register `reg`, rather than the actual target address `reg+offset`, thereby saving an instruction. To support this optimization, the prototype establishes *guard zones* at the top and bottom of each segment. To create the guard zones, virtual memory pages adjacent to the segment are unmapped (see Figure 3).

We also reduce runtime overhead by treating the MIPS stack pointer as a dedicated register. We avoid sandboxing the uses of the stack pointer by sandboxing this register whenever it is set. Since uses of the stack pointer to form addresses are much more plentiful than changes to it, this optimization significantly improves performance.

Further, we can avoid sandboxing the stack pointer after it is modified by a small constant offset as long as the modified stack pointer is used as part of a load or store address before the next control transfer instruction. If the modified stack pointer has moved into a guard zone, the load or store instruction using it will cause a hardware address fault. On the DEC Alpha processor, we apply these optimizations to both the frame pointer and the stack pointer.

There are a number of further optimizations that could reduce sandboxing overhead. For example, the transformation tool could remove sandboxing sequences from loops, in cases where a store target address changes by only a small constant offset during each loop iteration. Our prototype does not yet implement these optimizations.

3.4 Process Resources

Because multiple fault domains share the same virtual address space, the fault domain implementation must prevent distrusted modules from corrupting resources that are allocated on a per-address-space basis. For example, if a fault domain is allowed to make system calls, it can close or delete files needed by other code executing in the address space, potentially causing the application as a whole to crash.

One solution is to modify the operating system to know about fault domains. On a system call or page fault, the kernel can use the program counter to determine the currently executing fault domain, and restrict resources accordingly.

To keep our prototype portable, we implemented an alternative approach. In addition to placing each distrusted module in a separate fault domain, we require distrusted modules to access system resources only through cross-fault-domain RPC. We reserve a fault domain to hold trusted *arbitration* code that determines whether a particular system call performed by some other fault domain is safe. If a distrusted module's object code performs a direct system call, we transform this call into the appropriate RPC call. In the case of an extensible application, the trusted portion of the application can make system calls directly and shares a fault domain with the arbitration code.

3.5 Data Sharing

Hardware fault isolation mechanisms can support data sharing among virtual address spaces by manipulating page table entries. Fault domains share an address space, and hence a set of page table entries, so they can not use a standard shared memory implementation. Read-only sharing is straightforward; since our software encapsulation techniques do not alter load instructions, fault domains can read any memory mapped in the application's address space [3].

If the object code in a particular distrusted module has been sandboxed, then it can share read-write memory with other fault domains through a technique we call *lazy pointer swizzling*. Lazy pointer swizzling provides a mechanism for fault domains to share arbitrarily many read-write memory regions with no additional runtime overhead. To support this technique, we modify the hardware page tables to map the shared memory region into every address space segment that needs access; the region is mapped at the same offset in each segment. In other words, we alias the shared region into multiple locations in the virtual address space, but each aliased location has exactly the same low order address bits. As with hardware shared memory schemes, each shared region must have a different segment offset.

To avoid incorrect shared pointer comparisons in sandboxed code, the shared memory creation interface must ensure that each shared object is given a unique address. As the distrusted object code accesses shared memory, the sandboxing code automatically translates shared addresses into the corresponding addresses within the fault domain's data segment. This translation works exactly like hardware translation; the low bits of the address remain the same, and the high bits are set to the data segment identifier.

Under operating systems that do not allow virtual address aliasing, we can implement shared regions by introducing a new software encapsulation technique: *shared segment matching*. To implement sharing, we use a dedicated register to hold a bitmap. The bitmap indicates which segments the fault domain can access. For each unsafe instruction checked, shared segment matching requires one more instruction than segment matching.

3.6 Implementation and Verification

We have identified two strategies for implementing software encapsulation. One approach uses a compiler to emit encapsulated object code for a distrusted module; the integrity of this code is then verified when the module is loaded into a fault domain. Alternatively, the system can encapsulate the distrusted module by directly modifying its object code at load time.

[3] We have implemented versions of these techniques that perform general protection by encapsulating load instructions as well as store and jump instructions. We discuss the performance of these variants in Section 5.

Our current prototype uses the first approach. We modified a version of the gcc compiler to perform software encapsulation. Note that while our current implementation is language dependent, our techniques are language independent.

We built a verifier for the MIPS instruction set that works for both sandboxing and segment matching. The main challenge in verification is that, in the presence of indirect jumps, execution may begin on any instruction in the code segment. To address this situation, the verifier uses a property of our software encapsulation techniques: all unsafe stores and jumps use a dedicated register to form their target address. The verifier divides the program into sequences of instructions called *unsafe regions*. An *unsafe store region* begins with any modification to a dedicated store register. An *unsafe jump region* begins with any modification to a dedicated jump register. If the first instruction in a unsafe store or jump region is executed, all subsequent instructions are guaranteed to be executed. An unsafe store region ends when one of the following hold: the next instruction is a store which uses a dedicated register to form its target address, the next instruction is a control transfer instruction, the next instruction is not guaranteed to be executed, or there are no more instructions in the code segment. A similar definition is used for unsafe jump regions.

