Update Logging for Persistent Programming Languages: A Comparative Performance Evaluation*

Antony L. Hosking Eric W. Brown J. Eliot B. Moss

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

Abstract

If persistent programming languages are to be accepted they must provide many of the standard features of traditional database systems, including resilience in the face of system failures in which the volatile database (in-memory database buffers) is lost. Ensuring the consistency of the database requires the generation of recovery information sufficient to restore the database to a consistent state after a crash. This paper examines a range of schemes for the efficient generation of recovery information in persistent programming languages, 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]. Useful persistent programming languages must support resilience in the face of system failures, an important feature of any database system. A crash results in the loss of the volatile part of the database, including all updates to persistent data that have not yet been propagated to the stable database. Recovering from such a failure involves restoring the database to some consistent state from which processing can resume. This paper considers a number of schemes for checkpoint-based generation of recovery information within a persistent programming language. A checkpoint specifies a consistent state of the database; after a crash the restored database state will be that defined by the most recent checkpoint invocation. Resilience requires the implementation of several mechanisms: detection of updates, logging of updates to stable storage (i.e., disk), and merging of the log with the stable database upon recovery. We focus on the first two mechanisms here, and their realisation within a persistent programming language.

Traditional database systems generate detailed update information for each operation that modifies the database. A log record may be written immediately to disk, or left for asynchronous output at some point later in the execution. Generating detailed update information for each mutation of the database is acceptable when the update consists of a high-overhead database operation, invoked through a call to the database subsystem. However, programs written in a persistent programming language perform frequent updates directly in memory, and often to the same locations. Thus, it seems preferable to hold off the generation of detailed update information until a checkpoint (or transaction commit), at which time log records can be generated and written as a batch. At one extreme the system might assume that all memory-resident data is subject to modification, requiring that the entire volatile database be written at checkpoint time. Given an application that modifies only a small fraction of the database this scheme will be hopelessly inefficient. A better option is to have the language's run-time system keep track of updates as they occur, so that only the modified part of the volatile database needs to be logged upon checkpoint. This minimises the volume of the log, and allows repeated updates to the same location to be subsumed, thus minimising checkpoint latency. Nevertheless, questions remain as to the mechanism by which updates are tracked, and the granularity of the recorded updates.

Two basic alternatives present themselves: updates can

*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. Eric Brown is funded by the NSF Center for Intelligent Information Retrieval at the University of Massachusetts. The authors can be reached via Internet email addresses {hosking,brown,moss}@cs.umass.edu.

Proceedings of the 19th International Conference on Very Large Data Bases
Dublin, Ireland, 1993, pp. 429–440

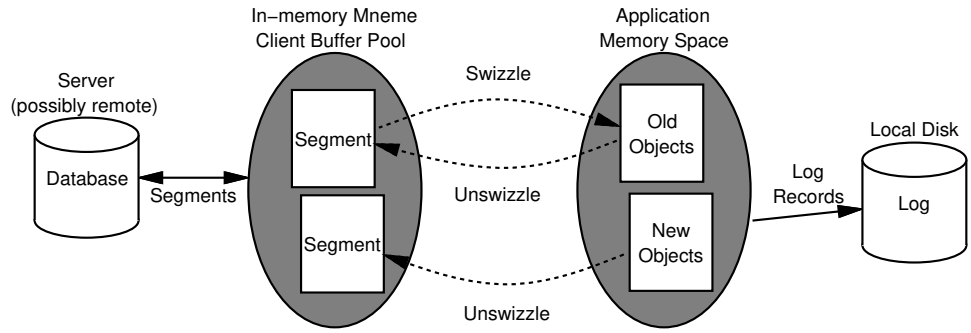

Figure 1: System architecture

be recorded on the basis of either *logical*[1] units (objects in this case since we consider an object-oriented paradigm) or *physical* units (such as virtual memory pages or some other unit of the virtual memory space). Naturally, the choice of implementation will influence both the CPU time consumed by a persistent program in the recording of updates, and the checkpoint latency in generating and writing log records.

In this paper we compare various schemes for efficiently generating recovery information in persistent programming languages. The next section describes our architecture and its rationale. We then discuss related work, drawing together influences from the fields of database systems and persistent programming languages. Succeeding sections present the competing implementations, the benchmarks used for their comparison, the experimental setup and methodology, results, and conclusions.

The paper's principal contributions include the exploration of several alternative mechanisms for the detection and logging of updates in persistent programming languages and a comprehensive comparison of their performance, in addition to performance results concerning a prototype persistent programming language, using established benchmarks recognised by the database community.

