Pointer Swizzling at Page Fault Time: Efficiently and Compatibly Supporting Huge Address Spaces on Standard Hardware *

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

...I could be bounded in a nut-shell, and count myself king of infinite space...

— W. Shakespeare, *Hamlet*, II:ii

Abstract

Pointer swizzling at page fault time is a novel address translation mechanism that exploits conventional address translation hardware. It can support huge address spaces efficiently without long hardware addresses; such large address spaces are attractive for persistent object stores, distributed shared memories, and shared address space operating systems. This swizzling scheme can be used to provide data compatibility across machines with different word sizes, and even to provide binary code compatibility across machines with different hardware address sizes.

Pointers are translated ("swizzled") from a long format to a shorter hardware-supported format *at page fault time*. No extra hardware is required, and no continual software overhead is incurred by presence checks or indirection of pointers. This pagewise technique exploits temporal and spatial locality in much the same way as a normal virtual memory; this gives it many desirable performance characteristics, especially given the trend toward larger main memories. It is easy to implement using common compilers and operating systems.

1 Huge Address Spaces

It is often desirable to support a larger virtual memory address space than the word size of the available hardware can specify directly. Applications of large address spaces include distributed shared memories (e.g., [Li86]) operating systems with a single shared address space (e.g., [CLLBH92]), and persistent object stores (e.g., [ABC+83, DSZ90]). Distributed shared memories provide a single address space for applications that span multiple machines, shared address space operating systems provide a single addressing model for all processes on one or more machines, and persistent object stores provide sharable, recoverable heap storage to eliminate the use of files for most purposes.

A common feature of these systems is the emphasis on simplifying programming by preserving pointer semantics—that is, object identity. The programmer's default view of data is of a heap of objects connected by pointers. Programs may traverse pointers freely, and complex data structures may be passed by reference.

It most current systems (without large shared address spaces) it is often necessary to write routines to flatten data structures into a low-level linear form and reconstruct them later, in order to communicate them from one machine to another, or to make them recoverable in the case of a crash, or to save them on disk so that they can be operated on later, perhaps by a different program. This tedious and error-prone coding is a large software development problem in conventional systems. (The explicitly programmed conversions typically lose most type information, as well as pointer semantics; they bypass type systems, leaving data structure consistency entirely up to the programmer.)

Shared and persistent memory systems avoid much of this difficulty, but they themselves pose challenges for system implementors. One problem in implementing such large memories is that it involves addressing a huge number of objects, potentially more than can be

*Proc. 1992 Int'l. Workshop on Object Orientation in Operating Systems, Paris, France, Sept. 24-25, 1992, pp. 364–377. (IEEE Press.)

specified by the hardware's address bits. Schemes for supporting large virtual addresses on normal hardware (e.g., LOOM [Kae81, Sta82], E [WD92], and Mneme [Mos90]) typically incur significant overhead.

Two approaches are commonly used to implement large address spaces in software. One is to indirect pointers through an *object table*, and translate large identifiers into object table offsets when objects are brought into memory. Untranslated identifiers (for which there isn't a corresponding table entry) may be flagged and translated into offsets lazily.

The other main approach is *pointer swizzling.* Rather than using an indirection through a table, long-format pointers are converted to actual hardware supported addresses in an incremental fashion.

In a conventional pointer swizzling scheme, a persistent pointer is converted into a virtual memory address only when the running program tries to use it; this entails bringing the object into the transient memory address space. This may involve checking each pointer at each use, to see if it is an actual address. White and Dewitt refer to this as *swizzling on discovery* [WD92]; it is also possible to swizzle all of the pointers in an object at once, the first time it is touched [Mos90]. In either case, pointer fields or objects must be checked very frequently to see if they've been swizzled yet, so that unswizzled pointers can be swizzled before being traversed.

We would like to avoid these costs, so that programs that do not access persistent data do not pay the freight, and so that programs that access persistent objects (or pages) many times do not incur additional checking costs at every access. Ideally, we would like this mechanism to operate efficiently on standard hardware, rather than requiring an exotic "object-oriented" hardware memory hierarchy such as that of the MUSHROOM project [WWH87] or capability-based addressing schemes (e.g., [Lev84], [Ros91].).

For simplicity, much of this paper is cast in terms of persistent memories, because our current implementation focus is on persistent object stores. The paper is actually about address space implementation more generally, however; the ideas are equally applicable to operating systems and distributed shared memories.

(We sometimes refer to conventional hardware-supported virtual memory as *transient* memory, distinguishing it from a larger, *persistent* memory. Conventional virtual memories are transient in that an address space ceases to exist when the (heavyweight) process it belongs to terminates. Persistent memories, like file systems, may outlive the processes that create them.[1].)

2 Address Translation at Page-Fault Time

Our approach is to fault pages into conventional virtual memory on demand, translating long persistent-memory addresses into normal hardware-supported addresses at page-fault time [Wil91].[2] This strategy exploits locality of reference in much the same way as a normal virtual memory; we believe this pagewise approach is increasingly attractive as main memories grow larger. (As the number of instructions executed between page faults goes up, the cost of address translation can be amortized across more useful program work.) In contrast, software schemes involving presence checks and indirections do not scale as nicely—most of their address translation cost is tied directly to the rate of program execution.

Any incremental faulting scheme must somehow detect references to objects in persistent memory, so that they can be copied into virtual memory before being operated on. We choose to use existing virtual memory hardware's pagewise access protection capability for this. This allows the checks to occur in parallel, as part of the normal functioning of the virtual memory system, and avoids continual overhead in software.

Our approach is analogous to that of Appel, Ellis, and Li's incremental garbage collection scheme [AEL88]. Their system copies live (reachable) objects incrementally from "fromspace" to "tospace," to separate them from garbage objects[3]; ours relocates pages of objects from persistent memory transient memory so that they may be directly addressed. Still, the basic principles of operation are the same.

To simplify the explanation of this technique we will assume for the moment that objects in virtual memory are the same size and shape (memory layout) as persistent-memory objects. Later we will explain how mismatches between representations (e.g., one-

[1]This terminology is appropriate in terms of our address translation scheme as well—long addresses conceptually endure over time, but they may be mapped in changing (transient) ways to different virtual memory addresses.

[2]A similar approach (developed independently) appears to be used in ObjectStore [LLOW91], a commercial product from Object Design, Inc. Few details of their scheme are available, however.

