Operating System Support for Small Objects

Paul R. Wilson
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712-1188
wilson@cs.utexas.edu

Abstract

Operating systems should support fine-grained objects in two important ways. One is the use of fairly small pages (e.g., 1KB) and fast, flexible virtual memory primitives. Another is the use of a binary code format that supports precise identification of pointers in registers and stacks.

These features can support many novel and efficient uses of conventional hardware. Pointer swizzling ***(address translation)*** at page fault time ***can eficiently support huge address spaces with modest word sizes.*** Compressed paging ***implements a new level in the memory hierarchy (compressed in-memory storage) to bridge the growing gap between RAM and disk.*** Adaptive prefetching ***promises to reduce page fault costs more effectively than increasing page sizes. Many other applications would also benefit, including garbage collection, checkpointing, distributed virtual memories, lazy evaluation, and memory striping to reduce cache conflicts.***

1 Introduction

Several recent developments in OS and language implementation depend heavily on virtual memory system flexibility and performance [AL91], and it's likely more will be invented soon. This means that old benchmarks and measurements may have little bearing on the future importance of virtual memory primitives and their implementation [ALBL91]. This position paper will briefly describe our own uses for advanced virtual memory features, and outline their performance characteristics. We conclude that OS kernel designers (and hardware architects) should pay very careful attention to virtual memory and trap-handling mechanisms.

We also believe that the standard format of compiled code should support the identification of pointers in registers and the stack, so that a true heap abstraction can be supported. This support could be specified either at the OS level or through a common runtime system layered on top of the OS.

Many of the costs of object oriented systems can be minimized by software techniques on standard hardware. To properly exploit the hardware, however, system designers must take pointers very seriously. Given a true pointer abstraction, software and hardware can be coordinated for improved performance.

2 Pointer Swizzling at Page Fault Time–A RISC address translation scheme

We have designed and implemented an address translation mechanism that uses virtual memory page protections to allow ***pointer swizzling*** (address translation) ***at page fault time*** [Wil90a]. This scheme can support essentially infinite address spaces on standard (e.g., 32-bit) hardware, without incurring either continual software overhead or frequent hardware traps. The basic idea is similar to (and inspired by) the Appel, Ellis and Li incremental garbage collection scheme [AEL88]. We have used it to implement a simple persistent store that incurs almost no overhead for programs with good locality.

The basic technique is to translate pointers from one (long) format to another (short,, hardware-supported) format at page fault time. When a page is brought into a processor's memory, all of the pointers in the page are ***immediately*** translated into the short. format. This requires reserving a page of the processor's address space for each referred-to page.

For example, if we touch a page A, and it. holds pointers to pages B and C, space must be reserved for B and C in the processor's address space, so that the pointers in page A can be translated into normal addresses. That is, before allowing the program

0-8186-2265-2/91 $1.OO ©1991 IEEE

to see page A, we have to ensure that it contains only hardware-supported addresses–i.e., pointers into other pages of the processor's address space. To allow this, we must reserve a place for each pointed-into page. Once we know where the page will be in the address space, we can translate pointers into actual addresses.

Reserved pages are access-protected, and their contents are only brought into the local memory if the page is actually referenced. Then the pointers in the page are "swizzled" as described above–any pointed-to pages must have corresponding pages in the local memory, and those reserved pages are access-protected until they're actually touched.

The result is that address translation occurs in a "wavefront" just ahead of the actual access patterns of the program. Only touched pages have their pointers swizzled, and pages immediately reachable from them have pages reserved. By translating pointers eagerly (one page's pointers per fault), the fault overhead is kept low.

This address translation is static, in a certain sense-addresses are not translated at each use, but only when pages are brought into a processor's (virtual) memory. While we currently use this scheme to translate pointers from a persistent store's format into normal virtual memory addresses, it's possible to imagine an architecture in which this is the ***only*** kind of address translation–the "translation lookaside buffer" would be used only for caching protection bits, not for address translation [Wil90a].

We view this as an extension of the RISC philosophy to address translation, in that it only requires minimal hardware, plus the ability to trap to software as needed. In the usual case, we don't need more than 32 address bits because address streams are highly redundant (i.e., there's locality of reference, hence little real information). We just use a 32-bit shorthand to represent much longer addresses, and trap to software when we encounter an unusual case. We incrementally build up a table of translations from long page numbers to short ones.