2 System architecture and rationale

Our architecture (see Figure 1) is designed to allow the language implementation maximum control over all objects being manipulated by a program, without having to go through a restrictive interface to the underlying storage manager. We use the Mneme persistent object store [12] to manage the storage and retrieval of objects from disk. Mneme groups persistent objects into *segments* for transfer to and from secondary storage, buffering the segments in main memory on the client machine as necessary. Objects are copied from the client buffer pool into the virtual memory address space of the program. This 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 are converted to direct memory pointers for manipulation by the program. This conversion of identifiers to pointers is known as *swizzling* [13]. For the purposes of this study objects are copied one at a time from the buffer pool, as opposed to all the objects of a segment being copied at once. Regardless of the comparative desirability of these alternatives, our goal was to keep this aspect of the implementation fixed, while varying the mechanisms for detecting and logging updates.

Similarly, we chose to fix on just one implementation of the *object faulting* mechanism used by the language implementation to detect references to objects not resident in the program's address space. The particular mechanism used here is not relevant to this study, since it is kept constant, while the recovery support schemes are varied. A comparison of alternative schemes for object faulting within this basic architecture appears elsewhere [9].

2.1 Persistent Smalltalk

Our implementation of Smalltalk is based on the definition of Goldberg and Robson [5]. The implementation consists of two components: the *virtual machine* and the *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, our implementation of the virtual machine differs somewhat from that described in [5]. 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

[1] We mean logical with respect to the logical granules of data in the database, not logical in the sense of operation logging.

by the virtual machine.

In a persistent implementation of Smalltalk the virtual image resides in the database, and the Smalltalk environment is booted by loading a subset of the objects in the image sufficient to resume execution by the virtual machine. The virtual machine is carefully augmented for persistence to make persistent objects resident as they are needed by the executing image.

2.2 Checkpoints

Our notion of recovery is dictated by certain assumptions about the behaviour 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 database recovery mechanism must restore the state of the database to that of the most recent checkpoint. Moreover, we assume that checkpoint latencies should be minimised 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 object. Log records are generated only if there are differences between the object and its original in the client buffer pool. Persistence in our system is based on *reachability*, so the unswizzling operation may encounter pointers to objects newly created since the last checkpoint. These objects are assigned a persistent identifier and unswizzled in turn, perhaps dragging further newly-created objects into the database, 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 the 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 slightly 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.

To integrate buffer management with the recovery model, we guarantee that a modified segment is flushed to the database 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 virtual memory address space of the program is described in [8]. That scheme is compatible with the recovery model, since modified objects that have been selected for replacement will be unswizzled and logged in a checkpoint fashion.

The recovery model is indifferent to concurrency, which can be introduced to the architecture in two ways. First, separate applications can share the same database, 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 easily be integrated into any transaction model based on the *database cache* [4].

3 Related work

This work is loosely related to the performance study by White and DeWitt [22] which compares the overall performance of various object faulting and pointer swizzling schemes for persistent programming languages. The systems considered in that study include version 1.2 of ObjectStore [11, 15], a commercially available object-oriented DBMS, and a number of software architectures based on the EXODUS Storage Manager (ESM) [2, 16].

Several of the architectures based on ESM require object updates to be carried out via a call interface, which modifies the object in the client buffer pool and generates a log record based on the changes. To avoid this call overhead White and DeWitt introduced a new architecture which they call *object caching*, and which bears a close resemblance to our own architecture. Objects are retrieved into the client buffer pool using the ESM interface, and then copied into the virtual memory of the application. The original in the buffer pool is unpinned, and a descriptor for the copied object is entered into a hash table based on the object's identifier. This descriptor contains a pointer to the object in memory along with a pair of values indicating the range of modified bytes in the object. Updates are applied directly to the object copies in virtual memory and noted by adjusting the byte range. At transaction commit, the hash table of cached objects is scanned. For each modified object, ESM is called to pin and update the object in the buffer pool and a log record is generated based on the range of modified bytes. Two versions of

this caching scheme were explored. 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 performs some pointer swizzling, with references to objects that are resident in the cache being converted to direct memory pointers. Whenever 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 object cache hash table is examined to see if it contains a pointer for the target object, in which case the location is updated with the pointer.

ObjectStore takes a dramatically different approach from the previously described architectures. Objects are faulted and pointers are swizzled using a page mapping scheme similar to virtual memory. Recovery information is generated by logging entire dirty pages. 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.

