Virtual Memory Primitives for User Programs

Andrew W. Appel and Kai Li

Department of Computer Science
Princeton University

Abstract

Memory Management Units (MMUs) are traditionally used by operating systems to implement disk-paged virtual memory. Some operating systems allow user programs to specify the protection level (inaccessible, read-only, read-write) of pages, and allow user programs to handle protection violations, but these mechanisms are not always robust, efficient, or well-matched to the needs of applications.

We survey several user-level algorithms that make use of page-protection techniques, and analyze their common characteristics, in an attempt to answer the question, "What virtual-memory primitives should the operating system provide to user processes, and how well do today's operating systems provide them?"

1 Introduction

The "traditional" purpose of virtual memory is to increase the size of the address space visible to user programs, by allowing only the frequently-accessed subset of the address space to be resident in physical memory. But virtual memory has been used for many other purposes. Operating systems can share pages between processes, make instruction-spaces read-only (and thus guaranteed re-entrant), make portions of memory zeroed-on-demand or copy-on-write, and so on [18]. In fact, there is a large class of "tricks" that operating systems can perform using the page protection hardware.

Modern operating systems allow user programs to perform such tricks too, by allowing user programs to provide "handlers" for protection violations. Unix, for example, allows a user process to specify that a particular subroutine is to be executed whenever a segmentation-fault signal is generated. When a program accesses memory beyond its legal virtual address range, a user-friendly error message can be produced by the user-provided signal handler, instead of the ominous "segmentation fault: core dumped."

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1991 ACM 0-89791-380-9/91/0003-0096...$1.50

This simple example of a user-mode fault handler is "dangerous," because it may lead the operating-system and hardware designers to believe that user-mode fault-handlers need not be entered efficiently (which is certainly the case for the "graceful error shutdown" example). But there are much more interesting applications of user-mode fault handlers. These applications exercise the page-protection and fault-handling mechanisms quite strenuously, and should be understood by operating-system implementors.

This paper describes several algorithms that make use of page-protection techniques. In many cases, the algorithms can substitute the use of "conventional" paging hardware for the "special" microcode that has sometimes been used. On shared-memory multiprocessors, the algorithms use page-protection hardware to achieve medium-grained synchronization with low overhead, in order to avoid synchronization instruction sequences that have noticable overhead.

We have benchmarked a number of systems to analyze how well today's operating systems support user-level page-protection techniques. Finally, from these algorithms we draw lessons about page-protection costs, the utility of memory-mapping mechanisms, translation-buffer shootdowns, page sizes and other aspects of operating system implementation.

2 Virtual memory primitives

Each of the algorithms we will describe require some of the following virtual-memory services from the operating system:

TRAP:	handle page-fault traps in user mode;
PROT1:	decrease the accessibility of a page;
PROTN:	decrease the accessibility of N pages;
UNPROT:	increase the accessibility of a page;
DIRTY:	return a list of dirtied pages since the previous call.
MAP2:	map the same physical page at two different virtual addresses, at different levels of protection, in the same address space.

Finally, some algorithms may be more efficient with a smaller PAGESIZE than is normally used with disk paging.

We distinguish between "decreasing the accessibility of a page" and "decreasing the accessibility of a batch of pages" for a specific reason. The cost of changing the protection of several pages simultaneously may be not much more than the cost of changing the protection of one page. Several of the algorithms we describe protect pages (make them less accessible) only in large batches. Thus, if an operating system implementation could not efficiently decrease the accessibility of one page, but could decrease the accessibility of a large batch at a small cost-per-page, this would suffice for some algorithms.

We do not make such a distinction for unprotecting single *vs.* multiple pages because none of the algorithms we describe ever unprotect many pages simultaneously.

Some multi-thread algorithms require that one thread have access to a particular page of memory while others fault on the page. There are many solutions to such a problem (as will be described later), but one simple and efficient solution is to map the page into more than one virtual address: at one address the page is accessible and at the other address it faults. For efficiency reasons, the two different virtual addresses should be in the same page table, so that expensive page-table context switching is not required between threads.

The user program can keep track of dirty pages using PROTN, TRAP, and UNPROT; we list DIRTY as a separate primitive because it may be more efficient for the operating system to provide this service directly.

3 Virtual memory applications

We present in this section a sample of applications which use virtual-memory primitives in place of software tests, special hardware, or microcode. The page protection hardware can efficiently test simple predicates on addresses that might otherwise require one or two extra instructions on every fetch and/or store; this is a substantial savings, since fetches and stores are very common operations indeed. We survey several algorithms so that we may attempt to draw general conclusions about what user programs require from the operating system and hardware.

Concurrent garbage collection

A concurrent, real-time, copying garbage collection algorithm can use the page fault mechanism to achieve medium-grain synchronization between collector and mutator threads [4]. The paging mechanism provides synchronization that is coarse enough to be efficient and yet fine enough to make the latency low. The algorithm is based on the Baker's sequential, real-time copying collector algorithm [6].

Baker's algorithm divides the memory heap into two regions, *from-space* and *to-space*. At the beginning of a collection, all objects are in from-space, and to-space is empty. Starting with the registers and other global roots, the collector traces out the graph of objects reachable from the roots, copying each reachable object into to-space. A pointer to an object from-space is *forwarded* by making it point to the to-space copy of the old object. Of course, some from-space objects are never copied into to-space, because no pointer to them is ever forwarded; these objects are garbage.

As soon as the registers are forwarded, the mutator thread can resume execution. Reachable objects are copied incrementally from from-space while the mutator allocates new objects at **new**. Every time the mutator allocates a new object, it invokes the collector to copy a few more objects from from-space. Baker's algorithm maintains the following invariants:

- The mutator sees only to-space pointers in its registers.
- Objects in the new area contain to-space pointers only (because new objects are initialized from the registers).
- Objects in the scanned area contain to-space pointers only.
- Objects in the unscanned area contain both from-space and to-space pointers.

