Object Fault Handling for Persistent Programming Languages: A Performance Evaluation*

Antony L. Hosking J. Eliot B. Moss

Object Systems Laboratory
Department of Computer Science
University of Massachusetts
Amherst, MA 01003

Abstract

A key mechanism of a persistent programming language is its ability to detect and handle references to non-resident objects. Ideally, this mechanism should be hidden from the programmer, allowing the transparent manipulation of all data regardless of its potential lifetime. We term such a mechanism *object faulting*, in a deliberate analogy with page faulting in virtual memory systems. This paper presents a number of mechanisms for detecting and handling references to persistent objects, and evaluates their relative performance within an implementation of Persistent Smalltalk.

1 Introduction

Persistent programming languages combine the features of database systems and programming languages to allow the seamless manipulation of data, without regard for its potential lifetime, be it transient or persistent [1]. To achieve this the language must provide a mechanism for the detection and handling of references to persistent data. Ideally, this mechanism should be hidden from the programmer, so that manipulation of persistent and non-persistent data is as transparent as possible. The term we use for such a mechanism is *object faulting* [9, 10]. The analogy with page faulting virtual memory is deliberate, since the intent is to provide the illusion of a persistent virtual heap of objects, potentially much larger than physical or even virtual memory. Access to those objects is detected and managed by the object faulting mechanism, which triggers automatic retrieval of objects from persistent storage (i.e., disk) on demand. In effect, persistent objects are cached in memory for manipulation by the program.

This paper considers a number of implementations of object faulting. We divide our attention between the mechanism by which references to non-resident objects are detected, and the way in which the object faults themselves are handled. We compare several schemes for detecting references to non-resident objects, not only through checks in software, but also by exploiting the page protection mechanism of the operating system to detect non-residency through the trapping of references to non-resident objects. We also explore an orthogonal design choice: just how many objects should be made resident per object fault? Naturally, faulting on a given object must make at least that object available to the program, however any number of additional objects might also be made available. Moreover, making one object resident may *require* that other objects also be resident. Such constraints must be observed by the object fault handler before program execution can resume. The advantage of faulting more than one object per object fault is straightforward: it may reduce the number of object faults required for execution of a given program. Yet it may also result in more data being made available to the program than is absolutely necessary for its execution.

In addition to the comparison of alternative implementations of object faulting, this paper's contributions include the description of our architecture and frame-

*This work is supported by National Science Foundation Grants CCR-8658074 and CCR-9211272, Digital Equipment Corporation's Western Research Laboratory and Systems Research Center, and Sun Microsystems. The authors can be reached via Internet addresses {hosking,moss}@cs.umass.edu.

Proceedings ACM Conference on Object-Oriented Programming Systems, Languages, and Applications
Washington, DC, Sep. 1993, pp. 288–303

work for persistence, and the performance evaluation, using established benchmarks, of a prototype persistent programming language implemented within this framework.

The remainder of the paper is organized as follows. The next section surveys related work, distinguishing this study from previous ones. Succeeding sections describe our system architecture and rationale, the competing implementations of object faulting, the benchmarks used for their comparison, the experimental setup, methodology, results, and our conclusions.

2 Related work

Many systems have attempted to extend the address space of programs beyond that which can be addressed directly by the available hardware. Virtual memory represents the extension of the memory address space of a program beyond that of physical memory. Virtual address translation allows transparent access to data regardless of its physical memory location. The operating system is responsible for trapping references to pages that are not resident in physical memory, whereupon the non-resident page is fetched from disk before the process which caused the fault can resume execution.

Object-oriented programming languages typically allow the dynamic allocation of data objects in a *heap*. Objects are referred to via unique *object identifiers* (OIDs), just as virtual memory is referred to via virtual addresses. If the entire heap can fit in virtual memory then OIDs can be represented as direct virtual memory pointers. However, this limits the size of the heap to that of the virtual address space. Extending the size of the heap beyond that which can be addressed in virtual memory requires OIDs that are not virtual memory addresses. Such systems must ultimately perform translation of OIDs to in-memory pointers to allow the program to manipulate the data.

LOOM [12, 11] represents one of the earliest attempts to extend the size of the heap beyond that addressable by a machine word. Its goal was to provide extended virtual memory support for Smalltalk systems on machines with a narrow (16-bit) word width. Object pointers are stored in 32 bits on disk, and an object table is used to translate between the short and long forms. When an object is brought into memory, its 32-bit persistent pointer is hashed to find an entry for it in the object table. All in-memory references to the object are then indirected through its object table entry. If the object contains references to other objects then they must be converted to short form, in a process which has since been termed *swizzling*. References to resident objects are converted to their in-memory short object pointer. References to non-resident objects are represented either as an in-memory pointer to a *leaf* or as a *lambda*. A leaf is a resident proxy object that represents an object on disk, containing sufficient information to locate that object. A lambda is a place-holder (actually the null pointer, 0) for a pointer to an object that has not yet been assigned a short pointer. Suppose a particular object *O* has a field which contains lambda. To determine the object to which that field refers means accessing the long form of *O* stored on disk to obtain the long address.

Object faults in LOOM are triggered via explicit checks, isolated to certain operations in the Smalltalk interpreter. Leaves are distinguished by a bit in their object table entry. Objects containing lambdas are also specially marked. An object fault is handled by retrieving the object from disk and converting it to its in-memory format. This object-at-a-time transfer of objects between memory and disk is actually the major downfall of the LOOM system, since Smalltalk objects are too small a unit for transfer. The implementors of LOOM acknowledge this and speculate on refinements to their system that would group objects together for transfer between memory and disk.

The Alltalk system [22] takes a similar approach to LOOM in its implementation of a *persistent* Smalltalk system, using an object table to translate between object pointers and memory addresses. Alltalk performs no swizzling: object pointers are always external identifiers that must be translated whenever they are dereferenced.

GemStone [17], is another effort to expand the Smalltalk heap to include objects on disk. However, it extends Smalltalk to provide considerable database functionality, including queries and a query execution model. Gemstone's integration with Smalltalk systems is not totally "seamless," since the virtual image is modified to include *proxy* objects that act as forwarders to GemStone objects. Proxies, because they are full-blown objects in the virtual image, are thus fully visible to applications programmers. Moreover, they carry the additional burden of delivering much of the database

functionality supported by GemStone.

Each of these systems have extended Smalltalk in some way to provide some form of persistence. None of them consider the performance overheads of persistence, accepting the costs as necessary to support the functionality they desire. Here, we are interested in exploring the design space for implementing persistent programming languages, by evaluating the performance of a number of mechanisms for object faulting.

White and DeWitt [26] have compared the overall performance of a number of architectures and systems that perform object faulting and pointer swizzling. The systems considered in that study include ObjectStore [13, 16], a commercially available object-oriented DBMS, and a number of software architectures based on the EXODUS Storage Manager (ESM) [2, 20].

