1995

Lightweight write detection and checkpointing for fine-grained persistence

Antony L. Hosking
Purdue University

J. Eliot B. Moss
University of Massachusetts - Amherst

Purdue University Department of Computer Sciences Technical Report 95-084

Lightweight write detection and checkpointing for fine-grained persistence

Antony L. Hosking
Purdue University
and
J. Eliot B. Moss
University of Massachusetts at Amherst

Many systems must dynamically track writes to cached data, for the purpose of reconciling those updates with respect to the permanent or global state of the data. For example, distributed systems employ coherency protocols to ensure a consistent view of shared data. Similarly, database systems log updates both for concurrency control and to ensure the resilience of those updates in the face of system failures. Here, we measure and compare the absolute performance of several alternative mechanisms for the lightweight detection of writes to cached data in a *persistent* system, and the relative overhead to log those writes to stable storage in the form of a *checkpoint*. A checkpoint defines a consistent state to which the system will be restored in the event of any subsequent failure. The efficient detection and logging of updates is critical to the performance of persistent systems that embody a fine-grained data model, since per-object overheads are typically very low. Our results reveal a wide range of performance for the alternatives, indicating that the right choice of mechanism is important. They also demonstrate that software write detection mechanisms can significantly outperform approaches that rely solely on the hardware and operating system.

Categories and Subject Descriptors: []:

General Terms:

Additional Key Words and Phrases:

1. INTRODUCTION

A *persistent system* [Atkinson et al. 1982; Atkinson et al. 1983; Atkinson et al. 1983; Atkinson and Buneman 1987] maintains data independently of the transitory programs that create and manipulate that data—data may outlive their creators, and be manipulated by yet other programs. To achieve this, persistent systems provide an abstraction of *persistent storage*, which programmers view as a stable

This work has been supported by the National Science Foundation under grants CCR-9211272, CCR-8658074 and DCR-8500332, and by the following companies and corporations: Sun Microsystems, Digital Equipment, Apple Computer, GTE Laboratories, Eastman Kodak, General Electric, ParcPlace Systems, Xerox, and Tektronix.
Name: Antony L. Hosking
Affiliation: Purdue University
Address: Department of Computer Sciences, Purdue University, West Lafayette, IN 47907-1398, hosking@cs.purdue.edu
Name: J. Eliot B. Moss
Affiliation: University of Massachusetts at Amherst
Address: Department of Computer Science, University of Massachusetts, Amherst, MA 01003-4610, moss@cs.umass.edu

(i.e., disk-resident) extension of memory in which they can dynamically allocate new data, but which persists from one program invocation to the next. A *persistent programming language* allows traversal and manipulation of the persistent data to be programmed *transparently*: i.e., without explicit calls to transfer the data between volatile main memory and stable persistent storage. Rather, the language implementation and run-time system contrive to make persistent data available in memory on demand, much as non-resident pages are automatically made resident in a paged virtual memory system. Moreover, a persistent program can modify persistent data and commit the modifications so that the updates are permanently recorded in persistent storage.

As in traditional database systems, persistent systems typically treat memory as a relatively scarce resource (at least with respect to the size of the database), maintaining a cache of frequently-accessed persistent data in volatile memory for efficient manipulation. Updates can be made cheaply in place, in memory, but ultimately must propagate back to stable storage for them to become permanent. Thus, every operation that modifies persistent data requires some immediate or subsequent action to commit the update to disk. While the system might write the modifications straight through to disk on every update, such an approach is likely to be unnecessarily expensive if updates are frequent or otherwise incur very little overhead. Moreover, applications exhibiting locality of update may benefit from an approach that groups related updates together for efficient batch transfer to disk, deferring writes until absolutely necessary, such as when the program issues an explicit *checkpoint* operation. While this approach reduces per-update overhead, checkpoint latencies should also be minimized so as to have the smallest possible impact. For example, in interactive environments checkpoints should not noticeably delay response times. Such overheads to manage updates to persistent data constitute a *write barrier* that can significantly impact the performance of persistent systems.

Adding persistence to conventional programming languages, such as those in the Algol family (including Pascal, C, Modula-2, and their object-oriented cousins C++, Modula-3, and even Smalltalk), is complicated by their *fine-grained* view of data—they provide fundamental data types and operations that correspond very closely to the ubiquitous primitive types and operations supported by all machines based on the von Neumann model of computation. This close correspondence means that many operations supported in the language can be implemented directly with as little as one instruction of the target machine. Integrating persistence with such languages poses new problems of performance arising out of the fine granularity of the types and operations supported by the language.

The principle of orthogonality mandates that even data values as fine-grained as a single byte (the smallest value typically addressable on current machines) ought to persist independently. Clearly, this situation is significantly different from that of traditional database systems [Date 1983; Date 1986], where the unit of persistence is the record, usually consisting of many tens, if not hundreds, of bytes. Where a relational database system can spend hundreds or even thousands of instructions implementing relational operators, an Algol-like persistent programming language must take an approach to persistence that does not swamp otherwise low-overhead and frequently-executed operations. Thus, implementations of the write barrier

for such languages must be sufficiently lightweight as to represent only marginal overhead to frequently-executed operations on fine-grained persistent data. Performance of the write barrier for persistence can be broken down into two components: the *run-time* overhead to track updates as they occur, and the *checkpoint* overhead required to flush those updates to disk. We explore the relative performance of a comprehensive set of alternative low-overhead write barrier implementations within a prototype persistent system. We precisely measure both run-time and checkpoint overheads for each alternative and characterize their tradeoffs. Our results show that the alternatives exhibit a wide range of performance, implying that the right choice of mechanism is important. Moreover, the results are somewhat counter-intuitive, because they reveal that write barrier mechanisms implemented in software can significantly outperform alternative approaches that rely on support from the hardware and operating system to track updates. The results also have general implications for the choice of write barrier mechanism in domains other than persistence, such as garbage collection and distributed systems. In addition to the direct performance results, we also offer our experimental methodology as representing a unique blend of performance measurement and simulation in characterizing the low-level behavior of a complex software system.

The remainder of the paper is organized as follows. Section 2 covers background material about persistent systems. Section 3 presents the methodology we use for the performance evaluation, including a description of the prototype implementation and the write detection mechanisms compared, and the benchmarks, experimental configuration, and performance metrics used. Section 4 presents the performance results, along with an analysis based on the simulations. Section 5 discusses related work. Section 6 suggests directions for future work, and section 7 offers final conclusions.

2. BACKGROUND

Persistent systems were born out of a fundamental convergence of programming language and database technology, integrating the data manipulation features of programming languages with the storage management features of databases. The following sections review the important architectural features of persistent programming languages and systems, and describe our particular architecture for persistence and its rationale.

2.1 Persistent programming languages

Persistent programming languages place the full type system of the language at the programmer's disposal in defining persistent data. Early on, Atkinson et al. [1983] characterized persistence as "an orthogonal property of data, independent of data type and the way in which data is manipulated." This particular characterization encourages the view that a language can be extended to support persistence with minimal disturbance of its existing syntax. As such, most persistent programming languages represent an attempt to extend the address space of programs beyond that which can be addressed directly by the available hardware, just as 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 and hardware

cooperate to trap references to pages that are not yet resident in physical memory). They provide an abstraction of persistent storage in terms of a persistent dynamic allocation heap: data in the heap are referred to by language-supported pointers.

If the entire heap can fit in virtual memory then it can be mapped directly, with pointers represented as direct virtual memory addresses. However, this limits the size of the heap to that of the virtual address space. Extended addressability requires pointers that are not necessarily virtual memory addresses, as well as a mechanism to perform translation of pointers to virtual addresses to allow the program to manipulate the data.

Either way, a persistent program may refer to both resident and non-resident persistent objects. Ideally, a memory-resident persistent object will be referred to by its virtual address, so that accessing the object can be as fast as accessing a non-persistent object. If the program traverses a reference to a non-resident object then it must be made available to the program in memory: we call this an *object fault*. Thus, persistent objects are faulted into memory on demand much as non-resident pages are automatically made resident by the virtual memory system. Several implementations of object faulting are possible, driven by software checks on pointer dereferences supported by the language implementation, or through user-level virtual memory primitives supported by the operating system and hardware [Appel and Li 1991; Hosking and Moss 1993a].

2.2 Storage management

Architectures for persistence typically have one component in common: a *storage manager*, responsible for maintaining data in some inexpensive stable storage medium such as magnetic disk and for fielding requests to retrieve and save specified data. *Object-oriented* storage managers allow retrieval of a data *object* based on its *identifier* (ID); such systems are called *persistent object stores*. Object-oriented database systems also support this style of access, but they are distinguished from persistent object stores by additionally providing full database functionality, including concurrency control, recovery, transactions, distribution, data access via associative queries, and languages for data definition and manipulation.

Our implementation uses the Mneme persistent object store [Moss 1990] to manage the storage and retrieval of objects, which are grouped into *segments* for efficient transfer to and from disk. Mneme is intended for tight integration with persistent programming languages through a procedural interface. Its primary abstraction is the persistent store as a persistent heap: objects persist so long as they are reachable from designated root objects.