To satisfy the invariant that the mutator sees only to-space pointers in its registers, every pointer fetched from an object must be checked to see if it points to from-space. If it does, the from-space object is copied to to-space and the pointer updated; only then is the pointer returned to the mutator. This checking requires hardware support to be implemented efficiently [25], since otherwise a few extra instructions must be performed on every fetch. Furthermore, the mutator and the collector must alternate; they cannot operate truly concurrently because they might simultaneously try to copy the same object to different places.

Instead of checking every pointer fetched from memory, the concurrent collector [4] uses virtual-memory page protections to detect from-space memory references and to synchronize the collector and mutator threads. To synchronize mutators and collectors, the algorithm sets the virtual-memory protection of the unscanned area's pages to be "no access." Whenever the mutator tries to access an unscanned object, it will get a page-access trap. The collector fields the trap and scans the objects on that page, copying from-space objects and forwarding pointers as necessary. Then it unprotects the page and resumes the mutator at the faulting instruction. To the mutator, that page appears to have contained only to-space pointers all along, and thus the mutator will fetch only to-space pointers to its registers.

The collector also executes concurrently with the mutator, scanning pages in the unscanned area and unprotecting them as each is scanned. The more pages

scanned concurrently, the fewer page-access traps taken by the mutator. Because the mutator doesn't do anything extra to synchronize with the collector, compilers needn't be reworked. Multiple processors and mutator threads are accommodated with almost no extra effort.

This algorithm requires TRAP, PROTN, UNPROT, and MAP2. Traps are required to detect fetches from the unscanned area; protection of multiple pages is required to mark the entire to-space inaccessible when the flip is done; UNPROT is required as each page is scanned. In addition, since the time for the user-mode handler to process the page is proportional to page size, it may be appropriate to use a small PAGESIZE to reduce latency.

We need multiple-mapping of the same page so that the garbage collector can scan a page while it is still inaccessible to the mutators. Alternatives to multiple-mapping are discussed in section 5.

Shared virtual memory

The access protection paging mechanism has been used to implement shared virtual memory on a network of computers, on a multicomputer without shared memories [21], and on a multiprocessor based on interconnection networks [14]. The essential idea of shared virtual memory is to use the paging mechanism to control and maintain single-writer and multiple-reader coherence at the page level.

Figure 1 shows the system architecture of an SVM system. On a multicomputer, each node in the system consists of a processor and its memory. The nodes are connected by a fast message-passing network.

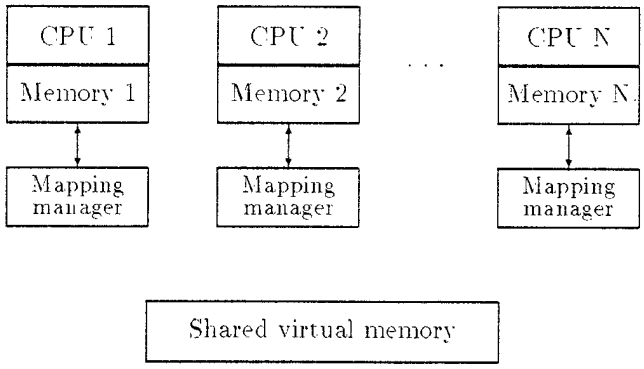

Figure 1: Shared virtual memory

The SVM system presents all processors with a large coherent shared memory address space. Any processor can access any memory location at any time. The shared memory address space can be as large as the memory address space provided by the MMU of the processor. The address space is coherent at all times, that is, the value returned by a read operation is always the same as the value written by the most recent write operation to the same address.

The SVM address space is partitioned into pages. Pages that are marked "read-only" can have copies residing in the physical memories of many processors at the same time. But a page currently being written can reside in only one processor's physical memory. If a processor wants to write a page that is currently residing on other processors, it must get an up-to-date copy of the page and then tell the other processors to invalidate their copies. The memory mapping manager views its local memory as a big cache of the SVM address space for its associated processors. Like the traditional virtual memory [15], the shared memory itself exists only *virtually*. A memory reference may cause a page fault when the page containing the memory location is not in a processor's current physical memory. When this happens, the memory mapping manager retrieves the page from either disk or the memory of another processor.

This algorithm uses TRAP, PROT1, and UNPROT; the trap-handler needs access to memory that is still protected from the client threads (MAP2), and a small PAGESIZE may be appropriate.

Concurrent checkpointing

The access protection page fault mechanism has been used successfully in making checkpointing concurrent and real-time [22]. This algorithm for shared-memory multiprocessors runs concurrently with the target program, interrupts the target program for small, fixed amounts of time and is transparent to the checkpointed program and its compiler. The algorithm achieves its efficiency by using the paging mechanism to allow the most time-consuming operations of the checkpoint to be overlapped with the running of the program being checkpointed.

First, all threads in the program being checkpointed are stopped. Next, the writable main memory space for the program is saved (including the heap, globals, and the stacks for the individual threads.) Also, enough state information is saved for each thread so that it can be restarted. Finally, the threads are restarted.

Instead of saving the writable main memory space to disk all at once, the algorithm avoids this long wait by using the access protection page fault mechanism. First, the accessibility of entire address space is set to "read only." At this point, the threads of the checkpointed program are restarted and a copying thread sequentially scans the address space, copying the pages to a separate virtual address space as it goes. When the copying thread finishes copying a page, it sets its access rights to "read/write."

When the user threads can make read memory references to the read-only pages, they run as fast as with no checkpointing. If a thread of the program writes a

page before it has been copied, a write memory access fault will occur. At this point the copying thread immediately copies the page and sets the access for the page to "read/write," and restarts the faulting thread.

