Fault Interpretation: Fine-Grain Monitoring of Page Accesses

Daniel R. Edelson*

INRIA Project SOR
Rocquencourt B.P. 105
78153 Le Chesnay CEDEX
FRANCE
Daniel.Edelson@inria.fr

9 November 1992

Abstract

This paper presents a technique for obtaining fine-grain information about page accesses from standard virtual memory hardware and UNIX operating system software. This can be used to monitor all user-mode accesses to specified regions of the address space of a process. Application code can intervene before and/or after an access occurs, permitting a wide variety of semantics to be associated with memory pages. The technique facilitates implementing complex replication or consistency protocols on transparent distributed shared memory and persistent memory. The technique can also improve the efficiency of certain generational and incremental garbage collection algorithms. This paper presents our implementation and suggest several others. Efficiency measurements show faults to be about three orders of magnitude more expensive than normal memory accesses, but two orders of magnitude less expensive than page faults. Information about how to obtain the code via anonymous ftp appears at the end of the paper.

*This work was performed while the author was visiting INRIA. The author's most recent affiliation is: Computer and Information Science, University of California, Santa Cruz CA 95064, *daniel@cse.ucsc.edu*.

Introduction

This paper shows how a program can use common UNIX virtual memory page protection and signal handling to monitor all accesses to selected pages of its address space. The technique has been encapsulated in a library called $\mathcal{FI}$ for *Fault Interpretation*. We discuss a number of applications for this technique including garbage collection and consistency/replication protocols for transparent distributed shared memory.

Virtual memory page protection has been used for similar reasons before [AEL88, AL91, DWH+90]. The difference with our approach is that most other techniques unprotect a protected page when a fault occurs. For some period of time thereafter, there is no monitoring of how many times and at what addresses the page is accessed. With fault interpretation, in contrast, a page does not remain unprotected. When an access causes a fault, the page is now unprotected and the access is performed. Then, the page is restored to its previous protection state and the application resumes at the subsequent machine instruction. A notification function, registered by the application, can intervene immediately before and/or after the access. It is as if the access succeeds and the application is informed that the access occurs.[1]

The remainder of this report is organized as follows: Section 1 gives an overview of the technique with a small example. Section 2 presents a C library interface that encapsulates the functionality. Section 3 discusses some applications. Then, Sect. 4 describes the implementation and Sect. 5 presents efficiency measurements. Section 6 discusses the availability of the library and some caveats, and Sect. 7 concludes the report.

1 Fault Interpretation: Memory Access Monitoring

Fault interpretation allows an application to detect all reads and/or writes to selected pages of its virtual address space. The library uses the mprotect system call to disallow accesses to monitored pages. An access to a protected page causes a fault, which UNIX passes to the application as a signal. The $\mathcal{FI}$ signal handler unprotects the page and notifies the application of the access.

[1] *Caveat:* This technique requires knowing the precise state of the CPU when a protection violation occurs. It may not be possible to implement this functionality on all RISC architectures. We have implemented it on the SPARC processor [Cyp90, Sun87].

Then, the faulting instruction is restarted; it succeeds because the page is unprotected. Control returns immediately to the $\mathcal{FI}$ library, which notifies the application again, re-protects the page, and resumes the application at the next instruction.

As just described, the application can be notified twice per access. These two function invocations are referred to as *pre-access* notification and *post-access* notification. The two calls permit a wide variety of semantics, for example, pre-access notification might be used to read a page over the network, to obtain a write-lock on a page, or simply to record the address of the access. Post-access notification might release a write lock or send an updated page to other hosts. It might also be used by a debugger to detect that a variable has been accessed.

Arguments to the notify function indicate the address of the access, its type (read, write or swap) and how many bytes are involved; the access type and number of bytes are obtained by decoding the instruction. During notification, the accessed page is unprotected, permitting the notify function to access the page without faulting.

$\mathcal{FI}$ utilizes the UNIX mprotect system call and traps the resulting signal, which is typically either SEGV or BUS. When the signal is caught, the operating system passes the handler information about the faulting context. To support fault interpretation, this information must include the program counter and the other registers. $\mathcal{FI}$ uses this information to determine what access the program was performing and to alter the usual flow of control.