[3]In a copying garbage collector, unreachable (garbage) data objects are reclaimed *implicitly*—rather than finding the garbage, the live (reachable) objects are moved (copied) into another area of memory (tospace), and the obsolete area (fromspace) is then reclaimed in its entirety.

vs. two-word pointer fields) can be handled.

Our scheme relocates objects into transient memory somewhat sooner than a straightforward (software-only) pointer swizzling scheme. This allows us to preserve one essential constraint—the running program is never allowed to encounter any pointers into persistent memory. Any pages that contain persistent pointers must be access-protected. If the program attempts to access a page that may contain persistent pointers, a trap handler is invoked; it translates all of the persistent pointers in that page into transient pointers, relocating their referents as needed; the page is then un-protected and the program may safely resume.

Rather than relocating particular *objects* referred to by pointers in the faulted-on page (as the Appel, Ellis and Li garbage collector does), our scheme relocates whole *pages* those objects are located in.[4] This reduces the size of tables required to hold mappings between transient and persistent addresses—only the page numbers must be recorded, not individual objects. (It also comports well with page faulting—we believe it is desirable to bring objects into memory a whole page at a time anyway; caching pages is usually more attractive than object faulting when main memories are not small [Sta82, Wil91, WD92].)

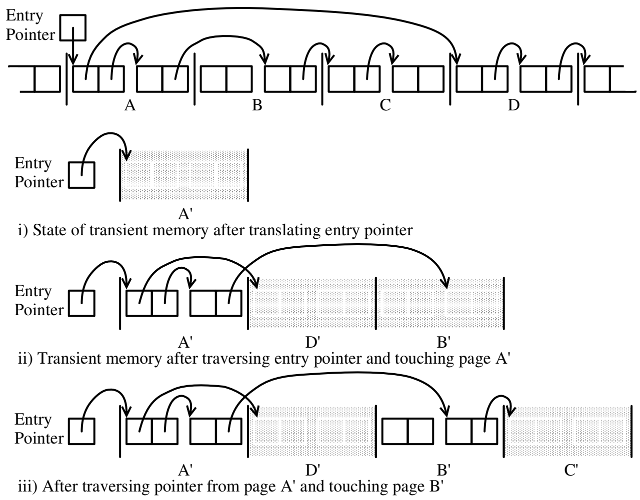

Figure 1: Incremental faulting and pointer swizzling

Figure 1 illustrates this mechanism. The top part of the figure shows some data objects in pages of the persistent store, with persistent-format (i.e., long) pointers between them. When a program is given access to the persistent store, any pages it holds "entry pointers" into must be relocated into virtual memory and access-protected. This allows the entry pointers to be translated into machine format so that the program can begin execution. The second part of the figure shows this state, with page A appearing in virtual memory as page A'.

When the program attempts to access such a page (in this case, A'), the access-protection handler translates *all* of the pointers in the page into the desired format. For all of the pointers to be translated, it must be known where in memory the referred-to objects will reside. That is, if the program accesses page A', and page A' contains a pointer into page B, page B must be relocated into transient memory (as B'). This allows persistent pointers from page A' into page B to be translated into absolute virtual memory addresses (in B'), so that page A' can be made accessible to the program. We simply treat each pointer in the faulted-on page the same way we treated the entry pointer. This process repeats whenever a pointer into a protected page is traversed, advancing the pagewise "read barrier" of protection one step ahead of the program's actual access patterns.

Relocation of pages thus occurs in a sort of "wavefront" just ahead of the running program, insulating it from untranslated pointers. Pointers can be dereferenced in the usual way, i.e., in a single machine instruction, with no extra overhead except at page faults.

3 Moving Data Lazily

Note that since the pages are access-protected at relocation time, we can simply reserve the space and note the mapping, *without actually copying their contents into transient memory.* The actual data can be moved more lazily, when the program first attempts to touch the page. The recording of persistent-to-transient page mappings thus runs "one pointer ahead" of the program's access patterns, but the movement of actual data into the persistent store is just as lazy as with a normal, demand-paged virtual memory. Since reserved-but-untouched pages are access protected anyway, they needn't have any storage behind them at all—not even swap disk. Establishing a persistent-to-transient page mapping amounts to a promise that we'll put a page there if and only if it's needed.

[4] Thanks to Ralph Johnson for this; our original conception adhered too slavishly to the Appel-Ellis-Li model and relocated individual objects. Johnson had devised a somewhat similar scheme, from which we got the idea to relocate whole pages instead. (Johnson, personal communication 1989.)

4 Handling the Pointer Size Mismatch

Since the contents of a page need not be translated at the time it is relocated, but only if it is accessed and its pointers are translated to the normal hardware-supported size, object formats in persistent memory may be different form those in transient memory. The main requirement is that a persistent page full of objects fit into a transient page of memory once they have had their formats translated. It is also necessary to be able to derive objects' eventual locations in transient pages from their locations in persistent pages.

The most straightforward possibility is making every pointer in persistent memory be twice the size of a transient memory pointer, e.g., use a field size of 64 bits. When a pointer is swizzled into a 32-bit native pointer, it only uses half the field. This is not a big cost for a statically typed language like C++, because most fields are not pointers.[5] The space cost is usually only ten to twenty percent.

5 Avoiding Exhaustion of Virtual Address Space

A potential problem with the basic scheme is that the transient memory could fill up with relocated pages that are used for a while, and then not used again for a long time. These pages could fill up the virtual memory, causing excessive paging. This is actually not much of a problem, because the process of swizzling (address translation) is nearly orthogonal to issues of levels in the storage hierarchy—an inactive page can still be paged out to backing disk. (It may be paged to a swap area temporarily, or it may be unswizzled and evicted back to the persistent store.)

The real problem, then, is not the exhaustion of hardware memory, but the *exhaustion of the hardware-supported virtual address space*. This is not just a problem for programs that actually touch millions of pages, because touching one page may cause the reserving of several pages of the address space so that addresses can be translated. In the worst case, a page contains nothing but pointers, causing the reserving of as many pages as there are pointers—in our current system, about 500 times as many pages may be reserved as are actually touched. While this is unlikely for most programs, it is conceivable and in fact rather near-fetched—pages holding multiway tree nodes may approximate the worst case.

To avoid using up the virtual address space, we have two strategies. (Neither is currently implemented, however—we currently have no applications requiring them, but expect to in the future.) The first strategy is to reduce the effective page size, and slow the rate of address space use. This strategy may not be entirely effective, so we have also devised an algorithm for reclaiming virtual address space incrementally and reusing it.