2.3 An architecture for persistence: object caching

Our architecture for persistence is not that unusual. For example, it bears a close resemblance to the object caching architectures of White and DeWitt [1992], Kemper and Kossman [1994] and Napier88 [Brown et al. 1991; Brown et al. 1990]. However, the architecture's *realization* is unique in that it allows the language implementation maximum control over all objects being manipulated by a program, without having to pass through a restrictive interface to the underlying storage manager. Much of this flexibility comes as a result of using Mneme, since it allows direct access to the objects in its buffers via virtual memory pointers.

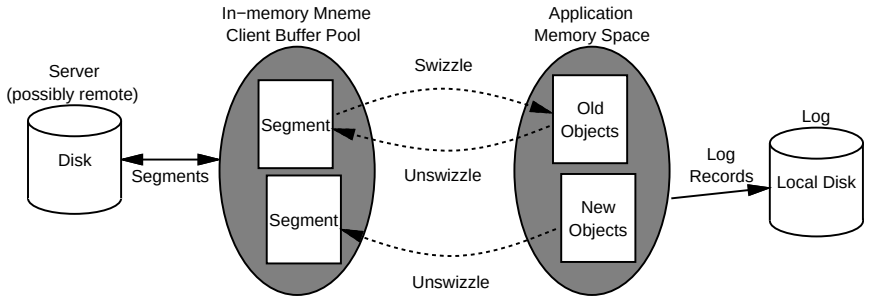


Fig. 1. System architecture

The architecture is illustrated in Figure 1. A Mneme client maintains a buffer of segments containing persistent objects in main memory as necessary. Object faults trigger the copying of objects from this client buffer pool into the virtual memory address space of the application program. The copying includes any translation needed to convert the objects into a form acceptable to the program. In particular, since Mneme uses object identifiers to refer to objects while the program uses virtual memory pointers, object references may be converted to direct memory pointers for manipulation by the program, in a process known as *swizzling*. The architecture permits standard programming language techniques for memory management, including those of garbage collection, to manage the objects resident in the program's virtual address space.

When an object is made resident its pointer fields are swizzled according to the mechanism being employed for triggering object faults. All fields referring to other resident objects can be converted to point directly to those objects—Mneme supports this mapping efficiently with a hash table. Otherwise, the reference must be converted to a form that will trigger an object fault when it is traversed.

2.4 Checkpoints

Our notion of recovery is dictated by specific assumptions about the behavior of persistent programs. We assume that a program will invoke a checkpoint operation at certain points throughout its execution to make permanent all modifications it has made to persistent objects. In the event of a system crash the recovery mechanism must restore the state of the persistent store to that of the most recent checkpoint. Moreover, we assume that checkpoint latencies should be minimized so as to have the smallest possible impact on the running time of the program. This last point is important in interactive environments where checkpoints may noticeably delay response times.

A checkpoint operation consists of copying and unswizzling modified and newly-created objects (or modified subranges of objects) back to the client buffer pool and generating log records describing the range and values of the modified regions of the objects. Log records are generated only if there are differences between an object and its original in the client buffer pool. Since persistence in our system is based on *reachability* the unswizzling operation may encounter pointers to objects newly created since the last checkpoint. Each such object is assigned a persistent

identifier and unswizzled in turn, perhaps dragging further newly-created objects into the persistent store, and a log record describing the new object is generated.

The precise format of the log records is not relevant to this study, since we are interested only in the mechanisms used to detect and log updates. However, we note that each log record is tagged by the persistent identifier of the modified object and encodes a range of modified bytes. Recovery involves applying these log records to the objects to which they pertain, in the order in which they occur in the log. Although alternative log-record formats might yield a more compact log, or allow more efficient recovery, our log is *minimal* in the sense that it records just enough information to reconstruct each modified object. Moreover, difference-based logging minimizes disk traffic at the cost of computing the differences. In preliminary performance studies we determined that the tradeoff is worthwhile. This result has also been confirmed in other settings [White and DeWitt 1995].

2.5 Extensions to the basic architecture

As described so far, the architecture supports single-user access with recovery. We now argue that the architecture can be extended to provided additional functionality such as buffer management and concurrency control.

2.5.1 *Buffer management.* To integrate buffer management with the recovery model, we guarantee that a modified segment is flushed to disk only after the log records associated with those modifications have been written. Outside of that constraint, the buffer manager is free to use any appropriate buffer replacement policy. Management of swizzled objects in the application's virtual memory must rely on techniques similar to garbage collection to determine which objects are subject to replacement [Hosking 1991]; this is compatible with the recovery model, since modified objects that have been selected for replacement will be unswizzled and logged to disk for inclusion in the next checkpoint.

2.5.2 *Concurrency control.* The recovery model is indifferent to concurrency, which can be introduced to the architecture in two ways. First, separate applications can share the same persistent store, arbitrated by a server. Locking is managed by the server and the application's view of recovery is unchanged, modulo some additional information required in a log entry to identify its owner. Second, a single application may be multi-threaded. Additional locks must be managed within the application if data is shared among threads. Again, the recovery model remains essentially unchanged, modulo some additional log entry information to identify the owner of the entry.

The recovery model and support for concurrency provide the foundation for any transaction model. The incorporation of transaction models in persistent programming languages remains an open topic of research. We are not directly concerned with that issue here, and merely remark that our recovery model could be extended to incoporate transactions similarly to the *database cache* [Elhardt and Bayer 1984], for which several transaction models exist. Like our system, the database cache was designed for fast transaction commit and rapid recovery after a crash.

3. METHODOLOGY

We evaluate the performance of several alternative write barrier implementations within a single prototype persistent system. Different instantiations of the prototype use different implementations of the low-level write barrier mechanism. All other aspects of the implementation are kept constant for each instantiation of the prototype. This allows a head-to-head comparison of alternative implementations where only the particular mechanism under study varies across all the instances of the prototype. In this respect, the prototype is a novel experimental test-bed for the exploration of persistent systems implementation, allowing direct comparison of alternatives.

Our experiments encompass measurements of elapsed time, as well as cache simulation and instruction profiling to obtain counts of both cache misses and execution frequency, per instruction address. Combined, these measurements allow precise determination of both absolute run-time overheads in terms of cycles per update, and relative checkpoint overheads, for each alternative write barrier implementation.

3.1 A prototype implementation: Persistent Smalltalk

The prototype persistent system used for this study is an implementation of the Smalltalk programming language and environment [Goldberg and Robson 1983], extended for persistence. The implementation has two components: a *virtual machine* and a *virtual image*.

The virtual machine implements the bytecode instruction set to which Smalltalk source code is compiled, along with certain *primitives* whose functionality is built directly into the virtual machine. These typically provide low-level access to the underlying hardware and operating system on which the virtual machine is implemented. For example, low-level floating point and integer arithmetic, indexed access to the fields of objects, and object allocation, are all supported as primitives. Notable features of our implementation of the Smalltalk virtual machine are its use of direct 32-bit object pointers, an improved scheme for managing Smalltalk stack frames (i.e., activation records) [Moss 1987], generation scavenging garbage collection [Ungar 1984; Ungar 1987], and dynamic translation of compiled methods from bytecodes to *threaded code* [Bell 1973]. Threaded code significantly improves the performance of the virtual machine by replacing an expensive decode-and-branch overhead for every bytecode instruction cycle with a straightforward indirect branch. The result is an interpreted Smalltalk system that exhibits performance around three times faster on the SPARCstation 2 than an equivalent implementation in which bytecode instructions are microcoded on the Xerox Dorado [McCall 1983].

The virtual image is derived from Xerox PARC's Smalltalk-80 image, version 2.1, of April 1, 1983, with minor modifications. It implements (in Smalltalk) all the functionality of a Smalltalk development environment, including editors, browsers, a debugger, the bytecode compiler, class libraries, etc.–all are first-class objects in the Smalltalk sense. Bootstrapping a (non-persistent) Smalltalk environment entails loading the entire virtual image into memory for execution by the virtual machine.

The persistent implementation of Smalltalk places the virtual image in the persis-

tent store, and the environment is bootstrapped by loading just that subset of the objects in the image sufficient for resumption of execution by the virtual machine. We retain the original bytecode instruction set and make only minor modifications to the virtual image. Rather, our efforts focus on the virtual machine, which is carefully augmented to fault objects into memory as they are needed by the executing image. The precise mechanism used for object faulting is not relevant to this study, except to say that we use a software approach that is kept constant across all variations of write barrier mechanism. Moreover, object faulting overheads are very low and restricted solely to the method invocation sequence—bytecode dispatch and execution are entirely free of object faulting overheads. Comparisons of alternative schemes for object faulting within the prototype appear elsewhere [Hosking and Moss 1993a; Hosking and Moss 1993b; Hosking 1995].

3.2 Implementing the write barrier: detecting and logging updates

Our lightweight mechanisms are inspired by similar solutions to the write barrier problem in garbage collection: the act of storing a pointer in an object is noted in order to minimize the number of pointer locations examined by the collector [Hosking et al. 1992]. Similarly, efficient logging requires keeping track of all updates to objects, to minimize the number of locations unswizzled when generating the log (recall that a log record is generated only if there are differences between the new version of an object and the original in the client buffer pool).

This study examines several implementations of the write barrier, including three approaches previously used in garbage collection and now applied for the first time to the problem of detecting and logging updates to persistent data. Note that since the log consists of difference information obtained by comparing old and new versions of objects, all schemes end up generating exactly the same log information. The schemes vary mostly in the granularity of the update information they record, and hence in the amount of unswizzling and comparison required to generate the log.