The $\mathcal{FI}$ signal handler can coexist peacefully with other signal handlers, provided they are not both trying to catch the same kinds of signals on the same memory pages. When $\mathcal{FI}$ traps a signal from a fault on an unmonitored page, the signal is propagated to any other handler that is installed for the signal.

The biggest difference between this and common uses of virtual memory (VM) protection is that the faulting instruction is (effectively) single-stepped, rather than resumed normally. After the instruction succeeds, control returns to the library's *reprotect block*, which performs the post-access notification, reprotects the page, and resumes the application. Thus, the page is only unprotected for the one instruction that faults (as well as during notification); all accesses to the page can be trapped. This effect could be accomplished using the ptrace system call but that doesn't permit a process to monitor itself; it can only monitor another process.

The best way to demonstrate the exact effect obtained is through an example. We present a small test application that obtains some protected

memory and causes faults. The handler displays the address and the type of every fault. The application is shown in Fig. 1. The output of the application follows in Fig. 2. As this example demonstrates, an application can very easily obtain a region of managed memory. Thereafter, the application will be notified upon every access to the region.

2 Library

The $\mathcal{FI}$ library encapsulates the functionality that is described in the previous section. The library includes calls to obtain managed memory, to change the state or attributes of the memory, and to release the memory when it is no longer needed.

When a fault occurs, the exact sequence of events is the following:

1. An instruction attempts to access a protected page; the instruction faults. The operating system invokes the $\mathcal{FI}$-installed signal handler.
2. The $\mathcal{FI}$ signal handler verifies that the fault occurred on a page that is managed by $\mathcal{FI}$. If not, the signal is propagated to any previously installed signal handler. If the page is managed, the page will not be unprotected.
3. If pre-access notification has been requested, the application is notified. The notification function is passed the fault address, the number of bytes, and flags indicating whether the access is a read or write (or both) and that the notification is pre-access. The notification function can examine or modify the page.
4. The faulting instruction is executed again. Since the page is not protected, the access succeeds. Control returns immediately to $\mathcal{FI}$.
5. If post-access notification has been requested, $\mathcal{FI}$ calls the notification function. The same arguments are passed except that the flags indicate post-access.
6. $\mathcal{FI}$ returns the page to its previous protection state and resumes the application. The application continues with the instruction following the one that caused the fault.

The library is written in C [ANS89, ISO90] using UNIX system call extensions. It can also be compiled as C++ code. In order to avoid name clashes, all external identifiers used in $\mathcal{FI}$ begin with fi_.

```
#include <stdio.h>
#include "fi.h"
#define PGSIZE 4096

/* notify prints the address and type of the access */
void notify(void * addr, size_t nb, fi_flags_t type) {
   printf("NOTIFY: Access 0x%p for %d bytes, type ",addr,nb);
   if (type & FI_PREREAD)   printf("PREREAD ");
   if (type & FI_PREWRITE)  printf("PREWRITE ");
   if (type & FI_POSTREAD)  printf("POSTREAD ");
   if (type & FI_POSTWRITE) printf("POSTWRITE ");
   printf("\n");
}

int main() {
    int i, * addr;

    fi_initialize();
    /* Allocate one page of managed memory */
    addr = (int*) fi_alloc(PGSIZE,fi_noaccess,notify,FI_ALL);
    printf("Causing four faults now!\n");
    addr[0] = 6;
    addr[121] = 999;
    i = addr[40];
    i = addr[400];
    printf("Permit READ accesses without faulting.\n");
    fi_setprot(addr, PGSIZE, fi_readonly);
    printf("Causing two faults now!\n");
    addr[0] = 6;
    addr[121] = 999;
    i = addr[40];    /* no fault: read access permitted */
    i = addr[400];   /* no fault: read access premitted */
    fi_free(addr);
    return 0;
}
```