5.1 Reducing the Effective Page Size

Large page sizes are a potential problem for our scheme, because large pages may have a high "fan out", i.e., hold pointers to many other pages. We can reduce the effective page size by only using part of each virtual address page when allocating objects with large numbers of pointers. For example, if we only use one fourth of each 4KB page, we reduce the fan-out by a factor of four. A naive implementation of this strategy would be very wasteful, of course, so it is desirable to avoid using actual RAM and disk storage sparsely.

A better strategy is to only use a fraction of each *virtual* page, but arrange the fragments in a complementary pattern so that several virtual pages can share a physical (RAM or disk) page. Suppose we wanted to implement 1KB fragments of 4KB pages; we could map four virtual pages to a single physical page, and use a different quarter of each virtual page for data. While the physical page as a whole would have four aliases (virtual page numbers), the nonoverlapping pattern of allocation would ensure that no object (or cache block) was actually aliased.

(To avoid fragmenting memory mappings and decreasing virtual memory performance, an entire range of pages could be aliased by four virtual memory address ranges, e.g., using the **mmap()** system call available in most modern versions of UNIX.)

This solution is not entirely satisfactory, for two reasons. First, it does not deal with objects whose size is comparable to a page or larger. Second, while it wastes little or no physical storage, it does decrease the effectiveness of translation lookaside buffers—each partially-used virtual page requires its own virtual-to-physical page mapping in the virtual memory system. While it defers the exhaustion of the address space in the sense of delaying the discovery and swizzling

[5] Our original implementation for Scheme (a dialect of Lisp) used a somewhat different approach, because most objects' fields must be large enough to hold pointers. In that implementation, *all* fields were twice as big in the persistent store [Wil91], and persistent pages were twice as big as normal virtual memory pages. This avoided wasting transient memory space (at a cost in persistent data density) while allowing simple conversion from persistent addresses to virtual memory addresses.

of pointers, it actually increases total address space usage (in the long run) by decreasing the usable size of each virtual page.

This strategy is therefore most attractive for small pointer-rich objects expected to be used in ways that create high-fanout pages. In particular, it should be useful for the nodes of trees used as large, sparsely searched indices.

5.2 Address Space Reuse

The easy way to deal with the exhaustion of the address space is simply to occasionally evict all of the pages from virtual memory, throw away the existing mappings, and then begin faulting pages in again in the normal way.[6] Pages that are no longer in use will not be faulted in again, but the current working set will be restored quickly. This incurs unnecessary and bursty traffic between the transient and persistent stores, however.

To avoid this, we take advantage of the fact that address translation and data caching are essentially orthogonal. We don't really have to write pages out to reclaim the corresponding pages of the virtual address space. Evicting pages from local (virtual memory) storage is easy; clean pages can simply be discarded, and dirty pages can be unswizzled and written back to the persistent store.

Rather than actually writing everything out, then, we can simply invalidate and incrementally rebuild the virtual memory mappings. That is, we "pretend" to write out all of the data, but we leave it cached locally, and actually just access-protect the pages. We can then incrementally fault on them to build a new set of mappings. If a page is faulted on and it is still in local storage (RAM or disk), so much the better—its pointers can simply be reswizzled according to the current mappings, in much the same way as when the page was originally faulted in. The "obsolete" mappings could be consulted, and could even be re-used in many cases. (If a page is not dirty, i.e., written to since it was faulted into transient memory, then it contains no pointers into pages that it did not previously contain pointers into. Low-fanout pages should thus have their referred-to pages recorded at swizzling time.)

[6] Note that we can't just evict a page from the virtual address space, because we do not keep track of pointer assignments—any page that in the virtual address space must be assumed to have pointers into it from other pages in the address space. We therefore cannot reuse that page we rebuild the mappings—we have to traverse the graph of pointers and rebuild the mappings to find out which pages are reclaimable.

Reclamation of pages can begin after the program has run a while, to recreate or revalidate all the mappings of its current "working set." Candidates for reclamation are pages that have not been referenced since the mass invalidation, and which are not directly reachable from pages that have been. The reclamation policy should probably favor evicting pages that are only directly reachable from pages that haven't been touched for a long time. To increase efficiency in systems where page fault traps are expensive, the (transient) virtual memory system's recency information might be consulted, and the most recently-touched pages could be assumed to be part of the current working set. These pages would have their addresses recomputed or revalidated immediately, (rather than being access-protected) to avoid most of the flurry of access protection faults immediately after the mass invalidation.

6 Objects that Cross Page Boundaries

One potential hitch in our scheme is that pages that if an object crosses page boundaries in the persistent store, the corresponding pages must be adjacent in the transient virtual memory as well. (If an object straddling a page boundary is not relocated as a contiguous object, indexing to access its fields will not work properly.) Lazy copying is particularly helpful here; address space must be reserved for the whole object, but there needn't actually be *any* physical (RAM or disk) memory used for untouched pages.

To support this, the language implementation must somehow support operations that find object boundaries, and/or maintain "crossing maps" to tell which pages hold part of an object continued from a previous page. These requirements are not much different from those of garbage collected systems that must perform pagewise (or "cardwise") operations [AEL88, WM89b] within the heap; there is no major difficulty supporting such operations efficiently for languages like Lisp or ML. Slightly conservative versions of these schemes will work well for languages with derived pointers and (limited) pointer arithmetic, in much the same way that conservative garbage collectors operate with languages like C [BW88]. The main modifications are to the allocation and deallocation routines, which must provide headers and/or groupings and/or alignment restrictions to allow objects to be identified.

Large objects still pose a potential problem for our system, in terms of exhaustion of the hardware address space. If a page is touched, and it holds pointers to several large (multi-page) objects, space must be reserved for all of those objects' pages, even if they are never touched. Programs that deal with many very large objects may therefore benefit from a larger hardware address space, to decrease the frequency of address space reuse. While we don't think that this is a serious problem for most programs on most computers, it is worth considering. As the following section shows, we can integrate machines which require large hardware addresses with those that don't, and allow the sharing of most data between them.

7 Compatibility

We see pointer swizzling at page fault time as part of a general purpose *reconciliation layer* that can be used to structure systems very flexibly at little performance cost [Wil91]. It can be used to support data formats that allow flexible sharing of data across machines with different word sizes, and even to support binary code compatibility within families of machines that have different address sizes. These capabilities only require very slight changes to existing programs and/or compilers.

Naturally, any program that actually requires a huge flat address space cannot be run on the 32-bit machines, for example, programs that need flat array indexing into multiple gigabyte arrays, or programs whose locality is very bad on the gigabyte scale. These programs are likely to require high-end processors anyway, and be executed on the larger machines. By and large, however, most programs and data could be shared across different-sized machines.