3.2.1 *Object-based schemes.* The first two schemes record updates at the logical level of objects. One approach is to mark updated objects by setting a bit in the header of the object when it is modified. The checkpoint operation must scan all cached objects to discover those marked as updated. A marked object must be unswizzled and compared to its original in the buffer pool to determine any differences to be logged. The drawback of this approach is the additional checkpoint overhead required to scan the cached objects to find those that are marked.

To avoid scanning, the second scheme uses a data structure called a *remembered set* [Ungar 1984] to record modified persistent objects. A checkpoint need only process the entries in the remembered set to locate the objects that must be unswizzled and possibly logged. The remembered set is implemented as a dynamic hash table.

So that the remembered set does not become too large, an inline filter is applied to record only updates to persistent objects, as opposed to newly-created transient objects—Smalltalk is a prodigious allocator, so the vast majority of updates are to transient objects. This requires a check to see that the updated object is located in the separately managed persistent area of the volatile heap, determined by taking the high bits of its address to index a table that contains such information. If the

updated object is indeed persistent then a subroutine is invoked to hash the object's pointer into the remembered set.

Remembered sets have the advantage of being both concise and accurate, at the cost of filtering and hashing to keep the sets small—repeated updates to the same object result in just one entry in the remembered set, but incur repeated overhead to filter and hash.

3.2.2 *Card-based schemes.* Object-based schemes concisely represent just those objects that have been modified, and so need to be unswizzled on checkpoint. However, updates to larger objects may suffer from poor locality with respect to the object size, resulting in unnecessary unswizzling and comparison upon checkpoint, bounded solely by the size of the object. An alternative is to record updates based on fixed-size units of the virtual memory space, by dividing the memory into aligned logical regions of size 2^k bytes—the address of the first byte in the region has k low bits zero. These regions are called *cards* after Sobalvarro [1988]. Each card has a corresponding entry in a card table indicating whether the card contains updated locations. Mapping an address to an entry in this table is simple: shift the address right by k bits and use the result as an index into the table. Whenever an object is modified, the corresponding card is *dirtied*.

One of the most attractive features of card marking is the simplicity of the write barrier. Ignoring cache effects, the per-update overhead is constant. Keeping this overhead to a minimum is highly desirable. By implementing the card table as a byte array (rather than a bitmap), and interpreting zero bytes as dirty entries and non-zero bytes as clean, a store can be recorded with just three SPARC instructions: a shift, index, and byte store of zero [Wilson and Moher 1989a; Wilson and Moher 1989b].[1]

The checkpoint operation scans only the dirty cards containing persistent objects, to perform unswizzling and obtain differences for logging. Unswizzling requires locating all pointers within the card. Moreover, the log records must be generated with respect to the modified objects in the card, recording the object identifier and contiguous ranges of modified bytes. Since the formats of the objects in the card are encoded in their object headers, the header of the first object within a given card must be located to start the scan. For this purpose, a table of card offsets parallel to the dirty card table records the location of the *last* (highest address) object header within each card. Thus, given a card for scanning, the header of the first object in the card can be found at the end of the last object in the previous card.

Dirty cards are marked clean after scanning. To reduce the overhead of scanning, contiguous dirty cards are scanned as a batch, running from the first to the last in one scan. Also, the implementation takes great pains to avoid unnecessary memory accesses when scanning the card table to locate a run of dirty cards, by loading an entire memory word of the table at a time.

The size of the cards is an important factor influencing checkpoint costs, since

[1]Some modern RISC architectures either do not provide a byte store instruction, or implement it by reading a full word, modifying the appropriate byte, and writing back the modified word. On such machines, it may be cheaper to code the read-modify-write explicitly as a sequence of instructions, or even revert to a bitmap implementation of the dirty card table.

large cards mean fewer cards and smaller tables. However, larger cards imply unnecessary checkpoint overhead to perform unswizzling and comparison of objects that are unmodified, but just happen to lie in a dirty card. Thus, an interesting question arises as to whether there exists an optimal card size that minimizes the sum of these competing overheads. Our performance evaluation answers this question for the program behaviors we consider.

3.2.3 *Page-protection schemes.* A variant of the card-based approach uses the hardware-supported page protection primitives of the operating system to detect stores into clean cards. A card in this scheme corresponds to a page of virtual memory. All clean pages are protected from writes. When a write occurs to a protected page, the trap handler dirties the corresponding entry in the card table and unprotects the page. Subsequent writes to the now dirty page incur no extra overhead.[2] Because the dirty page table is updated out of line in the trap handler it is probably less important that dirtying an entry in the page table incur minimal overhead, than that the dirty page table be kept as small as possible. Thus, one might prefer to reimplement the dirty page table as a bitmap instead of a byte-map, so reducing the table's size by a factor of eight. Here, we continue to use a byte table, for more direct comparison with the page-sized card scheme.

3.3 Benchmarks

The performance evaluation draws on the OO1 object operations benchmarks [Cattell and Skeen 1992] to compare the alternative implementation approaches. These benchmarks are retrieval-oriented and operate on substantial data structures, although the benchmarks themselves are simple, and so easily understood. Their execution patterns include phases of intensive computation so that memory residence is important. Although the OO1 benchmarks are relatively low-level, they are sufficient for an exhaustive exploration of the behavior of our minimal write barrier mechanisms.

3.3.1 *Benchmark database.* 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. We implement the part database and the benchmarks entirely in Smalltalk, including the B-tree used to index the parts.

The database, including the base Smalltalk virtual image as well as the parts data is around 6MB. Newly created objects are clustered into Mneme segments only as they are encountered when unswizzling, using an essentially breadth-first traversal similar to that of copying garbage collectors [Cheney 1970]. The part objects are 68 bytes in size (including the object header). The three outgoing connections are stored directly in the part objects. The string fields associated

[2] An operating system could more efficiently supply the information needed in the page protection scheme by offering appropriate calls to obtain the page dirty bits maintained by most memory management hardware [Shaw 1987].

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 Smalltalk Array object containing references to the parts that are the source of each incoming connection. The B-tree index for the 20,000 parts consumes around 165KB.

3.3.2 *Benchmark operations.* The 001 benchmarks comprise three separate operations and measure response time for execution of each operation. The first two operations are read-only, leaving the permanent database untouched. Since they are retrieval-oriented, and involve no modification to the database, they are not relevant to our study of write barrier mechanisms. We describe them merely for completeness:

—**Lookup** fetches 1,000 randomly chosen parts from the database. A null procedure is invoked for each part, 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). Similar 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.[3]

The third operation requires updating the permanent database, and so is more appropriate for comparing write barrier mechanisms:

—**Insert** allocates 100 new parts in the database, each with three connections to randomly selected parts as described in Section 3.3.1 (i.e., applying the same rules for locality of reference). The index structure must be updated, and the entire set of changes committed to disk.

Although this operation is a reasonable measure of update overhead, it is hampered by a lack of control over the number and distribution of the locations modified, and its mixing of updates to parts and the index. A more controlled benchmark is the following:

—**Update** [White and DeWitt 1992] operates in the same way as the Traversal measure, but instead of calling a null procedure it performs a simple update to each part object encountered, with some fixed probability. The update consists of incrementing the *x* and *y* scalar integer fields of the part. All changes must be committed to disk.

Here, the probability of update can vary from one run to the next to change the frequency and density of updates.

These benchmarks are intended to be representative of the data operations in many engineering applications. The Lookup benchmark emphasizes selective re-

[3] 001 also specifies a *reverse* Traversal operation, swapping "from" and "to" directions. This reverse Traversal operation is of minimal practical use because the random nature of connections means that the number of "from" connections varies among the parts—while every part has three *outgoing* connections, the number of *incoming* connections varies randomly. Thus, different iterations of the reverse Traversal vary randomly in the number of objects they traverse, and so the amount of work they perform.

trieval of objects based on their attributes, while the Traversal benchmark illuminates the cost of raw pointer traversal. The Update variant measures the costs of modifying objects and making those changes permanent. Additionally, the Insert benchmark measures both update overhead and the cost of creating new persistent objects.

OO1 calls for each benchmark measure to be *iterated* 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). Thus, before each run of ten iterations we execute a "chill" program on the client to read a 32MB file from the server, scanning first forward then backward; this flushes the operating system kernel buffer cache on both client and server, so that the first iteration is truly cold. Each successive iteration accesses a *different* set of random parts. Still, there may be some overlap in the parts accessed by different iterations, in which case implementations that cache data from one iteration to the next will exhibit a warming trend, with improved performance for later *warm* iterations that access data cached by earlier iterations.

In addition to the ten cold through warm iterations, it is useful to measure performance for *hot* iterations of the benchmarks, by beginning each hot iteration at the same initial part used in the last of the warm iterations. The hot runs are guaranteed to access only resident objects, and so are free of any overheads due to the handling of object faults.

3.4 Experimental configuration

We ran our experiments on a SPARCstation 2 (Cypress Semiconductor CY7C601 integer unit clocked at 40MHz) running SunOS 4.1.3.[4] The SPARCstation 2 has a 64KB unified cache (instruction and data) with a line size of 32 bytes. Read misses cost 24-25 cycles. On a hit, writes update both cache and memory—a 16-byte write buffer reduces this overhead, though if the buffer is full the processor will stall for 4-5 cycles for the completion of one slow memory cycle. Write misses invalidate, but do not allocate, the corresponding cache line.