Figure 1: A small $\mathcal{FI}$ application

```
Causing four faults now!
NOTIFY: Access at 0x7000 for 4 bytes of type PREWRITE
NOTIFY: Access at 0x7000 for 4 bytes of type POSTWRITE
NOTIFY: Access at 0x71e4 for 4 bytes of type PREWRITE
NOTIFY: Access at 0x71e4 for 4 bytes of type POSTWRITE
NOTIFY: Access at 0x70a0 for 4 bytes of type PREREAD
NOTIFY: Access at 0x70a0 for 4 bytes of type POSTREAD
NOTIFY: Access at 0x7640 for 4 bytes of type PREREAD
NOTIFY: Access at 0x7640 for 4 bytes of type POSTREAD
Change page to READONLY.
Causing two faults now!
NOTIFY: Access at 0x7000 for 4 bytes of type PREWRITE
NOTIFY: Access at 0x7000 for 4 bytes of type POSTWRITE
NOTIFY: Access at 0x71e4 for 4 bytes of type PREWRITE
NOTIFY: Access at 0x71e4 for 4 bytes of type POSTWRITE
```

Figure 2: Output of the small $\mathcal{FI}$ application

Managed memory is obtained in segments whose size is an integral number of pages. Within a segment, the protection state, notification, and notify function of each page may be independently specified.

The library interface defines a small number of types, constants and functions. The first type is an enumeration that indicates what protection state the application requires for a page. The type is defined as follows:

```
typedef enum {
    fi_noaccess,
    fi_readonly,
    fi_readwrite
} fi_prot_t;
```

The enumeration constants mean:

`fi_noaccess`	No accesses to the page are permitted, meaning all accesses result in faults.
`fi_readonly`	Read accesses do not fault.
`fi_readwrite`	Both reads and writes are permitted without faulting.

Another set of flags defines the types of notification. The flags are bit values that may be ORed together. The values of the constants have been elided.

```
typedef unsigned char fi_flags_t;

#define FI_PREREAD   /* Pre-access notification for reads         */
#define FI_PREWRITE  /* Pre-access notification for writes        */
#define FI_PRE       /* Pre-access notification for all accesses  */
#define FI_POSTREAD  /* Post-access notification for reads        */
#define FI_POSTWRITE /* Post-access notification for writes       */
#define FI_POST      /* Post-access notification for all accesses */
#define FI_READ      /* Pre and post notification for reads       */
#define FI_WRITE     /* Pre and post notification for writes      */
#define FI_ALL       /* Pre and post notification for all accesses*/
```

When obtaining pages of managed memory, the application supplies a pointer to a notification function. The type of that function pointer is the following:

```
typedef void (*fi_notify_t)(caddr_t, size_t, fi_flags_t);
```

The caddr_t argument is the address of the fault. The size_t argument is the number of bytes involved in the access. The fi_flags_t argument indicates the type of access and whether the notification is pre-access or post-access.

Finally, the last part of the interface is the prototypes of the library functions. These prototypes are summarized in Fig. 3. The meanings of the functions are the following:

fi_initialize This function must be called first to initialize the library.

fi_alloc This routine allocates new monitored memory. The function returns a pointer to the allocated pages. The initial protection state and notify function are parameters to the function, as is the number of pages to allocate.

fi_addpages As with fi_alloc this function adds more managed memory. However, this routine allows the user to supply the address of the memory, rather than obtaining the memory from valloc or sbrk.

fi_free This free routine tells the library to stop using a set of pages. If the pages were obtained with fi_alloc they are deallocated.

```
void  fi_initialize(void);
void* fi_alloc(size_t, fi_prot_t, fi_notify_t, fi_flags_t);
void* fi_addpages(void*, size_t, fi_prot_t, fi_notify_t, fi_flags_t);
int   fi_free(void* addr);
int   fi_setprot(void* pgaddr, size_t nb, fi_prot_t nw);
int   fi_setnotify(void* pageaddr, size_t nb, fi_notify_t nw);
int   fi_setflags(void* pgaddr, size_t nb, fi_flags_t nw);
int   fi_getprot(void* pgaddr, fi_prot_t* old);
int   fi_getnotify(void* pageaddr, fi_notify_t* old);
int   fi_getflags(void* pgaddr, fi_flags_t* old);
```

Figure 3: $\mathcal{FI}$ function prototypes

---

fi_setprot This function sets the protection state of one or more managed pages. This determines what kinds of accesses, reads or writes, cause faults.

fi_setnotify This function sets the notify function pointer associated with one or more pages.

fi_setflags The fi_setflags interface is used to set the kind of notification required: pre-access and/or post-access.

fi_getprot This function returns the protection state of a page.

fi_getnotify This function returns the notify function pointer associated with a page.

fi_getflags This routine returns the notification flags of a page.