(We don't claim to have a panacea—the problems of data sharing are difficult and deep in the general case. On the other hand, the trend toward standard numeric formats and byte addressing is encouraging. 32-bit integers, 64-bit IEEE floats, and swizzled pointers would support a much higher "greatest common divisor" than the currently ubiquitous streams of bytes.)

7.1 Address Size-Independent Data Formats

For compatibility across different machines, it may be desirable to have a single data format that can be used, irrespective of the address word size of the machine operating on the data. This is particularly attractive for a shared persistent store or a distributed virtual memory.

By using pointer swizzling to adjust pointer sizes, it is easy to accomplish this. When pages are transferred from one machine to another, it is only necessary to translate the pointers in a page into the native format of the receiving machine.

Pointer swizzling only requires that it be easy to find the pointers in a page, and that it be easy to convert a large persistent pointer into the hardware supported format. This is done by translating a the high order bits (page number) to the shorter bit pattern of the transient page number, and adjusting the low-order bits that represent the offset within the page. The simplest way of ensuring this ability is to have the persistent data format be the same as the transient format, so that the offset part of a pointer does not change at all.

This can be done for multiple pointer sizes by simply leaving enough room for the largest hardware-supported pointer size, whether it is needed on all machines or not. So a 64-bit field can be used on 64-bit machines, and also on 32-bit machines—but only half of the field is used for transient pointers on 32-bit machines. (As mentioned above, the other half of the field goes to waste, but this space cost is relatively small.)

This is similar to the approach used in the Commandos [MG89] operating system, where object identifiers are used on disk, but they are swizzled to actual pointers in memory. Commandos does not use page-fault time swizzling, however, and incurs overhead in checking for unswizzled pointers. (Using object identifiers rather than persistent addresses also makes translations more expensive.)

7.2 Binary Code Compatibility Across Hardware Address Sizes

Not only is it possible to define compatible data formats that can be used by programs running on hardware with different address sizes, it is even possible to define instruction set architectures (ISA's) so that the same *compiled code* can run on machines with different word sizes.

For architecture families like the current MIPS processors (i.e., 32-bit R3000 and 64-bit R4000) or the IBM/Apple/Motorola PowerPC, a few instructions added to the 32-bit machine can provide an addressing mode that can be reconciled with 64-bit addressing, by using pointer swizzling. In each of these families, the 64-bit instruction set architecture is a superset of the 32-bit ISA. The same register designations are used, but on the 64-bit machine the registers are twice as long. (Backward compatibility is ensured by defining most opcodes in such a way that on the 64-bit machine they operate on a 64-bit register, but the effect on the low-order 32 bits is the same as the operation on the smaller machine's 32-bit registers. Any program compiled for the 32-bit ISA still runs on the 64-bit ISA, because it doesn't depend on the values in the upper halves of registers. Some new opcodes are added as well, such as 64-bit loads and stores.)

As described previously, data objects can be laid out in a compatible way, with a 64-bit pointer field, only half of which is used on 32-bit machines. It is only necessary to define the 64-bit machines' opcodes on the 32-bit machines to "do the right thing" for the actual hardware address size. The opcodes for 64-bit loads and stores on the 32-bit machines should do *32-bit* loads and stores. (Equally important, the loads and stores should use 32-bit register values for addressing.)

The strategy here is to use the 64-bit opcodes as "compatibility opcodes," which use whatever hardware address size is best on a particular machine. That is, they become 64-bits-if-you've-got-them opcodes. For compatibility with existing compilers and libraries, code compiled to use 32-bit addressing on 32-bit machines (using the plain 32-bit opcodes) will work on either machine, in the normal way. Code compiled to run on 32-bit machines using pointer swizzling will use 64-bit opcodes instead, to load 32-bit values using 32-bit addresses. The pointer swizzling scheme will ensure that any pointer field the processor can see will in fact have a 32-bit pointer in the relevant half of the word. But the same opcodes actually mean 64-bit loads and stores on 64-bit machines, and the pointer swizzling scheme will ensure that those fields do hold 64-bit addresses on those machines.

It's not actually *quite* this simple, because of the possibility of performing pointer arithmetic on these addresses. All that is necessary, though, is to use the relevant arithmetic opcodes in the same way as load and store opcodes.[7]

One of the great attractions of this scheme is that it is not actually necessary to use any new operations. A slight change to opcode decoding will do the trick; 64-bit instruction opcodes, which currently trap as illegal instructions on a 32-bit machine, should instead do the equivalent operation, but using 32-bit addressing and registers. This should not break any existing 32-bit code, and requires no change to the 64-bit architecture.[8]

The implications for compilers are that a "compatibility mode" back end can easily be implemented by slightly modifying existing back ends. The instruction set should be limited to instructions available on both 64-bit and 32-bit versions of the architecture—that is, it should be limited to the 32-bit ISA—except when dealing with addresses. 64-bit fields should be used for addresses, including stack-allocated variables, and 64-bit opcodes should be emitted to operate on them. The resulting code should also work on 32-bit architectures, with pointer swizzling at run time.

While the above scheme would provide maximum performance by simply aliasing existing operations with new opcodes, another variation could provide compatibility on an installed base of 32-bit machines. Many machines provide for the fast trapping of unimplemented opcodes, so that they can be emulated in software. This trapping could be used to emulate the compatibility opcodes, rather than actually having a new opcode. On 32-bit machines, 64-bit opcodes would trap to routines that simply executed the corresponding 32-bit instructions. While less efficient than hardware aliasing, this would still allow the generation of compatible binaries using 64-bit opcodes as compatibility opcodes.[9]

7.3 Using Existing Languages and Compilers

While pointer swizzling at page fault time is obviously applicable to languages like Lisp and Smalltalk, which use tagged pointers, it can also be used for strongly-typed languages such as Modula-3, ML, and (with slight restrictions) C or C++.[10] The success of conservative garbage collectors shows that most C programs require little or no modification to meet the necessary constraints.[11] Some compiler optimizations may mutilate pointers beyond recognition, but there are ways around this.[12]

[7] In some architecture families, 32-bit machines will never attempt to actually do 64-bit operations, so the 64-bit opcodes can be used as pseudonyms for 32-bit operations. In other families, the same opcodes are used anyway. Either way, those opcodes effectively become "compatibility mode" address arithmetic instructions, which do the right thing on either architecture.