The system has 32MB of main memory (DRAM), sufficient for the entire benchmark database to be cached in memory. Thus, buffer management policies can be ignored when interpreting the experimental results. The log file is written locally to an internal SUN0424 SCSI disk (414,360KB unformatted capacity, 2.5-3.0MB/s peak data rate, ~2.9MB/s sustained data rate,[5] 14ms average seek time). The log records are buffered and written as a batch, using calls to `write` followed immediately by a call to `fsync` to force the log data to the local disk before the checkpoint completes. Thus, checkpoints break down into two phases, *unswizzling* and *writing*, which are measured separately.

The database is stored locally (i.e., the client is its own server), for the simple

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

[5] The data rate varies because the disk has a constant linear density for all tracks, with outer tracks yielding a faster bit rate than inner tracks. Peak data rate is the data rate possible for a single sector. Sustained data rate is calculated as the number of 512-byte sectors per track multiplied by the angular velocity—because the number of sectors per track decreases from outer to inner tracks this calculation is only approximate.

reason that the experimental apparatus is thus easier to obtain and control, and also because results for a remote database would differ only in the network latency for retrieval of objects from a remote server's disk. Local disk latencies are sufficiently high to demonstrate the caching effects inherent in the implementation. The database resides on an external SUN1.3G SCSI disk (1,336,200KB unformatted capacity, 3.25-4.5MB/s peak data rate, ~3.5MB/s sustained data rate, 11ms average seek time).

3.5 Metrics

We measure elapsed time for the execution of the benchmark operations using a version of the Smalltalk virtual machine instrumented with calls to the SunOS system call `gettimeofday`. This directly accesses the system hardware clock, which has a resolution of 1μs for the SPARCstation 2. Such fine-grained accuracy permits each phase of execution (running, swizzling, disk retrieval, logging, etc.) to be measured separately with minimal disturbance of the results due to measurement overheads. Moreover, benchmarks are run with the client in single-user mode and disconnected from the network, to minimize interference from network traffic, virtual memory paging, and other operating system activity. All times are reported in seconds (unless stated otherwise), and exclude the time to initialize the Smalltalk system prior to beginning the benchmark.

A benchmark *run* consists of the ten cold through warm iterations, plus a single hot iteration. To guarantee repeatability, the permanent database is kept entirely static (updates are written only to the log, never propagated to the permanent database) so that different runs are always presented with the same physical database. Moreover, every run begins with the same random number seed, so the nth iteration of any given benchmark run always accesses the same parts as the nth iteration within any other benchmark run. In other words, although a different set of random parts are accessed from one cold/warm iteration to the next *within* a run, corresponding iterations from *different* runs always access the same set of parts, so they are directly comparable.

Nevertheless, there may be other uncontrollable variations in system behavior from one run to the next. For example, variations in the disk state (track and block position of the disk read/write arm) may affect read performance. To get an idea of the significance of this variation in the comparison of different implementations we repeat the runs for each implementation and calculate the results as 90% confidence intervals for the mean elapsed time.

As well as measuring elapsed time we also use the Shade instruction-set simulator [Cmelik and Keppel 1994] to obtain precise execution profiles for each run. Using modified versions of Shade's tools for cache simulation and instruction profiling we obtain precise counts, per instruction address, of both execution frequency and cache misses (for the SPARCstation 2 cache configuration described above).

4. RESULTS AND ANALYSIS

Our experiments use the Insert and Update benchmarks, with update probabilities of 0.00, 0.05, 0.10, 0.15, 0.20, 0.50, and 1.00, to measure the performance of object marking (objects), remembered sets (remsets), access-protected virtual memory pages (pages), and card marking (cards-n, where $n = 4^k$ bytes is the card size, for

Table I. Schemes compared

Scheme	On update	On checkpoint
scan	do nothing	scan all objects
objects	set bit in header of changed object	scan for changed objects
remsets	enter object reference into update set	iterate over update set
cards-n $16 \leq n = 4^k \leq 4096$ bytes	dirty card table entry	scan dirty table for dirty cards (process objects/fragments in cards)
pages 4096 bytes	dirty and unprotect page	scan dirty table for dirty pages (process objects/fragments in pages)

All schemes write only differences, for minimal log volume.

$k = 2, 3, 4, 5, 6$). As a "worst-case" scenario, we also include a scheme (scan) which does not track individual updates at all, but simply scans the entire set of resident persistent objects on checkpoint, and compares each object with its unmodified copy in the buffer cache. The virtual memory page size on the SPARCstation 2 is 4096 bytes. These schemes are summarized in Table I.

In addition to the cold through warm and hot iterations, which perform one update traversal per checkpoint, we also measure the performance for longer transactions, varying the number of hot update traversals performed as a single transaction. This allows the derivation of a precise estimate of the run-time overheads (excluding swizzling and disk accesses for retrieval) for the different write barrier mechanisms. We measure the total elapsed time for 5, 10, 15, 20, 50, 100, and 500 iterations per checkpoint.

4.1 Insert

Results for Insert are plotted in Figure 2, and summarized in Table II. The baseline non-persistent implementation treats a checkpoint as a null operation.

For cold transactions, the results for the pages scheme reveal the high cost of calls to the operating system to manage page protections. Only as the system warms up does performance for pages fall below the "worst-case" scan approach, which pays consistently high overhead to scan the entire set of resident objects on checkpoint.

The hot transaction makes insertions to the same set of objects as the last of the warm transactions. Thus, the hot transaction includes no object faults or swizzling, so the page protection scheme is no longer penalized for having to manipulate page protections during swizzling, and therefore achieves performance closer to that of the other schemes—it still incurs page traps when clean pages are modified, and must manipulate page protections at every checkpoint. For the other schemes, there is little to distinguish one from another in Figure 2 for the cold through warm transactions.

Differences are better discerned by considering the raw elapsed times given in Table II. For the cold Insert, the pages scheme is significantly more expensive (even than scan) because of the overhead to manage page protections as objects are made resident. Best is remsets closely followed by objects, since neither of these schemes incur any overhead as the resident set of objects grows, whereas

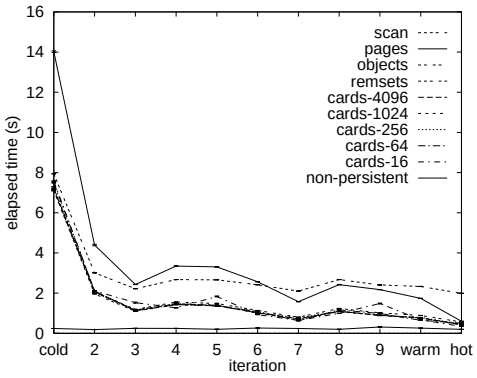


Fig. 2. Insert

Table II. Insert

Scheme	Cold	Warm	Hot
non-persistent	$0.2426 \pm_{90\%} 0.0001$	$0.26014 \pm_{90\%} 0.00009$	$0.19914 \pm_{90\%} 0.00006$
scan	$7.942 \pm_{90\%} 0.008$	$2.333 \pm_{90\%} 0.005$	$1.993 \pm_{90\%} 0.001$
objects	$7.161 \pm_{90\%} 0.007$	$0.887 \pm_{90\%} 0.003$	$0.562 \pm_{90\%} 0.002$
remsets	$7.10 \pm_{90\%} 0.02$	$0.766 \pm_{90\%} 0.004$	$0.436 \pm_{90\%} 0.002$
cards-16	$7.61 \pm_{90\%} 0.02$	$0.673 \pm_{90\%} 0.003$	$0.345 \pm_{90\%} 0.003$
cards-64	$7.55 \pm_{90\%} 0.01$	$0.646 \pm_{90\%} 0.003$	$0.511 \pm_{90\%} 0.003$
cards-256	$7.327 \pm_{90\%} 0.009$	$0.794 \pm_{90\%} 0.003$	$0.454 \pm_{90\%} 0.003$
cards-1024	$7.189 \pm_{90\%} 0.009$	$0.766 \pm_{90\%} 0.003$	$0.425 \pm_{90\%} 0.004$
cards-4096	$7.24 \pm_{90\%} 0.01$	$0.764 \pm_{90\%} 0.004$	$0.436 \pm_{90\%} 0.005$
pages	$14.03 \pm_{90\%} 0.02$	$1.722 \pm_{90\%} 0.002$	$0.585 \pm_{90\%} 0.005$

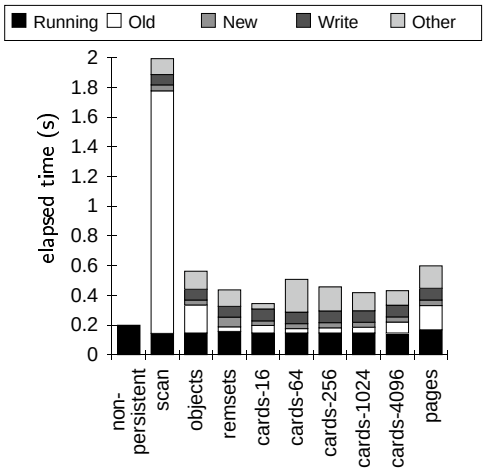


Fig. 3. Insert: Hot breakdown

the card schemes must grow their tables to cover the expanding set of resident objects. For warmer iterations, checkpoint cost is more important, so the smaller granularity schemes that require no scanning are to be preferred since they focus the unswizzling effort. Notice how these two factors influence the results among the card schemes: moderately large cards (cards-1024) are better when the system is cold, and the resident set of objects grows frequently, since growing the card tables is cheaper; smaller cards are better for the warm (cards-64) and hot (cards-16) iterations where checkpoint overheads dominate.