The verifier analyzes each unsafe store or jump region to insure that any dedicated register modified in the region is valid upon exit of the region. For example, a load to a dedicated register begins an unsafe region. If the region appropriately sandboxes the dedicated register, the unsafe region is deemed safe. If an unsafe region can not be verified, the code is rejected.

By incorporating software encapsulation into an existing compiler, we are able to take advantage of compiler infrastructure for code optimization. However, this approach has two disadvantages. First, most modified compilers will support only one programming language (gcc supports C, C++, and Pascal). Second, the compiler and verifier must be synchronized with respect to the particular encapsulation technique being employed.

An alternative, called *binary patching*, alleviates these problems. When the fault domain is loaded, the system can encapsulate the module by directly modifying the object code. Unfortunately, practical and robust binary patching, resulting in efficient code, is not currently possible [LB92]. Tools which translate one binary format to another have been built, but these tools rely on compiler-specific idioms to distinguish code from data and use processor emulation to handle unknown indirect jumps[SCK+93]. For software encapsulation, the main challenge is to transform the code so that it uses a subset of the registers, leav-

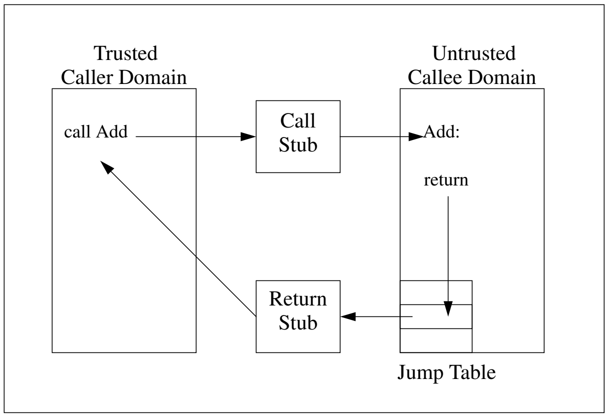

Figure 4: Major components of a cross-fault-domain RPC.

ing registers available for dedicated use. To solve this problem, we are working on a binary patching prototype that uses simple extensions to current object file formats. The extensions store control flow and register usage information that is sufficient to support software encapsulation.

4 Low Latency Cross Fault Domain Communication

The purpose of this work is to reduce the cost of fault isolation for cooperating but distrustful software modules. In the last section, we presented one half of our solution: efficient software encapsulation. In this section, we describe the other half: fast communication across fault domains.

Figure 4 illustrates the major components of a cross-fault-domain RPC between a trusted and distrusted fault domain. This section concentrates on three aspects of fault domain crossing. First, we describe a simple mechanism which allows a fault domain to safely call a trusted stub routine outside its domain; that stub routine then safely calls into the destination domain. Second, we discuss how arguments are efficiently passed among fault domains. Third, we detail how registers and other machine state are managed on cross-fault-domain RPCs to insure fault isolation. The protocol for exporting and naming procedures among fault domains is independent of our techniques.

The only way for control to escape a fault domain is via a *jump table*. Each jump table entry is a control transfer instruction whose target address is a legal entry point outside the domain. By using instructions whose target address is an immediate encoded in the instruction, the jump table does not rely on the use of a dedicated register. Because the table is kept in the (read-only) code segment, it can only be modified by a trusted module.

For each pair of fault domains a customized call and return stub is created for each exported procedure. Currently, the stubs are generated by hand rather than using a stub generator [JRT85]. The stubs run unprotected outside of both the caller and callee domain. The stubs are responsible for copying cross-domain arguments between domains and managing machine state.

Because the stubs are trusted, we are able to copy call arguments directly to the target domain. Traditional RPC implementations across address spaces typically perform three copies to transfer data. The arguments are marshalled into a message, the kernel copies the message to the target address space, and finally the callee must de-marshall the arguments. By having the caller and callee communicate via a shared buffer, LRPC also uses only a single copy to pass data between domains [BALL91].

The stubs are also responsible for managing machine state. On each cross-domain call any registers that are both used in the future by the caller and potentially modified by the callee must be protected. Only registers that are designated by architectural convention to be preserved across procedure calls are saved. As an optimization, if the callee domain contains no instructions that modify a preserved register we can avoid saving it. Karger uses a trusted linker to perform this kind of optimization between address spaces [Kar89]. In addition to saving and restoring registers, the stubs must switch the execution stack, establish the correct register context for the software encapsulation technique being used, and validate all dedicated registers.