3 Applications

Possible applications of this technique include: write-detection in generational or incremental garbage collection, and consistency/replication protocols for shared memory.

3.1 Generational Garbage Collection

The idea behind generational garbage collection (GC) is that some objects are likely to remain reachable for the immediate future, thus, attempting to reclaim their memory is not worthwhile. [DWH+90, LH83, Moo84]. Typically, young objects are expected to become garbage relatively soon [Ung84], therefore, the garbage collector concentrates its effort on the young objects.

A garbage collection of the young objects (the younger *generation*) requires locating all pointers to young objects. Such pointers are of three types:

1. pointers on the stack, in global data, and in registers,
2. pointers in young objects, or,
3. pointers in old objects.

Pointers of the first two types are common to all GC algorithms and do not introduce new difficulties. Pointers of the third kind are called *back pointers* and they introduce a problem that is unique to generational garbage collectors. These pointers must be located to avoid erroneously reclaiming live objects. However, since the collector is concentrating on young objects, it does not want to examine the old objects to locate these pointers. Thus, the task is to efficiently locate the set of all these pointers.

Some collectors add a run-time test to every (pointer) assignment to see if a back pointer is being created. Other collectors do not attempt to locate each individual pointer, but rather identify the set of pages that might contain such pointers, the *remembered pages*. During garbage collection, every object on a remembered page is scanned for back pointers. This has been implemented using page protection [DWH+90]. The garbage collector write-protects all of the older-generation pages. Every fault indicates that there has been an assignment to an older generation object; the page is added to the remembered set. Upon collection, the remembered pages are scanned for back pointers. If a page contains no back pointers, then then it is deleted from the remembered set. Otherwise, it is left in the set.

This implementation of the remembered set unprotects a page every time a fault occurs, permitting any number of writes to the page. Since it doesn't know what addresses were written, the collector must scan every object on every remembered page looking for back pointers. Even if only one word on the page is modified, the collector still must check every field of every object. In contrast, through memory access monitoring, the collector can

have available the exact list of address that are modified. It is not necessarily desirable to remember the exact list, since that could be quite expensive. Instead, the collector can keep N remembered addresses per page. For the first N faults that occur on a page, the collector stores the fault address. Upon the next fault after that, the collector unprotects the page and treats the page the same as in the old system. This bounds the maximum time and space overhead due to faulting.

The exact value of N depends on two things: the efficiency of handling a fault, and the cost of scanning a page. If every field on a page can be scanned in less time than it takes to handle a fault, then fault interpretation should not be used. However, if scanning objects is relatively expensive, then remembering several stored addresses may improve efficiency.

3.2 Incremental Garbage Collection

Incremental garbage collection is a family of algorithms in which the collector never stops the application for an extended period of time. The first such algorithm was Baker's copying collector [Bak78] with many other algorithms based on it. To avoid annoying pauses, the collector does its work in short chunks. Incremental garbage collectors are often concurrent, in which case protected pages of memory can serve as medium grain synchronization mechanism between the collector and the application [AEL88].

3.2.1 Incremental Mark-and-Sweep Collection

Incremental mark-and-sweep garbage collection has been implemented previously using virtual memory page protection [BDS91]. The normal implementation provides one bit of information per page: there was or was not a fault. Pages on which a fault occurred must be entirely rescanned. This is another case in which fault interpretation can provide finer granularity information, possibly increasing the efficiency of the algorithm.

Incremental mark-and-sweep collectors do their work in short bursts. During each burst, the collector follows pointers and may discover that some additional objects are accessible. The collector marks the accessible objects so they will not be deallocated at the end of the collection. After chasing some pointers and marking some objects, the cycle ends and the collector returns control to the application. A burst in which the collector runs out of pointers signals the end of the mark phase.

