Hardware and Software Support for Efficient Exception Handling

Chandramohan A. Thekkath and Henry M. Levy

Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Technical Report 94-07-05
July 17, 1994

Also appears in the Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems, October 1994.

Hardware and Software Support for Efficient Exception Handling

Chandramohan A. Thekkath[†] and Henry M. Levy
Department of Computer Science and Engineering, FR-35
University of Washington
Seattle, WA 98195

Abstract

Program-synchronous exceptions, for example, breakpoints, watchpoints, illegal opcodes, and memory access violations, provide information about *exceptional* conditions, interrupting the program and vectoring to an operating system handler. Over the last decade, however, programs and run-time systems have increasingly employed these mechanisms as a performance optimization to detect *normal* and *expected* conditions. Unfortunately, current architecture and operating system structures are designed for exceptional or erroneous conditions, where performance is of secondary importance, rather than normal conditions. Consequently, this has limited the practicality of such hardware-based detection mechanisms.

We propose both hardware and software structures that permit efficient handling of synchronous exceptions by user-level code. We demonstrate a software implementation that reduces exception-delivery cost by an order-of-magnitude on current RISC processors, and show the performance benefits of that mechanism for several example applications.

1 Introduction

Modern processors provide program interruption mechanisms for two event types: *asynchronous* events (interrupts), normally caused by external input/output, and *synchronous* events (exceptions), which are caused by internal execution of the current program. For historical and perhaps simplicity reasons, the handling of these two event types has been intimately bound together: both asynchronous and synchronous events cause interruption of the current program, and the hardware dispatches to the operating system kernel to process the event. Within the operating system, a handler is responsible for saving state, deciding what event occurred if necessary, constructing an environment to handle that event, transferring control to event-specific code, either within the operating system or within the user-level program, and finally restoring state and returning from the exception or interrupt.

This uniform event handling would be fine if the synchronous events were all *exceptional*, i.e., either unusual events that could be handled leisurely, or errors that require the operating system to terminate the program. However, applications and run-time systems are using exception mechanisms increasingly and in novel ways; for example, exceptions are being used for garbage collection [Appel et al. 88], conditional watchpoints [Wahbe 92], transaction support [Chang & Mergen 88], persistent storage management [Wilson & Kakkad 92], and distributed virtual memory [Li & Hudak 89]. To compound the problem, several studies

[†]Author's current address: DEC Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301.

This work was supported in part by the National Science Foundation (Grants No. CCR-8907666, CDA-9123308, and CCR-9200832), the Washington Technology Center, Digital Equipment Corporation, Boeing Computer Services, Intel Corporation, Hewlett-Packard Corporation, and Apple Computer. C. Thekkath was supported in part by a fellowship from Intel Corporation.

	Time (μseconds)					
Operation	**R3000 Ultrix**	**R3000 Mach MK/UX**	**R3000 Mach MK**	**R4000 NT**	**SPARC-10 SunOS**	**DECchip 21064 OSF/1 V1.3**
Deliver Simple Exception to Null User Handler	49	1937	48	64	45	150
Deliver Write Prot. Exception To Null Handler	60	1992	80	84	106	160
Return from Null Handler	31	64	208	136	24	25
Simple Exception Round-Trip Delivery and Return	80	2001	256	200	69	175

Table 1: Performance of basic exception functions on modern systems.

have shown that the relative cost of exception handling on modern RISCs has actually increased, when compared to past-generation architectures [Ousterhout 90, Anderson et al. 91].

A crucial difference between interrupts and exceptions is that, in the latter case, the information needed to respond typically lies within the application program or run-time system, *not* within the kernel. The kernel is thus the wrong transfer target for an exception; vectoring to the kernel only adds unnecessary time and complexity to the processing of the exception. On modern systems, the kernel may need user-level knowledge even for page fault handling [Young 89]. As operating system structures evolve to move more functions to user level, the situation will even worsen; for example, on micro-kernel operating systems, an application exception must be sent from the kernel, to the operating system environment server managing the application, to the application itself, and then back through all the levels.

Table 1 shows the magnitude of the problem. We measured the time to deliver a simple exception to a null user-level signal handler on several hardware/software systems: the Ultrix 4.2A and Mach/UX (MK83/UX41) operating systems on the 25 MHz DECstation 5000 (MIPS R3000 processor); the SunOS operating system (SunOS 4.1.3) on the 36 MHz Sun SPARC-10; the Microsoft Windows/NT operating system on a 40 MHz MIPS R4000-based computer; and the DEC OSF/1 V1.3 operating system on the 200 MHz DEC AXP 3000/500X. The time to deliver a write protection exception is shown as well, since write protection is often used to detect dirty pages for garbage collection [Appel et al. 88], pointer swizzling [Wilson & Kakkad 92], and other applications. From the bottom row, we see that the delivery to and return from a null user-level exception handler takes 69 μseconds in the best case of SunOS, to 2 milliseconds for Mach/UX. The Mach/UX is significantly slower, because the exception travels to the Unix server and then to the application. While Windows/NT has a micro-kernel as well, most exceptions are handled directly in the NT kernel. For comparison, the raw performance of the Mach exception handling without the Unix server is shown in the fourth column (256 μs).

Responding to these problems requires two things: (1) realizing that exception handling is important and can have an impact on application performance, and (2) designing an integrated hardware/software architecture to efficiently vector exceptions to “the right place.” This paper studies the issues in designing efficient exception mechanisms and their implications for applications.

This paper is structured as follows. In the following section, we discuss architectural issues in user-level exception vectoring. We begin by presenting, as an example, the exception architecture of the Tera computer [Alverson et al. 90], which includes user-level exception delivery. We then extend that design to meet the needs of our target applications on more conventional RISC architectures. Section 3 describes a software architecture for fast user-level exception delivery, which we have implemented on the MIPS-based DECstation. Our implementation offers an order-of-magnitude performance improvement over current software schemes and is close to what the hardware is capable of delivering. In Section 4, we examine several applications that use exceptions and demonstrate the performance impact of the software architecture discussed in Section 3. Finally, we conclude and summarize our results in Section 5.

For the most part, this paper discusses precise exceptions, in which the hardware state on exception reflects conditions that existed just prior to the faulting instruction. With imprecise exceptions, however, instructions following the exception may have modified state, complicating the handler and the continuation of the process. Typically, imprecise exceptions are confined to faults encountered during arithmetic operations; for example, in the DECchip 21064, all non-arithmetic synchronous exceptions, such as protection violation, unaligned access, and breakpoint, are precise. These are the exceptions used by the applications we are attempting to benefit.