Our system must also be robust in the presence of fatal errors, for example, an addressing violation, while executing in a fault domain. Our current implementation uses the UNIX signal facility to catch these errors; it then terminates the outstanding call and notifies the caller's fault domain. If the application uses the same operating system thread for all fault domains, there must be a way to terminate a call that is taking too long, for example, because of an infinite loop. Trusted modules may use a timer facility to interrupt execution periodically and determine if a call needs to be terminated.

5 Performance Results

To evaluate the performance of software-enforced fault domains, we implemented and measured a prototype of our system on a 40MHz DECstation 5000/240 (DEC-MIPS) and a 133Mhz Alpha 400 (DEC-ALPHA).

We consider three questions. First, how much overhead does software encapsulation incur? Second, how fast is a cross-fault-domain RPC? Third, what is the performance impact of using software enforced fault isolation on an end-user application? We discuss each of these questions in turn.

5.1 Encapsulation Overhead

We measured the execution time overhead of sandboxing a wide range of C programs, including the C SPEC92 benchmarks and several of the Splash benchmarks [Ass91, SWG91]. We treated each benchmark as if it were a distrusted module, sandboxing all of its code. Column 1 of Table 1 reports overhead on the DEC-MIPS, column 6 reports overhead on the DEC-ALPHA. Columns 2 and 7 report the overhead of using our technique to provide general protection by sandboxing load instructions as well as store and jump instructions[4]. As detailed in Section 3, sandboxing requires 5 dedicated registers. Column 3 reports the overhead of removing these registers from possible use by the compiler. All overheads are computed as the additional execution time divided by the original program's execution time.

On the DEC-MIPS, we used the program measurement tools `pixie` and `qpt` to calculate the number of additional instructions executed due to sandboxing [Dig, BL92]. Column 4 of Table 1 reports this data as a percentage of original program instruction counts.

The data in Table 1 appears to contain a number of anomalies. For some of the benchmark programs, for example, `056.ear` on the DEC-MIPS and `026.compress` on the DEC-ALPHA, sandboxing *reduced* execution time. In a number of cases the overhead is surprisingly low.

To identify the source of these variations we developed an analytical model for execution overhead. The model predicts overhead based on the number of additional instructions executed due to sandboxing (*s-instructions*), and the number of saved floating point interlock cycles (*interlocks*). Sandboxing increases the available instruction-level parallelism, allowing the number of floating-point interlocks to be substantially reduced. The integer pipeline does not provide interlocking; instead, delay slots are explicitly filled with nop instructions by the compiler or assembler. Hence, scheduling effects among integer instructions will be accurately reflected by the count of instructions added (*s-instructions*). The expected overhead is computed as:

$$\frac{(\textit{s-instructions} - \textit{interlocks}) / \textit{cycles-per-second}}{\textit{original-execution-time-seconds}}$$

[4]Loads in the libraries, such as the standard C library, were not sandboxed.

The model provides an effective way to separate known sources of overhead from second order effects. Column 5 of Table 1 are the predicted overheads.

As can be seen from Table 1, the model is, on average, effective at predicting sandboxing overhead. The differences between measured and expected overheads are normally distributed with mean 0.7% and standard deviation of 2.6%. The difference between the means of the measured and expected overheads is not statistically significant. This experiment demonstrates that, by combining instruction count overhead and floating point interlock measurements, we can accurately predict average execution time overhead. If we assume that the model is also accurate at predicting the overhead of individual benchmarks, we can conclude that there is a second order effect creating the observed anomalies in measured overhead.

We can discount effective instruction cache size and virtual memory paging as sources for the observed execution time variance. Because sandboxing adds instructions, the effective size of the instruction cache is reduced. While this might account for measured overheads higher than predicted, it does not account for the opposite effect. Because all of our benchmarks are compute bound, it is unlikely that the variations are due to virtual memory paging.

The DEC-MIPS has a physically indexed, physically tagged, direct mapped data cache. In our experiments sandboxing did not affect the size, contents, or starting virtual address of the data segment. For both original and sandboxed versions of the benchmark programs, successive runs showed insignificant variation. Though difficult to quantify, we do not believe that data cache alignment was an important source of variation in our experiments.

We conjecture that the observed variations are caused by *instruction cache mapping conflicts*. Software encapsulation changes the mapping of instructions to cache lines, hence changing the number of instruction cache conflicts. A number of researchers have investigated minimizing instruction cache conflicts to reduce execution time [McF89, PH90, Sam88]. One researcher reported a 20% performance gain by simply changing the order in which the object files were linked [PH90]. Samples and Hilfinger report significantly improved instruction cache miss rates by rearranging only 3% to 8% of an application's basic blocks [Sam88].