[8] Thanks to Rich Oehler and Keith Diefendorff for pointing out a flaw in an earlier version of this compatibility mode scheme, and suggesting details of this one.

[9] In marketing terms, this would allow the distribution of compatible binaries that would run on the installed base, with a two-level upgrade strategy. A cheap upgrade would consist of swapping the CPU chip for a pin-compatible chip with the opcode aliases in hardware. A more expensive upgrade would be a full-blown 64-bit CPU.

[10] The main restriction is the avoidance of untagged unions holding pointers in the variant part. Another is the avoidance of storing intermediate values from pointer arithmetic expressions, without retaining an actual pointer to the object [BW88, Boe91].

[11] Untagged unions are not very attractive in C++, because of its object-oriented features, and most pointer arithmetic that breaks garbage collection (or pointer swizzling) relies on unportable assumptions anyway.

[12] The easy way is to forgo certain advanced optimizations, as is done by several systems that use C as an intermedi-

It is only necessary for heap allocation routines to ensure that data objects within heap pages can be recognized, and that the pointers to and within those objects can be found. The techniques for this are well understood, having been developed for the purpose of garbage collecting statically-typed languages.

It is even possible to do pointer swizzling for programs compiled with standard off-the-shelf compilers. It is only necessary to treat the stack *conservatively*, as is done by garbage collectors developed for off-the-shelf compilers [BW88, Bar88, Det90, WH91, Det92]. Any bit pattern within the stack which *could* represent a pointer must be conservatively assumed to *be* a pointer. The pointed-to page is then "pinned" in the transient address space, because the page can't be relocated without invalidating the (possible) pointer. This does *not* require the page to actually remain in transient memory—just that its persistent-to-transient mapping (for a particular transient memory) not be changed.[13]

Interestingly, it is possible to cast pointers to integers, *and back to pointers again*, without breaking the pointer swizzling system. If pointer casting to an integer is implemented as an unswizzling into the persistent address bit pattern, and if casting an integer into a pointer swizzles it according to the prevailing page mappings, then all is well.[14] This requires the use of an integer long enough to hold the persistent form of the address, which is probably fine if only a 64-bit address space is required. It can break copying garbage collectors, however; in this respect, the requirements of the swizzling system are actually weaker than those of a copy collector.

7.4 Linking to Existing Binaries

Because pointer swizzling at page fault time requires no changes to objects' data formats or the code that manipulates them, it allows swizzled and unswizzled objects to be used in the same programs, with a few restrictions on how they may interact. Persistence is a property of individual objects, not of classes—a persistent object is simply one that is allocated on the persistent heap rather than the conventional (transient) heap.

We have found this to be extremely convenient in the process of developing our persistent store—we can intermix conventional C++ code and persistent C++ code, even linking to existing binaries. It is not necessary to recompile libraries, for example, as long as transient objects are not expected to persist or survive across crashes. Transient objects may hold pointers to persistent objects, and vice versa, as long as they follow a few simple rules [SKW92]. (Naturally, more alternatives are possible if the source is available. In that case, it can be run through a precompiler to record object layouts, and linked with our heap allocator; this allows all objects to be made persistent, if desired.)

8 Truly Enormous Address Spaces

In the scheme described above, the size of pointer fields is determined primarily by the size of hardware-supported virtual addresses. (We currently use a 64-bit field because it is large enough to hold either a 32-bit or a 64-bit address.) The size of persistent addresses can be arbitrarily large, however; we believe it will be desirable to have 96 or 128-bit addressing in the future. This is easy to do with pointer swizzling at page fault time.[15]

To support enormous address spaces, it is only necessary that the persistent object format be able to hold the maximum-sized pointers. The uniform 64-bit transient pointer fields can still be supported, but pages may take up more space in persistent memory than in a processor's virtual memory—that is, the size of a page may change when it is swizzled. Variable-sized persistent pages complicate page lookup, because they complicate the mapping of conceptual pages to disk blocks. We do not believe that this is a serious problem in the long run, however, because of current trends in the evolution of file systems—namely *block mobility* and *compressed storage*.

Block mobility is increasingly important, as in the case of *log structured* file systems—that is, the location of a block may change over time to maximize effective write bandwidth [RO91] and possibly adapt

ate language—but this incurs a small performance penalty. A better technique in the long term is to ensure that compilers don't violate the invariants unnecessarily [Boe91], or that they record clarifying information when they do [DMH92, WH91, BMBC91]; we believe compilers should retain this kind of information anyway, to support "de-optimizing" in source-level debuggers (e.g., [HCU92]).

[13] It is not difficult to swap a page from one machine's address space to another's, and back. The page's mappings (to transient pages) do not change in either space, but when a page is transferred, the pointers in it are unswizzled from the source space representation and reswizzled into the destination space representation. It might be possible to make other format changes at the same time, such as endianness reversal when dealing with different machine architectures.

[14] Thanks to David Chase for suggesting this implementation.

[15] Our attitude toward address space might be summed up as "crunch all you want—we'll make more."

to observed patterns of disk usage to reduce read latencies [Gri89, Wil91].

Compressed storage is likely to become increasingly important, whether at the level of file storage or as part of the virtual memory system [AL91, Wil91]. The problems of variable-sized units of storage are thus likely to be solved anyway, for entirely different reasons. Compressed storage is attractive for reducing storage costs, and for reducing average latency (by storing more data in fast memory and/or decreasing communication costs). If compression is used, long persistent addresses should typically take up considerably *less* space than the in-memory format—presumably, the actual information content will not be much higher despite the very horizontal pointer format.

While such compressed storage techniques are beyond the scope of this paper (but see [WLM91, Wil91]), we would like to make one observation about why these problems are not as difficult as they may seem at first glance: it is not necessary to store the compressed pages contiguously. Pages only need to be contiguous and "the right size" when they are made accessible to a program, so that hardware addressing works correctly. When stored in compressed format, they may be broken into smaller blocks to reduce fragmentation problems.

9 Related Work

While there are quite a few persistent storage systems (see, e.g., [DSZ90]), to the best of our knowledge only one uses virtual memory techniques to allow pointer swizzling at page fault time (avoiding continual runtime overhead) and supports a very large address space. That is ObjectStore, a commercially available system [LLOW91] from Object Design, Inc. Their system was designed independently of ours, and apparently predates it. We believe that it operates by similar means, but no details have been published.