Several benchmark programs have been used to measure the performance of this algorithm on the DEC Firefly multiprocessors [33]. The measurements show that about 90% of the checkpoint work is executed concurrently with the target program while no thread is ever interrupted for more than .1 second at a time.

This method also applies to taking incremental checkpoints; saving the pages that have been changed since the last checkpoint. Instead of protecting all the pages with "read-only," the algorithm can protect only "dirtied" pages since the previous checkpoint. Feldman and Brown [17] implemented and measured a sequential version for a debugging system by using reversible executions. They proposed and implemented the system call DIRTY.

This algorithm uses TRAP, PROT1, PROTN, UNPROT, and DIRTY ; a medium PAGESIZE may be appropriate.

Generational garbage collection

An important application of memory protection is in generational garbage collection[23], a very efficient algorithm that depends on two properties of dynamically allocated records in LISP and other programming languages:

1. Younger records are much more likely to die soon than older records. If a record has already survived for a long time, it's likely to survive well into the future: a new record is likely to be part of a temporary, intermediate value of a calculation.
2. Younger records tend to point to older records, since in LISP and functional programming languages the act of allocating a record also initializes it to point to already-existing records.

Property 1 indicates that much of the garbage collector's effort should be concentrated on younger records, and property 2 provides a way to achieve this. Allocated records will be kept in several distinct areas G_i of memory, called *generations*. Records in the same generation are of similar age, and all the records in generation G_i are older than the records in generation G_{i+1}. By observation 2 above, for $i < j$, there should be very few or no pointers from G_i into G_j. The collector will usually collect in the youngest generation, which has the highest proportion of garbage. To perform a collection in a generation, the collector needs to know about all pointers into the generation; these pointers can be in machine registers, in global variables, and on the stack. However, there very few such pointers in older generations because of property 2 above.

The only way that an older generation can point to a younger one is by an assignment to an already-existing record. To detect such assignments, each modification of a heap object must be examined to see whether it violates property 2. This checking can be done by special hardware [25,35], or by compilers [34]. In the latter case, two or more instructions are required. Fortunately, non-initializing assignments are rare in Lisp, Smalltalk, and similar languages [25,35,30,3] but the overhead of the instruction sequence for checking (without special hardware) is still on the order of 5–10% of total execution time.

Virtual memory hardware can detect assignments to old objects. If DIRTY is available, the collector can examine dirtied pages to derive pointers from older generations to younger generations and process them. In the absence of such a service, the collector can use the page protection mechanism [30]: the older generations can be write-protected so that any store into them will cause a trap. The user trap-handler can save the address of the trapping page on a list for the garbage collector; then the page must be unprotected to allow the store instruction to proceed. At garbage-collection time the collector will need to scan the pages on the trap-list for possible pointers into the youngest generation. Variants of this algorithm have exhibited quite good performance [30,11]: as heaps and memories get larger the this scheme begins to dominate other techniques[37].

This technique uses the TRAP, PROTN, and UNPROT features, or just DIRTY. In addition, since the time for the user-mode handler to process the page is independent of page size, and the eventual time for the garbage collector to scan the page is proportional to the page size, it may be appropriate to use a small PAGESIZE.

Persistent stores

A *persistent store* [5] is a dynamic allocation heap that persists from one program-invocation to the next. An execution of a program may traverse data structures in the persistent store just as it would in its own (in-core) heap. It may modify objects in the persistent store, even to make them point to newly-allocated objects of its own; it may then *commit* these modifications to the persistent store, or it may *abort*, in which case there is no net effect on the persistent store. Between (and during) executions, the persistent store is kept on a stable storage device such as a disk so that the "database" does not disappear.

It is important that traversals of pointers in the persistent store be just as fast as fetches and stores in main memory; ideally, data structures in the persistent store should not be distinguishable by the compiled code of a program from data structures in core. This can be accomplished through the use of virtual memory: the persistent store is a memory-mapped disk file; pointer traversal through the persistent store is just the same as pointer traversal in core, with page faults if new parts

of the store are examined.

However, when an object in the persistent store is modified, the permanent image must not be altered until the *commit*. The in-core image is modified, and only at the *commit* are the "dirty" pages (possibly including some newly-created pages) written back to disk. To reduce the number of new pages, it is appropriate to do a garbage collection at commit time.

A *database* is a storage management system that may provide, among other things, locking of objects, transactions with abort/commit, checkpointing and recovery. The integration of virtual memory techniques into database implementations has long been studied [24,31].

Compiled programs can traverse the data in their heaps very quickly and easily, since each access operation is just a compiled *fetch* instruction. Traversal of data in a conventional database is much slower, since each operation is done by procedure call; the access procedures ensure synchronization and abortability of transactions. Persistent stores can be augmented to cleanly handle concurrency and locking; such systems (sometimes called *object-oriented databases*) can be quickly traversed with *fetch* instructions but also can provide synchronization and locking; efficiency of access can be improved by using a garbage collector to group related objects on the same page, treat small objects differently than large objects, and so on[13].

These schemes requires the use of TRAP and UNPROT as well as file-mapping with copy-on-write (which, if not otherwise available, can be simulated using PROTN, UNPROT, and MAP2.

Extending addressability

A persistent store might grow so large that it contains more than (for example) 2^{32} objects, so that it cannot be addressed by 32-bit pointers. Modern disk drives (especially optical disks) can certainly hold such large databases, but conventional processors use 32-bit addresses. However, *in any one run of a program* against the persistent store, it is likely that fewer than 2^{32} objects will be accessed.

One solution to this problem is to modify the persistent store mechanism so that objects in core use 32-bit addresses and objects on disk use 64-bit addresses. Each disk page is exactly twice as long as a core page. When a page is brought from disk to core, its 64-bit disk pointers are translated to 32-bit core pointers using a translation table. When one of these 32-bit core pointers is dereferenced for the first time, a page fault may occur; the fault handler brings in another page from disk, translating it to short pointers.