Beyond this effect, there were statistically significant differences among programs. On average, programs which contained a significant percentage of floating point operations incurred less overhead. On the DEC-MIPS the mean overhead for floating point intensive benchmarks is 2.5%, compared to a mean of 5.6% for the remaining benchmarks. All of our benchmarks are

Benchmark		DEC-MIPS Fault Isolation Overhead	DEC-MIPS Protection Overhead	DEC-MIPS Reserved Register Overhead	DEC-MIPS Instruction Count Overhead	DEC-MIPS Fault Isolation Overhead (predicted)	DEC-ALPHA Fault Isolation Overhead	DEC-ALPHA Protection Overhead
`052.alvinn`	FP	1.4%	33.4%	-0.3%	19.4%	0.2%	8.1%	35.5%
`bps`	FP	5.6%	15.5%	-0.1%	8.9%	5.7%	4.7%	20.3%
`cholesky`	FP	0.0%	22.7%	0.5%	6.5%	-1.5%	0.0%	9.3%
`026.compress`	INT	3.3%	13.3%	0.0%	10.9%	4.4%	-4.3%	0.0%
`056.ear`	FP	-1.2%	19.1%	0.2%	12.4%	2.2%	3.7%	18.3%
`023.eqntott`	INT	2.9%	34.4%	1.0%	2.7%	2.2%	2.3%	17.4%
`008.espresso`	INT	12.4%	27.0%	-1.6%	11.8%	10.5%	13.3%	33.6%
`001.gcc1.35`	INT	3.1%	18.7%	-9.4%	17.0%	8.9%	NA	NA
`022.li`	INT	5.1%	23.4%	0.3%	14.9%	11.4%	5.4%	16.2%
`locus`	INT	8.7%	30.4%	4.3%	10.3%	8.6%	4.3%	8.7%
`mp3d`	FP	10.7%	10.7%	0.0%	13.3%	8.7%	0.0%	6.7%
`psgrind`	INT	10.4%	19.5%	1.3%	12.1%	9.9%	8.0%	36.0%
`qcd`	FP	0.5%	27.0%	2.0%	8.8%	1.2%	-0.8%	12.1%
`072.sc`	INT	5.6%	11.2%	7.0%	8.0%	3.8%	NA	NA
`tracker`	INT	-0.8%	10.5%	0.4%	3.9%	2.1%	10.9%	19.9%
`water`	FP	0.7%	7.4%	0.3%	6.7%	1.5%	4.3%	12.3%
Average		**4.3%**	**21.8%**	**0.4%**	**10.5%**	**5.0%**	**4.3%**	**17.6%**

Table 1: Sandboxing overheads for DEC-MIPS and DEC-ALPHA platforms. The benchmarks `001.gcc1.35` and `072.sc` are dependent on a pointer size of 32 bits and do not compile on the DEC-ALPHA. The predicted fault isolation overhead for `cholesky` is negative due to conservative interlocking on the MIPS floating-point unit.

compute intensive. Programs that perform significant amounts of I/O will incur less overhead.

5.2 Fault Domain Crossing

We now turn to the cost of cross-fault-domain RPC. Our RPC mechanism spends most of its time saving and restoring registers. As detailed in Section 4, only registers that are designated by the architecture to be *preserved* across procedure calls need to be saved. In addition, if no instructions in the callee fault domain modify a preserved register then it does not need to be saved. Table 2 reports the times for three versions of a NULL cross-fault-domain RPC. Column 1 lists the crossing times when all data registers are caller saved. Column 2 lists the crossing times when the preserved integer registers are saved. Finally, the times listed in Column 3 include saving all preserved floating point registers. In many cases crossing times could be further reduced by statically partitioning the registers between domains.

For comparison, we measured two other calling mechanisms. First, we measured the time to perform a C procedure call that takes no arguments and returns no value. Second, we sent a single byte between two address spaces using the pipe abstraction provided by the native operating system and measured the round-trip time. These times are reported in the last two columns of Table 2. On these platforms, the cost of cross-address-space calls is roughly three orders of magnitude more expensive than local procedure calls.

Operating systems with highly optimized RPC implementations have reduced the cost of cross-address-space RPC to within roughly two orders of magnitude of local procedure calls. On Mach 3.0, cross-address-space RPC on a 25Mhz DECstation 5000/200 is 314 times more expensive than a local procedure call [Ber93]. The Spring operating system, running on a 40Mhz SPARCstation2, delivers cross-address-space RPC that is 73 times more expensive than a local leaf procedure call [HK93]. Software enforced fault isolation is able to reduce the relative cost of cross-fault-domain RPC by an order of magnitude over these systems.