For a better understanding of the behavior of the hot results, Figure 3 shows the breakdown of the elapsed time for each phase of execution of the benchmark:

—*running*: time spent in the interpreter executing the program, as opposed to unswizzling old and new objects to generate differences and writing those differences to the log (note that running includes the cost of noting modifications as they occur);

—*old*: time to unswizzle old modified objects and generate log entries for them;

—*new*: time to unswizzle new objects and generate log entries for them;

—*write*: time to flush the log entries to disk; and

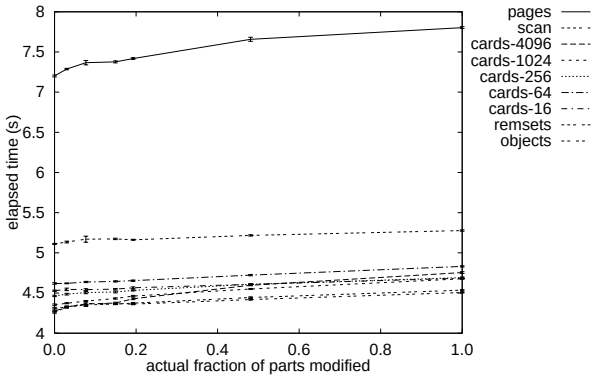
—*other*: time for any remaining bookkeeping activities, such as modifying page protections, and scavenging free transient memory space.

The most interesting feature of Figure 3 is the *old* component, which reflects the amount of scanning required to determine the differences between a cached object and its original in the client buffer pool. For the card-based schemes there is an evident tradeoff between the size of the card table and the card size: small cards require more overhead to scan the card table but less overhead in scanning the cards themselves; larger cards have a smaller table, but more overhead to scan the larger cards. Variation among the schemes in the other components is due less to the intrinsic costs of the schemes than to subtle underlying hardware cache effects: that the *other* component exhibits such variation is a result of the scavenging of transient space having markedly different cache performance across schemes. Thus, we refrain from further discussion of the results for Insert, and move on to those for the Update benchmark, which affords more precise control of benchmark parameters.

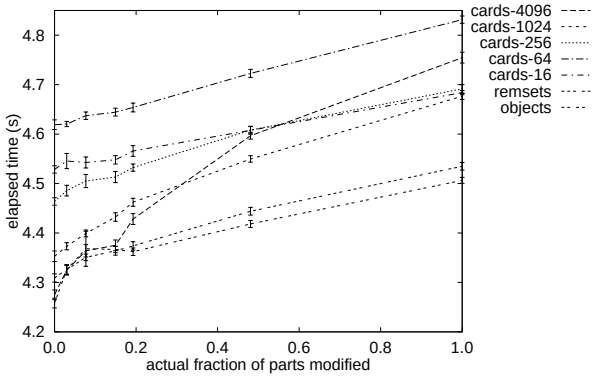
4.2 Update

Comparison of the results for the implementation alternatives is easier if we consider the key cold, warm, and hot results separately.

4.2.1 *Cold Update.* Figure 4 presents the elapsed time for the first (cold) iteration at each of the update probabilities; Figure 4(b) repeats the plot at an expanded scale omitting the results for pages and scan. There is little variation with update probability, since the cold times are dominated by I/O and swizzling costs. Nevertheless, pages is significantly more expensive (worse even than scan) due to the overheads of page protection management, both to trap updates to protected pages and to manipulate page protections during swizzling. Best overall is objects, closely followed by remsets—neither of these schemes incurs any additional overhead as the



(a) all schemes



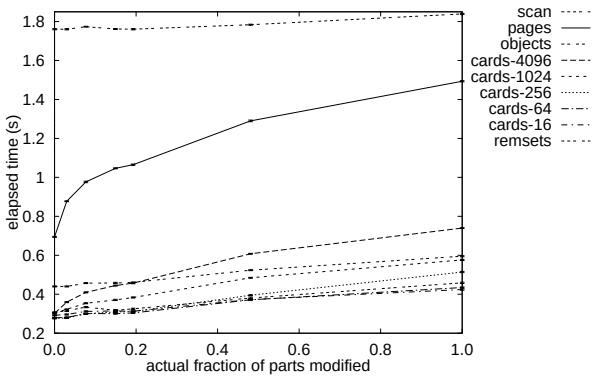
(b) expanded scale

Fig. 4. Cold Update

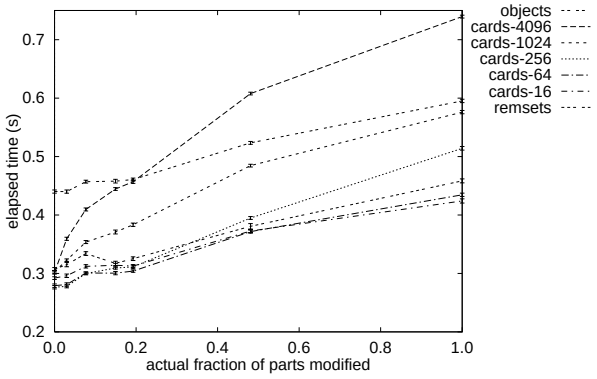
resident set of objects grows; in contrast, the card schemes must grow their tables to cover the expanding cache of resident objects.

4.2.2 *Warm Update.* At the tenth (warmest) iteration (Figure 5), checkpoint cost becomes more important. Worst overall is scan, since it must unswizzle all resident objects to generate the log. Next worst is pages, again because of the overhead to manage page protections. The card schemes are ranked by size, with smaller cards providing more precise information as to which objects are modified. The remsets scheme has performance very close to that of the smaller card schemes, because it concisely records just those objects that are modified.

4.2.3 *Hot Update.* The hot transaction traverses exactly the same parts as the tenth (warm) iteration, by beginning at the same part. Thus, the hot transaction incurs no object faults or swizzling. The hot results (Figure 6) are similar to those for the warm transaction, except that with all objects needed by the traversal having already been cached, no fetching and swizzling of objects occurs. Thus, the page protection scheme is no longer penalized for having to manipulate page protections during swizzling, and therefore achieves performance closer to that of the page-sized card scheme. The remaining difference between these schemes is



(a) all schemes

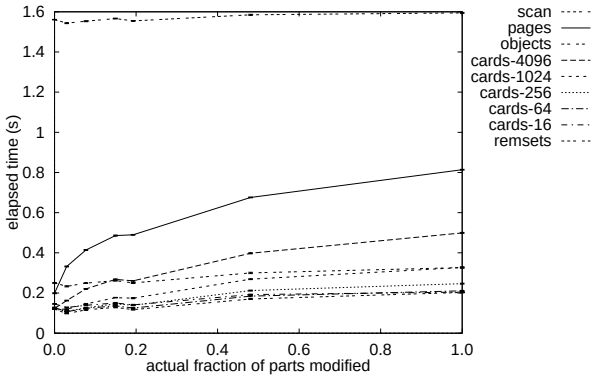


(b) expanded scale

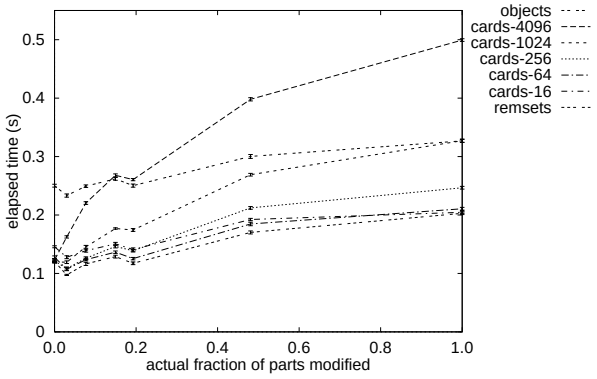
Fig. 5. Warm Update

explained by the need to manipulate the protection of dirty pages on checkpoint.

The breakdowns of the elapsed time for the hot Update at each update probability (p) are plotted in Figures 7(a)-(g) (note that the scale changes as update probability increases), omitting the results for scan and non-persistent. As we saw with Insert, checkpoint latency dominates, with the *old* component being the decisive difference among schemes, particularly at larger update probabilities. The tradeoff between card table size and card size is once again evident. For lower update probabilities the cost of scanning the card table exerts more influence; schemes with small cards but a larger card table fare worse than larger cards. At higher update probabilities there are more dirty cards to process, so unswizzling overheads dominate those of scanning the card table, with larger cards requiring more unswizzling to generate differences than smaller cards. The tradeoff is most pronounced for the 16-byte cards, which are substantially smaller than the average object size, so that unswizzling costs outweigh card table scanning costs only at the higher update probabilities. Overall, remembered sets offer the most concise record of updates, allowing modified objects to be unswizzled without scanning. The scanning overhead is clearly evident for the object marking scheme, especially at low update probabilities, where objects has the worst performance.