Several of the architectures based on ESM require the program to manipulate objects through a call interface, with modifications being performed in the client buffer pool of ESM, as opposed to the virtual memory space of the application. White and DeWitt introduce a new scheme (EPVM 2.0), which avoids this call overhead through *object caching*. Objects are still retrieved into the client buffer pool using the ESM interface. However, they are then copied into the virtual memory of the application, while the originals in the buffer pool are unpinned. Modifications can then be made directly in virtual memory. At transaction commit, for each modified object in virtual memory the corresponding original is pinned and updated in the ESM buffer pool through a call to ESM. White and DeWitt explored two versions of this caching scheme. The first copies objects one at a time from the buffer pool into virtual memory as they are accessed by the application. The second copies all of the objects on a given page of the buffer pool when the first object on the page is accessed.

White and DeWitt's object caching scheme also performs some pointer swizzling, in which references to objects that are resident in the cache are converted to direct memory pointers. Each object includes a bit table indicating which of its slots contain direct pointers and which contain unswizzled OIDs. Translating an OID means probing a hash table containing pointers for all cached objects, and caching the object if it is not already resident. EPVM 2.0 performs swizzling upon *discovery*: when a location containing an unswizzled reference to a persistent object is discovered (usually as a result of loading the reference to perform some operation on it) the location is updated with a direct pointer to the object.

ObjectStore, the final architecture considered by White and DeWitt, takes a dramatically different approach. Objects are faulted and pointers are swizzled using a page mapping scheme similar to virtual memory. We do not have exact details of the proprietary mechanisms for object faulting and swizzling, but the approach is similar to that used in the Texas system, described in more detail below.

The results obtained by White and DeWitt indicate that object caching is an attractive architecture for persistent programming languages. For small databases, in which the entire database can fit in main memory, caching objects a page at a time seems best, since there is little extra overhead in copying pages versus objects, with fewer copying operations being needed. However, for larger databases that do not fit in main memory, page caching will copy some objects unnecessarily. This results in *double paging*: pages are first cached in virtual memory by the object caching mechanism, and then paged out by the virtual memory manager.

The comparison with ObjectStore produced mixed results. Cold database performance (obtained by running benchmarks against a database that starts out entirely on disk at the possibly remote database server) was worse for ObjectStore than for the architectures based on ESM. For a small database ObjectStore exhibited the best warm performance; for the large database its performance was the worst. White and DeWitt suggest that these results indicate the cost of mapping data into a process's address space. We speculate that it is also due to the high overhead of fielding page protection traps from the operating system to fault non-resident pages.[1]

In contrast to White and DeWitt, who consider the *overall* performance of several different architectures, we have chosen to keep our basic architecture fixed while varying the mechanisms used to detect and handle object faults. Our architecture, as described in the next section, is similar to the object caching architecture of White and DeWitt. However, the representations we use for references to non-resident objects are much more lightweight than those of White and DeWitt, as are the mechanisms we use for fault detection.

[1] $\sim 250\mu$s round trip as measured in a tight loop under Ultrix 4.1 on the DECstation 3100. We note that this is generally acknowledged to be one of the best operating system implementations for trapping page protection faults.

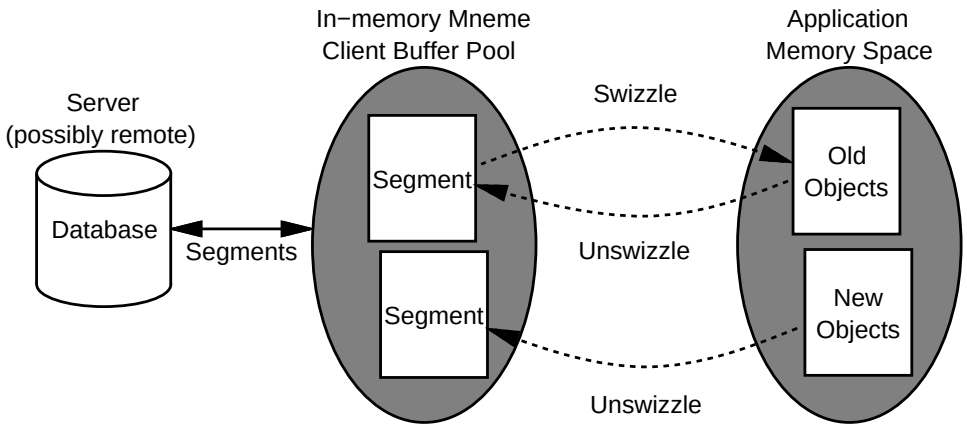


Figure 1: System architecture

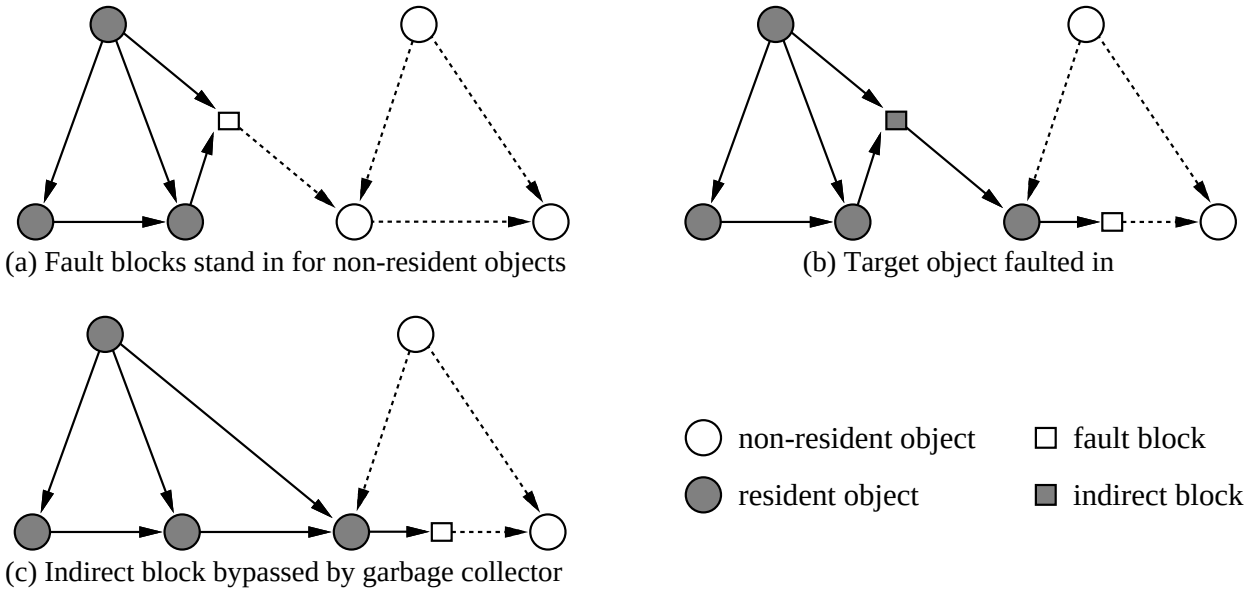
The Texas system [21, 27] uses a page mapping scheme similar to ObjectStore to fault objects and swizzle pointers. When a persistent object is to be assigned a virtual address, a page of virtual memory is reserved (and access protected) for the page in the persistent store that contains the object. The offset of the object in the persistent page is known, allowing the virtual address of the object in the reserved virtual memory page to be calculated. Accessing the page triggers a virtual memory page trap. Texas handles this trap by reading in the persistent page from the store and mapping it into the previously reserved virtual page. All pointers in that page are then swizzled by reserving virtual memory pages for the objects to which they refer (assuming the referenced pages are not already mapped into virtual memory). The persistent references can then be replaced with virtual memory addresses, the faulted page is unprotected, and execution resumes. As execution proceeds, pages are reserved in a "wave-front" just ahead of the most recently faulted and swizzled pages, guaranteeing that the application will only ever see virtual memory addresses.