5.3 Using Fault Domains in POSTGRES

To capture the effect of our system on application performance, we added software enforced fault domains to the POSTGRES database management system, and measured POSTGRES running the Sequoia 2000 benchmark [SFGM93]. The Sequoia 2000 benchmark

	Cross Fault-Domain RPC				
Platform	Caller Save Registers	Save Integer Registers	Save Integer+Float Registers	C Procedure Call	Pipes
DEC-MIPS	1.11μs	1.81μs	2.83μs	0.10μs	204.72μs
DEC-ALPHA	0.75μs	1.35μs	1.80μs	0.06μs	227.88μs

Table 2: Cross-fault-domain crossing times.

Sequoia 2000 Query	Untrusted Function Manager Overhead	Software-Enforced Fault Isolation Overhead	Number Cross-Domain Calls	DEC-MIPS-PIPE Overhead (predicted)
Query 6	1.4%	1.7%	60989	18.6%
Query 7	5.0%	1.8%	121986	38.6%
Query 8	9.0%	2.7%	121978	31.2%
Query 10	9.6%	5.7%	1427024	31.9%

Table 3: Fault isolation overhead for POSTGRES running Sequoia 2000 benchmark.

contains queries typical of those used by earth scientists in studying the climate. To support these kinds of non-traditional queries, POSTGRES provides a user-extensible type system. Currently, user-defined types are written in conventional programming languages, such as C, and dynamically loaded into the database manager. This has long been recognized to be a serious safety problem[Sto88].

Four of the eleven queries in the Sequoia 2000 benchmark make use of user-defined polygon data types. We measured these four queries using both unprotected dynamic linking and software-enforced fault isolation. Since the POSTGRES code is trusted, we only sandboxed the dynamically loaded user code. For this experiment, our cross-fault-domain RPC mechanism saved the preserved integer registers (the variant corresponding to Column 2 in Table 2). In addition, we instrumented the code to count the number of cross-fault-domain RPCs so that we could estimate the performance of fault isolation based on separate address spaces.

Table 3 presents the results. Untrusted user-defined functions in POSTGRES use a separate calling mechanism from built-in functions. Column 1 lists the overhead of the untrusted function manager without software enforced fault domains. All reported overheads in Table 3 are relative to original POSTGRES using the untrusted function manager. Column 2 reports the measured overhead of software enforced fault domains. Using the number of cross-domain calls listed in Column 3 and the DEC-MIPS-PIPE time reported in Table 2, Column 4 lists the estimated overhead using conventional hardware address spaces.

5.4 Analysis

For the POSTGRES experiment software encapsulation provided substantial savings over using native operating system services and hardware address spaces. In general, the savings provided by our techniques over hardware-based mechanisms is a function of the percentage of time spent in distrusted code (t_d), the percentage of time spent crossing among fault domains (t_c), the overhead of encapsulation (h), and the ratio, r, of our fault domain crossing time to the crossing time of the competing hardware-based RPC mechanism.

$$savings = (1 - r)t_c - ht_d$$

Figure 5 graphically depicts these trade-offs. The X axis gives the percentage of time an application spends crossing among fault domains. The Y axis reports the relative cost of software enforced fault-domain crossing over hardware address spaces. Assuming that the execution time overhead of encapsulated code is 4.3%, the shaded region illustrates when software enforced fault isolation is the better performance alternative.

Software-enforced fault isolation becomes increasingly attractive as applications achieve higher degrees of fault isolation (see Figure 5). For example, if an application spends 30% of its time crossing fault domains, our RPC mechanism need only perform 10% better than its competitor. Applications that currently spend as little as 10% of their time crossing require only a 39% improvement in fault domain crossing time. As reported in Section 5.2, our crossing time for the DEC-MIPS is 1.10μs and for the DEC-ALPHA 0.75μs. Hence,

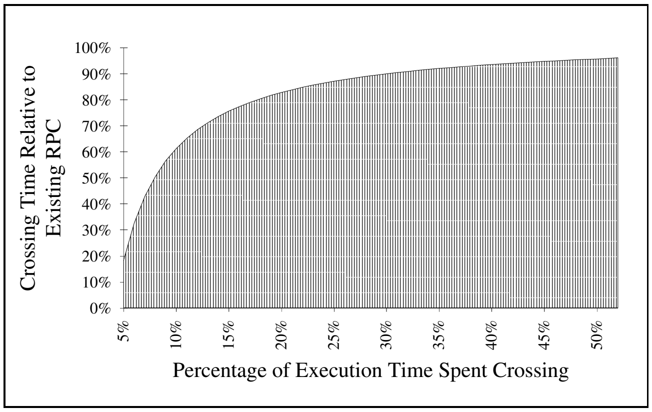

Figure 5: The shaded region represents when software enforced fault isolation provides the better performance alternative. The X axis represents percentage of time spent crossing among fault domains (t_c). The Y axis represents the relative RPC crossing speed (r). The curve represents the break even point: $(1-r)t_c = ht_d$. In this graph, $h = 0.043$ (encapsulation overhead on the DEC-MIPS and DEC-ALPHA).

for this latter example, a hardware address space crossing time of 1.80μs on the DEC-MIPS and 1.23μs on the DEC-ALPHA would provide better performance than software fault domains. As far as we know, no production or experimental system currently provides this level of performance.