(a) all schemes



(b) expanded scale

Fig. 6. Hot Update

Apart from the pages scheme, variation among the schemes in the *running* component is not significant, indicating that checkpoint overheads are the dominating influence for this short transaction benchmark. The pages scheme does incur significant run-time overhead because of the high cost of the traps to note updates. Note that although no *part* objects are modified for $p = 0.0$, some updates do occur to other objects in the system, which cause page traps.

Fig. 7. Update: Hot breakdown

4.2.4 *Long transactions.* Run-time overheads come into play only when transactions are long enough for computation to dominate checkpoint overhead. The final set of results concerns the experiments in which multiple hot update traversals are performed as a single transaction. We generalize the results, by obtaining linear regression fits for each scheme, for the model $y = a+bx$, where y is the total elapsed time, and x the number of update traversals per transaction. As expected, since a hot traversal will have constant cost no matter how many times it is performed, the fits are excellent. The y-axis intercept, a, approximates the checkpoint latency, and has the familiar form we have seen for short-running transactions (see Figure 8).

The slope b is a measure of the per-traversal run-time costs of each scheme, plotted in Figure 9(a). The remsets scheme has the highest overhead to note updates. The card schemes are clustered together in the mid-range of overhead, while objects has the least measured overhead apart from scan. Curiously, pages has the second worst measured overhead per update, even though it incurs no additional cost for subsequent updates to modified pages. The overhead to field these page protection traps is thus constant for each value of x in the regression, so the trap overhead is extracted as a component of the a coefficient.

As it turns out, the pages scheme is the victim of anomalous hardware cache behavior. Indeed, taking per-traversal instruction references as our measure (plotted in Figure 9(b)), pages has overhead equivalent to that of scan. The anomaly is revealed in Figures 9(c) and (d), where we see that there is contention between data and instructions for cache lines in the unified instruction and data cache of the SPARCstation 2. Subsequent inspection of the cache simulation results indicates that the contention is restricted to a single instruction and data location, both of which are accessed for each part modified in the Update traversal—the slopes of the lines for pages in Figures 9(c) and (d) are approximately 1 miss per modified part. There is a similar anomaly for remsets, but only at update probabilities 0.15 and 0.2—because the remembered set data structure grows dynamically, contention between code and data also varies dynamically.

Taking linear regressions for the per-traversal elapsed time and instruction references, versus the actual number of parts modified (i.e., the slopes of the lines in Figures 9(a) and (b)), we obtain a measure of the run-time overhead for each scheme per part modified, given in Table III. Taking the non-persistent/scan results as the base level of overhead required to perform an unrecorded update, we calculate the net update overhead that can be attributed to each of the schemes as:

$$\text{net overhead per update} = \frac{\text{overhead per part} - \text{scan overhead per part}}{2}$$

The division by 2 reflects the fact that modifying a part actually consists of two updates: one to increment the x attribute, and one to increment the y attribute. The net overhead per update is also given in Table III. The adjusted pages result is obtained by subtracting 25 cycles to correct for the two anomalous read misses per modified part incurred by the pages scheme ($\frac{2\times 25}{2} = 25$ cycles).

As validation of these results, the observed instruction overheads for the different schemes correspond to the actual instruction overheads revealed upon inspection of the code generated by the compiler. Indeed, the cards-256 scheme requires only three instructions instead of the four used for the other card sizes because it shares

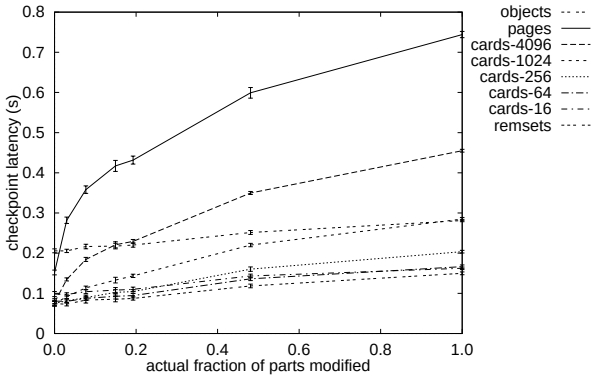
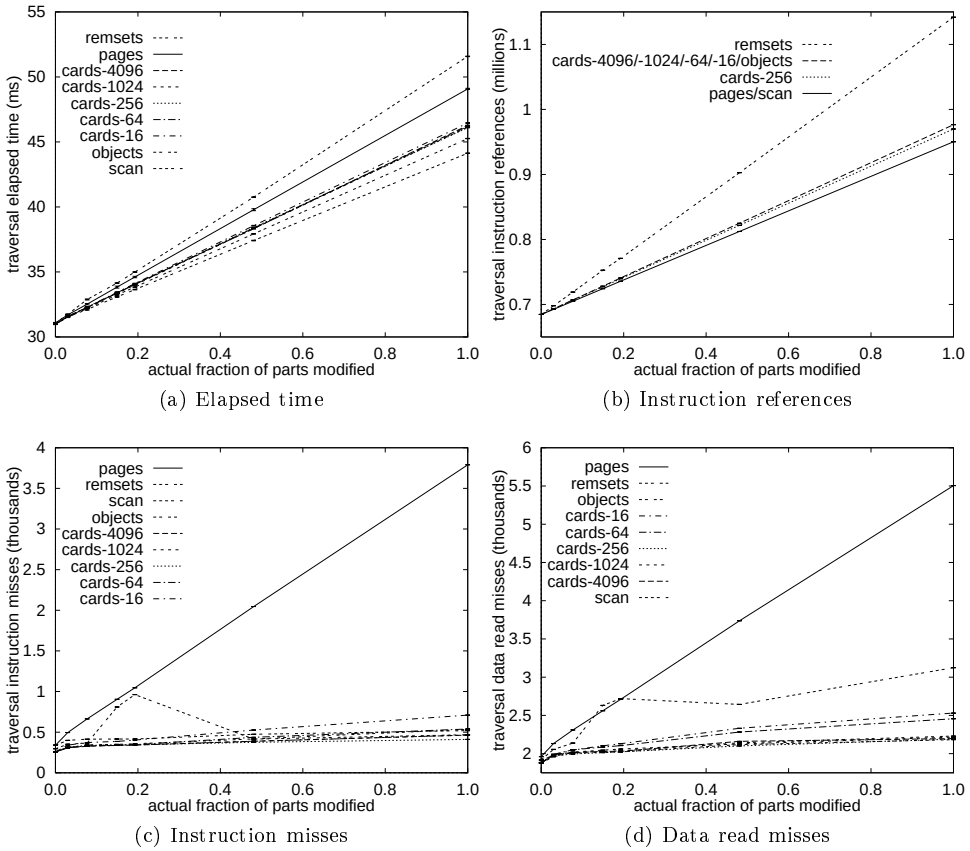


Fig. 8. Long-running Update: checkpoint latency



(a) Elapsed time (b) Instruction references

(c) Instruction misses (d) Data read misses

Fig. 9. Run-time overheads

Table III. Long-running Update: run-time overheads

	per part modified		Net (per update)	
Scheme	Time (cycles)	Instructions	Time (cycles)	Instructions
non-persistent	$159 \pm_{90\%} 0$	$81 \pm_{90\%} 0$	$0 \pm_{90\%} 0$	$0 \pm_{90\%} 0$
scan	$159 \pm_{90\%} 1$	$81 \pm_{90\%} 0$	$0 \pm_{90\%} 1$	$0 \pm_{90\%} 0$
objects	$173 \pm_{90\%} 1$	$89 \pm_{90\%} 0$	$7 \pm_{90\%} 1$	$4 \pm_{90\%} 0$
remsets	$249 \pm_{90\%} 4$	$139 \pm_{90\%} 1$	$45 \pm_{90\%} 2$	$29 \pm_{90\%} 1$
cards-16	$188 \pm_{90\%} 2$	$89 \pm_{90\%} 0$	$15 \pm_{90\%} 1$	$4 \pm_{90\%} 0$
cards-64	$185 \pm_{90\%} 1$	$89 \pm_{90\%} 0$	$13 \pm_{90\%} 1$	$4 \pm_{90\%} 0$
cards-256	$183 \pm_{90\%} 2$	$87 \pm_{90\%} 0$	$12 \pm_{90\%} 1$	$3 \pm_{90\%} 0$
cards-1024	$184 \pm_{90\%} 1$	$89 \pm_{90\%} 0$	$13 \pm_{90\%} 1$	$4 \pm_{90\%} 0$
cards-4096	$185 \pm_{90\%} 2$	$89 \pm_{90\%} 0$	$13 \pm_{90\%} 1$	$4 \pm_{90\%} 0$
pages (raw)	$219 \pm_{90\%} 1$	$81 \pm_{90\%} 0$	$30 \pm_{90\%} 1$	$0 \pm_{90\%} 0$
pages (adjusted)	$169 \pm_{90\%} 1$	$81 \pm_{90\%} 0$	$5 \pm_{90\%} 1$	$0 \pm_{90\%} 0$

$$\text{net overhead per update} = \frac{\text{overhead per part} - \textsf{scan overhead per part}}{2}$$

code with the write barrier for the generation scavenging garbage collector, which also uses 256-byte cards.