Wilson and Kakkad [27] report promising preliminary performance results for an implementation of persistent C++ using Texas. The beauty of Texas is that it requires little or no modification to an existing language to support persistence. As we have already indicated, fielding a page protection trap from the operating system is an expensive operation. Whether software-mediated object faults (realized by augmenting the programming language implementation) can offer competitive performance is a question we explore here.

3 System architecture and rationale

Our architecture (see Figure 1) bears a close resemblance to the object caching architecture of White and DeWitt [26]. Objects are copied on demand into the virtual memory address space of the program from the buffer pool of the persistent storage manager, in this case the Mneme persistent object store [14]. This copying includes any translation needed to convert the objects into a form acceptable to the program, including pointer swizzling. Our choice of such an architecture was driven by a desire to give the language implementation maximum control over all objects being manipulated by an application, without having to go through a restrictive interface to the underlying storage manager. In particular, standard programming language techniques for memory management, including those of garbage collection, can be used to manage the objects resident in the program's virtual address space [8].

The unit of transfer between the permanent database and Mneme's buffers is the *physical segment*, which may have arbitrary size (up to some large system-defined limit). Thus a physical segment may contain any number of objects. Objects within a physical segment are further grouped into *logical* segments. A logical segment may contain at most 255 objects; all logical segments within a physical segment must be full, except possibly the last, in which new objects are allocated. Grouping of objects for transfer between disk and memory eliminates the performance bottleneck experienced by LOOM, which retrieved objects one at a time.



(a) Fault blocks stand in for non-resident objects

(b) Target object faulted in

(c) Indirect block bypassed by garbage collector

Figure 2: Node marking

3.1 Detecting object faults

As mentioned previously, object faulting requires some mechanism to distinguish between references to resident and non-resident objects. These mechanisms may be loosely divided into two categories, depending on the strategy they adopt. For the purposes of this discussion we view the persistent heap as a *directed graph*: the objects are the *nodes* and the references between the objects are the *edges*.

Edge marking schemes take the approach of tagging the references between the objects. If tagged as *swizzled*, then a reference is a direct pointer to the corresponding object in memory; if *non-swizzled* then the reference consists of an OID. This is the approach used by EPVM 2.0 [26]. An apparent disadvantage of edge marking is that OIDs can be fetched from the pointer fields of objects, passed around, and stored, without accessing the target object. When the target object finally is accessed the origin of the reference may no longer be known. White and DeWitt got around this through swizzling upon discovery (when a reference is loaded from a location), assuming that the load is a precursor to performing some operation on the target object. However, their solution may swizzle too eagerly, since the ultimate reason for loading a reference cannot always be determined at the time of the load.

Node marking schemes require that all object references in resident objects be converted to pointers. In ObjectStore and Texas this is achieved by reserving (although not necessarily allocating) virtual pages for the objects referred to by the pointers, and protecting those pages to trap all access to those pages. Another approach, similar to LOOM's leaf objects, is to have small proxy objects (we call them *fault blocks*) stand in for non-resident objects, as illustrated in Figure 2(a). A fault block contains the OID of the target object, and is distinguishable from an ordinary object. Whenever a reference is followed, if it refers to a fault block, then the target object is made resident (copied and swizzled as necessary). The fault block is changed to point to the now-resident object (see Figure 2(b)). We call the updated fault block an *indirect block*. If a reference to be followed refers to an indirect block then the target object can be located at the cost of an indirection. Occasional scanning (possibly by a garbage collector) can be used to bypass indirect blocks, as shown in Figure 2(c).

References to tagged OIDs and fault blocks may be detected via explicit checks upon pointer dereference. Alternatively, fault blocks can be allocated in protected virtual memory pages, so that dereferencing a pointer to a fault block is trapped, and handled by making the target object available. Another approach is to exploit the indirection implicit in the method invocation schemes

of object-oriented programming languages, folding residency checks into the overhead of method invocation (this approach is used to good effect in the persistent Smalltalk system used for this study, and will be described in detail in the next section).

3.2 Swizzling

When an object is made resident its pointer fields are swizzled according to the mechanism being employed for fault detection. All fields referring to objects that are already resident are converted to point directly to those objects—Mneme supports this mapping efficiently with a hash table. Otherwise, for edge marking we convert the reference to a tagged OID; for node marking, the reference is converted to point to a fault block for the non-resident object (a fault block is allocated if one does not yet exist for the target object).

The architecture leaves open the possibility of copying and swizzling any number of objects at one time from the Mneme buffer pool into memory. For this study we consider the granularities naturally inherent in this architecture: individual objects, logical segments, and physical segments. Swizzling just one object at a time has the advantage of copying and swizzling only those objects needed immediately by the program for it to continue execution. This will serve to minimize object fault latencies (including swizzling), as well as memory consumption.

Swizzling a logical or physical segment at a time may take advantage of any clustering present in the physical layout of objects in the database. Since all the objects in a segment are mapped before they are swizzled, any *intra*-segment references will be converted to direct pointers. If the static clustering is a good approximation to the dynamic locality of access by the program then the speed of program execution will improve since fewer object faults will occur.

4 Persistent Smalltalk

The prototype persistent programming language used for these experiments is an implementation of Smalltalk with extensions to support persistence. The underlying permanent storage is managed by the Mneme persistent object store [14]. Our Smalltalk implementation is based on the definition of Goldberg and Robson [6], and consists of two components: a *virtual machine* and a *virtual image*. The virtual machine implements a bytecode instruction set to which Smalltalk source code is compiled, as well as other primitive functionality. While we have retained the standard bytecode instruction set of Goldberg and Robson [6], our implementation of the virtual machine differs somewhat from their original definition.

The virtual image is derived from an early commercial version of Smalltalk with minor modifications. It implements (in Smalltalk) all the functionality of a Smalltalk development environment, including editors, browsers, the bytecode compiler, class libraries, etc., all of which are first-class objects in the Smalltalk sense. Booting a Smalltalk environment involves loading the virtual image into memory for execution by the virtual machine.