Each time the collector returns control to the application, the application

is free to modify marked objects. The application may store in a marked object the only pointer to an unmarked object. If the collector never again examines the marked object, the pointer won't be discovered: the unmarked object remains unmarked and is incorrectly deallocated by the collector. Thus, marked objects that are subsequently modified must be reexamined.

VM protection can be used to detect this case. Any page that contains marked objects is write-protected. If a fault occurs, the page is flagged. After the mark phase has nominally finished, all the flagged pages are scanned for marked objects with pointers to unmarked objects. When any such pointer is found, the data structure reachable from the pointer is marked.

Fault interpretation can be used to remember the first N fault addresses per page. Only N addresses per page are remembered to bound the total time spent servicing faults. After the mark phase has terminated, the pages that had between 1 and N faults can be serviced very quickly because the addresses of the writes have been saved.

3.3 Consistency and Replication Control

$\mathcal{FI}$ can be used to implement arbitrary replication and consistency protocols on top of transparent distributed shared memory [LH89]. The contribution of $\mathcal{FI}$ is the ability to execute application code before and after memory pages are accessed. This code might, for example, implement a voting algorithm [Lon88]. The consistency protocol runs transparently; the client accesses the memory with normal load and store instructions.

One possible implementation is the following. Shared memory pages are replicated on all the participating sites. Upon a write, the pre-access handler of the process that is writing sends packets over the network to lock the location. When the lock is obtained, the write executes. Then, the post-access handler unlocks the location. For reads, if there are currently no locks on a page, the page does not need to be read-protected. If there is at least one lock on a page, the page is protected so that read accesses can't occur concurrently with a write access at the same location. The pre-access handler for reads checks that the location is not locked, and if it is not, allows the read to complete. If the location is locked, the handler blocks until the location is unlocked. Post-access read notification is not required.

4 Implementation

There are a number of ways that fault interpretation can be implemented. By and large, they are architecture specific and require reading the state of the CPU when the fault occurs. Thus, this technique is less portable and less general than those discussed by Appel [AL91]. Nonetheless, it has several uses and may let some programs run more efficiently.

4.1 Code Modification

When the signal handler is invoked after a fault, it determines what instruction has faulted. The instruction immediately following the faulting instruction is overwritten with an unconditional branch to the block of handler code called the *reprotect* block.[2] Then, the signal handler unprotects the page and returns, allowing the operating system to resume the application.

When it resumes, the application re-executes the instruction that caused the fault. Since the page is now unprotected, this succeeds. Then, control follows the branch to the reprotect block. This block performs post-access notification, reprotects the page, restores the instruction sequence that was modified, and branches back to the application.

4.2 Register Modification

The SPARC architecture permits a much simpler implementation that does not require code modification. The SPARC has a register called `npc` for *next program counter*. This register contains the address of the instruction that will execute after the current instruction completes. This register is used to implement delayed branches. The `npc` register makes it particularly easy to implement $\mathcal{FI}$.

Upon a fault, the signal handler can read and modify the CPU state at the faulting instruction. This state includes the contents of `npc`. The previous value of this register is saved, and the address of the reprotect block is assigned to the register. Then, the signal handler unprotects the page and returns. The application again executes the instruction that faulted; this time the access succeeds. Since `npc` points to handler code, control jumps to the reprotect block. As before, the application is notified, the page is restored to its former protection state, and control branches back to the

[2] On delayed branch architectures, a `nop` is written after the branch.

application. This is the implementation used in the current version of the $\mathcal{FI}$ library.

4.3 Instruction interpretation

Another way of executing a single instruction is to parse and interpret the instruction. On a RISC processor this is not very difficult or inefficient, provided the operating system makes the entire context of the faulting instruction available. This also requires being able to restart the instruction following the faulted instruction. One advantage of this is the interpreter can take advantage of extra information. For example, if the fault page is also mapped without protection elsewhere in the address space [AL91, Wil92], the interpreter can use that version to avoid needing to unprotect and reprotect the page.

4.4 Parallelization

The $\mathcal{FI}$ code is currently sequential. However, the majority of it could be parallelized. There are two main issues that must be resolved. The first is the use of global data. Two parts of the $\mathcal{FI}$ library communicate through global variables. In a parallel implementation, this data would have to be replicated on a per-thread basis.