The translation table has entries only for those objects accessed in a single execution: that is why 32-bit pointers will suffice. Pointers in core may point to not-yet-accessed pages; such a page is not allocated in core, but there is an entry in the translation table showing what (64-bit pointer) disk page holds its untranslated contents.

The idea of having short pointers in core and long pointers on disk, with a translation table for only that subset of objects used in one session, originated in the LOOM system of Smalltalk-80 [20]. The use of a page-fault mechanism to implement it is more recent [19]. This algorithm uses TRAP, UNPROT, PROT1 or PROTN, and (in a multi-threaded environment) MAP2, and might work well with a smaller PAGESIZE.

Data-compression paging

In a typical linked data structure, many words point to nearby objects; many words are nil. Those words that contain integers instead of pointers often contain small integers or zero. In short, the information-theoretic entropy of the average word is small; furthermore, a garbage collector can be made to put objects that point to each other in nearby locations, thus reducing the entropy per word to as little as 7 bits[9].

By the use of a data-compression algorithm, then, a page of 32-bit words might be compressible to about one-quarter of a page. Instead of paging less-recently-used pages directly to the disk, they could be compressed instead and put back into main memory.[36] Then, when those virtual pages are again needed, it might take much less time to uncompress them than it would to fetch from the disk. Of course, compressed pages could be sent out to disk after a long period of disuse.

Of course, data-compression paging might be done inside the operating system transparently to the user process[28]. But since a garbage collector can move objects to minimize their entropy, much better results might be obtained if the user process can have some control over how and when compression is done.

This algorithm requires TRAP, PROT1 (or perhaps PROTN with careful buffering), TRAP, and UNPROT. It is necessary to determine when pages are not recently used; this can be done by occasionally protecting pages to see if they are referenced, or with help from the operating system and hardware.

Heap overflow detection

The stack of a process or thread requires protections against overflow accesses. A well-known and practical technique used in most systems is to mark the pages above the top of the stack invalid or no-access. Any memory access to these pages will cause a page fault. The operating system can catch such a fault and inform the user program of a stack overflow. In most implementations of Unix, stack pages are not allocated until first used; the operating-system's response to a page

fault is to allocate physical pages, mark them accessible, and resume execution without notifying the user process (unless a resource limit is exceeded).

This technique requires TRAP, PROTN and UNPROT. But since the faults are quite rare (most processes don't use much stack space), efficiency is not a concern.

The same technique can be used to detect heap overflow in a garbage-collected system[2]. Ordinarily, heap overflow in such a system is detect by a compare and conditional-branch performed on each memory allocation. By having the user process allocate new records in a region of memory terminated by a guard page, the compare and conditional-branch can be eliminated. When the end of the allocatable memory is reached, a page-fault trap invokes the garbage collector. It can often be arranged that no re-arrangement of memory protection is required, since after the collection the same allocation area can be re-used. Thus, this technique requires PROT1 and TRAP.

Here, efficiency of TRAP is a concern. Some language implementations allocate a new cell as frequently as every 50 instructions. In a generational garbage collector, the size of the allocation region may be quite small in order to make the youngest generation fit entirely in the data cache; a 64 Kbyte allocation region could hold 16k 8-byte list cells, for example. In a very-frequently-allocating system (e.g. one that keeps activation records on the heap), such a tiny proportion of the data will be live that the garbage-collection time itself will be small. Thus, we have:

Instructions executed before heap overflow:

$$(64k/8) \times 50 = 400k.$$

Instructions of overhead, using compare and branch:

$$(64k/8) \times 2 = 16k.$$

If a trap takes 1200 cycles to handle (as is typical—see section 4), then this technique reduces the overhead from 4% to 0.3%, a worthwhile savings. If a trap were to take much longer, this technique would not be as efficient.

Since there are other good techniques for reducing heap-limit-check overhead, such as combining the limit checks for several consecutive allocations in an unrolled loop, this application of virtual memory is perhaps the least interesting of those discussed in this paper.

4 VM primitive performance

Almost all the algorithms we described in this paper fall into one of the two categories. The first category of algorithms protect pages in large batches, then upon each page-fault trap they unprotect one page. The second category of algorithms protect a page and unprotect a page individually. Since PROTN or PROT, TRAP, and UNPROT are always used together, an operating system in which one of the operations is extremely efficient, but others are very slow will not be very competive.

We performed two measurements for overall user-mode virtual-memory performance. The first is the sum of PROT1, TRAP, and UNPROT, as measured by 100 repititions of the following benchmark program:

- access a random protected page, and
- in the fault-handler, protect some other page and unprotect the faulting page.

This process is repeated 100 times to obtain more accurate timing.

The second measurement is the sum of PROTN, TRAP, and UNPROT. The benchmark program measures:

- protect 100 pages,
- access each page in a random sequence, and
- in the fault-handler, unprotect the faulting page.

Before beginning the timing, both programs write each page to eliminate transient effects of filling the cache and TLB.

We compared the performance of Ultrix, SunOS, and Mach on several platforms in the execution of these benchmarks. For calibration, we also show the time for a single instruction (ADD), measured using a 20-instruction loop containing 18 adds, a compare, and a branch. Where we have the data, we also show the time for a trap-handler that does not change any memory protections; this would be useful for *heap-overflow detection*. The results are shown in Table 1. Note that this benchmark is not an "overall operating system throughput" benchmark [27] and should not be influenced by disk speeds; it is measuring the performance of CPU-handled virtual memory services for user-level programs.

We also tried mapping a physical page at two different virtual addresses in the same process, using the shared memory operations (**shmop**) on SunOS and Ultrix, and on Mach using **vm_map**. SunOS and Mach permit this, but Ultrix would not permit us to attach (**shmat**) the same shared-memory object at two different addresses in the same process.