2 Architectural Support for User-Level Exceptions

One possible solution to inefficient exception handling is the direct hardware vectoring of exceptions to user-mode handlers, rather than the kernel. Direct user-level delivery could eliminate much of the current overhead: exceptions would arrive rapidly at the appropriate user-level code, that code could be specialized to “expect” the exception, the amount of state saving could be greatly reduced, and the return through the kernel would be eliminated. By itself, however, user-mode delivery may be insufficient for some exception-handling needs; in the case of memory management faults, for example, privileged memory management state may need to be manipulated by the handler. We address such issues later in this section.

2.1 The Tera Architecture

We briefly describe exception handling in the Tera architecture [Alverson et al. 90] as an introduction and example of direct user-mode exception delivery. The Tera supports multi-threading in hardware and each processor contains complete hardware state for 128 threads. The processor cycles through a queue of runnable threads, executing one instruction from each thread on each cycle. The objective is to maintain high processor utilization through improved pipeline performance (by spreading dependencies) and to reduce the impact of memory latency (because the processor overlaps memory accesses from some threads with computation from others).

Exceptions in the Tera system are dispatched directly to the thread causing the exception, since only that thread will have access to the state needed to process the exception. If the exception handler requires operating system aid, it then calls the operating system within its context. The architectural support for exceptions in Tera is simple. In addition to general-purpose registers, the hardware context of each thread contains: (1) two condition registers that hardware loads with the cause of an exception, (2) 6 registers that the exception handler can use without saving, and (3) an exception target register that user-level software loads with its exception handler address. On an exception, the hardware simply loads the condition registers with exception information and then *exchanges* the contents of the program counter and the exception target register. As a result of the exchange, the PC is saved in the exception target register, and the next instruction executed for the faulting thread is its exception handler entry. When the exception handler wishes to return, it simply exchanges the PC and exception target register once again.

2.2 General Issues for Direct User Exception Delivery

In this section we consider architectural issues for direct user-mode exception vectoring on conventional RISC architectures. Because of its multi-threaded execution, the Tera processor requires a large amount of per-thread state; in contrast, a conventional processor contains only per-processor state (i.e., one set of registers) that is multiplexed by the operating system among different software processes. Adding user-mode exception vectoring to conventional processors requires handling for several special conditions, which we consider in this section.

The first issue in user-mode exception vectoring is the specification of the exception handler address, which can be managed in several ways. Like the Tera, an additional (user-accessible) register could provide the transfer address, while a second register indicates which exception occurred, permitting quick dispatching of different exceptions. Or, an additional register could point to the base of a process-local vector table, permitting direct vectoring of exceptions to the proper user-level code (although this seems to increase complexity with little likely performance gain). Alternatively, the hardware could vector to a fixed, architecturally-defined address in each virtual address space; the run-time system would be required to load an exception handler or dispatcher at this address. In either case, the cost of a few additional processor registers or the per-process memory vector table is inconsequential on modern processors, many of which already include exception condition registers.

Recursive exceptions—exceptions in user-mode code that is processing an existing exception—must be handled as well. A simple solution is to consider these recursive exceptions to be true error conditions, and to dispatch them to the kernel. One or more bits of the processor status word are thus needed to indicate that the processor is already servicing a user-mode exception, so that the hardware can decide in which mode to deliver the exception. The kernel can still send such recursive exceptions up to user level, giving the application run-time system a last chance to handle the exception. Recursive exceptions would thus be delivered at a slower speed, and possibly to a different user-mode handler established through an operating system call.

Virtual memory presents additional difficulties as well to the user-mode exception mechanism. In particular, the exception handling code (and vector table, if one exists) must be pinned in primary memory to avoid page faults, at least within the first-level user-mode handler. Obviously any user-mode memory management code and data must be pinned as well. This in itself is not particularly difficult, given the amount of primary memory common on contemporary computers. Furthermore, although these pages are pinned, they do not need to remain resident when a process is swapped out or not running, since exceptions can occur only synchronously with program execution. Pinning of the user-mode handlers might permit the exception handler address to be specified physically rather than virtually, thus avoiding the need for translation and the possibility of a TLB miss on an exception; however, this scheme would require the exception address register to be privileged. The tradeoff is not crucial; if the operating system must validate and load the exception address register, user-mode software can simulate a change to that register by placing an indirect jump in the first instruction of its exception handler.

Finally, complete user-level exception handling for the common cases required by many of the intended applications will require user-level access to the processor's memory management state. Most schemes that use memory management, such as garbage collection, debugging, or distributed virtual memory, require that the run-time system manipulate the protection bits for a page, either removing all privileges in order to detect any access, or removing write access in order to catch attempted modification. User-level TLB access for these manipulations can in fact be provided in a straightforward and protected way. Each TLB entry must contain one additional bit, which when set by the kernel permits user-mode code to amplify or restrict read and write permission on that TLB entry. User code is still not allowed to modify the translation information in the TLB entry; only protection bits can be modified. This mechanism requires a tagged TLB, so that only TLB entries for the executing process can be modified.

In summary, we believe that the rapid vectoring of exceptions to user-level code is both sensible and practical, and that such vectoring could be provided rather easily in existing architectures. Furthermore, a simple change to the TLB would permit many exception-based applications to process access detection exceptions completely at user level through controlled modification of read/write protection bits in the TLB. We examine some example uses of these features and their performance benefit in Section 4.

3 A Software Implementation of User-Level Exceptions

Conventional operating systems, like conventional hardware architectures, use a uniform approach to exception handling. Systems such as Unix, for example, combine handling of simple synchronous exceptions with that of asynchronous interrupts and inter-process signals, adding to the cost of user-level exception delivery.

In this section we present a software implementation of fast user-level exception delivery, and demonstrate how an order-of-magnitude performance improvement can be achieved with a software-only approach. This exception handling mechanism could be viewed as an emulation of the hardware approach, or as a slower alternative given the lack of hardware support. While we estimate that perhaps another two- or three-fold performance improvement can be achieved with the hardware approach described previously, the software approach may be useful for many applications as well. In any case, our software prototype allows us to experiment with fast user-level exception delivery and measure or estimate its impact on several simple applications. A key factor in our improved performance is our reduction of state saving and user-kernel boundary crossings, a strategy similar to that used in [Massalin & Pu 89] and [Patience 93].