The second issue is the following: If any thread is executing when a page is unprotected, the thread can access the page without being monitored. Thus, whenever $\mathcal{FI}$ unprotects a page, it must first stop all the threads in the system. They remain stopped until the page's protection is restored.

5 Efficiency

The key operations in terms of efficiency are changing the protection state of a page and handling a fault. The times for these two operations are presented in Table 1. The timing information was obtained with the SunOS version 4.1.1 *getrusage* system call. The tests were performed on a Sun IPX with a cycle time of 25 ns (40Mhz). The cycles-per-operation figures are obtained by dividing the time per operation by the cycle time.

The time to protect a page was obtained by making the *mprotect* system call in a loop. The time this call requires to execute depends on whether the page in question is accessed or not, and whether it is clean or dirty. Therefore, this test was repeated for unaccessed pages, pages that had been

read from, and pages that had been written to. In each case, the page was entirely initialized to zeros before beginning the test. The data for each class of page are presented. This was repeated several times with the total time and the total number of iterations summed and averaged.

The time for handling a fault was obtained by writing a fault handler that leaves the page protected $N-1$ times so that restarting the instruction causes another fault. Then, on the N^{th} iteration, the handler unprotects the page and the instruction completes successfully.

The time for *protect+fault+unprotect* was obtained by protecting a page, faulting, and unprotecting the page, all in a loop. This is a test whose efficiency is also measured in [AL91] and is repeated here to provide a baseline for comparison.

The time for *fault interpret* is the time to interpret a fault, i.e. to access a protected page and have the application's notify function informed that the access has occurred, while finishing with the page still protected. This consists of *fault+unprotect+protect+small overhead*. The application's notify function for this test returns immediately.

Lastly, we present the time for handling a page fault. This data was obtained obtained by allocating more virtual memory than the machine has physical memory and repeatedly sequentially touching every page. This was done once with pages being read and once with pages being read and written. In both cases, page accesses are sequential. This information is provided to offer a comparison between the efficiency of handling protection faults and page faults.

The data show that this implementation of fault interpretation is about 5% more expensive than standard fault handling (for substantially greater functionality). Nonetheless, protection faults are very expensive, costing approximately 20,000 cycles each. This cost in terms of memory references is much different, of course, probably closer to 8000 memory references. Therefore, if taking a fault can save more than 8000 memory references, there will be an increase in efficiency.

What is really clear is how expensive page faults are. If we can save a single page fault, then we can interpret over 30 protection faults and still see an increase in efficiency (based on the relative costs of a page fault and fault interpretation). A generational collector that stores generation counters in objects, or an incremental mark-and-sweep collector that stores mark bits with objects, could significantly improve efficiency with fault interpretation. For transparent persistent memory, the fault time is inconsequential compared to the time to write the data to disk. Similarly, assuming that the

Table 1: Efficiency of the component operations

Operation	Count	Total Time	Time per Operation	Cycles per Operation
mprotect, unaccessed	80,000	4.0s	50μs	2000
mprotect RO-RW, clean	80,000	14.4s	179μs	7160
mprotect RW-RW, clean	80,000	22.1s	275μs	11000
mprotect RW-RW, dirty	80,000	21.2s	265μs	10600
handle a fault	500,000	81.7s	163μs	6520
protect+fault+unprotect	500,000	258.5s	517μs	20680
fault interpret	500,000	270.0s	540μs	21600
page fault, reading	20,480	480.0s	23,437μs	937,480
page fault, writing	20,480	757.0s	36,963μs	1,478,520

The measurements were taken on a 40Mhz Sun IPX. *Unaccessed* means the page has neither been read nor written. *Clean* means the page has been read since the last call to mprotect. *Dirty* means the page has been written since the last call to mprotect. *RW-RW* means successive calls to mprotect always grant full access to the page. *RO-RW* means successive calls to mprotect alternate between restricting access and restoring access.

time for a network message for a relatively fast protocol such as UDP is on the order of 1.5ms [Mak89], fault handling should not be the bottleneck in implementing distributed shared memory.