Our scheme also bears some resemblance to the Moby address space designed and implemented by Richard Greenblatt at Lisp Machines Incorporated. Moby used pagewise relocation, but exploited the tagged architecture of the Lisp machine to swizzle pointers one at a time, rather than translating all of the pointers in a page with a single page protection fault. A similar scheme (designed and implemented by Bob Courts) was later used on the TI Explorer. Apparently, no papers were ever published about either of these systems.[16]

At a higher level of abstraction, pointer swizzling at page fault time is really a *compression* of the stream of addresses going by on the address bus (and also those within the pagewise wavefront around it). It is therefore philosophically akin to *dynamic base register caching* [FP91], but entirely different in its implementation. Our scheme incrementally constructs a compression table (mapping long persistent page numbers to short transient ones), which is rebuilt when its alphabet is exhausted (that is, when the transient page numbers are used up).

The pagewise wavefront of compression mappings is a purely pagewise analogue of Baker's "read barrier" technique for traversing objects in incremental tracing garbage collection [Bak78, Bak92, Wil92]. Our pagewise read barrier is an variation of Appel, Ellis, and Li's incremental garbage collection algorithm, which uses a pagewise wavefront with objectwise relocation.

10 Discussion

Page-fault-time swizzling is essentially a RISC addressing scheme, exploiting the common cases of programs' memory referencing behavior, i.e., mostly repeated touches to a subset of all pages; it provides extremely high performance in the usual case—with no impact on cycle time—and traps to software in the unusual cases. Ironically, it could have been used at any time in the last twenty or thirty years, to avoid the hardware and software woes of segmented architectures.

It can be argued that while it would have been a very good idea in the past, it will soon be unnecessary because 64-bit hardware addressing will become ubiquitous over the next few years. We think that it's still an excellent idea, for several reasons: the large installed base of narrow machines, the possibility that narrow machines may become necessary again if hardware trends change, and the likelihood that even 64 bits will not be enough. (We also think it's simply the right way to look at memory, freeing the notion of a pointer from its realization as a hardware address.)

The current trend toward 64-bit architectures may well be irreversible, and not as expensive as it might seem at first glance, because most of the costs of a chip are not directly proportional to address width. Still, we believe our scheme to be valuable even in a world dominated by 64-bit machines.

[16] Thanks to David Moon, Richard Greenblatt, George Carrette, and Doug Johnson for information about these systems.

32-bit machines will be manufactured for quite a while, if only for palmtop computers and embedded applications.[17] Their small address spaces should not prevent them from sharing a truly global address space with their larger cousins, perhaps over infrared links within a building, or via a cellular telephone network. Pointer swizzling can extend also the life of 64-bit architectures indefinitely, even if many bits are stolen from the address words and devoted to bit addressing, processor ID's, type tags, and so on.

As a reconciliation layer, swizzling can allow an indefinite number of machines (say, all the computers in the world) to share an indefinitely large address space, without each having to map potentially shared pages into the same place in each hardware-supported address space. And even the lowly embedded 32-bit single-chip computers of the coming decade should be able to share data with the largest 64-bit compute servers. And if current trends change toward lower-density, higher-speed materials (e.g., GaAs or Josephson junctions), word size may again become critical, and we can provide a graceful "downgrade" path.

In [CLLBH92], Chase *et al.* argue that 64-bit computers eliminate the need to play tricks with address spaces, either in the conventional way of interpreting addresses differently for each process, or by pointer swizzling. They point out that even if storage is used at a rate of 100 megabytes per second, it will take several thousand years to use up a 64-bit address space—2^{64} is 16 billion billion, and that's a very large number.

While we agree that a single global address space is very desirable, and greatly simplifies the design of sophisticated systems, we don't believe that 64 bits is enough in the long run, or that the global address space should be equated with conventional virtual memory addressing. (We also disagree with the contention that pointer swizzling interferes with sharing and protection; different protection domains on a node can share data pages—and swizzling mappings—while using different virtual memory protections for the swizzled pages.)

We believe it is possible, and even *likely* that 64 bit address spaces will be exhausted in the foreseeable future. Eventually, computers will encompass multimedia systems, and (for example) high-definition digital video data may be mapped into memory. HDTV information rates will be tremendous—consider megapixels per frame, in color, at tens of frames per second—and 100 MB per second ceases to seem all that unusual. Even with compression, a relatively small number of cameras could achieve that rate of data capture. (We also believe that the uncompressed image should be mapped into memory, even if the actual storage is in compressed form.) A single locally-networked site may therefore capture hundreds of megabytes of raw data per second; an international network of several thousands such sites could exhaust a 64-bit address space in a single year.

We also believe that it may be desirable to adopt a *no-reuse* policy toward virtual addresses—that is, when an object is deallocated, its (long, conceptual) address should not be reused. Just as [CLLBH92] advocates the separation of addressing from protection, we advocate the separation of addressing and protection from *storage*. Reusing memory should not necessarily entail reusing part of the global address space, even within a single thread of computation.

This separation of concerns could have several benefits. One would be that debugging long-lived processes would be much easier, because distinct objects would have distinct addresses over time—an address would not be reused for a new object. (This would be especially desirable in the context of a time-travel debugging system [WM89a, FB88, TA90].)

Along with this ability to distinguish distinct objects over time comes the possibility of detecting most uses of dangling pointers. If empty virtual pages are not reused, they can simply be access protected and have their storage reclaimed. An attempt to dereference a stale pointer will then cause an access protection fault, rather than silently operating on a completely different object.[18]

Many other uses of enormous address spaces are possible, and the cheap availability of indefinitely large spaces will probably encourage the development of many more. We are particularly interested in the lazy construction of large memory-mapped data structures, whose incremental materialization is triggered by access-protection faults. (For example, programs might see an area of address space as a gigabyte array, but the elements of the array would be computed on an as-needed basis, triggered by actual memory referencing.)

[17] The recent introduction of new "subnotebook" computers based on *16-bit* CPUs suggests that it will be quite a while before 64-bit machines take over the entire market. Even 16-bit machines could be supported with our scheme.

[18] This will not detect all uses of dangling pointers, because the page cannot be access protected until it is completely empty. We believe that the clustered births and deaths of objects [Hay91] will frequently cause entire pages to become empty, however; small pages and/or sub-page protections would also make this more effective.

11 Current Status and Future Work

Our initial prototype persistent store for Scheme [Wil91] lacked recovery features and a body of data-intensive programs to truly stress it, so was little more than an existence proof, showing that it is possible (and, in fact, easy) to perform address translation at page fault time. (Performance problems with our (bytecoded) Scheme system also prevented us from precisely measuring the overheads of pointer swizzling—the swizzling cost were too low compared to the overhead of interpretation.)