The software mechanism described here was implemented on the DECstation MIPS R3000 processor running the DEC Ultrix 4.2A kernel. The implementation is compatible with the standard Ultrix signal mechanism; that is, standard Ultrix applications can continue to handle signals as before, and applications that use our mechanisms can receive conventional Unix signals if desired. In addition, the delivery of interrupts and other asynchronous events is completely unchanged by our software.

To motivate the need for a simple operating system mechanism, and to illustrate the difference between our mechanism and a standard Unix-like operating system, we begin with a discussion of signal handling in Ultrix on the MIPS-based DECstation. With minor modifications, our description applies as well to SunOS on SPARCs or to other similar Unix-like systems on RISCs. We then describe our simple exception handling mechanism, including its support for user-level protection modification of TLB entries, and for sub-page-granularity access detection. Finally, we describe the performance of our mechanism.

3.1 Conventional Handling of Synchronous Exceptions

This section describes the handling of a simple exception in the Ultrix operating system on the DECstation. Our objective, as just noted, is to highlight the amount of work inherent in exception handling on a conventional operating system.

Consider, as an example, the delivery of an unaligned data access to a user-level signal handler. The Unix signal handling mechanism is fairly complex and is intended as a general-purpose mechanism to handle simple synchronous exceptions, such as data alignment errors, as well as asynchronous events. Consequently, signal handling within the kernel is divided into three phases—posting a signal, recognizing a signal, and delivering the signal. For asynchronous signals, the posting of the signal is done typically in the context of the signaling process, while the recognition and delivery phases are done in the context of the signaled process. For synchronous exceptions, many of these separate phases are redundant. However, for simplicity, they are treated alike in the kernel.

On encountering an unaligned data access exception, the hardware saves the execution state (the PC, the processor status word, and other supervisor-accessible registers), enters supervisor mode, disables all exceptions and interrupts, and vectors to a fixed kernel address. The kernel's general-purpose exception handler, located at that fixed address, initializes a kernel stack and saves several general-purpose registers. Because the R3000 hardware vectors all exceptions (other than a TLB miss) to the same kernel address, the handler must decode the exception condition before further dispatch. In Ultrix, control then passes to another general-purpose routine inside the kernel, which saves additional user registers on the kernel stack and re-enables exceptions before calling a kernel C language procedure.

The C language routine translates the hardware exception code into a Unix-specific signal; in this case, the unaligned access fault is turned into a `SIGBUS` signal that is posted to the affected process causing a bit to be set in a per-process flag. Since the affected process is the active process, the call to post a signal completes without any scheduling action. After the posting phase, the kernel prepares to deliver the signal to the process. (We ignore, for the purposes of this discussion, the fact that the Ultrix kernel optionally tries to fix up unaligned access exceptions.)

In the delivery phase, the kernel (again in the context of the current process) copies registers saved on the kernel stack at the time of exception into user-accessible memory. Next, it locates the user-specified signal handler from a per-process data area. The kernel modifies the kernel-saved exception state such that (1) on exception return, a fixed register contains the user's signal handler address and (2) a return from exception transfers control to a fixed piece of user-runtime code, called the trampoline code.

Finally, the kernel C language routine returns to its caller restoring the registers saved on exception, switches to the user stack, and executes a return from exception. On exception exit, control passes to the trampoline code in the user runtime, which calls the user-specified handler. The user handler is passed a structure that contains the user-visible saved machine state at the time of the exception. In the course of exception handling, the handler may change the values in the registers, which will be restored on handler return. For example, the handler might choose to advance the saved exception PC, so that the exception is not retaken. On return from the user handler, the trampoline code makes a system call back into the kernel to restore the modified register values, including the exception PC. Figure 1 shows a pictorial view of the process.

Unix signal-handling is a useful and general-purpose facility; however, as the above description makes clear, the generality makes it somewhat cumbersome and expensive. In particular, the Unix signals unify synchronous exceptions with inter-process signals. We emphasize simply that the richness of this general-purpose signal handling mechanism is overkill for simple synchronous exceptions.

3.2 Implementing Efficient User-Level Exceptions

We have modified the Ultrix kernel's general-purpose exception handler to efficiently dispatch synchronous exceptions to a user-specified handler. Our exception-handling mechanism is organized around two observations. First, the kernel need not save user state (as opposed to the Ultrix handler which saves all user registers, some of them twice), because the exception is dispatched to the currently executing process. The process can itself decide what to save, based on its needs. Second, decoding and dispatching to a user-level handler is a trivial operation that requires few instructions. In particular, the decoding and vectoring has

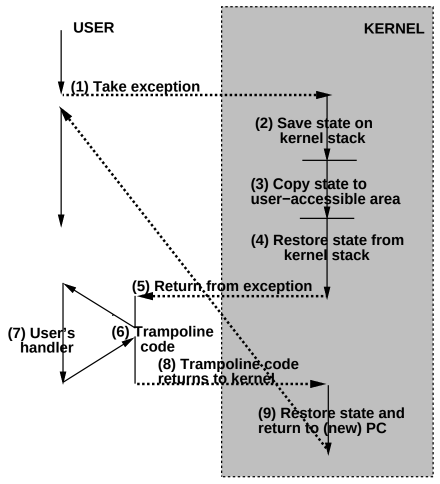
USER
KERNEL
(1) Take exception
(2) Save state on kernel stack
(3) Copy state to user−accessible area
(4) Restore state from kernel stack
(5) Return from exception
(6) Trampoline code
(7) User's handler
(8) Trampoline code returns to kernel
(9) Restore state and return to (new) PC

Figure 1: Sketch of exception handling in Unix with multiple user-kernel domain crossings and register saves.

little impact on the rest of the operating system; that is, the rest of the operating system need not be aware that a synchronous exception occurred.

In our prototype, user processes must explicitly enable synchronous exception delivery. To do so, a process makes a system call specifying a handler address and a list of synchronous exceptions it wishes to handle. In addition, the call specifies a region of user memory in which the kernel stores the saved PC and condition register on exception. The kernel ensures that this memory region is user accessible and pinned. We chose to use one 4K page, because that is the granularity of memory protection on the MIPS; the page contains a communication area (an exception frame) for each exception type enabled. Our implementation does not explicitly prohibit recursive exceptions, therefore, a nested exception of the same type will overwrite the information saved by the kernel on the first exception of that type.