Our persistent implementation of Smalltalk places the virtual image in a Mneme database, and the Smalltalk environment is booted by loading that subset of the objects in the image sufficient to resume execution by the virtual machine. We have retained the original bytecode instruction set, and changes to the virtual image have been minor. Rather, all extensions for persistence have been to the virtual machine, which has been carefully augmented to make persistent objects resident as they are needed by the executing image.

4.1 Object faulting

Computation in Smalltalk proceeds by sending *messages* to objects. A message consists of a *message selector* and a number of arguments. The effect of sending a message is to invoke a *method* on the receiver of the message. Invoking a method may be thought of as a procedure call. The method to be executed is determined dynamically, based on the message selector and the class of the receiver. Every class object in Smalltalk has a pointer to a *method dictionary* which associates selectors with *compiled methods*. A compiled method consists of the bytecodes that implement the method, along with a *literal frame*, containing the shared variables, constants, and message selectors used by the method's bytecodes. Determining which method to execute when a message is sent proceeds as follows. The receiver's class is checked to see if its method dictionary contains the message selector. If it does then the corresponding compiled method is invoked. Otherwise,

the search continues in the superclass of the object, and so on, up the class hierarchy. If no matching selector is found then a run-time error is signalled.

As described so far, the method lookup process is very expensive. To reduce this lookup cost a method lookup cache is used. Entries in the cache store a selector, class, and compiled method. Before proceeding to a full method lookup, the selector and class are hashed to index an entry in the cache. If the selector and class of the cache entry match those of the message send, then the compiled method has been found. If they do not, then a full lookup takes place, updating the corresponding cache entry as well.

Our discussion of message sends has illustrated just how many objects must be accessed as computation proceeds. For performance reasons it is crucial that the bytecode interpreter not perform a residency check for every object reference it must follow. To overcome this we impose certain residency constraints on critical objects, restricting residency checks to message sends as follows.

Because computation is driven by the sending of messages, most objects will become resident only when a message is sent to them. The send bytecodes must load the receiver's class for method lookup. When an object is made resident, we require that its class also be resident, so that its class field can be swizzled to a direct pointer. In this way we eliminate the need for a residency check on the class when probing the method lookup cache.

4.1.1 Edge marking

Smalltalk implementations typically avoid allocating individual objects for such things as integers by tagging object pointers, and representing the integer value directly in the tagged pointer.[2] Such objects have been termed *immediate*, since their value may be obtained immediately from their object reference. To cope with this, message sends must always check the pointer tag of the receiver. Immediate values are mapped to their class based on the tag, rather than by dereferencing the object pointer to obtain the class.

For edge marking, references to non-resident objects are represented as tagged immediate OIDs,[3] which we map to a special "class" (represented by the null pointer), whose only "method" primitively responds to all messages by faulting the target object and forwarding the message to it. Since the method lookup cache is loaded with this response the first time a message is sent to an OID, subsequent message sends can proceed without an explicit residency check. Only the *full* method lookup must deal with the case when the class is null, priming the method cache appropriately.

4.1.2 Node marking

We use a similar trick to obtain check-free message sends for node marking. Fault and indirect blocks are distinguished from other objects by their "class" field, which instead of containing a direct pointer to some class, contains a tagged OID or indirect pointer instead.[4] Similarly to our implementation of edge marking, we arrange for fault blocks to respond to all messages by faulting the corresponding object and forwarding the message to the now-resident object. Once again, only the *full* method lookup performs residency checks to detect fault and indirect blocks, priming the method cache appropriately so that all future sends to the fault or indirect block will occur without additional checks.

Our implementation of the page protection variation for fault blocks achieves the same effect, but makes sure that the virtual machine sees only resident objects. Loading the "class" of a fault or indirect block will cause a trap. The trap handler unprotects the pages containing fault and indirect blocks, overwrites the offending fault block with an indirect block, and arranges for the load instruction that caused the fault to be restarted with a direct pointer to the resident object. The fault and indirect block pages are then reprotected before resuming execution in the virtual machine.

In addition to elimination of indirections by the garbage collector, a fault block implementation can be more aggressive in its elimination of indirections. At each object fault our system scans all transient (i.e., non-persistent) objects (including active stack frames) to eliminate any references to fault blocks that have been converted to indirect blocks. We also maintain a *remembered set* [24, 25] for each page of allocated fault blocks, recording all persistent objects whose pointer

[2] We use an immediate representation for SmallInteger, Character, nil, true and false.

[3] Mneme OIDs are only 28 bits, leaving plenty of room for the tag on a 32-bit machine.

[4] Mneme's 28-bit OIDs allow us to keep the size of fault blocks to 32 bits.

fields have been swizzled to refer to a fault block in the page. At each object fault the objects in the remembered set are scanned, and any fields that contain pointers to (ex-fault) indirect blocks are updated to bypass the indirection. In this way the source locations of fault block references are swizzled, so avoiding repeated loading and faulting on those references, without having to adopt the over-eager swizzle-on-discovery approach of White and DeWitt. We expect this to be particularly important for the page protection variant, by preempting unnecessary expensive page traps.

4.1.3 Residency constraints

In addition to the constraint that an object must always contain a direct pointer to its class, we impose further restrictions to elide other residency checks in the bytecodes of the virtual machine. Whenever a byte-compiled method is made resident (usually through its invocation), we make the literals in its literal frame resident along with it. This forces the selectors, constants, and shared variables[5] referred to by the bytecodes to be resident. It does not force the objects *referred to* by the shared variables to be resident. This permits the bytecodes accessing the selectors, constants, and shared variables of the literal frame to do so without performing residency checks. In short, there is no need for residency checks in the stack bytecodes. Stack frames are also objects in the Smalltalk system, and so may be persistent. Requiring all stack frames of an active process to be resident further eliminates residency checks in the return bytecodes.

In summary, by preloading objects that are critical to the forward progress of computation, we are able to restrict all residency checks to message sends.[6]

5 Experiments

We compared several versions of the virtual machine, varying the schemes for object fault detection (tagged OIDs, fault blocks, and page protection), the granularity of swizzling (object, logical segment, and physical segment at a time), and whether the virtual machine is running against a completely resident virtual image (ordinary non-persistent Smalltalk) or against an image that is faulted in on demand (persistent Smalltalk). Table 1 enumerates the variants.

As mentioned earlier, our fault block schemes (FB and PF variants) eliminate indirections at each object fault by scanning transient space, and processing the remembered set of the page containing the faulted-on fault block. We apply this technique in the explicitly checked FB schemes as well as their page-trapping PF counterparts, in order to obtain a straight comparison. This is despite the fact that the explicitly checked FB schemes can cheaply bypass indirections as they are encountered, while the scanning and remembered set processing adds substantial additional overhead at each fault. In contrast, the page-trapping PF schemes must be aggressive in eliminating indirections, since the indirect blocks reside in protected pages, to which any access will be trapped.