We have recently constructed a new, highly-portable persistent store for C++, called Texas [SKW92]. The current implementation, running on unmodified ULTRIX, incurs unnecessary costs due to difficulties in avoiding allocating backing store and bypassing file system caching. Despite these flaws, we have run a few benchmarks that suggest that our pointer swizzling techniques perform well.[19]

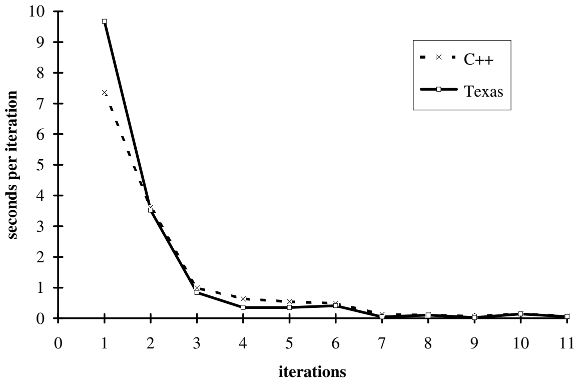

Figure 2: Small OO1 database traversal times

Figure 2 shows the results of running a simple benchmark with our persistent store, and with non-persistent C++. (In both cases, the test system is a DECstation 5000/200 with 16MB main memory. The virtual memory system and file caches were flushed to disk by reading and writing unrelated data before the start of the benchmark.)

The plots show the time taken to perform each of 10 iterations of the "traversal" component of the OO1 benchmark [Cat91], using the standard small OO1 test database. The database consists of 20,000 "part" objects, representing parts in a hypothetical engineering database application. The parts are indexed by part number, with the index implemented as an AVL tree[20]. Each part object also has pointers to several other parts that it is conceptually "connected" to, and a pointer to a set object that holds backpointers to objects that are connected to it. There is locality in the connections, based on mostly nearby part numbers, and some randomness as well.

The standard benchmark searches the database several times in the following way: a part number is selected at random, and then its connections to other parts are traversed recursively to find three other parts. This traversal continues transitively for seven levels (with a branching factor of three), touching a total of 3280 connected parts.

The graph shows the times to perform successive traversals, each starting from a different randomly-selected part and tracing out its connections.

Our persistent C++ system (Texas) outperforms non-persistent C++ in most of these early iterations, which should not be possible, but this is apparently an artifact of the fetch policy used in the underlying file system. (We fetch faulted-on pages from the persistent store, which is implemented as file data. C++ simply faults them in from virtual memory as 4KB pages.)

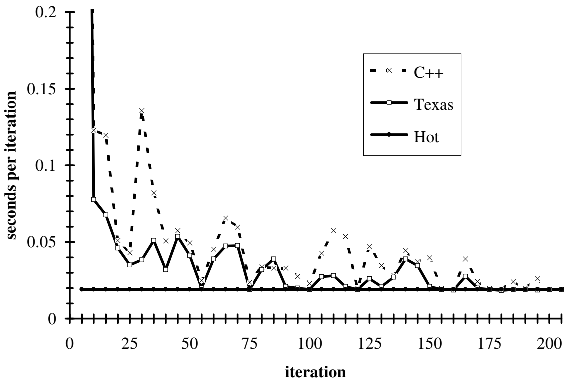

Figure 3: More iterations traversing small database

Figure 3 shows what happens when more iterations of the benchmark are performed. Both C++ and Texas approach the same speed as the virtual memory "warms up", i.e., as more and more pages have been faulted into main memory. This is to be expected, since Texas incurs little or no overhead except at page fault time. Following White and Dewitt [WD92], we also show the "hot" traversal time, traversing only

[19] We are currently implementing several improvements; these will be reported in [SKW92], including with details of our new log-structured storage manager and its effects on checkpointing costs.

[20] We used the standard AVLMap template (parameterized) class from the Free Software Foundation's GNU C++ library. Only the calls to **new()** were changed, so that objects would be allocated on the persistent heap

pages that are already in memory by repeating previous iterations.

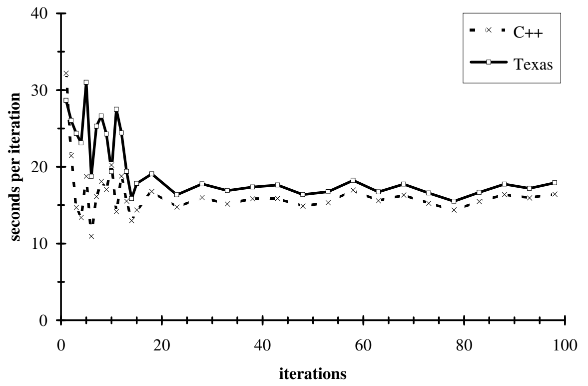

Figure 4: OO1 Large database traversal times

Figure 4 shows the same kind of traversal performed against the large OO1 database (including 200,000 part objects), which will not fit in memory [Cat91]. The startup effects of filesystem fetching quickly diminish, and the performance of both systems is dominated by virtual memory paging costs. After the first 30 iterations or so, the performance of Texas is about 93% of the performance of C++. Again, the hot times were essentially identical, at about 19ms. This indicates that the time overhead of our system is nearly zero in a large main memory, at least when address space reuse is not necessary during the execution of a program.

We should stress that the above measurements are very preliminary—they are the first measurements we've made, of an unoptimized, untuned system. Still, they suggest that pointer swizzling at page fault time will be quite efficient for many applications, and in particular that it is competitive with a conventional virtual memory.

Conclusions

Pointer swizzling at page fault time can support enormous address spaces on standard hardware, using only commonly available compilers and operating sytem features. We believe it can be an important part of a radical and long-overdue reorganization of the relationships between software and hardware, with a high-performance RISC approach to address translation. Nonetheless, it is of immediate practical value given existing hardware, operating systems and compilers, as a reconciliation layer between otherwise incompatible system components and abstractions.

Acknowledgements

We are grateful to Seth White, who provided the OO1 benchmark code, and especially to Vivek Singhal, who implemented much of the Texas persistent store.

References

[ABC+83] M.P. Atkinson, P.J. Bailey, K.J. Chisholm, P. W. Cockshott, and R. Morrison. An approach to persistent programming. *The Computer Journal*, 26(4):360–365, December 1983.

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

[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 Operating Systems (ASPLOS IV)*, pages 96–107, April 1991. Santa Clara, CA.