Since we allow potential nesting of exceptions, it is possible for a user process to loop indefinitely, bouncing between the kernel and user-level. This situation is no different in principle than an infinite loop in the user program, which the operating system cannot prevent. Many Unix systems, including DEC Ultrix and SunOS, permit infinite loops to occur with standard Unix signal handlers. Thus, our mechanism is no worse than existing ones in this respect. In any case, such looping can be handled in Unix by signals that unconditionally terminate a process.

A user process can choose to handle any synchronous exception supported by the MIPS architecture [Kane & Heinrich 92], with the exception of system calls, co-processor unusable exceptions, and page faults. Our system does deliver to the user any TLB faults caused by page protection violations. While delivering page faults is no more difficult than page protection violations, additional interface changes would be needed in Ultrix before the user could really exploit this mechanism. The set of exceptions that our mechanism supports includes unaligned accesses, protection faults, and breakpoints, and is sufficient for the needs of most applications that exploit exception handling mechanisms.

3.2.1 Handling Simple Exceptions

Handling exceptions that do not affect the VM system or involve the TLB is particularly simple. On the MIPS R3000, these include integer overflow, unaligned data access, and breakpoint exceptions, all of which can be used to advantage by applications.

When an exception occurs, the hardware vectors to our modified kernel exception handler. The complete work done in the kernel is as follows:

- The kernel handler decodes the exception to ensure that it is an enabled user-mode exception; otherwise, the standard Ultrix exception handling mechanism is invoked.
- The handler then saves the exception PC, the cause of the exception, and the contents of a few scratch registers into the user's shared area. These scratch registers are saved only as a convenience so that the kernel handler can be coded easily. The bulk of the user state is untouched.
- The handler locates the user-specified handler address from a per-process data area, loads the exception PC with that address, and returns from the exception.

At this point, as far as the hardware is concerned, the exception has been handled. Kernel involvement is negligible, barring the few instructions to decode the exception and save state. The only complication is that TLB misses encountered in the kernel handler could alter the contents of the exception PC and other status registers, so this must be handled.

The return from exception enters the user-level handler, which can save additional registers if appropriate and can call arbitrary functions or issue system calls. When the user handler has finished executing, it restores appropriate registers and simply jumps to the exception PC (or a different location), *without* re-entering the kernel. Handling exceptions in this way is fast and effectively emulates, with a small amount of work, direct hardware vectoring to the user handler. Figure 2 shows a pictorial view of the process.

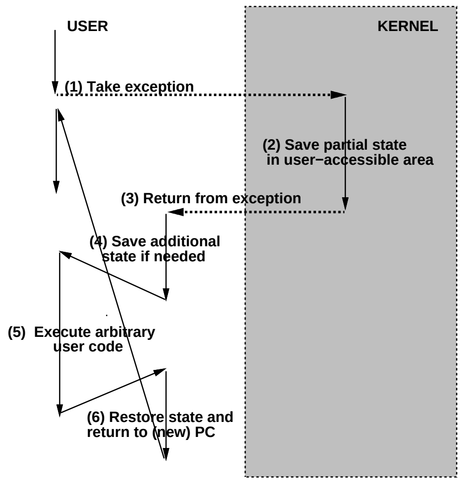
USER
KERNEL
(1) Take exception
(2) Save partial state in user−accessible area
(3) Return from exception
(4) Save additional state if needed
(5) Execute arbitrary user code
(6) Restore state and return to (new) PC

Figure 2: Sketch of fast exception handling.

3.2.2 Handling Virtual Memory and TLB Exceptions

An important class of synchronous exceptions involve memory protection faults and page faults. We wish to distinguish between memory protection faults, which our prototype can deliver to user level, and page faults, which are always handled by the kernel in the normal way. Both types of exceptions are treated by the R3000 hardware as TLB exceptions and are vectored to the kernel's general-purpose exception handler where they are treated differently.

A memory protection fault occurs when a process makes an *illegal* access to a virtual address. This can occur because (1) the virtual address is marked invalid for some reason, (2) the access to the virtual address is incompatible with its protection, e.g., a store on a write-protected page, or (3) the virtual address is not part of the process' address space. In contrast, a page fault occurs when a program makes a *legal* access to a part of its address space that is not resident in main memory. Typically, under Unix, a memory protection violation results in a signal (usually `SIGSEGV`), while the kernel responds to a page fault by initiating a disk operation and creating a new TLB mapping.

Memory protection faults are similar to simple exceptions, but they require the kernel handler to read per-process page tables in kernel memory. Furthermore, the presence of Unix shared memory implies that the handler must perform additional checks before an exception can be correctly dismissed. Consequently, our emulation requires an additional call into a C language routine, which in turn necessitates more state to be saved than for simple exceptions.

3.2.3 User-Accessible TLB Modification

Applications such as distributed shared memory and garbage collection are typical users of virtual memory exceptions. In particular, these applications need to change VM page protection, which they do through kernel calls. We mentioned in Section 2.2 that hardware support for protected user-level access to TLB entries is straightforward. However, in the absence of such hardware, it is still possible to provide support for TLB modification through software emulation of unused opcodes in the kernel. Depending on the application, though, a software approach may not provide acceptable performance in this case, because user's page tables and other kernel data structures must be accessed and manipulated.

It is possible to incorporate an additional optimization, which we call *eager amplification*, that can benefit some applications. When a memory protection exception is received by the kernel, the kernel permits access (i.e., amplifies protection) on that page before vectoring to the user-level handler. An application wishing to re-protect the page would then need to make an additional system call; however, applications such as garbage collectors, persistent store manages, and others can benefit from this minor change in semantics, because they don't always need to re-enable protection checks following the exception [Appel & Li 91]. This optimization works well in environments such as ours, where address spaces are not multithreaded. Even in multithreaded environments this optimization may be useful, because it can be made optional under user program control.

3.2.4 Emulating Subpage-Granularity Protection

Previous studies have noted that architects are increasing page size at the same time that software wants smaller pages, in order to reduce protection granularity and false sharing [Anderson et al. 91]. We demonstrate that with a small amount of kernel support, it is possible to provide subpage granularity protection in a simple way.

In our prototype, users can make kernel calls to protect a memory region composed of "logical" pages that are 1K bytes long. The kernel translates these into appropriate protection settings on hardware-supported 4K byte pages. An illegal access to any protected hardware page causes a hardware exception. If the accessed virtual address is not within a protected logical sub-page, the kernel emulates the instruction, and returns from the exception. The kernel has read/write access to the data, by default, so no protection state need be altered to do the emulation. If the accessed virtual address is within a protected subpage, the kernel enables user access to the entire page and vectors to the user handler. A subsequent call from the user can re-enable protection checks on the logical page.