Clearly, there are wide variations between the performance of these operating systems even on the same hardware. This indicates that there may be considerable room for improvement in some or all of these systems. Furthermore, several versions of operating systems do not correctly flush their translation buffer after an **mprotect** call, indicating that many operating systems implementors don't take this feature seriously.

It is important that these operating system services be made efficient. The argument here is much more specific than a vacuous "Efficiency is good." For disk-paging, a page fault usually implies a 20-millisecond wait for the disk to spin around to the right sector;

Machine	OS	ADD	TRAP	TRAP +PROT1 +UNPROT	TRAP +PROTN +UNPROT	MAP2	PAGESIZE
Sun 3/60	SunOS 4.0	0.12	760	1238	1016	yes	8192
Sun 3/60	SunOS 4.1	0.12		2080	1800	yes	8192
Sun 3/60	Mach 2.5(xp)	0.12		3300	2540	yes	8192
Sun 3/60	Mach 2.5(exc)	0.12		3380	2880	yes	8192
SparcStn 1	SunOS 4.0.3c	0.05		*919	*839	yes	4096
SparcStn 1	SunOS 4.1	0.05	†230	1008	909	yes	4096
SparcStn 1	Mach 2.5(xp)	0.05		1550	1230	yes	4096
SparcStn 1	Mach 2.5(exc)	0.05		1770	1470	yes	4096
DEC 3100	Ultrix 4.1	0.062	210	393	344	no	4096
DEC 3100	Mach 2.5 (xp)	0.062		937	766	no	4096
DEC 3100	Mach 2.5 (exc)	0.062		1203	1063	no	4096
μVax 3	Ultrix 2.3	0.21	314	612	486	no	1024
i386 on iPSC/2	NX/2	0.15	172	302	252	yes	4096

Table 1: Benchmark data.

Elapsed time in microseconds is given for each operation. For Mach we measured both the exception-port mechanism ("exc") and the external-pager interface ("xp"). Time for TRAP is the operating system overhead, plus the user-mode part of trap-handler as provided in standard library. For the PROT1 and PROTN benchmarks, the we show the time *per page*. MAP2 is whether the system supports mapping the same page at different addresses; see section 4. PAGESIZE is as reported by the operating system.

* The implementation of **mprotect** (which changes memory protection) on SunOS 4.0 incorrectly fails to flush the TLB.

† estimated.

so a 3- or 5-millisecond fault handling overhead would be hardly noticed as a contributor to fault-handling latency. But in the algorithms surveyed in the paper, the fault will be handled entirely within the CPU. For example, we have implemented a garbage collector that executes about 10 instructions per word of to-space. For a pagesize of 4096 bytes (1024 words) on a 20 MIPS machine, the computation time to handle a fault will be approximately $10 * 0.05 * 1024$ or about 500 microseconds. If the operating system's fault-handling and page-protection overhead is 1200 microseconds (as is average), then the operating system is clearly the bottleneck.

If the program exhibits good locality of reference, then the garbage-collection faults will be few, and the operating system overhead will matter less. But for real-time programs, which must satisfy strict constraints on latency, even an occasional "slow fault" will cause problems. For example, if the client program must never be interrupted for more than a millisecond, then a fault-handler computation time of 500 microseconds doesn't leave room for an operating-system overhead of 1200 microseconds! (This issue gets more complicated when we consider multiple consecutive faults; see [11] for an analysis.)

In order to compare virtual memory primitives on different architectures, we have normalized the measurements by processor speed. Figure 4 shows the number

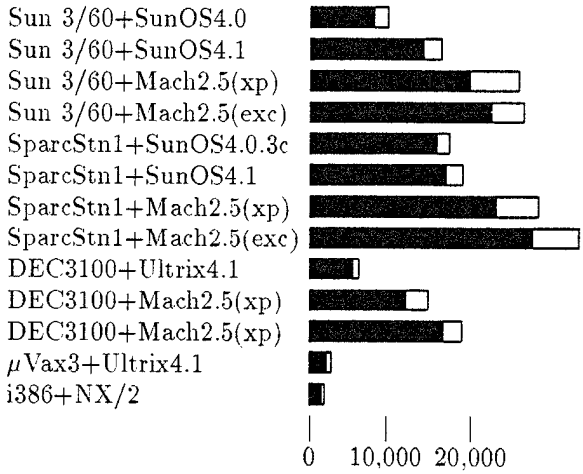

Figure 2: Instructions per PROT + TRAP + UNPROT.

The black bars show the results when pages are protected in large batches (PROTN), and the white bars are the additional time taken when pages are protected one at a time (PROT1).

of ADDs each processor could have done in the time it takes to protect a page, fault, and unprotect a page.

Our benchmark shows that there is a wide range of efficiency in implementing virtual memory primitives. Intel 80386-based machine running NX/2 operating system [29] (a simple operating system for the iPSC/2 hypercube multicomputer) is the best in our benchmark. Its normalized benchmark performance is about ten times better than the worst performer (Mach on the Sparcstation). Clearly, there is no inherent reason that these primitives must be slow. Hardware and operating system designers should treat memory-protection performance as one of the important tradeoffs in the design process.

5 System design issues

We can learn some important lessons about hardware and operating system design from our survey of virtual-memory applications. Most of the applications use virtual memory in similar ways; this makes it clear what VM support is needed—and just as important, what is unnecessary.

TLB Consistency

Many of the algorithms presented here make their memory less-accessible in large batches, and make memory more-accessible one page at a time. This is true of *concurrent garbage collection*, *generational garbage collection*, *concurrent checkpointing*, *persistent store*, and *extending addressability*.