Further, Figure 5 assumes that the entire application was encapsulated. For many applications, such as POSTGRES, this assumption is conservative. Figure 6 transforms the previous figure, assuming that 50% of total execution is spent in distrusted extension code.

Figures 5 and 6 illustrate that software enforced fault isolation is the best choice whenever crossing overhead is a significant proportion of an application's execution time. Figure 7 demonstrates that overhead due to software enforced fault isolation remains small regardless of application behavior. Figure 7 plots overhead as a function of crossing behavior and crossing cost. Crossing times typical of vendor-supplied and highly optimized hardware-based RPC mechanisms are shown. The graph illustrates the relative performance stability of the software solution. This stability allows system developers to ignore the performance effect of fault isolation in choosing which modules to place in separate fault domains.

6 Related Work

Many systems have considered ways of optimizing RPC performance [vvST88, TA88, Bla90, SB90, HK93, BALL90, BALL91]. Traditional RPC systems based

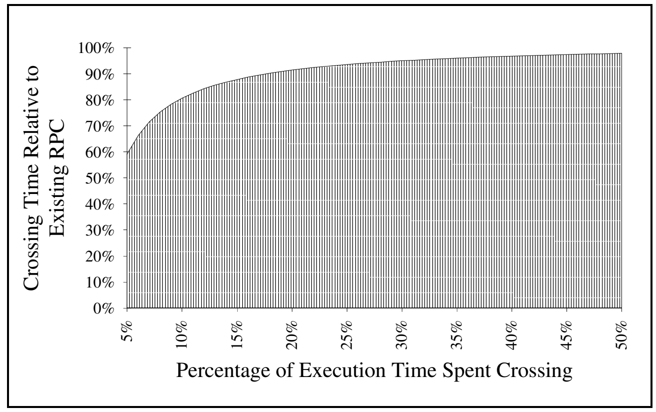

Figure 6: The shaded region represents when software enforced fault isolation provides the better performance alternative. The X axis represents percentage of time spent crossing among fault domains (t_c). The Y axis represents the relative RPC crossing speed (r). The curve represents the break even point: $(1-r)t_c = ht_d$. In this graph, $h = 0.043$ (encapsulation overhead on the DEC-MIPS and DEC-ALPHA).

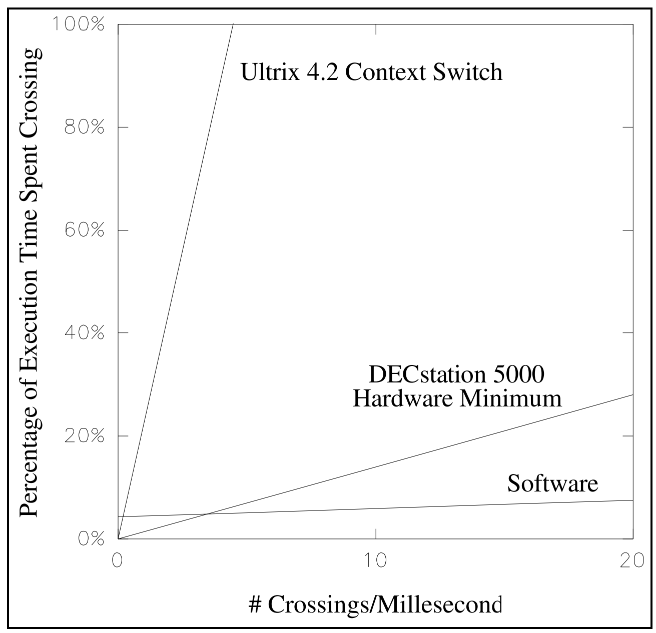

Figure 7: Percentage of time spent in crossing code versus number of fault domain crossings per millisecond on the DEC-MIPS. The hardware minimum crossing number is taken from a cross-architectural study of context switch times [ALBL91]. The Ultrix 4.2 context switch time is as reported in the last column of Table 2.

on hardware fault isolation are ultimately limited by the minimal hardware cost of taking two kernel traps and two hardware context switches. LRPC was one of the first RPC systems to approach this limit, and our prototype uses a number of the techniques found in LRPC and later systems: the same thread runs in both the caller and the callee domain, the stubs are kept as simple as possible, and the crossing code jumps directly to the called procedure, avoiding a dispatch in the callee domain. Unlike these systems, software-based fault isolation avoids hardware context switches, substantially reducing crossing costs.