The kernel emulation of exception-causing instructions in a "non-protected" logical subpage is extremely simple, because only a load or store could cause such an exception. (Technically a jump into a protected page could also cause an exception. Our current implementation does not handle this.) If the memory instruction is in a branch delay slot, then the MIPS architecture causes an exception before the branch is taken. In such cases, the kernel must emulate the branch in addition to the load/store.

There are costs in both space and time to this mechanism. The space cost is fairly small; the kernel needs one bit of information per subpage of user virtual address space. For example, a typical program with a data segment size of 64 Mbytes requires only two pages of overhead (for emulating 1K subpages on 4K hardware pages). The time cost depends on the application and can be more significant. In taking a fault on a logical subpage that is not protected by the user, the kernel must emulate the instruction that caused the fault. This has two component costs: taking the exception and emulating the load/store instruction, as well as a possible branch instruction. The direct cost of subpage handling is quite minimal, that is, as shown in Section 3.3, a subpage exception can be delivered to a user-level handler as efficiently as an ordinary memory protection exception. However, the indirect cost, i.e., the cost of the additional emulation and unwanted vectoring, could be expensive if there is a lot of activity on unrelated logical sub-pages. By providing a mechanism for subpage protection, however, we enable application writers to use it selectively if there is a potential benefit.

3.3 Performance of Software Mechanisms

We implemented our software emulation of user-level exceptions on a 25 MHz MIPS R3000 running a stock Ultrix 4.2A kernel. This section reports on the low-level performance of our mechanisms.

Operation	**Time (μseconds)**	
	Fast Exceptions	**Ultrix**
Deliver Simple Exception to Null User Handler	5	49
Deliver Write Prot. Exception To Null Handler	15	60
Deliver Subpage Exception To Null Handler	19	NA
Return from Null Handler	3	31
Simple Exception Round-Trip Delivery and Return (Rows 1 + 4)	8	80

Table 2: Performance of exception functions. Values in the second column refer to Ultrix times from Table 1.

Table 2 shows the performance of our exception handling implementation for the same operations described in Table 1 of Section 1. For comparison, an Ultrix null kernel call (e.g., `getpid`) is 12 μs. The first row of Table 2 gives the time to dispatch a simple exception to a user-mode handler. The second row presents the cost of delivering a write protection fault without subpage protection handling, while the third row represents the same delivery cost with subpage protection handling. All measurements were made on warm caches, as were the measurements in Table 1. In our implementation, the majority of state is saved by a low-level user handler before it calls the null C handler. Applications can customize their handlers in our system to save less state if appropriate; we saved the same state as Ultrix for these measurements to make the comparison fair.

Compared to Ultrix running on identical hardware, the null user-level exception delivery and return in our mechanism is an order of magnitude faster (8 μs versus 80 μs), and 33% faster than a simple null Ultrix system call. (By comparison, the architectural limit for an exception that enters the kernel and returns immediately is about 2 μs, so we've added only 6 μs to the minimum possible time.) Our performance improvement for delivering memory protection faults is relatively less, but is still 4 times faster than the underlying Ultrix system (15 μs versus 60 μs). The handling of protection faults is slower, relatively, because emulating these faults involves accessing kernel data structures.

Operation	**Instruction Count**
Decode Exception	6
Compatibility Check	11
Save Partial State	31
Floating Point Check	6
Check for TLB Fault	8
Vector to User	3

Table 3: Kernel exception handler instruction count summary.

Our simple (non-TLB) exception handling implementation takes 65 instructions within the kernel; the breakdown of instructions is shown in Table 3. The decode exception phase verifies that the exception is indeed a user-mode synchronous exception. The Ultrix compatibility check ensures that the process has enabled emulation for this particular exception by accessing a per-process flag. The sum of these two phases (17 instructions, of which 4 are no-ops) represents the overhead that we add to the default processing of Ultrix exceptions. The floating point check determines whether the floating point registers need to be saved. The check for a TLB fault determines which type of TLB exception has occurred, if any, and dispatches to an appropriate handler to read user page tables if needed. In addition to these instructions in the kernel, specific user-level handlers will require additional instructions depending on how much state they wish to save.

Relative to other operations, saving state requires the most instructions. This is primarily because the kernel handler must ensure that any TLB misses generated by its execution do not destroy the original exception state information. Given our implementation on one architecture, it is difficult to speculate on the effort needed on other platforms. However, a SPARC-like architecture with an extra set of registers for exception handling would reduce the cost of saving state.

4 Applications of Fast Exception Handling

We believe that low-cost exception handling can have both a qualitative and quantitative impact on applications. First, low-cost exceptions could enable the use of techniques previously considered impractical in contemporary systems, due to their high exception handling cost. Second, techniques that have traditionally used exception-based mechanisms can be made to run more efficiently.

As we have noted, many uses for exception handling have been previously discussed in the literature.

In this section, our goal is simply to examine a small number of applications for exception handling, and to show how improved exception performance can shift the balance between exception handling and alternative mechanisms. We consider two different types of exceptions — memory protection faults and unaligned address faults — and show uses and performance implications for each.

4.1 Using Memory Protection Faults

The use of memory protection faults in applications such as distributed shared memory and garbage collection has received much attention [Appel & Li 91, Hosking & Moss 93]. In this section we report on the performance of a garbage collector that uses our software delivery mechanism to handle protection violation exceptions. For our measurements, we use a conservative garbage collector, distributed by Xerox, which is meant to be used with C and C++. We trivially modified the standard Xerox version to enable generational and incremental collection support for Ultrix. Otherwise, the operation of the collector is similar to that described in [Boehm & Weiser 88].

A generational garbage collector separates heap objects as belonging to multiple generations. Empirical evidence has shown that most of the garbage is created in "younger" (more recently allocated) rather than "older" generations [Lieberman & Hewitt 83, Ungar 84]. Hence, most of the collection can be performed by scanning the newer generations. Occasionally, there are pointers from outside the collected area, i.e., from older generations back into younger ones. The collector must consider these during a collection. The Xerox collector tracks these locations in the older generations by write-protecting memory pages that contain old generations. Thus, depending on the nature of the application, the cost of handling protection violations can be a factor in overall performance.