5.1 The benchmark database

Our benchmarks are drawn from the OO1 object operations benchmarks [3]. The OO1 benchmark database consists of a collection of 20,000 "part" objects, indexed by part numbers in the range 1 through 20,000, with exactly three "connections" from each part to other parts. The connections are randomly selected to produce some locality of reference: 90% of the connections are to the "closest" 1% of parts, with the remainder being made to any randomly chosen part. Closeness is defined as parts with the numerically closest part numbers. The part database and the benchmarks are implemented entirely in Smalltalk, including the B-tree used to index the parts.

The Mneme database, including the Smalltalk image as well as the parts data, consumes 179 physical segments, for a total size of just over 6 Mbytes. Each physical segment is at least 32 Kbytes in size, although some may be larger since Smalltalk objects larger than 32 Kbytes are allocated in their own private segment. There are on average three or four logical segments per physical segment. Newly created objects are clustered into segments only as they are encountered when unswizzling, using an essentially breadth-first traversal similar to that of copying garbage collectors [4]. The part objects are 68 bytes in size (including the object header). The three outgoing connections are stored di-

[5] Shared variables are represented as Association objects with two fields, one for a name and one for a value.

[6] Primitive methods must perform additional residency checks on any objects they need to access other than the receiver of the message.

Variant	Description
non-persistent	Non-persistent
ID-resident	Non-persistent, augmented with checks needed for tagged OIDs
FB-resident	Non-persistent, augmented with checks needed for fault blocks
PF-resident	Non-persistent, augmented with the page trap handling code, plus necessary support to decode load instructions that might cause a trap
ID-OBJ	Persistent, tagged OIDs, swizzle 1 object at a time
ID-LSEG	Persistent, tagged OIDS, swizzle 1 logical segment at a time
ID-PSEG	Persistent, tagged OIDs, swizzle 1 physical segment at a time
FB-OBJ	Persistent, fault blocks, swizzle 1 object at a time
FB-LSEG	Persistent, fault blocks, swizzle 1 logical segment at a time
FB-PSEG	Persistent, fault blocks, swizzle 1 physical segment at a time
PF-OBJ	Persistent, fault blocks allocated in protected pages, swizzle 1 object at a time
PF-LSEG	Persistent, fault blocks allocated in protected pages, swizzle 1 logical segment at a time
PF-PSEG	Persistent, fault blocks allocated in protected pages, swizzle 1 physical segment at a time

Table 1: Schemes measured in experiments

rectly in the part objects. The string fields associated with each part and connection are represented by references to separate Smalltalk objects of 24 bytes each. Similarly, a part's incoming connections are represented as a separate object containing references to the parts that are the source of the connections. The B-tree index for the 20,000 parts consumes around 165 Kbytes.

5.2 Benchmarks

We used the Lookup and Traversal portions of the OO1 benchmarks, which operate as follows:

- **Lookup** fetches 1,000 randomly chosen parts from the database. For each part a null procedure is invoked, taking as its arguments the *x*, *y*, and *type* fields of the part.
- **Traversal** fetches all parts connected to a randomly chosen part, or to any part connected to it, up to seven hops (for a total of 3,280 parts, with possible duplicates). Similarly to the Lookup benchmark, a null procedure is invoked for each part, taking as its arguments the *x*, *y*, and *type* fields of the part.

These benchmarks are intended to be representative of the data operations in many engineering applications. The Lookup benchmark emphasizes selective retrieval of objects based on their attributes, while the Traversal benchmark illuminates the cost of raw pointer traversal.

Each measure is typically run ten times, the first when the system is cold, with none of the database cached (apart from any schema or system information necessary to initialize the system). Each successive iteration fetches a *different* set of random parts. Before the first run of each series of benchmark iterations a "chill" program is executed on the client to sequentially read a 32 Mbyte file from the server. This ensures that the operating system file buffers of both client and server have been flushed of all database segments, so that the first iteration is truly cold.

In addition to the ten cold-warm iterations, we measured the elapsed time for several *hot* iterations of the Traversal benchmark, by beginning each hot iteration at the same initial part used in the last of the warm iterations. These hot runs are guaranteed to traverse only resident objects, and so will be free of any overheads due to swizzling and retrieval of non-resident objects. We varied the number of hot iterations performed per

data point gathered, in order to obtain a linear measure of the CPU overheads of fault detection (excluding swizzling and disk accesses) for each of the schemes.

5.3 Experimental setup

The client machine on which the benchmarks were run was a DECstation 3100 (MIPS R2000A CPU[7] clocked at 16.67MHz) running ULTRIX 4.1.[8] The system has 24 Mbytes of main memory, 10% of which is used for operating system disk buffers. The Smalltalk interpreter is coded in C and compiled with the GNU C compiler (gcc) version 2.3.3 at optimization level 2. The benchmarks were run with the client system in single user mode and the process's address space was locked in main memory to prevent paging.

The database is accessed remotely via NFS. The server is a SPARCstation 2 running SunOS 4.1.2,[9] with 32 Mbytes of main memory,and the database resides on a 1.3 Gbyte external SCSI disk. The client and server were connected via a private ethernet.

We measured elapsed time on the client machine using a custom timer board having a resolution of 100 ns. The fine-grained accuracy of this timer allowed us to measure the elapsed time of each phase of execution separately: running time, swizzling, and time spent retrieving physical segments from disk.

The experiments were repeated several times for each configuration, and the results averaged. Each run is presented with exactly the same database (no updates are ever committed). Note also that the *n*th iteration of any given benchmark run will always access the same parts as the *n*th iteration within any other benchmark run, since the script that controls the execution of the benchmarks presents the same sequence of random part identifiers to each run.

6 Results

We now report on the results for each of the benchmarks. All times reported are in seconds, and exclude any Smalltalk initialization time prior to beginning the benchmark. In all of the figures, the schemes are identified by their names as specified in Table 1.

[7]MIPS and R2000 are trademarks of MIPS Computer Systems.

[8]DECstation and ULTRIX are registered trademarks of Digital Equipment Corporation. The operating system had some official patches installed that fix bugs in the mprotect system call.

[9]SPARCstation is a trademark of SPARC International, licensed exclusively to Sun Microsystems. SunOS is a trademark of Sun Microsystems.

Scheme	Elapsed time (s)	
	Average	
non-persistent	0.565	
ID-resident	0.557	
FB-resident	0.556	
PF-resident	0.567	
	Cold	Warm
ID-OBJ	6.75	1.771
ID-LSEG	7.65	1.448
ID-PSEG	7.56	1.431
FB-OBJ	379.79	40.647
FB-LSEG	26.21	0.558
FB-PSEG	19.81	0.569
PF-OBJ	390.93	41.579
PF-LSEG	27.21	0.573
PF-PSEG	20.27	0.593

Table 2: Lookup

6.1 Lookup

The results for the Lookup benchmark are summarized in Table 2. We give the average elapsed time of the ten iterations for the non-persistent variants (since the database is always resident and warm), and the cold and warm times for the persistent variants. The non-persistent variants exhibit marginal variation in their performance, indicating that the overhead of the run-time residency checks is negligible. It is curious that both the ID-resident and FB-resident schemes perform slightly better than non-persistent Smalltalk, since they have been augmented with residency checks. We can only speculate that the improvement is due to underlying cache effects.