This is a good thing, especially on a multiprocessor, because of the translation lookaside buffer (TLB) consistency problem. When a page is made more-accessible, outdated information in TLBs is harmless, leading to at most a spurious, easily patchable TLB miss or TLB fault. [1] But when a page is made less-accessible, outdated information in TLBs can lead to illegal accesses to the page. To prevent this, it is necessary to flush the page from each TLB where it might reside. This "shootdown" can be done in software by interrupting each of the other processors and requesting it to flush the page from its TLB, or in hardware by various bus-based schemes[7,32].

Software shootdown can be very expensive if there are many processors to interrupt. Our solution to the shootdown problem is to batch the shootdowns; the cost of a (software) shootdown covering many pages simultaneously is not much greater than the cost of a single-page shootdown: the cost per page becomes neglible when the overhead (of interrupting the processers to notify them about shootdowns) is amortized over many pages. The algorithms described in this paper that protect pages in batches "inadvertantly" take advantage of batched shootdown.

Batching suggested itself to us because the of the structure of the algorithms described here, but it can also solve the shootdown problem for "traditional" disk paging. Pages are made less-accessible in disk paging (they are "paged out") in order to free physical pages for re-use by other virtual pages. If the operating system can maintain a large reserve of unused physical pages, then it can do its paging-out in batches (to replenish the reserve); this will amortize the shootdown cost over the entire batch.[2] Thus, while it has been claimed that software solutions work reasonably well but might need to be supplanted with hardware assist [7], with batching it is likely that hardware would not be necessary.

Optimal page size

In many of the algorithms described here, page faults are handled entirely in the CPU, and the fault-handling time (exclusive of overhead) is a small constant times the page size.

When a page fault occurs for paging between physical memories and disks, there is a delay of tens of milliseconds while the disk rotates and the head moves. A computational overhead of a few milliseconds in the page fault handler will hardly be noticed (especially if there are no other processes ready to execute). For this reason—and for many others, including the addressing characteristics of dynamic RAMs—pages have traditionally been quite large, and fault-handling overhead has been high.

For user-handled faults that are processed entirely by user algorithms in the CPU, however, there is no such inherent latency. To halve the time of each fault (exclusive of trap time), it suffices to halve the page size. The various algorithms described here might perform best at different page sizes.

The effect of a varying page size can be accomplished on hardware with a small page size. (In the VMP system, the translation buffer and the cache are the same thing, with a 128-byte line size [8]; this architecture might be well-suited to many of the algorithms described in this paper.) For PROT and UNPROT operations, the small pages would be used; for disk paging,

[1] On some architectures, in which a TLB entry can be present but provide no access, it will be useful for the operating system's fault handler to flush the TLB line for the faulting page. Otherwise, the user-mode fault handler might make the page accessible, but the stale TLB entry would cause a second fault. Flushing the TLB entry of the faulting page should not add significantly to fault-handling overhead. On architectures (e.g. MIPS) with software handling of TLB misses, this extra complication is not present.

[2] This algorithm must be carefully implemented to handle the case in which a page is referenced after it is put in the reserve but before it is shot down: in this case the page may be dirty in some of the TLB's and must be removed from the reserve by the shootdown procedure.

contiguous multi-page blocks would be used (as is now common on the Vax).

When small pages are used, it is particularly important to trap and change page protections quickly, since this overhead is independent of page size while the actual computation (typically) takes time proportional to page size.

Access to protected pages

Many algorithms, when run on a multiprocessor, need a way for a user-mode service routine to access a page while client threads have no access. These algorithms are *concurrent garbage collection*, *extending addressability*, *shared virtual memory*, and *data-compression paging*.

There are several ways to achieve user-mode access to protected pages (we use the *concurrent garbage collection* algorithm to illustrate):

- Multiple mapping of the same page at different addresses (and at different levels of protection) in the same address space. The garbage collector has access to pages in to-space at a "nonstandard" address, while the mutators see to-space as protected.
- A system call could be provided to copy memory to and from a protected area. The collector would use this call three times for each page: once when copying records from from-space to to-space; once prior to scanning the page of to-space: and once just after scanning, before making the page accessible to the mutators. This solution is less desirable because it's not very efficient to do all that copying.
- In an operating system that permits shared pages between processes, the collector can run in a different heavyweight process from the mutator, with a different page table. The problem with this technique is that it requires two expensive heavyweight context switches on each garbage-collection page-trap. However, on a multiprocessor it may suffice to do an RPC to another processor that's already in the right context, and this option might be much more attractive.
- The garbage collector can run inside the operating-system kernel. This is probably most efficient, but perhaps that's not the appropriate place for a garbage collector; it can lead to unreliable kernels, and every programming language has a different runtime data format that the garbage collector must understand.

We advocate that for computer architectures with physically addressed caches, the multiple virtual address mapping in the same address space is a clean and efficient solution. It does not require heavyweight context switches, data structure copies, nor running things in the kernel. There is the small disadvantage that each physical page will require two different entries in the page tables, increasing physical memory requirements by up to 1%, depending on the ratio of page-table-entry size to page size.

With a virtually-addressed cache, the multiple virtual address mapping approach has a potential for cache inconsistency since updates at one mapping may reside in the cache while the other mapping contains stale data. This problem is easily solved in the context of the concurrent garbage-collection algorithm. While the garbage collector is scanning the page, the mutator has no access to the page; and therefore at the mutator's address for that page, none of the cache lines will be filled. After the collector has scanned the page, it should flush its cache lines for that page (presumably using a cache-flush system call). Thereafter, the collector will never reference that page, so there is never any danger of inconsistency.

Is this too much to ask?