We measured the performance of two simple synthetic applications, written in C, using the Xerox garbage collector with and without our optimized exception delivery. The first application simulates the behavior of simple Lisp operators, such as `cons`, `car`, and `cdr`. It repeatedly creates large Lisp-like data structures without explicit garbage collection. In the process, it runs the garbage collector about 80 times and generates over 2000 protection violation exceptions. The second benchmark creates a large array (1 MB) and randomly replaces elements in the array. Each replacement operation creates garbage and some of the replacement operations cause page protection violations to occur. Relative to the overall running time of the test, this benchmark creates many more older-to-younger pointer stores than the first application and generates about 2000 exceptions. For each benchmark, we built two versions. One version uses the standard Ultrix `SIGSEGV` signal for detecting faults and the `mprotect` system call to change page protection. The other version relies on our fast protection exception delivery with eager amplification.

Table 4 shows the performance impact of page protection mechanisms on the garbage collector. The times shown are CPU times, not wall clock time, of the complete application. It is clear that the performance is highly dependent on application behavior; i.e., in some applications, relative to the running time of the application, older-to-younger pointers may not be created often enough for protection exceptions to be a factor.

There are several other techniques of tracking older-to-younger generation pointer stores that do not rely on page protection exceptions. One commonly used scheme is to insert software checks before each store. Depending on the relative costs of a software check and a exception delivery, a software checking scheme may be preferable to the exception delivery scheme.

Hosking and Moss have compared the relative performance of the two approaches on two applications running on the DECstations 3100 under Ultrix [Hosking & Moss 93]. Both applications are written in Smalltalk. The first program ("Tree") is a synthetic benchmark based on tree creation and destruction. The

Application	**CPU Time of Application (seconds)**		
	Ultrix SIGSEGV	**Fast Exceptions**	**Percentage Improvement**
Lisp Operations	24	23	4%
Array Test	2	1.8	10%

Table 4: Comparative performance of generational garbage collection.

second program (“Interactive”) is a standard benchmark suite used to compare the relative performance of Smalltalk environments.

The DECstation 5000/200 used in our experiments has the same instruction set architecture and operating system, but different memory systems and performance compared to the DECstation 3100 used by Hosking and Moss. Thus, a particular software check on the 3100 will require the same number of instructions, but not the same number of machine cycles as the 5000/200. Also, a given application using page protection will generate the same number of protection violations on both systems, because they run identical operating systems and the same application with the same page reference pattern. Unlike page faults, memory protection violations are not affected by external, real-time variation, such as the time to fetch data from disk.

We can therefore use application characteristics reported in [Hosking & Moss 93] to compare the performance of software checks and page protection using our software exception mechanism. Depending on the relative costs of software checks and exceptions, and the number of exceptions, a break-even point can be calculated for the two techniques. Following the notation in [Hosking & Moss 93], let x be the number of cycles required for each software check and c the number of software checks needed for a particular application. Let t be the number of exceptions required by the same application using a page protection scheme and let f and y represent the machine clock frequency in megahertz and the cost of a exception in microseconds. Then, protection exception has better performance than software checking if: $y < cx / ft$.

In the Hosking and Moss study, a typical software check takes 5 instructions on the DECstation 3100; we assume, conservatively, that the same check takes exactly 5 (25 MHz) cycles on the DECstation 5000/200, i.e., x is 5 and f is 25. Given these assumptions, Table 5 shows the break-even point between software checks and exceptions for the two applications. As a reference, an exception and re-enable of protection takes 18 μs using the eager amplification optimization in our system. Thus, our software emulation scheme appears to offer a competitive alternative to software checks for these applications on our hardware/software combination.

	Application	
	Tree	**Interactive**
Store Checks (c)	59646	654245
Page Exception (t)	864	1656
Break-even Point	$y < 14\ \mu$s	$y < 79\ \mu$s

Table 5: Break-even points for page exception and software checks. y is the cost of handling a protection fault.

4.2 Using Unaligned Access Exceptions

Many architectures raise exceptions when an unaligned data access is attempted. For example, loading or storing a 4-byte word on an odd-byte or odd-halfword boundary causes an exception on the MIPS-based DECstation. Unaligned data exceptions, if delivered quickly to user level, can serve as an effective basis for implementing unbounded data structures, futures, full/empty bits, and other useful features of programming languages and systems. We briefly discuss several of these examples below.

4.2.1 Unbounded Data Structures

Potentially unbounded data structures, such as streams, are a valuable computational paradigm in languages such as Scheme [Abelson & Sussman 85]. Given a software/hardware architecture that delivers unaligned word access exceptions to user level, it is easy to build data structures that are incrementally augmented on demand. Thus, unbounded data structures can be built in a language-independent fashion, without requiring the programmer to make explicit calls to create the next element.

A simple example of an unbounded linked list is described below, but the idea can be extended to other structures as well. Each linked list element consists of two word-aligned fields: (1) a datum that can hold an arbitrary value, and (2) a word pointer to the next element of the list. At any point in time, we expect only part of the list to have been evaluated. The evaluated part is stored in the usual manner with the pointer field in each cell pointing to the next element in the list. The unevaluated part of the list is not allocated but is denoted by an unaligned pointer field in the last element of the evaluated part of the list. A program that tries to access the unevaluated part takes an unaligned access fault that extends the list appropriately.

The mechanism of unaligned exceptions can be extended in a conceptually similar way to implement constructs such as futures [Kranz et al. 89]. One obvious approach is to represent an unresolved future as an unaligned pointer. When the value of the future is available, the pointer is updated and aligned. In fact, the APRIL processor used in the Alewife machine does precisely this [Agarwal et al. 90]. With fast user-level exception delivery, similar mechanisms can be exploited in more conventional architectures as well.

Another example of the use of unaligned references is to implement synchronization through full/empty bits, as is done on some processors [Alverson et al. 90, Agarwal et al. 90]. An attempt to read from an empty location or write to a full location causes the reader or writer to block until the location is filled or emptied, respectively. While this is accomplished on special-purpose processors using additional tag bits on each memory word, it could as well be implemented using indirect and potentially unaligned pointers. Blocking for both write-on-full and read-on-empty requires two separate pointers. On the one hand, the storage overhead is greater with unaligned pointers for words requiring such synchronized access, however the hardware scheme requires tag bits on all memory words, which will likely cost more storage overall.