The results for the persistent schemes are naturally more interesting. The FB-OBJ and PF-OBJ schemes are a clear loss, since object-at-a-time faulting results in more frequent object faults, and fewer objects are made resident per fault. Thus, even at the warmest iteration the object-at-a-time schemes still experience object faults. Performance is poor since each object

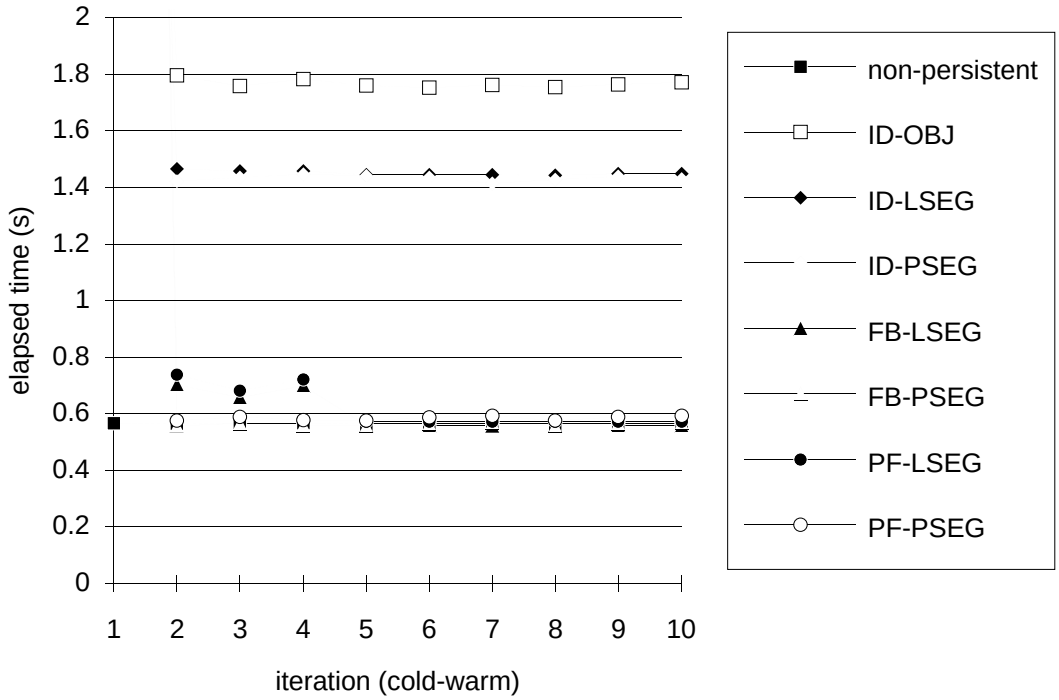


Figure 3: Lookup

fault incurs significant overhead to eliminate indirections. Still, FB-OBJ is better than PF-OBJ, because the page-trapping approach incurs significant overhead to trap object faults and to manipulate page protections when swizzling.

We have found that FB-OBJ behaves much less poorly if we refrain from eliminating indirections at every object fault, even though indirect blocks will frequently be encountered when traversing references, since dereferencing an extra level of indirection can be performed relatively cheaply. Similarly, the cold times for all the FB schemes can be improved substantially by not performing indirection elimination, so that they also outperform the ID schemes for cold starts. Thus, it may be preferable to expend effort to eliminate indirections for the FB schemes only as the system gets warmer, when the cost of traversing indirections becomes more important. In contrast, for the PF schemes an expensive page protection trap occurs every time an indirect block is encountered, making early elimination of indirections much more important.

To compare the schemes more effectively we have plotted their performance in Figure 3, expanding the scale to focus on the warm run performance, and omitting the poorly performing FB-OBJ and PF-OBJ variants. The non-persistent Smalltalk results are also plotted as a baseline. The ID schemes are ranked by their eagerness to swizzle, since swizzling more objects at a time reduces the number of locations containing OIDs. Still, the ID schemes are significantly less competitive overall. The FB and PF schemes behave very similarly, with warm performance close to optimal, due to the aggressive approach taken to eliminate references to indirect blocks. Nevertheless, the software-mediated FB schemes are marginally better than the page-trapping PF approach for the warmest runs, which incur no object faults or swizzling. The reason is that for the page trap handler to decode the contents of the registers at the time of a fault (in order to fix the faulting reference), we have had to impose a less than natural code sequence at each potential fault site in the interpreter. This yields a slight run-time performance penalty for the PF schemes.

The schemes illustrated in Figure 3 show almost immediate warmup, since the first iteration touches enough of the database to bring most of the database's physical segments into Mneme's client buffers, whence objects can be swizzled very quickly. Only the FB-LSEG and PF-LSEG schemes exhibit noticeable further warming effects after the first iteration, as additional objects are swizzled from the buffers. By the fifth iteration the