Address space identifier tags can be used to reduce hardware context switch times. Tags allow more than one address space to share the TLB; otherwise the TLB must be flushed on each context switch. It was estimated that 25% of the cost of an LRPC on the Firefly (which does not have tags) was due to TLB misses[BALL90]. Address space tags do not, however, reduce the cost of register management or system calls, operations which are not scaling with integer performance[ALBL91]. An important advantage of software-based fault isolation is that it does not rely on specialized architectural features such as address space tags.

Restrictive programming languages can also be used to provide fault isolation. Pilot requires all kernel, user, and library code to be written in Mesa, a strongly typed language; all code then shares a single address space [RDH+80]. The main disadvantage of relying on strong typing is that it severely restricts the choice of programming languages, ruling out conventional languages like C, C++, and assembly. Even with strongly-typed languages such as Ada and Modula-3, programmers often find they need to use loopholes in the type system, undercutting fault isolation. In contrast, our techniques are language independent.

Deutsch and Grant built a system that allowed user-defined measurement modules to be dynamically loaded into the operating system and executed directly on the processor [DG71]. The module format was a stylized native object code designed to make it easier to statically verify that the code did not violate protection boundaries.

An interpreter can also provide failure isolation. For example, the BSD UNIX network packet filter utility defines a language which is interpreted by the operating system network driver. The interpreter insulates the operating system from possible faults in the customization code. Our approach allows code written in any programming language to be safely encapsulated (or rejected if it is not safe), and then executed at near full speed by the operating system.

Anonymous RPC exploits 64-bit address spaces to provide low latency RPC and *probabilistic* fault isolation [YBA93]. Logically independent domains are placed at random locations in the same hardware address space. Calls between domains are anonymous, that is, they do not reveal the location of the caller or the callee to either side. This provides probabilistic protection – it is unlikely that any domain will be able to discover the location of any other domain by malicious or accidental memory probes. To preserve anonymity, a cross domain call must trap to protected code in the kernel; however, no hardware context switch is needed.

7 Summary

We have described a software-based mechanism for portable, programming language independent fault isolation among cooperating software modules. By providing fault isolation within a single address space, this approach delivers cross-fault-domain communication that is more than an order of magnitude faster than any RPC mechanism to date.

To prevent distrusted modules from escaping their own fault domain, we use a software encapsulation technique, called *sandboxing*, that incurs about 4% execution time overhead. Despite this overhead in executing distrusted code, software-based fault isolation will often yield the best overall application performance. Extensive kernel optimizations can reduce the overhead of hardware-based RPC to within a factor of ten over our software-based alternative. Even in this situation, software-based fault isolation will be the better performance choice whenever the overhead of using hardware-based RPC is greater than 5%.

8 Acknowledgements

We thank Brian Bershad, Mike Burrows, John Hennessy, Peter Kessler, Butler Lampson, Ed Lazowska, Dave Patterson, John Ousterhout, Oliver Sharp, Richard Sites, Alan Smith and Mike Stonebraker for their helpful comments on the paper. Jim Larus provided us with the profiling tool `qpt`. We also thank Mike Olson and Paul Aoki for helping us with POSTGRES.

References

[ACD74] T.L. Adam, K.M. Chandy, and J.R. Dickson. A comparison of list schedules for parallel processing systems. *Communications of the ACM*, 17(12):685–690, December 1974.

[ALBL91] Thomas Anderson, Henry Levy, Brian Bershad, and Edward Lazowska. The Interaction of Architecture and Operating System Design.

In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 108–120, April 1991.

[Ass91] Administrator: National Computer Graphics Association. *SPEC Newsletter*, 3(4), December 1991.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.

[BALL90] Brian Bershad, Thomas Anderson, Edward Lazowska, and Henry Levy. Lightweight Remote Procedure Call. *ACM Transactions on Computer Systems*, 8(1), February 1990.

[BALL91] Brian Bershad, Thomas Anderson, Edward Lazowska, and Henry Levy. User-Level Interprocess Communication for Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(2), May 1991.

[Ber93] Brian Bershad, August 1993. Private Communication.

[BL92] Thomas Ball and James R. Larus. Optimally profiling and tracing. In *Proceedings of the Conference on Principles of Programming Languages*, pages 59–70, 1992.

[Bla90] David Black. Scheduling Support for Concurrency and Parallelism in the Mach Operating System. *IEEE Computer*, 23(5):35–43, May 1990.

[BN84] Andrew Birrell and Bruce Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[Cla92] J.D. Clark. *Window Programmer' Guide To OLE/DDE*. Prentice-Hall, 1992.