4.2.2 Pointer Swizzling for Persistent Object Storage Systems

Persistent object stores provide sharable, recoverable, heap storage to programs. In such systems, heap data is moved between stable disk and volatile memory transparently to the user program. Typically, pointers to objects on disk and pointers to objects in memory have different representations. The object storage system provides a transparent mechanism, called *pointer swizzling*, that manages the complexity of dealing with two representations. A pointer is “swizzled” to change it from on-disk format to in-memory format (i.e., a virtual address); it is “unswizzled” to change it from in-memory format to on-disk format (e.g., an object identifier). Persistent systems use several common approaches to handling this transparent data movement and the swizzling of pointers.

In the first approach, the compiler inserts software checks at each potential pointer dereference site [White & DeWitt 92]. If the user dereferences a pointer that refers to data on stable storage, the check will detect an unswizzled pointer and cause that data to be brought into memory. The pointer value is then swizzled so that subsequent software checks will indicate that the object is memory resident.

An alternative approach is to swizzle pointers using protection faults. In this scheme, dereferencing a pointer to a non-resident object causes a fault. The handler then loads the referenced page from disk, swizzling the pointer and allowing it to be dereferenced without overhead.

The checking scheme has the disadvantage that software checks are performed on every pointer dereference, whether or not the referenced object is memory resident. For example, a procedure that deferences a pointer to the heap may be called many times with a pointer to the same object; all but the first access may refer to an in-memory object, in which case all but one of the checks are unnecessary.

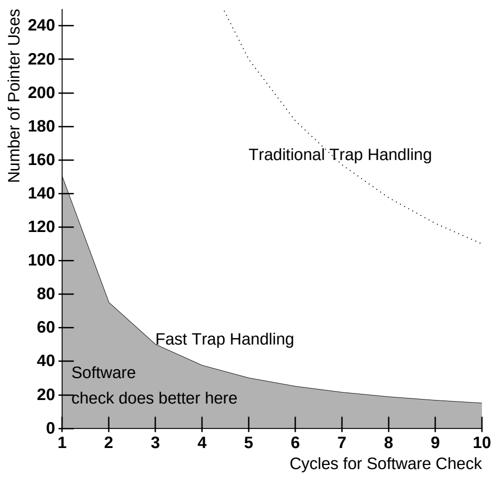

Figure 3: Exceptions Versus Software Checking for Swizzling

We illustrate the tradeoff between using exceptions and software checks by presenting a simple quantitative comparison. To underscore the flexibility of user-level exception handling, we installed a specialized user-level "swizzling handler" for unaligned exceptions. Because it is specialized, this handler needs to save only a few registers in addition to the kernel-saved state. In particular, callee-saved registers are not saved. Using this handler, the cost of taking an unaligned exception and calling and returning from a null C procedure is 6 μs (2 μs less than the time shown in Table 2). We assume that a software check for non-residency can be done in c cycles and that a particular pointer is used u times. Then in our current implementation on the 25 MHz processor, the swizzling approach is superior as long as $cu > 25 \times 6$. Figure 3 graphs the breakeven point between software checks and exceptions as a function of c (cycles per check) and u (number of uses per pointer). The higher curve shows the breakeven point between software checks and exceptions given the exception handling cost in Ultrix (software checks are better below the curve); as we can see, software checks will be superior in this case, unless each pointer is used hundreds of times, or unless software checks are very expensive. In contrast, the lower curve shows the same tradeoff for our exception mechanism; as the graph shows, the performance difference in our mechanism has greatly shifted the balance point, making exception-based swizzling superior for a much smaller number of pointer uses and smaller check costs. For our mechanism, software checks pay only in the shaded area.

Irrespective of whether swizzling is done using exceptions or software checks, there are two variations of swizzling: *eager swizzling* and *lazy swizzling*. In eager swizzling [Wilson & Kakkad 92], when an object is loaded, all the pointers within that object are found. Non-resident objects referenced by those pointers are then assigned virtual address space, and the pointers are swizzled to point to those non-resident virtual pages. Pointers in the loaded object to memory-resident objects are simply swizzled to those objects' memory addresses. In lazy swizzling [Cockshot et al. 84], each pointer within a newly-loaded object remains in unswizzled format until it is first used; at that point the pointer is swizzled.

Relative to lazy swizzling, the eager swizzling scheme has the disadvantage that many pointers that may never be used may be swizzled when an object is loaded into memory. This is a potential problem if objects are traversed sparsely.

Lazy swizzling using exceptions is currently unattractive because of the prohibitive cost of taking a exception for each pointer. Lazy swizzling suffers (relative to eager swizzling) if all of the pointers on a page are eventually used, since each one will cause a separate exception on first use. Using our mechanism, lazy swizzling could be supported using unaligned addresses. Addressing a non-resident object will cause a fault, which will load the object if necessary and repair the address. The next time the address is used, there is no penalty.

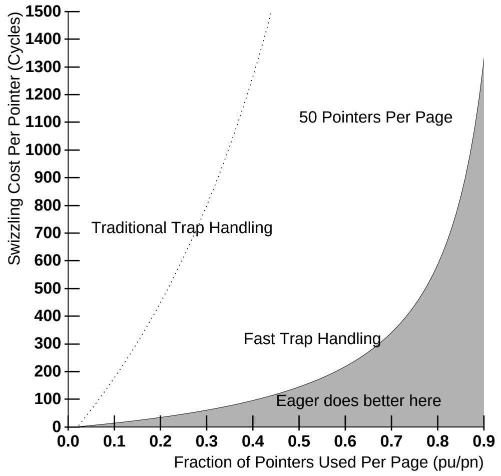

Figure 4: Eager Versus Lazy Swizzling Using Exceptions

Again, fast fault handling changes the breakeven point. In this case, eager swizzling will perform many more swizzling operations (in the worse case), but each at less cost. Let t be the time per exception, s be cost to swizzle a pointer, pn be the number of pointers per page, and pu be the number of pointers actually used per object, on average. Then eager swizzling should be used as long as $t + pn \times s < pu\,(t{+}s)$. Figure 4 graphs the breakeven point as a function of swizzling cost and the percentage of pointers per object that are used, for both the traditional and optimized exception handling approaches. This graph assumes that there are 50 pointers per page. The leftmost curve shows the breakeven point for traditional exception handling (i.e., Ultrix); to the right of that curve, eager swizzling performs better, while to the left, lazy swizzling performs better. The rightmost curve graphs the breakeven point for our software mechanism; once again, we see the strong shift caused by the greatly reduced exception handling cost, which in this case makes lazy swizzling advantageous for a broader range of parameter values.