Lastly, we observe that disk and network latencies do not scale with processor speeds, whereas fault handling latency does increase with faster CPUs, subject to memory access time. Thus, relative to disk and network I/O, the efficiency of fault interpretation will improve with faster CPUs. It will also improve if operating system implementors provide faster trap handling.

6 Availability

The $\mathcal{FI}$ library has been implemented for the SPARC processor. The code will compile either as an ANSI/ISO C program or as a C++ program. The source code is available via anonymous ftp from ftp.cse.ucsc.edu (128.114.134.19) in pub/csl/vm-trace.tar.Z. It can also be obtained from ftp.inria.fr (128.93.1.26) in INRIA/c++-gc/vm-fault.tar.Z. The code is not public domain, but may be used without fee for any purpose, commercial or otherwise.

All of the test programs that were used for our efficiency measurements are available with the library. The names of the files (and their purposes) are as follows:

File	Purpose
t0.c	Measure the efficiency of mprotect
t1.c	Sample $\mathcal{FI}$ application, obtain and exercise managed memory
t2.c	Measure the efficiency of trapping the signal upon a memory protection fault
t3.c	Measure the time required to protect a page, fault on it, and then unprotect it
t4.c	Measure the time required to interpret a fault
t5.c	Measure the time required to handle a page fault when reading sequential pages
t6.c	Measure the time required to handle a page fault when writing sequential pages
t7.c	Measure the efficiency of mprotect (more detail than t0.c)

7 Conclusion

We present a library that provides more functionality than is usually obtained from standard virtual memory hardware and operating system software. Given sufficiently fast trap handling, this technique can be used to improve the efficiency of incremental or generational garbage collectors. It may also be useful for persistent object stores, coherent distributed shared virtual memory, and other algorithms.

Acknowledgements

This work was supported in part by Esprit project 5279 *Harness*.

Biography

The author is a Ph.D. student in Computer and Information Science at the University of California, Santa Cruz. He plans to graduate in 1993. From 1991 to 1992, he spent 12 months as a visiting researcher at INRIA Rocquencourt, where this work was done. He also programs C++ on contracts and teaches C++ classes in industry. Formerly, he was an engineer in the operating systems group at the Santa Cruz Operation.

References

[AEL88] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. In *Proc. PLDI '88*, pages 11–20, July 1988. SIGPLAN Not. 23(7).

[AL91] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In *ASPLOS Inter. Conf. Architectural Support for Programming Languages and Operating Systems*, pages 96–107, Santa Clara, CA, April 1991. SIGPLAN Not. 26(4).

[ANS89] ANSI X3.159-1989, 1989. American national standard for the C programming language.

[Bak78] H. G. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[BDS91] Hans-J. Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. In *Proc. PLDI '91*, pages 157–164. ACM, June 1991. SIGPLAN Not. 26(6).

[Cyp90] Cypress Semiconductor. SPARC risc users guide, 1990.

[DWH+90] Alan Demers, Mark Weiser, Barry Hayes, Hans Boehm, Daniel Bobrow, and Scott Shenker. Combining generational and conservative garbage collection: Framework and implementations. In *Proc. POPL '90*, pages 261–269. ACM, January 1990.

[ISO90] ISO 9899-1990, 1990. International standard for the C programming language.

[LH83] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[LH89] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[Lon88] Darrell D. E. Long. *The Management of Replication in a Distributed System.* Ph.D. dissertation, University of California at San Diego, August 1988.

[Mak89] Mesaac Mounchili Makpangou. *Protocoles de communication et programmation par objets: l'exemple de SOS.* PhD thesis, Université Paris VI, Paris (France), February 1989.

[Moo84] David Moon. Garbage collection in a large LISP system. In *Symp. Lisp and Functional Programming*, pages 235–246. ACM, 1984.

[Sun87] Sun Microsystems, Inc. The SPARC architecture manual, 1987. Part No. 800–11399–07.

[Ung84] David Ungar. Generation Scavenging: A non–disruptive high performance storage reclamation algorithm. In *ACM SIGPLAN/SIGSOFT Symp. Practical Software Development Environments*, pages 157–167. ACM, April 1984. SIGPLAN Not. 19(2).

[Wil92] Paul Wilson, 1992. Personal communication.