Some implementations of Unix on some machines have had a particularly clean and synchronous signal-handling facility: an instruction that causes a page-fault invokes a signal handler without otherwise changing the state of the processor; subsequent instructions do not execute, etc. The signal handler can access machine registers completely synchronously, change the memory map or machine registers, and then restart the faulting instruction. However, on a highly pipelined machine there may be several outstanding page faults [26], and many instructions *after* the faulting one may have written their results to registers even before the fault is noticed; instructions can be *resumed*, but not *restarted*. When user programs rely on synchronous behaviour, it is difficult to get them to run on pipelined machines: *Modern UNIX systems ... let user programs actively participate in memory management functions by allowing them to explicitly manipulate their memory mappings. This ... serves as the courier of an engraved invitation to Hell*[26]

If the algorithms described are indeed incompatible with fast, pipelined machines, it would be a serious problem. Fortunately, all but one of the algorithms we described are sufficiently asynchronous. Their behaviour is to fix the faulting page and resume execution, without examining the CPU state at the time of the fault. Other instructions that may have begun or completed are, of course, independent of the contents of the faulting page. In fact, the behaviour of these algorithms, from the machine's point of view, is very much like the behaviour of a traditional disk-pager: get a fault, provide the physical page, make the page accessible in the page table, and resume.

The exception to this generalization is *heap overflow detection*: a fault initiates a garbage collection that

Methods	TRAP	PROT1	PROTN	UNPROT	MAP2	DIRTY	PAGESIZE
Concurrent GC	√		√	√	√		√
SVM	√	√		√	√		√
Concurrent checkpoint	√		√	√		‡	√
Generational GC	√		√	√		‡	√
Persistent store	√	√		√	√		
Extending addressability	√	*	*	√	√		√
Data-compression paging	√	*	*	√	√		
Heap overflow	√		†				

Table 2: Usages of virtual memory system services

* *Extending addressibility* and *data-compression paging* usePROT1 only to remove inactive pages; the batching technique described in section 5 could be used instead.

† Virtual memory-based *heap-overflow detection* can be used even without explicit memory-protection primitives, as long as there is a usable boundary between accessible and inaccessible memory (e.g. the "break" in vanilla Unix).

‡ Dirty-page bookkeeping can be simulated by using PROTN, TRAP, and UNPROT.

modifies registers (by forwarding them to point at the new locations of heap records), then resumes execution. The register containing the pointer to the next-allocatable word is adjusted to point to the beginning of the allocation space. The previously-faulting instruction is re-executed, but this time it won't fault because it's storing to a different location.

The behaviour is unacceptable on a highly-pipelined machine (unless, as on the VAX 8800 [10], there is hardware for "undoing" those subsequent instructions or addressing-mode side-effects that have already completed). In fact, even on the Motorola 68020 the use of page faults to detect heap overflow is not reliable.

Thus, with the exception of *heap overflow detection*, all of the algorithms we present pose no more problem for the hardware than does ordinary disk paging, and the invitation to Hell can be returned to sender; however, the operating system must make sure to provide adequate support for what the hardware is capable of; semi-synchronous trap-handlers should resume faulting operations correctly.

Other primitives

There are other virtual memory primitives that operating systems can provide. For a persistent store with transactions, it might be useful to *pin* a page[16] in core so that it is not written back to the backing store until the transaction is complete.

The Mach external-pager interface[1] provides at least one facility which is lacking from the primitives we describe: the operating system can tell the client which pages are least-recently-used and (therefore) about to be paged out. The client might choose to destroy those pages rather than have them written to disk. This would be particularly useful for *data-compression paging*, and *extending addressibility*. Also, in a system with garbage collection, the client might know that a certain region contains only garbage and can safely be destroyed[12].

In general, the external-pager interface avoids the problem in general of the operating-system pager (which writes not-recently-used pages to disk) needlessly duplicating the work that the user-mode fault handler is also doing.

6 Conclusions

Where virtual memory was once just a tool for implementing large address spaces and protecting one user process from another, it has evolved into a user-level component of a hardware- and operating-system interface. We have surveyed several algorithms that rely on virtual memory primitives; such primitives have not been paid enough attention in the past. In designing and analyzing the performance of new machines and new operating systems, page-protection and fault-handling efficiency must be considered as one of the parameters of the design space; page size is another important parameter. Conversely, for many algorithms the configuration of TLB hardware (e.g. on a multiprocessor) may not be particularly important.

Table 2 shows the usages and requirements of these algorithms. Some algorithms protect pages one at a time (PROT1), while others protect pages in large batches (PROTN), which is easier to implement efficiently. Some algorithms require access to protected pages when run concurrently (MAP2). Some algorithms use memory protection only to keep track of modified pages (DIRTY), a service that could perhaps be provided more efficiently as a primitive. Some algorithms might run more efficiently using a smaller page size than is commonly used

(PAGESIZE).

Many algorithms that make use of virtual memory share several traits:

1. Memory is made less-accessible in large batches, and made more-accessible one page at a time; this has important implications for TLB consistency algorithms.
2. The fault-handling is done almost entirely by the CPU, and takes time proportional to the size of a page (with a relatively small constant of proportionality); this has implications for preferred page size.
3. Every page fault results in the faulting page being made more accessible.
4. The frequency of faults is inversely related to the locality of reference of the client program: this will keep these algorithms competitive in the long run.
5. User-mode service routines need to access pages that are protected from user-mode client routines.
6. User-mode service routines don't need to examine the client's CPU state.

All the algorithms described in the paper (except *heap overflow detection*) share five or more of these characteristics.

Most programs access only a small proportion of their address space during a medium-size span of time. This is what makes traditional disk paging efficient; in different ways, it makes the algorithms described here efficient as well. For example, the concurrent garbage collection algorithm must scan and copy the same amount of data regardless of the mutator's access pattern [4], but the mutator's locality of reference reduces the fault-handling overhead. The "write barrier" in the generational collection algorithm, concurrent checkpointing, and persistent store algorithms takes advantage of locality if some small subset of objects accounts for most of the updates. And the shared virtual memory algorithms take advantage of a special kind of partitioned locality of reference, in which each processor has a different local reference pattern.