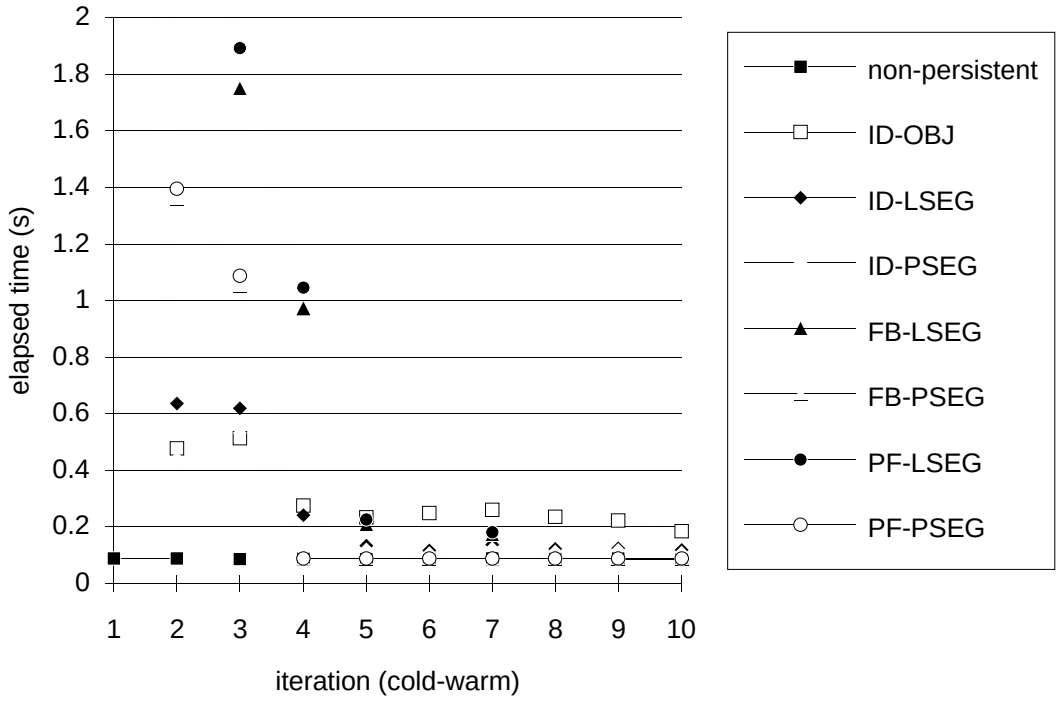


Figure 4: Traversal

LSEG schemes have made sufficient objects resident to proceed without further object faults. The ID schemes are dominated by the overhead to convert OIDs to direct pointers, masking any faults that might occur.

6.2 Traversal

We summarize the Traversal benchmark results in Table 3. Once again, the non-persistent variants show marginal differences in elapsed time, indicating that the overhead of the run-time residency checks is slight. Also, the results for the persistent variants show that the object-at-a-time faulting schemes still have the worst performance, due to the increased per-fault swizzling costs imposed by indirection elimination. Thus, we again omit FB-OBJ and PF-OBJ in plotting the results in Figure 4.

The warming effect is slower than for the Lookup benchmark, despite the fact that each iteration accesses more parts. This is due to the locality of references encoded in the connections between parts, which has been replicated in the clustering used to group objects into physical segments. Thus, traversals mostly touch parts whose physical segments are already resident in Mneme's client buffers, with only a few connection

Scheme	Elapsed time (s)	
	Average	
non-persistent	0.0880	
ID-resident	0.0880	
FB-resident	0.0880	
PF-resident	0.0886	
	Cold	Warm
ID-OBJ	4.01	0.184
ID-LSEG	5.17	0.119
ID-PSEG	5.46	0.110
FB-OBJ	19.64	6.717
FB-LSEG	15.52	0.0883
FB-PSEG	15.61	0.0883
PF-OBJ	21.30	8.090
PF-LSEG	16.08	0.0877
PF-PSEG	15.88	0.0884

Table 3: Traversal

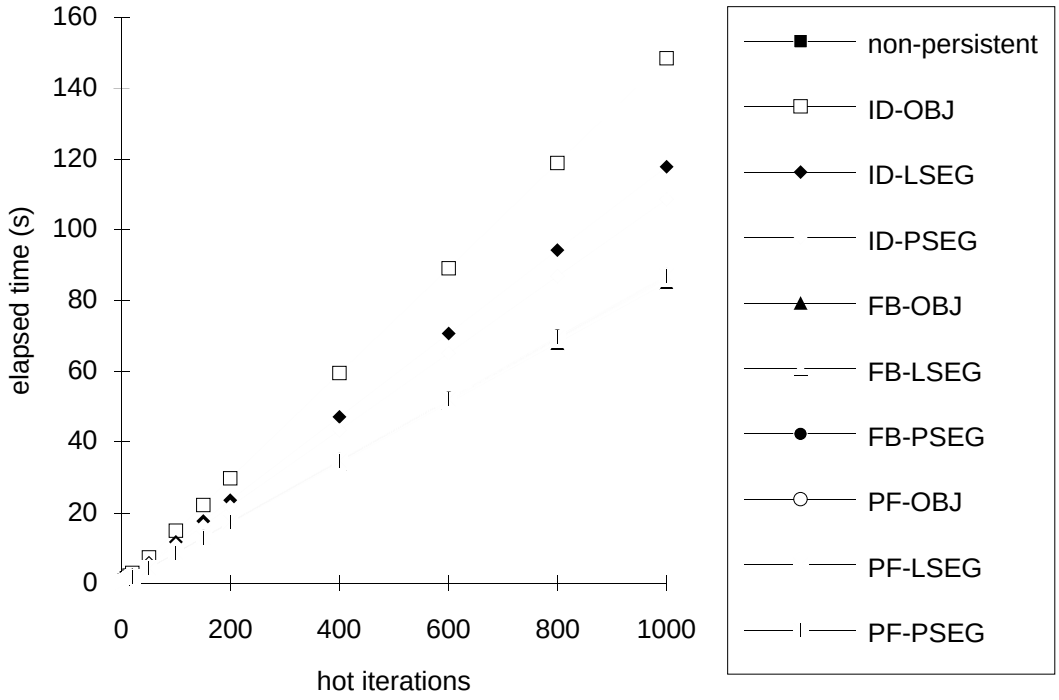


Figure 5: Hot traversal

traversals needing to be serviced by a disk access. The ID schemes warm up more quickly, although their performance is bounded by the overhead of translating OIDs to pointers, while the FB and PF schemes are penalized for indirection removal. Nevertheless, indirection removal pays off by the fourth iteration for FB-PSEG and PF-PSEG, and by the eighth iteration for FB-LSEG and PF-LSEG, when all resident part references have been converted to direct pointers, and enough of the database has been made resident for execution to proceed without object faults.

6.3 Hot traversals

The hot Traversal results (plotted in Figure 5) give some idea of the run-time CPU costs for the schemes, in the absence of any object faults or swizzling overheads. We have obtained excellent linear regression fits for these results, for the model $y = a + bx$, where y is the elapsed time, and x the number of hot iterations per run. As expected, the fitted y-axis intercepts a are close to zero. More interesting is the slope b, which is a measure of time per traversal, given in Table 4.

The results confirm the drawbacks of the ID schemes, showing that OID conversion is a significant run-time

Scheme	Slope b
non-persistent	0.0865
ID-resident	0.0865
FB-resident	0.0865
PF-resident	0.0871
ID-OBJ	0.1486
ID-LSEG	0.1178
ID-PSEG	0.1086
FB-OBJ	0.0850
FB-LSEG	0.0867
FB-PSEG	0.0866
PF-OBJ	0.0872
PF-LSEG	0.0862
PF-PSEG	0.0870

Table 4: Estimated elapsed time per hot traversal

overhead. Also, the ID schemes are ranked by their eagerness to swizzle, since swizzling a whole physical or logical segment at one time allows many intra-segment references to be converted to direct pointers rather than OIDs; recall that the OIDs are never updated with direct pointers. The FB and PF schemes have hot performance close to that of non-persistent Smalltalk since they convert all resident object references to direct pointers. Most importantly, the software-mediated residency checks used by the FB schemes pose insignificant overhead for hot execution.