These results precisely measure the run-time overheads of each of the schemes. The page protection scheme (pages) offers the least overhead per update of all the schemes that record updates (i.e., apart from scan and non-persistent), since each transaction entails many repeated updates to the same locations, so that only the first update to a location causes a page trap. Remaining updates proceed with no additional overhead. Note that the remaining 5 cycles overhead for pages is probably anomalous, unless it indicates lingering cache disturbance due to page protection traps. Meanwhile, the software-mediated card schemes show only marginally higher overhead, while the remsets scheme incurs the high cost of a call to hash the updated location into the remembered set on every update.

Lastly, we also determine the break-even point between pages and cards-4096 for the hot Update benchmark, as the number of Update traversals, t, required for the up-front cost of pages (incurred in managing page protections at checkpoints as well as the per-transaction run-time cost of the page traps) to equal the per-update run-time costs of cards-4096. This is calculated by taking their difference in checkpoint latency (as plotted in Figure 8), say $p - c$, and dividing by the cards-4096 net cost per update (from Table III), $\frac{13 \text{ cycles}}{40\text{MHz}}$, times the number of updates per traversal, $2m$ where m is the number of parts modified per traversal:

$$t = \frac{p - c}{\frac{13}{40000000} \times m \times 2}$$

The calculations are summarized in Table IV. Note how the frequency/density of update affects the tradeoff between the high per-transaction cost of pages versus the run-time costs of cards-4096: amortization of the up-front per-transaction overheads occurs with fewer Update traversals for higher update probabilities. The high break-even points show that pages is preferable only in extreme cases, when transactions are particularly long or updates extremely frequent and dense.

Maintaining the card table base in a register will shift the breakeven points

Table IV. Long-running Update: break-even points for cards-4096 versus pages

update probability	checkpoint latency (seconds) pages (p)	cards-4096 (c)	parts modified per traversal (m)	break-even point (traversals, t)
0.00	$0.152 \pm_{90\%} 0.006$	$0.077 \pm_{90\%} 0.006$	0	∞
0.05	$0.282 \pm_{90\%} 0.008$	$0.135 \pm_{90\%} 0.004$	98	$2308 \pm_{90\%}$ 366
0.10	$0.358 \pm_{90\%} 0.009$	$0.184 \pm_{90\%} 0.005$	252	$1062 \pm_{90\%}$ 167
0.15	$0.42 \pm_{90\%} 0.01$	$0.221 \pm_{90\%} 0.008$	490	$625 \pm_{90\%}$ 105
0.20	$0.43 \pm_{90\%} 0.01$	$0.229 \pm_{90\%} 0.005$	632	$489 \pm_{90\%}$ 74
0.50	$0.60 \pm_{90\%} 0.01$	$0.350 \pm_{90\%} 0.004$	1577	$244 \pm_{90\%}$ 32
1.00	$0.744 \pm_{90\%} 0.008$	$0.455 \pm_{90\%} 0.003$	3280	$136 \pm_{90\%}$ 16

$$t = \frac{p - c}{\frac{13}{40000000} \times m \times 2}$$

even more in favor of cards, by eliminating the two instructions to load the base, leaving just two instructions to index and store to the appropriate table location. Eliminating the load also removes the possibility of a read miss on the data cache, so it is reasonable to expect that the ratio of cycles to instructions per update will also improve.

4.3 Summary

The results show a clear ranking among the alternative schemes, with approaches that record updates at smaller granularities having a significant advantage when the transactions are short and the update locality is poor, since they greatly reduce the checkpoint overheads of unswizzling and generation of differences for the log. For short transactions, the remembered set scheme is best over all update probabilities, since it provides a very concise summary of just those objects that have been modified. Still, small granularity cards also offer robust performance across a range of update probabilities, and have the advantage of lower and precisely bounded run-time overhead.

For longer transactions, the run-time costs of update detection come into play. Thus, the remembered set scheme loses its appeal due to the relatively high expense of managing the remembered set. The page protection scheme has the advantage that detection overhead is paid for up front in the page protection violation trap on the first write to a clean page, and subsequent updates proceed without cost. Meanwhile, the overheads of the card and object marking schemes change very little as update probability varies, with any difference being due to hardware cache effects. Even so, the differences in run-time overheads of the schemes are slight when compared to checkpoint overheads.

The transaction length is an important factor because of this tension between the run-time and checkpoint overheads of the various schemes. Long transactions are likely to produce correspondingly more updates, increasing the checkpoint latency. Only when the volume of modified data is small with respect to the length of transaction should the run-time costs of the schemes be permitted to guide the choice of update detection mechanism. The overwhelming influence of unswizzling and generation of log records indicates that the general bias should be towards the more accurate smaller granularities than to schemes with low run-time overheads.

With respect to the hardware approach embodied in the page protection scheme we have seen that it can involve substantial extra overhead for "typical" operations as represented by the benchmarks. In the abstract, the hardware approach is an attractive one. However, current realizations which must use expensive calls to the operating system seem to be limited in their effectiveness. Moreover, the large page size remains the most serious deficiency of this scheme, even if improved operating system support can succeed in lowering the costs of managing the update information through access to page dirty bits (Hosking and Moss [1993b] explore the ramifications of such support in more detail).

We offer three guidelines for the generation of recovery information in persistent systems:

—Avoid large granules of update detection, to minimize checkpoint latency.

—Choose a checkpoint frequency corresponding to the rate of generation of new update information, so that checkpoint delays are tolerable. Long-running applications that perform few updates need infrequent checkpoints.

—Use page protection mechanisms only where update locality is good and checkpoints are infrequent.

5. RELATED WORK

White and DeWitt [1992] compare the overall performance of various object faulting and pointer swizzling schemes for C++, as supported by several different persistent object stores. While our basic architecture is similar to the object caching scheme of White and DeWitt, the thrust of our study is significantly different. Instead of comparing several different architectures for persistence we keep the architecture fixed, while varying the mechanisms to generate log information. The representations we use for references to non-resident objects are much more lightweight than those of White and DeWitt, as are our mechanisms to support generation of the log.

Nevertheless, White and DeWitt's results do suggest that the method used to generate recovery information can have a significant impact on the performance of the system, with fine-grained update information being most beneficial when transactions are short and there is poor update locality. We have explored this issue directly here, addressing the specific question of which mechanisms are best, and what factors determine a method's effectiveness, within the fixed framework of a single persistent programming language implementation.

In a subsequent study, White and DeWitt [1994] speculate on the advantages to be gained in augmenting the language implementation to generate log information instead of relying on memory mapping mechanisms to generate per-page difference records. This is precisely what we have done here with our studies of different mechanisms for noting updates for logging.

Most recently, White and DeWitt [1995] consider several approaches to generating recovery information in a client-server environment with full-blown database concurrency control mechanisms. Their system places the log at the server, for increased availability in the case that a client crashes—the server can continue to process the log requests of other clients. With the log placed at the database server network transmission times tend to dominate, swamping any differences in

the mechanisms used to track updates and generate log information. Rather, White and DeWitt focus on the actual log information generated and its effect on performance, where we produce the same simple difference log for all write detection mechanisms under consideration. There is nothing to prevent us using similar techniques to those of White and DeWitt to reduce the amount of information actually written to the log.

Zekauskas et al. 1994 explore the performance of software approaches to the detection of writes to fine-grained data for the maintenance of cache coherency in a distributed shared memory system. Their results show that software write detection can support such sharing with lower overhead than a corresponding trap-based scheme. Moreover, they confirm our own experience that page-sized granularities incur unnecessary overhead when processing the writes, which is the dominant factor affecting performance in their system. In their case, fine-grained update information minimizes the amount of data that must be transferred at synchronization points to maintain consistency of the distributed shared memory.

Thekkath and Levy [1994] describe an approach to performing efficient handling of synchronous exceptions by user-level code. Their implementation reduces the overhead of exception handling by an order of magnitude, which may be sufficient to make a hardware trap-based approach to update detection more acceptable. Nevertheless, the issue of page-sized granularity is still of concern, since this is the critical factor affecting checkpoint latencies. Thekkath and Levy discuss the possibility of sub-page protection mechanisms driven by the paging hardware. In their proposed implementation, accessing an unprotected sub-page that lies in the same page as some protected sub-page still causes an exception, whereupon the kernel must emulate the offending instruction before returning. It is unclear if this additional overhead will degrade performance to unacceptable levels.

6. FUTURE WORK

Several avenues of research deserve further exploration: expanding the range of processors and operating systems measured, to see if there are any shifts in behavior; comparing the tradeoffs between the byte-map used here for the card marking and page trap approaches with a more compact bit-map implementation; keeping a pointer to the base of the bit-/byte-map in a register to reduce indexing overhead; and obtaining similar measurements in a compiled language setting.

The fact that the results have been obtained for an interpreted language cannot be taken lightly, since run-time overheads for interpretation 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. We acknowledge that compilation will shrink the running-time portion of total execution, so that the overheads of write detection will become more pronounced *relative* to total execution time. However, the various mechanisms should retain their rankings with respect to one another, since their absolute costs will remain the same (modulo shifting and possibly spurious hardware cache effects).