(In effect, this is an adaptive compression of the dynamic stream of addresses; it is philosophically similar to the hardware mechanism called ***dynamic base register allocation,*** used to compress addresses communicated between a CPU and memory modules. Unlike dynamic base register allocation,, it requires no special hardware support, yet is more aggressive–it compresses addresses in memory, within the pagewise wavefront, as well as those dynamically used by the running program.)

One potential problem with this scheme is the exhaustion of address space. If too many pages are reserved, a processor's address space may be exhausted even if most of those pages go untouched. To avoid this, we have designed an incremental scheme for continuously invalidating and rebuilding address mappings, so that address space is never exhausted. This "reswizzling" is not free, however, and must be done more often if many unused pages get reserved.

In the worst case, the cost of incremental reswizzling is proportional to the number of pointers per page. In this case, every word of each touched page contains a pointer to a different page, which will not be touched. If there are n pointers per page, then maximum memory utilization is l/(n+l), because each touched page will require a page of address space for itself, plus n more pages of reserved address space for the pages it holds pointers into.

On 32-bit hardware using 1KB pages, the worst case allows us to touch about sixteen thousand 1KB pages before reswizzling. That's probably bearable–you can usually execute a lot of instructions without touching sixteen thousand pages. On the other hand, if pages are 8KB, as in some current machines, the worst case is very much worse: pages hold 2048 pointers, so only l/2049 of the address space can be used, or roughly 2 megabytes–only 256 8KB pages. Clearly, the worst-case performance of pointer swizzling is much improved by using smaller pages. The likely cases are significantly improved as well.

Even with 64-bit hardware architectures, pointer swizzling is still valuable. In some systems, especially Lisp- and Smalltalk-like systems, many things must be the size of a pointer. It may be desirable to use 32-bit hardware addressing, and be able to load or store two values–say, a whole Lisp cons cell–with one 64-bit machine operation. The costs of pointer swizzling may be more than repaid by the reduction in instructions and memory bandwidth requirements.

For other systems, with very large amounts of memory and worse locality, increasing addresses past 32 bits may be desirable. Pointer swizzling is still attractive, because it will allow a fixed number of bits to address an indefinitely large amount of data. A worldwide network of millions of machines, each holding billions of objects, could easily be addressed without *ever* requiring another increase in hardware addressing size. In fact, a handful of those bits could even be devoted to processor numbers and bit addressing support, with no problem. 64 bits would be the last hardware address size.

It is interesting to note that a swizzling facility needn't cost anything if it is not actually necessary. In a system with a "large enough" address word, page mappings may never need to be changed. In that case, the actual swizzling of pointers can be skipped when pages are faulted in. A swizzled system is more flexible, however, because it can grow beyond hardware word-size restrictions without any problem. It can also be used for reconciling conflicting mappings between systems–it is not necessary to ensure that any page always occupies the same part of virtual memory across different machines or local networks.

3 Checkpointing and Reconstructive Memory

Another application of virtual memory facilities is for checkpointing. In [WM89a], we presented a scheme for efficiently checkpointing garbage collected heaps, to build a *reconstructive memory* for *time-travel debugging* [FB88, WM89a, TA90].[1] A time travel debugger uses checkpointing and replay to provide the illusion of storing all past states of a program, allowing the programmer to jump backward in time to find the point where a program went astray. Checkpointing is also valuable for fault tolerance and concurrency control.

Since its original publication, our scheme has changed somewhat; we now avoid relying so heavily on copy-on-write techniques. We changed this to avoid overhead in SunOS trap handling, and also to avoid locality problems on machines with large pages. We now use a dirty bit scheme that saves copies of dirty areas of memory as our primary checkpointing mechanism (as did Feldman and Brown), but we implement the dirty bits in software to allow checkpointing smaller areas of memory. (This is much cheaper than it sounds, because most memory writes don't actually require setting a dirty bit.) We only use copy-on-write for the initial copy of pages faulted in from disk.

(When using a dirty page scheme rather than copy-on-write, there must be two copies of everything, because you don't know until later what has been changed. Since you can't catch a write before it happens, you can't be as lazy about copying-a write could destroy information that you'll need later. (For initial page contents faulted in from disk, e.g., the system image or a persistent store, the original version on disk can be used as one of the copies.))

To keep marking costs low, we actually use a whole byte as a dirty bit, so that we can exploit byte store instructions, and set a mark with only three instructions. (We got this idea from David Ungar, who refined our "card marking" scheme in this way before incorporating it into his latest garbage collector. We also use card marking for garbage collection purposes, killing two birds with one stone.)