[DG71] L. P. Deutsch and C. A. Grant. A flexible measurement tool for software systems. In *IFIP Congress*, 1971.

[Dig] Digital Equipment Corporation. *Ultrix v4.2 Pixie Manual Page*.

[Dys92] Peter Dyson. Xtensions for Xpress: Modular Software for Custom Systems. *Seybold Report on Desktop Publishing*, 6(10):1–21, June 1992.

[FP93] Kevin Fall and Joseph Pasquale. Exploiting in-kernel data paths to improve I/O throughput and CPU a vailability. In *Proceedings of the 1993 Winter USENIX Conference*, pages 327–333, January 1993.

[HC92] Keiran Harty and David Cheriton. Application-controlled physical memory using external page-cache management. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.

[HK93] Graham Hamilton and Panos Kougiouris. The Spring nucleus: A microkernel for objects. In *Proceedings of the Summer USENIX Conference*, pages 147–159, June 1993.

[HKM+88] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–82, February 1988.

[Int86] Intel Corporation, Santa Clara, California. *Intel 80386 Programmer's Reference Manual*, 1986.

[JRT85] Michael B. Jones, Richard F. Rashid, and Mary R. Thompson. Matchmaker: An interface specification language for distributed processing. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 225–235, January 1985.

[Kar89] Paul A. Karger. Using Registers to Optimize Cross-Domain Call Performance. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 194–204, April 3–6 1989.

[Kle86] Steven R. Kleiman. Vnodes: An Architecture for Multiple File System Types in SUN UNIX. In *Proceedings of the 1986 Summer USENIX Conference*, pages 238–247, 1986.

[LB92] James R. Larus and Thomas Ball. Rewriting executable files to measure program behavior. Technical Report 1083, University of Wisconsin-Madison, March 1992.

[McF89] Scott McFarling. Program optimization for instruction caches. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–191, April 1989.

[MJ93] Steven McCanne and Van Jacobsen. The BSD Packet Filter: A New Architecture for User-Level Packet Capture. In *Proceedings of the 1993 Winter USENIX Conference*, January 1993.

[MRA87] J. C. Mogul, R. F. Rashid, and M. J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of the Symposium on Operating System Principles*, pages 39–51, November 1987.

[PH90] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 16–27, White Plains, New York, June 1990. Appeared as SIGPLAN NOTICES 25(6).

[RDH+80] David D. Redell, Yogen K. Dalal, Thomas R. Horsley, Hugh C. Lauer, William C. Lynch, Paul R. McJones, Hal G. Murray, and Stephen C. Purcell. Pilot: An Operating System for a Personal Computer. *Communications of the ACM*, 23(2):81–92, February 1980.

[Sam88] A. Dain Samples. Code reorganization for instruction caches. Technical Report UCB/CSD 88/447, University of California, Berkeley, October 1988.

[SB90] Michael Schroeder and Michael Burrows. Performance of Firefly RPC. *ACM Transactions on Computer Systems*, 8(1):1–17, February 1990.

[SCK+93] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. Binary translation. *Communications of the ACM*, 36(2):69–81, February 1993.

[SFGM93] M. Stonebraker, J. Frew, K. Gardels, and J. Meridith. The Sequoia 2000 Benchmark. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, May 1993.

[Sto87] Michael Stonebraker. Extensibility in POSTGRES. *IEEE Database Engineering*, September 1987.

[Sto88] Michael Stonebraker. Inclusion of new types in relational data base systems. In Michael Stonebraker, editor, *Readings in Database Systems*, pages 480–487. Morgan Kaufmann Publishers, Inc., 1988.

[SWG91] J. P. Singh, W. Weber, and A. Gupta. Splash: Stanford parallel applications for shared-memory. Technical Report CSL-TR-91-469, Stanford, 1991.

[TA88] Shin-Yuan Tzou and David P. Anderson. A Performance Evaluation of the DASH Message-Passing System. Technical Report UCB/CSD 88/452, Computer Science Division, University of California, Berkeley, October 1988.

[Thi92] Thinking Machines Corporation. *CM-5 Network Interface Programmer's Guide*, 1992.

[vCGS92] T. von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, 1992.

[vvST88] Robbert van Renesse, Hans van Staveren, and Andrew S. Tanenbaum. Performance of the World's Fastest Distributed Operating System. *Operating Systems Review*, 22(4):25–34, October 1988.

[Web93] Neil Webber. Operating System Support for Portable Filesystem Extensions. In *Proceedings of the 1993 Winter USENIX Conference*, January 1993.

[YBA93] Curtis Yarvin, Richard Bukowski, and Thomas Anderson. Anonymous RPC: Low Latency Protection in a 64-Bit Address Space. In *Proceedings of the Summer USENIX Conference*, June 1993.