Moreover, compile-time derivation of control-flow information may reveal opportunities for the merging or elimination of explicit write barrier code, For example, where several updates to a given object occur within one basic block, just one operation to note the update is necessary. Such optimizations will reduce the importance

of the run-time overheads of the software schemes, so that checkpoint overheads become the dominant factor influencing the choice of write detection scheme. We are exploring the effects of compile-time optimization in our implementation of Persistent Modula-3 [Hosking and Moss 1990; Hosking and Moss 1991; Moss and Hosking 1994].

7. CONCLUSIONS

We have explored several lightweight implementations of the write barrier for fine-grained persistence. Our experiments used established benchmarks to compare the performance of alternative realizations of these mechanisms within a prototype persistent programming language. Most importantly, we have demonstrated that software-mediated techniques can be a competitive alternative to hardware-assisted techniques.

The results have implications beyond the realm of persistence. At a general level, the results are an indictment of the performance of operating system virtual memory primitives. Both the overhead to field page protection violations within user-level signal handlers, and the high cost of calls to the operating system to modify page protections mean that applications relying on these primitives pay unnecessarily high overheads. However, much of the problem lies with the large granularity (relative to fine-grained objects) of virtual memory pages in modern operating systems and architectures, which significantly affects overall performance. As processor speeds improve, and physical memories grow, page sizes are likely to become larger, further degrading the performance of virtual memory solutions in applications that have a naturally smaller granularity.

It is worthwhile noting that similar results have been obtained in other domains, such as efficient implementation of data breakpoints for debuggers [Wahbe 1992] and generational garbage collection [Hosking et al. 1992; Hosking and Moss 1993b]. We concede that sub-page protection and dirty bits, along with appropriate operating system interfaces, might somewhat overcome the performance disadvantages we observed [Thekkath and Levy 1994]. However, it is clear that while user level virtual memory primitives offer transparent solutions to various memory management problems, requiring no modification of the programming language implementation (except in the run-time to handle protection violations) they do not *necessarily* offer the best performance.

In this particular setting, the overhead required for software write detection is more than made up for with the precision of the information obtained. Additionally, the card marking approaches have other advantages: bounded overhead per write, and elimination of conditional code at the update site. These advantages are sure to become more important if the current trend in architectural advances continues.

REFERENCES

Appel, A. W. and Li, K. 1991. Virtual memory primitives for user programs. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, California, pp. 96–107.

Atkinson, M., Chisolm, K., and Cockshott, P. 1982. PS-Algol: an Algol with a persistent heap. *ACM SIGPLAN Notices 17,* 7 (July), 24–31.

Atkinson, M. P., Bailey, P. J., Chisholm, K. J., Cockshott, P. W., and Morrison, R.

1983. An approach to persistent programming. *The Computer Journal 26,* 4 (Nov.), 360–365.

Atkinson, M. P. and Buneman, O. P. 1987. Types and persistence in database programming languages. *ACM Computing Surveys 19,* 2 (June), 105–190.

Atkinson, M. P., Chisholm, K. J., Cockshott, W. P., and Marshall, R. M. 1983. Algorithms for a persistent heap. *Software: Practice and Experience 13,* 7 (March), 259–271.

Bell, J. R. 1973. Threaded code. *Commun. ACM 16,* 6 (June), 370–372.

Brown, A. L., Dearle, A., Morrison, R., Munro, D. S., and Rosenberg, J. 1990. A layered persistent architecture for Napier88. Technical report (May), University of St Andrews, Bremen, Germany.

Brown, A. L., Mainetto, G., Matthes, F., Mueller, R., and McNally, D. J. 1991. An open system architecture for a persistent object store. Research Report CS/91/9, University of St Andrews.

Cattell, R. G. G. and Skeen, J. 1992. Object operations benchmark. *ACM Transactions on Database Systems 17,* 1 (March), 1–31.

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

Cmelik, B. and Keppel, D. 1994. Shade: A fast instruction-set simulator for execution profiling. In *Proceedings of the ACM Conference on the Measurement and Modeling of Computer Systems*, pp. 128–137.

Date, C. J. 1983. *An Introduction to Database Systems*, Volume II. Addison-Wesley.

Date, C. J. 1986. *An Introduction to Database Systems* (Fourth ed.), Volume I. Addison-Wesley. Corrected in 1987.

Elhardt, K. and Bayer, R. 1984. A database cache for high performance and fast restart in database systems. *ACM Transactions on Database Systems 9,* 4 (Dec.), 503–525.

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

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

Hosking, A. L. 1995. *Lightweight Support for Fine-Grained Persistence on Stock Hardware.* Ph.D. thesis, University of Massachusetts at Amherst. Available as Department of Computer Science Technical Report 95-02.

Hosking, A. L., Brown, E., and Moss, J. E. B. 1993. Update logging for persistent programming languages: A comparative performance evaluation. In *Proceedings of the International Conference on Very Large Data Bases*, Dublin, Ireland, pp. 429–440. Morgan Kaufmann.

Hosking, A. L. and Moss, J. E. B. 1990. Towards compile-time optimisations for persistence. In A. Dearle, G. M. Shaw, and S. B. Zdonik (Eds.), *Proceedings of the International Workshop on Persistent Object Systems*, Martha's Vineyard, Massachusetts, pp. 17–27. *Implementing Persistent Object Bases: Principles and Practice*, Morgan Kaufmann, 1990.

Hosking, A. L. and Moss, J. E. B. 1991. Compiler support for persistent programming. Technical Report 91-25 (March), Department of Computer Science, University of Massachusetts at Amherst.

Hosking, A. L. and Moss, J. E. B. 1993a. Object fault handling for persistent programming languages: A performance evaluation. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Washington, DC, pp. 288–303.

Hosking, A. L. and Moss, J. E. B. 1993b. Protection traps and alternatives for memory management of an object-oriented language. In *Proceedings of the ACM Symposium on Operating Systems Principles*, Asheville, North Carolina, pp. 106–119.

Hosking, A. L., Moss, J. E. B., and Stefanović, D. 1992. A comparative performance evaluation of write barrier implementations. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Vancouver, Canada, pp. 92–109.

KEMPER, A. AND KOSSMAN, D. 1994. Dual-buffering strategies in object bases. In *Proceedings of the International Conference on Very Large Data Bases*, Santiago, Chile. Morgan Kaufmann.

MCCALL, K. 1983. The Smalltalk-80 benchmarks. In G. KRASNER (Ed.), *Smalltalk-80: Bits of History, Words of Advice*, Chapter 9, pp. 153–173. Addison-Wesley.

MOSS, J. E. B. 1987. Managing stack frames in Smalltalk. In *Proceedings of the ACM Symposium on Interpreters and Interpretive Techniques*, St. Paul, Minnesota, pp. 229–240.

MOSS, J. E. B. 1990. Design of the Mneme persistent object store. *ACM Transactions on Information Systems 8,* 2 (April), 103–139.

MOSS, J. E. B. AND HOSKING, A. L. 1994. Expressing object residency optimizations using pointer type annotations. In M. ATKINSON, D. MAIER, AND V. BENZAKEN (Eds.), *Proceedings of the International Workshop on Persistent Object Systems*, Workshops in Computing, Tarascon, France, pp. 3–15. Springer-Verlag, 1995.

SHAW, R. A. 1987. Improving garbage collector performance in virtual memory. Technical Report CSL-TR-87-323 (March), Stanford University.

SOBALVARRO, P. G. 1988. A lifetime-based garbage collector for LISP systems on general-purpose computers. B.S. Thesis, Dept. of EECS, Massachusetts Institute of Technology, Cambridge.

THEKKATH, C. A. AND LEVY, H. M. 1994. Hardware and software support for efficient exception handling. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, pp. 110–119.

UNGAR, D. 1984. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the ACM Symposium on Practical Software Development Environments*, Pittsburgh, Pennsylvania, pp. 157–167.

UNGAR, D. M. 1987. *The Design and Evaluation of a High Performance Smalltalk System.* ACM Distinguished Dissertations. MIT Press, Cambridge, Massachusetts.

WAHBE, R. 1992. Efficient data breakpoints. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, Massachusetts, pp. 200–212.

WHITE, S. J. AND DEWITT, D. J. 1992. A performance study of alternative object faulting and pointer swizzling strategies. In *Proceedings of the International Conference on Very Large Data Bases*, Vancouver, Canada, pp. 419–431. Morgan Kaufmann.

WHITE, S. J. AND DEWITT, D. J. 1994. QuickStore: A high performance mapped object store. In *Proceedings of the ACM International Conference on Management of Data*, Minneapolis, Minnesota, pp. 395–406.

WHITE, S. J. AND DEWITT, D. J. 1995. Implementing crash recovery in QuickStore: A performance study. In *Proceedings of the ACM International Conference on Management of Data*, San Jose, California, pp. 187–198.

WILSON, P. R. AND MOHER, T. G. 1989a. A "card-marking" scheme for controlling intergenerational references in generation-based garbage collection on stock hardware. *ACM SIGPLAN Notices 24,* 5 (May), 87–92.

WILSON, P. R. AND MOHER, T. G. 1989b. Design of the opportunistic garbage collector. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, New Orleans, Louisiana, pp. 23–35.

ZEKAUSKAS, M. J., SAWDON, W. A., AND BERSHAD, B. N. 1994. Software write detection for a distributed shared memory. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, Monterey, California, pp. 87–100.