4 Adaptive Prefetching and Replacement

Adaptive Prefetching schemes can also exploit small pages and fast virtual memory operations. We believe that the major advantages of large pages are better achieved by transferring several small pages at a time. Using smaller pages also gives you the flexibility to choose *which* pages are grouped together, and to change those groupings dynamically.

A very simple and cheap form of adaptive prefetching is to recluster pages according to the actual access patterns that bring them into memory. Future faults on a page will bring in a whole cluster of pages. If access patterns tend to be similar across time, this should be an effective heuristic.

This technique was simulated in the mid-seventies [BS76], with disappointing results. We believe the poor performance of the scheme was caused by accidents of the experimental design, however, and artifacts of mid-seventies memory technology. In particular, the page sizes used for the simulations were unrealistically small *relative to the total memory sizes simulated;* reserving pages for prefetching was thus very expensive. The cost for typical 1991 memories would be more than an order of magnitude smaller, because of the much larger number of pages or blocks at each level of the memory hierarchy.

We have found supporting evidence for the desirability of adaptive prefetching. [HH87] describes a prefetching policy that obeys an inclusion property;[2] like (non-prefetching) LRU, this allows many sizes of memory to be simulated efficiently in a single pass through a trace. While their policy was intended as an approximation of sequential One Block Lookahead, their modifications (to preserve inclusion) made it work significantly *better* than OBL.

We believe that the superiority of this prefetch policy is because it approximates what we call a "fool me

[1]**The term "time-travel debugging," which we like very much, is from Tolmach and Appel, but the concept was explicit in [WM89a], as was the coordination of checkpointing and garbage collection (though Tolmach and Appel neatly invert the relationship between them).**

[2]**That is, the page faults you get for one size memory are a subset of the ones you'd get for any smaller memory.**

once" policy. If a prefetched page is not touched, it will typically not be prefetched next time. This is a very simple form of adaptive prefetching.

Another technique, analogous to Baer and Sager's, is to reorganize objects within pages according to dynamic access patterns. This has been simulated for Smalltalk [WWH87] with good results (but see [WLM91]), and comparable techniques have been used with Baker-style incremental garbage collectors, exploiting the gc's ability to move objects when they're accessed by the running program [Whi80, Cou88, Joh91].

While these techniques (like Baer and Sager's) have been described as "reorganization," we think they should be viewed as a variety of adaptive non-linear prefetching that happens to have some very convenient properties. (In particular, it's easy to prefetch things without an extra seek, because they've been laid out consecutively on a disk.)

Rather than seeing these techniques as reorganizing objects within pages, we conceptualize them as transferring multiple tiny (object-sized) pages within larger units of disk transfer.

Given this perspective, it is natural to ask whether this is the right granularity for prefetching, and we suspect that it's not. In the first place, it requires specialized hardware support and some overhead per object; in the second place, fine-grained reorganization is most advantageous when memories are small. Larger-grained reorganization should work better for large memories, as are increasingly common, for essentially the same reasons that optimal page sizes get larger as memories do. We therefore choose to use very cheap static clustering [WLM91] to organize small objects within pages, but we believe that dynamic grouping of pages could yield significant additional benefits.

Clustering pages (rather than objects) limits the effectiveness of prefetching/reorganization schemes, but the lower frequency of traps could allow the use of more sophisticated clustering–doing a little more computation at each page fault could pay off handsomely if it results in a better grouping. For example, it may be better not to reorganize things that are already satisfactorily grouped; this might adapt to one access pattern at the expense of another.

We would also like to investigate adaptive replacement policies, which recognize patterns of use, and base replacement on expected time until future use. The very first virtual memory replacement policy (the "loop detecting" policy on the Ferranti ATLAS) attempted to do this, in a crude way, but did not work as well as LRU. We think that this failure may be due to the fact that the ATLAS policy keyed off of the highest observable frequency component, of access pat,-terns, rather than the lowest, leading to premature eviction of pages.[3]

We are particularly interested in applying adaptive prefetching and replacement to large persistent object stores and object-oriented databases. It seems likely that the virtual memory could he designed to respond appropriately to both stereotypical "database-like" reference patterns and more program-like "navigational" access patterns.

5 Compressed Paging

Another approach to improving paging performance is to add a level to the memory hierarchy, with price and performance intermediate between normal RAM and disk. Our candidate for this is compressed *in-memory storage* [WLM91]; we have discovered that heap data can be easily and cheaply compressed by an algorithm that is tuned to the garbage collector's placement of data with pages. This exploits the fact that pointer data typically hold very little real information, in much the same way that CDR-coding does for Lisp linked lists.