7 Conclusions

Our results are conclusive in establishing that software object fault detection mechanisms can provide performance very close to optimal, even surpassing the performance of comparable hardware-assisted schemes. This has been achieved through careful assumptions about residency. In particular, the object-oriented execution paradigm allows many residency checks to be elided, with residency checks being restricted mostly to method invocation. This approach can be applied to any language that includes dynamic binding of method calls, by arranging for fault blocks to respond to all methods by first faulting the target object and then forwarding the invocation to it. We have also shown that it pays to be eager in object swizzling, by swizzling *related* objects in advance of the application's need for them.

7.1 Compilation

The fact that the results have been obtained for an interpreted language cannot be taken lightly, since run-time overheads are several times higher than those of compiled programs. Nevertheless, we see no reason why the results will not carry over to a compiled setting; only the *relative* overheads of object fault detection and handling will change with respect to total execution time. However, some languages (e.g., Modula-3 [15, 7], C++ [23]) do not enforce the pure object-oriented style of execution that enables residency checks to be piggy-backed with method invocation. Operations on an object can be performed without necessarily invoking a method on it. This means that explicit residency checks must be compiled into the code in advance of such operations, to ensure that the object is resident. Compiler optimizations [19, 18, 9, 10] may allow these explicit residency checks to be merged or eliminated. For example, control-flow information may reveal that multiple traversals of a particular object reference along a given execution path require only one residency check, rather than a check per traversal. We look forward to exploring the effect of such techniques in our forthcoming implementation of Persistent Modula-3.

7.2 Other architectures

We acknowledge that our architectural framework is one of several that might be considered. For example, we have chosen a copy swizzling approach, whereas it may be possible for applications to manipulate objects directly in the client buffer pool. We have already discussed the reasons for our choice, on the grounds of flexibility in the management of resident objects. Moreover, the performance study of White and DeWitt [26] indicated that such an architecture was superior to the others they considered.

We also recognize that our page trapping approach, which allocates fault blocks in protected pages, does not compare directly with the memory-mapped file approaches of ObjectStore and Texas. In particular, ObjectStore makes some effort to allow objects to be mapped directly into the same memory locations in which they were originally allocated, thus reducing or eliminating the need to swizzle pointers upon object fault.

Nevertheless, our results stand as a relative comparison of object faulting techniques within a fixed architectural framework. It is reasonable to assume that the relative standing of our fault detection mechanisms will remain the same even if the underlying persistent object storage architecture changes.

7.3 Summary

In summary, we have explored the design space of mechanisms for detecting and handling references to persistent objects, and compared their performance within a prototype persistent programming language. Most importantly, we have demonstrated that software-mediated object faulting can be a viable alternative to hardware-assisted techniques, and that adding persistence to a programming language does not necessarily imply degradation of performance.

8 Acknowledgements

We thank Eric Brown for his enhancements to Mneme in support of this work. We also thank the Western Research Laboratory of Digital Equipment Corporation, and Jeff Mogul in particular, for giving us the high resolution timing board and the necessary supporting software.

References

[1] 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, Nov. 1983.

[2] M. J. Carey, D. J. DeWitt, J. E. Richardson, and E. J. Shekita. Storage management for objects in EXODUS. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, chapter 14, pages 341–369. ACM Press/Addison-Wesley, New York, New York, 1989.

[3] R. G. G. Cattell and J. Skeen. Object operations benchmark. *ACM Trans. Database Syst.*, 17(1):1–31, Mar. 1992.

[4] C. J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, 13(11):677–678, Nov. 1970.

[5] A. Dearle, G. M. Shaw, and S. B. Zdonik, editors. *Proceedings of the Fourth International Workshop on Persistent Object Systems*, Martha's Vineyard, Massachusetts, Sept. 1990. Published as *Implementing Persistent Object Bases: Principles and Practice*, Morgan Kaufmann, 1990.

[6] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[7] S. P. Harbison. *Modula-3*. Prentice Hall, New Jersey, 1992.

[8] A. L. Hosking. Main memory management for persistence, Oct. 1991. Position paper presented at the OOPSLA '91 Workshop on Garbage Collection.

[9] A. L. Hosking and J. E. B. Moss. Towards compile-time optimisations for persistence. In Dearle et al. [5], pages 17–27.

[10] A. L. Hosking and J. E. B. Moss. Compiler support for persistent programming. COINS Technical Report 91-25, University of Massachusetts, Amherst, MA 01003, Mar. 1991.

[11] T. Kaehler. Virtual memory on a narrow machine for an object-oriented language. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 87–106, Portland, Oregon, Sept. 1986. *ACM SIGPLAN Not. 21*, 11 (Nov. 1986).

[12] T. Kaehler and G. Krasner. LOOM—large object-oriented memory for Smalltalk-80 systems. In G. Krasner, editor, *Smalltalk-80: Bits of History, Words of Advice*, chapter 14, pages 251–270. Addison-Wesley, 1983.

[13] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Commun. ACM*, 34(10):50–63, Oct. 1991.

[14] J. E. B. Moss. Design of the Mneme persistent object store. *ACM Trans. Inf. Syst.*, 8(2):103–139, Apr. 1990.

[15] G. Nelson, editor. *Systems Programming with Modula-3*. Prentice Hall, New Jersey, 1991.

[16] Object Design, Inc. *ObjectStore User Guide*, Oct. 1990. Release 1.0.

[17] A. Purdy, B. Schuchardt, and D. Maier. Integrating an object server with other worlds. *ACM Trans. Office Inf. Syst.*, 5(1):27–47, Jan. 1987.

[18] J. E. Richardson. Compiled item faulting: A new technique for managing I/O in a persistent language. In Dearle et al. [5], pages 3–16.

[19] J. E. Richardson and M. J. Carey. Persistence in the E language: Issues and implementation. *Software: Practice and Experience*, 19(12):1115–1150, Dec. 1990.

[20] D. Schuh, M. Carey, and D. DeWitt. Persistence in E revisited—implementation experiences. In Dearle et al. [5], pages 345–359.

[21] V. Singhal, S. V. Kakkad, and P. R. Wilson. Texas, an efficient, portable persistent store. In *Proceedings of the Fifth International Workshop on Persistent Object Systems*, pages 11–33, San Miniato, Italy, Sept. 1992.

[22] A. Straw, F. Mellender, and S. Riegel. Object management in a persistent Smalltalk system. *Software: Practice and Experience*, 19(8):719–737, Aug. 1989.

[23] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[24] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, Pittsburgh, Pennsylvania, Apr. 1984. *ACM SIGPLAN Not. 19*, 5 (May 1984).

[25] D. M. Ungar. *The Design and Evaluation of a High Performance Smalltalk System*. ACM Distinguished Dissertations. The MIT Press, Cambridge, MA, 1987. Ph.D. Dissertation, University of California at Berkeley, February 1986.

[26] S. J. White and D. J. DeWitt. A performance study of alternative object faulting and pointer swizzling strategies. In *Proceedings of the Eighteenth International Conference on Very Large Data Bases*, pages 419–431, Vancouver, Canada, Aug. 1992. Morgan Kaufmann.

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