5 Summary and Conclusions

Exception handling has become a commonplace mechanism for supporting the needs of runtime systems. Unfortunately, conventional hardware and software systems have not optimized exception handling performance. In particular, exceptions are delivered to the operating system, which often has insufficient information to handle the exception. The typical result is that general-purpose kernel code performs substantial (and perhaps unnecessary) work before dispatching the exception to the user-level runtime system; furthermore, the user-level handler must return through the kernel to dismiss the exception and retry the faulting instruction. On some new operating system structures, these costs could be substantially compounded if the exception is required to pass from the kernel to an operating system environment process before the application runtime system can be invoked.

In this paper we have described two approaches to improved exception performance on conventional RISC processors and operating systems. The first approach requires a straightforward architectural change, in which hardware vectors synchronous exceptions directly to user level. We described some of the issues with this approach, and presented a mechanism for safe user-level modification of TLB state, in order to permit user-level management of protection exceptions. The second approach is a software implementation of that architectural approach, which quickly dismisses the exception, vectoring it back to the user. In both cases, user-level code can save only the state that is needed, process the exception, and return directly to application processing. Should kernel action be required, the handler simply calls the kernel through the normal system call mechanism; however, even in this case, exception handling time would decrease, because a system call is much faster than the time to dispatch an exception to the user on current systems.

We have implemented the software approach and showed that it achieves an order-of-magnitude improvement over exception handling mechanisms in several current operating systems. In addition, we have presented as examples several uses of memory protection and unaligned address faults, and have evaluated their performance through measurement or analysis. By greatly reducing the time to process an exception, we shift the tradeoff point between exceptions and alternative mechanisms for accomplishing the same function. In some cases, performing this shift may enable new uses of exceptions that were formerly prohibitive, benefiting application performance, structure, or both. We may also enable new operating system structures by permitting kernel-resident functions to be relocated to user level, thereby giving runtime systems more flexibility in handling application needs.

We should note that exceptions are always a performance optimization. That is, exceptions can in all cases be replaced by some sequence of instructions that explicitly check for the exceptional condition. This is true even for arithmetic exceptions, such as overflow, or even more troublesome, for memory violations. In these cases in particular, it is clear that we do not want to pay the cost of software checks for every arithmetic or memory access instruction. Exceptions are thus implemented in hardware so that the *normal case* does not suffer from executing those additional instructions. But in all cases, we can evaluate the tradeoff based on the frequency of traps, the cost of exception, the frequency of the required checks, and other factors.

Acknowledgments

We would like to thank many people who greatly aided us in this study. Jeff Dean and Dave Grove of the CECIL project helped evaluate the impact of traps on garbage collection. Gail Alverson and Burton Smith of Tera Computer shared the details of the Tera's exception architecture. David Keppel and Burton Smith suggested many applications of fast traps. We also thank Rob Fowler, Povl Koch, and Niels Elgaard for pointing out an error in an earlier version of Table 1. Jeff Chase and Ashutosh Tiwary were instrumental in our study of pointer swizzling. Jeff Chase, Ed Lazowska, and Burton Smith provided valuable comments on early versions of the paper.

References

[Abelson & Sussman 85] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, Cambridge Massachusetts, 1985.

[Agarwal et al. 90] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz. April: A processor architecture for multiprocessing. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 104–114, May 1990.

[Alverson et al. 90] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. *International Conference on Supercomputing*, pages 1–6, June 1990.

[Anderson et al. 91] T. E. Anderson, H. M. Levy, B. N. Bershad, and E. D. 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.

[Appel & Li 91] A. W. Appel and K. Li. Virtual memory primitives for user programs. In *Proceedings of the 4th International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 96–107, April 1991.

[Appel et al. 88] A. W. Appel, J. R. Ellis, and K. Li. Real-time concurrent collection on stock multiprocessors. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 11–20, June 1988.

[Boehm & Weiser 88] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software—Practice and Experience*, 18(9):807–820, September 1988.

[Chang & Mergen 88] A. Chang and M. F. Mergen. 801 storage: Architecture and programming. *ACM Transactions on Computer Systems*, 6(1):28–50, February 1988.

[Cockshot et al. 84] W. Cockshot, M. Atkinson, K. Chisholm, P. Bailey, and R. Morrison. Persistent object management system. *Software—Practice and Experience*, 14(1):251–272, January 1984.

[Hosking & Moss 93] A. L. Hosking and J. E. B. Moss. Protection traps and alternatives for memory management of an object-oriented language. In *Proceedings of the Fourteenth ACM Symposium on Operating*, pages 106–119, December 1993.

[Kane & Heinrich 92] G. Kane and J. Heinrich. *MIPS RISC Architecture*. Prentice Hall, Englewood Cliffs, New Jersey 07632, 1992.

[Kranz et al. 89] D. A. Kranz, R. H. Halstead, and E. Mohr. Mul-T: A high-performance parallel Lisp. In *Proceedings of SIGPLAN '89 Symposium on Progamming Languages Design and Implementation*, pages 81–90, June 1989.

[Li & Hudak 89] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[Lieberman & Hewitt 83] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[Massalin & Pu 89] H. Massalin and C. Pu. Threads and input/output in the Synthesis kernel. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 191–201, December 1989.

[Ousterhout 90] J. K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proceedings of the Summer 1990 USENIX Conference*, pages 247–256, June 1990.

[Patience 93] S. Patience. Redirecting system calls in Mach 3.0, an alternative to the emulator. In *Proceedings of the USENIX Mach III Symposium*, pages 57–74, April 1993.

[Ungar 84] D. M. Ungar. Generation scavenging: A non-disruptive high-performance storage reclamation algorithm. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, April 1984.

[Wahbe 92] R. Wahbe. Efficient data breakpoints. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 200–212, October 1992.

[White & DeWitt 92] S. J. White and D. J. DeWitt. A performance study of alternative object faulting and pointer swizzling strategies. In *Proceedings of the 18th VLDB Conference*, pages 419–431, 1992.

[Wilson & Kakkad 92] P. R. Wilson and S. V. Kakkad. Pointer swizzling at page fault time: Efficiently and compatibly supporting huge address spaces on standard hardware. In *Proceedings of the 1992 International Workshop on Object Orientation in Operating Systems*, pages 364–377, September 1992.

[Young 89] M. W. Young. *Exporting a User Interface to Memory Management from a Communication-Oriented Operating System*. Ph.D. dissertation, Carnegie Mellon University, November 1989. Technical Report CMU-CS-89-202.