One of the interesting characteristics of compressed paging is that it comports well with adaptive memory management policies. Compressed disk storage allows more pages to be prefetched, and only the pages that are actually referenced need be uncompressed.

As processor cycles become progressively cheaper relative to disk seeks, it becomes increasingly attractive to keep more pages in RAM in compressed form, adding a new level to the memory hierarchy. The proportion of memory devoted to compressed storage can be varied dynamically, adapting to different locality characteristics. (Due to space limitations, we will not discuss this further-interested readers should see [Wil90b] and [WLM91] .)

6 Garbage Collection

Garbage collectors can benefit from the use of virtual memory primitives in several ways, as described in

[3]Lower frequencies should be more important, unless they are too low to be relevant to paging decisions. Consider data referenced in the bodies of nested loops-the period of the *outer* loop is the important one, because it may predict another occurrence of the whole spate of higher-frequency accesses in the inner loop. If the outer loop's frequency is lower than the time pages typically spend in memory, however, the pages should be evicted anyway. (See also [Wil90b].)

[AL91]. One application is to use pagewise protection to support incremental copy collection, with scanning a page of tospace as the increment of work [AEL88]. Unfortunately, this does not yield true real-time performance because the increments of work can be fairly large[4] and very closely spaced [WM89b]. Johnson has recently modified the Appel, Ellis and Li algorithm to put a much lower bound on the increment of work, but it is still proportional to the page size [Joh].

Another garbage collector application is the use of virtual memory dirty bits to help a generational garbage collector locate recently-created pointers from one generation to another [Sha88].[5] Shaw's technique can suffer greatly from poor locality of writes, and we believe that our card marking (software dirty bit) scheme is superior [WM89a] because it keeps track of smaller areas of memory. (Ungar's variant of our scheme is faster, but the bitmaps are really bytemaps, and therefore much larger. This incurs extra costs in scanning the maps to find dirty cards. This cost could be decreased by combining Shaw's technique with ours, and using virtual memory dirty bits to guide the scanning of (software) bytemaps–only dirty pages hold any dirty cards. This amounts to a two-level dirty bit scheme, with the virtual memory dirty bits providing the coarser resolution.)

7 Conclusions

We've discussed several applications that rely on advanced virtual memory primitives to support large heaps of small objects, and this only scratches the surface. Clearly, the performance of these techniques depends directly on the efficiency of trap handling, protection setting, and so on. Most depend on page size as well, with the ideal page size being smaller if traps are cheaper.

Many more applications of virtual memory primitives are possible if small areas of memory can be protected and traps are truly cheap, e.g., lazy evaluation and bounds checks, memory striping to reduce cache conflicts [Wil91] or reblocking to reduce false sharing in distributed systems.

We therefore view virtual memory primitives as more than just a means of implementing a memory hierarchy. They are the fundamental hook into basic parallel hardware (memory protection checking) that can support a large variety of features. This suggests that hardware and OS designers should consider separating out issues of protection from those of address translation, perhaps having multiple protectable blocks per page.

In addition, many of these desirable applications would benefit from the ability to unambiguously discriminate between object references and non-references. This can be done in software, without any specialized hardware support for objects (e.g., a tagged architecture or "object-oriented" memory hierarchy).

Object-oriented systems can be efficient on stock hardware, but hardware and OS designers must realize that the heap abstraction is very important, even if it's not implemented in hardware. Programs should be written in languages that support precise pointer finding, so that implementors have more freedom in implementing object references. This is not a very restrictive requirement–it does not require writing programs in dynamically typed languages like Lisp or Smalltalk. Modula-3, ML, Oberon, Eiffel, Ada, and even FORTRAN-90 all provide the necessary level of abstraction; unfortunately, C does not.

Languages should discriminate between object pointers and raw addresses, so that garbage collectors and other low-level runtime facilities are not confused by casts, etc. In particular, C++ should be extended to support classes which are garbage collected, and whose pointers are swizzle-able, in much the same way that Modula-3 does.[6]

If we are ever to rid ourselves of the scourge of least,-common-denominator C-and-UNIX, we need to clean up our act with respect to virtual memory and pointers. If we do, operating systems can support, **small** objects efficiently on standard hardware.

Acknowledgements

I would like to thank Ralph Johnson, Eliot. Moss, Andrew Appel, David Ungar, and Tom Moher for informative conversations during the development, of these ideas.

[4]When a page of tospace is scanned, all of the objects it holds pointers to are also moved to tospace. If the objects are large, this can involve considerable work.