We believe that, because these algorithms depend so much on locality of reference, they will scale well. As memories get larger and computers get faster, programs will tend to *actively* use an even smaller proportion of their address space, and the overhead of these algorithms will continue to decrease. It is important that hardware and operating system designers make the virtual memory mechanisms required by these algorithms robust, and efficient.

Acknowledgements

Thanks to Rick Rashid, David Tarditi, and Greg Morrisett for porting various versions of our benchmark program to Mach and running it on Mach machines, and to Larry Rogers for running it on SunOS. Rafael Alonso, Brian Bershad, Chris Clifton, Adam Dingle, Mary Fernandez, John Reppy, and Carl Staelin made helpful suggestions on early drafts of the paper.

Andrew W. Appel was supported in part by NSF Grant CCR-8806121. Kai Li was supported in part by NSF Grant CCR-8814265 and by Intel Supercomputer Systems Division.

References

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. In *Proc. Summer Usenix*, July 1986.

[2] Andrew W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, 1987.

[3] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software—Practice/Experience*, 19(2):171–183, 1989.

[4] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. *SIGPLAN Notices (Proc. SIGPLAN '88 Conf. on Prog. Lang. Design and Implementation)*. 23(7):11–20, 1988.

[5] Malcom Atkinson, Ken Chisholm, Paul Cockshott, and Richard Marshall. Algorithms for a persistent heap. *Software—Practice and Experience*, 13(3):259–271, 1983.

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

[7] David L. Black, Richard F. Rashid, David B. Golub, Charles R. Hill, and Robert V. Brown. Translation lookaside buffer consistency: A software approach. In *Proc. 3rd Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 113–122. 1989.

[8] David R. Cheriton. The vmp multiprocessor: Initial experience, refinements and performance evaluation. In *Proceedings of the 14th Annual Symposium on Computer Architecture*, 1988.

[9] D. W. Clark and C. C. Green. An empirical study of list structure in Lisp. *IEEE Trans. Software Eng.*, SE-5(1):51–59, 1977.

[10] Douglas W. Clark. Pipelining and performance in the VAX 8800 processor. In *Proc. 2nd Intl. Conf. Architectural Support for Prog. Lang. and Operating Systems*. pages 173–179, 1987.

[11] Brian Cook. Four garbage collectors for Oberon. Undergraduate thesis, Princeton University, 1989.

[12] Eric Cooper. personal communication, 1990.

[13] George Copeland, Michael Franklin, and Gerhard Weikum. Uniform object management. In *Advances in Database Technology—EDBT '90*, pages 253–268. Springer-Verlag, 1990.

[14] A.L. Cox and R.J. Fowler. The implementation of a coherent memory abstraction on a numa multiprocessor: Experiences with platinum. In *Proceedings of the Twelfth Symposium on Operating Systems Principles*, pages 32–44, December 1989.

[15] Peter J. Denning. Working sets past and present. *IEEE Trans. Software Engineering*, SE-6(1):64–84, 1980.

[16] Jeffrey L. Eppinger. *Virtual Memory Management for Transaction Processing Systems*. PhD thesis, Carnegie Mellon University, February 1989.

[17] S. Feldman and C. Brown. Igor: A system for program debugging via reversible execution. *ACM SIGPLAN Notices, Workshop on Parallel and Distributed Debugging*, 24(1):112–123, January 1989.

[18] R. Fitzgerald and R.F. Rashid. The integration of virtual memory management and interprocess communication in accent. *ACM Transactions on Computer Systems*, 4(2):147–177, May 1986.

[19] Ralph Johnson and Paul R. Wilson. personal communication, 1990.

[20] Glenn Krasner. *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley, Reading, MA, 1983.

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

[22] Kai Li, Jeffrey Naughton, and James Plank. Concurrent real-time checkpoint for parallel programs. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 79–88, Seattle, Washington, March 1990.

[23] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 23(6):419–429, 1983.

[24] Raymond A. Lorie. Physical integrity in a large segmented database. *ACM Trans. on Database Systems*, 2(1):91–104, 1977.

[25] David A. Moon. Garbage collection in a large LISP system. In *ACM Symposium on LISP and Functional Programming*, pages 235–246, 1984.

[26] Mike O'Dell. Putting UNIX on very fast computers. In *Proc. Summer 1990 USENIX Conf.*, pages 239–246, 1990.

[27] John Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *Proc. Summer 1990 USENIX Conf.*, pages 247–256, 1990.

[28] Karin Petersen. personal communication, 1990.

[29] Paul Pierce. *The NX/2 Operating System*, pages 51–57. Intel Corporation, 1988.

[30] Robert A. Shaw. Improving garbage collector performance in virtual memory. Technical Report CSL-TR-87-323, Stanford University, 1987.

[31] Michael Stonebraker. Virtual memory transaction management. *Operating Systems Review*, 18(2):8–16, April 1984.

[32] Patricia J. Teller. Translation-lookaside buffer consistency. *IEEE Computer*, 23(6):26–36, 1990.

[33] Charles Thacker, Lawrence Stewart, and Edwin Satterthwaite. Firefly: A multiprocessor workstation. *IEEE Transactions on Computers*, 37(8):909–920, August 1988.

[34] David Ungar. Generation scavenging: a non-disruptive high performance storage reclamation algorithm. *SIGPLAN Notices (Proc. ACM SIGSOFT/SIGPLAN Software Eng. Symp. on Practical Software Development Environments)*, 19(5):157–167, 1984.

[35] David M. Ungar. *The Design and Evaluation of a High Performance Smalltalk System*. MIT Press, Cambridge, Mass., 1986.

[36] Paul R. Wilson. personal communication, 1989.

[37] Benjamin Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms*. PhD thesis, University of California at Berkeley, November 1989.