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

[Bak92] Henry G. Baker, Jr. The Treadmill: Real-time garbage collection without motion sickness. *ACM SIGPLAN Notices*, 27(3):66–70, March 1992.

[Bar88] Joel F. Bartlett. Compacting garbage collection with ambiguous roots. Technical Report 88/2, Digital Equipment Corporation Western Research Laboratory, Palo Alto, California, February 1988.

[BMBC91] Hans-Juergen Boehm, Eliot Moss, Joel Bartlett, and David Chase. Panel discussion: Conservative vs. accurate garbage collection. (Summary appears in Wilson and Hayes' OOPSLA '91 GC workshop report.), 1991.

[Boe91] Hans-Juergen Boehm. Hardware and operating system support for conservative garbage collection. In *International Workshop on Memory Management*, pages 61–67, Palo Alto, California, October 1991. IEEE Press.

[BW88] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, September 1988.

[Cat91] R. G. G. Cattell. An engineering database benchmark. In Jim Gray, editor, *The Benchmark Handbook for Database and Transaction Processing Systems*, pages 247–281. Morgan-Kaufman, 1991.

[CLLBH92] Jeffrey S. Chase, Henry M. Levy, Edward D. Lazowska, and Miche Baker-Harvey. Lightweight shared objects in a 64-bit operating system. In *ACM SIGPLAN 1992 Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '92)*, Vancouver, British Columbia, Canada, October 1992.

[Det90] David L. Detlefs. Concurrent garbage collection for C++. Technical Report CMU-CS-90-119, Carnegie-Mellon University, May 1990.

[Det92] David L. Detlefs. Garbage collection and runtime typing as a C++ library. In *USENIXC++ Conference*, Portland, Oregon, August 1992.

[DMH92] Amer Diwan, Eliot Moss, and Richard Hudson. Compiler support for garbage collection in a statically-typed language. In *SIGPLAN Symposium on Programming Language Design and Implementation*, pages 273–282, San Francisco, California, June 1992.

[DSZ90] Alan Dearle, Gail M. Shaw, and Stanley B. Zdonik, editors. *Implementing Persistent Object Bases: Principles and Practice (Proceedings of the Fourth International Workshop on Persistent Object Systems)*. Morgan Kaufman, Martha's Vineyard, Massachusetts, September 1990.

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

[FP91] Matthew Farrens and Arvin Park. Dynamic base register caching: A technique for reducing address bus width. In *Proc. 18th Annual Int'l Symposium on Computer Architecture*, pages 128–137, May 1991. Toronto, Canada.

[Gri89] Knut S. Grimsrud. Multiple prefetch adaptive disk caching with strategic data layout. Master's thesis, Brigham Young University, December 1989.

[Hay91] Barry Hayes. Using key object opportunism to collect old objects. In *ACM SIGPLAN 1991 Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '91)*, pages 33–46, Phoenix, Arizona, October 1991. ACM Press.

[HCU92] Urs Hoelzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. In *SIGPLAN Symposium on Programming Language Design and Implementation*, pages 32–43, San Francisco, California, June 1992.

[Kae81] T. Kaehler. Virtual memory for an object-oriented language. *Byte*, 6(8):378–387, August 1981.

[Lev84] Henry M. Levy. *Capability-Based Computer Systems*. Digital Press, Bedford, Massachusetts, 1984.

[Li86] Kai Li. *Shared Virtual Memory on Loosely-Coupled Processors*. PhD thesis, Yale University, 1986.

[LLOW91] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore database system. *Communica-*

tions of the ACM, 34(10):50–63, October 1991.

[MG89] Jose Alves Marques and Paulo Guedes. Extending the operating system to support an object-oriented environment. In *ACM SIGPLAN 1989 Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '89)*, pages 113–122, New Orleans, Louisiana, October 1989.

[Mos90] J. Eliot B. Moss. Working with objects: To swizzle or not to swizzle? Technical Report 90–38, University of Massachusetts, Amherst, Massachusetts, May 1990.

[RO91] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *Thirteenth ACM Symposium on Operating Systems Principles*, pages 1–15, Pacific Grove, California, October 1991.

[Ros91] John Rosenberg. Architectural support for persistent objects. In *1991 Int'l Workshop on Object Orientation in Operating Systems*, pages 48–60. IEEE Computer Society Press, October 1991. Palo Alto, California.

[SKW92] Vivek Singhal, Sheetal V. Kakkad, and Paul R. Wilson. Texas: an efficient, portable persistent store. In *Fifth International Workshop on Persistent Object Systems*, Pisa, Italy, September 1992.

[Sta82] James William Stamos. A large object-oriented virtual memory: Grouping strategies, measurements, and performance. Technical Report SCG-82-2, Xerox Palo Alto Research Centers, Palo Alto, California, May 1982.

[TA90] Andrew Tolmach and Andrew Appel. Debugging Standard ML without reverse engineering. In *SIGPLAN Symposium on LISP and Functional Programming*, 1990.

[WD92] Seth J. White and David J. Dewitt. A performance study of alternative object faulting and pointer swizzling strategies. In *18th International Conference on Very Large Data Bases*, Vancouver, British Columbia, Canada, October 1992. To appear.

[WH91] Paul R. Wilson and Barry Hayes. The 1991 OOPSLA Workshop on Garbage Collection in Object Oriented Systems (organizers' report). In *Addendum to the proceedings of OOPSLA '91*, Phoenix, Arizona, 1991.

[Wil91] Paul R. Wilson. Operating system support for small objects. In *International Workshop on Object Orientation in Operating Systems*, Palo Alto, California, October 1991. IEEE Press. Revised version to appear in *Computing Systems*.

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In *International Workshop on Memory Management*, St. Malo, France, September 1992. Springer-Verlag Lecture Notes in Computer Science series.

[WLM91] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Effective static-graph reorganization to improve locality in garbage-collected systems. In *SIGPLAN Symposium on Programming Language Design and Implementation*, pages 177–191, Toronto, Canada, June 1991.

[WM89a] Paul R. Wilson and Thomas G. Moher. Demonic memory for process histories. In *SIGPLAN Symposium on Programming Language Design and Implementation*, pages 330–343, Portland, Oregon, 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 *ACM SIGPLAN 1989 Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '89)*, pages 23–35, New Orleans, Louisiana, October 1989.

[WWH87] Ifor W. Williams, Mario I. Wolkzko, and Trevor P. Hopkins. Dynamic grouping in an object-oriented virtual memory hierarchy. In *1987 European Conference on Object-Oriented Programming*, pages 87–96, Paris, France, June 1987. Springer-Verlag.