[5]This is necessary so that the generational garbage collector can collect young generations often, without actually traversing data in older generations [LH83].

[6]In Modula-3, this is the default, but programmers can explicitly manage storage for un-movable objects when necessary, with the proper declarations to notify the compiler and runt ime system. In C++, the default should probably be reversed, but programmers should be encouraged to use only these pointer-clean classes in order to benefit from garbage collection and/or pointer swizzling.

References

[AEL88] Andrew W. Appel, John R. Ellis, and Kai Li. Realtime concurrent garbage collection on stock multiprocessors. In ***Proceedings of SIGPLAN '88 Conference on Programming Language Design and Implementation,*** pages 11-20, 1988.

[AL91] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In ***Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operat***ing ***Systems (ASPLOS IV),*** pages 96-107, April 1991. Santa Clara, CA.

[ALBL91] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska. The interaction of architecture and operating systems design. In ***Fourth International Conference on Architectural Suppport for Programming Languages and Operating Systems,*** pages 108-120, April 1991. Santa Clara, CA.

[BS76] Jean-Loup Baer and Gary R. Sager. Dynamic improvement of locality in virtual memory systems. ***IEEE Transactions on Software Engineering, SE-2(1),*** March 1976.

[Cou88] Robert Courts. Improving locality of reference in a garbage-collecting memory management system. ***Communications of the ACM,*** 31(9):1128-1138, September 1988.

[FB88] Stuart I. Feldman and Channing B. Brown. IGOR: A system for program debugging via reversible execution. In ***Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging,*** May 1988. Also distributed as ***SIGPLAN Notices*** 24, 1, Jan. 1989, pp. 112-123.

[HH87] R. Nigel Horspool and Ronald M. Huberman. Analysis and development of demand prepaging policies. ***Journal of Systems and Software,*** 7:183-194, 1987.

[Joh] Ralph E. Johnson. Reducing the latency of a real-time collector. Submitted to ***ACM Letters on Programming Languages and Systems,*** 1991.

[Joh91] Douglas Johnson. The case for a read barrier. In ***Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV),*** pages 96-107, April 1991. Santa Clara, CA.

[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.

[Sha88] Robert A. Shaw. ***Empirical Analysis of a Lisp System.*** PhD thesis, Stanford University, Stanford, CA, February 1988. Computer Systems Laboratory tech report, CSL-TR-88-351.

[TA90] Andrew Tolmach and Andrew Appel. Debugging Standard ML without reverse engineering. In ***Proceedings of 1990 ACM Symposium on Lisp and Functional Programming,*** 1990.

[Whi80] Jon L. White. Address/memory management for a gigantic Lisp environment, or, GC considered harmful. In ***Conference Record of the 1980 Lisp Conference,*** pages 119-127, 1980. Redwood Estates, CA.

[Wil90a] Paul R. Wilson. Pointer swizzling at page fault time: Efficiently supporting huge address spaces on standard hardware. Technical Report UIC-EECS-90-6, University of Illinois at Chicago EECS Dept., December 1990. Also in ***Computer Architecture News,*** June 1991.

[Wil90b] Paul R. Wilson. Some issues and strategies in heap management and memory hierarchies. In ***OOPSLA/ECOOP '90 Workshop on Garbage Collection in Object-Oriented Systems,*** 1990. Also in SIGPLAN Notices ***23,*** 1, Mar. 1991, pp. 45-52.

[Wil91] Paul R. Wilson. ***Heap Management and Memory Hierarchies.*** PhD thesis, University of Illinois at Chicago, December 1991.

[WLM91] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Effective static-graph reorganization to improve locality in garbage-collected systems. In Proceed***ings of the ACM SIGPLAN '91 Conference***

on Programming Language Design and Implementation, pages 177-191, June 1991. Toronto, Canada.

[WM89a] Paul R. Wilson and Thomas G. Moher. Demonic memory for process histories. In ***Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation,*** pages 330-343, June 1989. Also distributed as ***SIGPLAN Notices 24***, 7, July 1989.

[WM89b] Paul R. Wilson and Thomas G. Moher. Design of the opportunistic garbage collector. In ***Proceedings of OOPSLA '89,*** pages 23-35, October 1989. New Orleans, Louisiana.

[WWH87] Ifor W. Williams, Mario I. Wolkzko, and Trevor P. Hopkins. Dynamic grouping in an object-oriented virtual memory hierarchy. In ***Proceedings of the 1987 European Conference on Object-Oriented Programming.*** Springer-Verlag, 1987.