Our architecture is similar to the object caching scheme of White and DeWitt, but the thrust of our study is significantly different. While their results do suggest that the method used to generate recovery information can have a significant impact on the performance of the system, their focus is on the *overall* performance of different architectures for faulting and swizzling. In contrast, we have chosen to keep the architecture constant, while varying the mechanisms used to generate log information. This allows us to study the effects of alternative log-generation schemes in isolation, without concern for the effects of other variations in the architecture. Nevertheless, White and DeWitt's results do indicate that maintaining finer-grained update information is most beneficial when transactions are short and there is poor update locality. We explore this issue directly here, addressing the specific question of which method of generating recovery information is best, and what factors determine a method's effectiveness, all in the context of persistent programming languages.

The Texas system [18, 23] 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. When the page is actually referenced a virtual memory page trap occurs, and is handled by Texas, which reads in the persistent page from the store and maps it into the 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 page just faulted 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.

Texas tracks updates to persistent objects by write protecting virtual memory pages that may be updated. On the first write to a protected page an access violation occurs, which is handled by unprotecting the page and making a copy of it in a *clean version buffer*. At transaction commit, a modified page will have a clean copy in the clean version buffer, which is compared with the modified page to generate a log entry.

The general approach to logging and recovery used by all these systems was originally devised by Elhardt and Bayer for the *database cache* [4]. The database cache was designed for fast transaction commit and rapid recovery after a crash. Modifications are always applied to copies of original database pages in main memory (the *cache*) so that transaction abort merely requires deletion of the copies. Transactions commit by flushing dirty copies to the *safe*: a log of updated pages on stable storage. Once a dirty copy has been flushed to the safe, it becomes a changed original. Dirty copies are never flushed to the permanent database. Thus the permanent database contains only the effects of committed transactions. Similarly, the buffer manager may only select unpinned originals for replacement in the cache, flushing any changed originals to the database as necessary. Recovery involves reconstructing the cache from the safe. To keep the safe to a manageable size, it is periodically cleaned by removing log entries that are no longer necessary for recovery.

Elhardt and Bayer require locking and logging at the granularity of a page. Moss et al. [14] extend the database cache algorithms to allow locking and logging at a finer granularity. The goal of their extension is to increase concurrency and, ultimately, performance. For future work, they propose an investigation of the effects of different lock and log granularities on system performance, which we partially address here. Our system is not an exact implementation of the database cache, but we use a similar buffer management protocol and our checkpoints have semantics similar to the database cache transaction commit. Here we go beyond a comparison of logging granularities and investigate different methods for noting modifications.

The database cache, and in fact all of the logging schemes described here, are versions of the *write ahead log* from traditional database systems [6, 21, 7]. However, they are distinguished by the fact that they are intended for non-traditional database applications, in which the characteristics of data access and manipulation are quite different from traditional (e.g., relational) database systems, requiring new

mechanisms and semantics for transactions, logging, and recovery.

Motivating this study is a concern that the aforementioned schemes for detection of updates and generation of log records will prove to be overly expensive, unduly affecting the performance of persistent programming languages that use them. We are interested in quantifying the overheads of more lightweight mechanisms, inspired by our experience as programming language implementors in facing similar problems of update detection in other domains such as garbage collection [10].

4 Noting updates

The lightweight mechanisms used for detecting updates in this study are drawn from solutions to the *write barrier* problem in garbage collection: the act of storing a pointer in an object is noted in order to minimise the number of pointer locations that must be examined during any given garbage collection [10]. Similarly, efficient checkpointing requires keeping track of all updates to objects, to minimise the number of locations that must be unswizzled and logged. 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. We have implemented several versions of the three most common write barrier approaches. 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.

4.1 Object-based schemes

The first two schemes each 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. Upon checkpoint all cached objects must be scanned to find objects that are marked as updated. A marked object must be unswizzled and compared to its original in the buffer pool to determine any differences that must be logged. The drawback of this approach is additional checkpoint overhead required to scan the cached objects to find those that have been marked.

To avoid scanning, the second scheme uses a *remembered set* [20] to record persistent objects that have been modified. 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.

In order that the remembered set does not become too large we 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 an address and indexing a table containing such information. If the updated object is persistent then a subroutine is invoked to hash the object's pointer into the remembered set. On the MIPS R2000 this involves twelve inline instructions at every update.

Remembered sets have the advantage of conciseness and accuracy, achieved 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.

4.2 Card-based schemes

For small objects the object-based schemes are ideal. However, updates to large objects may suffer from poor locality with respect to the object size, resulting in unnecessary unswizzling and comparison upon checkpoint. These checkpoint overheads are bounded solely by the size of the object. For this reason, we also consider schemes that record updates based on fixed-size units of the virtual memory space. They divide the memory into aligned logical regions of size 2^k bytes—the address of the first byte in the region will have k low bits zero. These regions are called *cards*, after [19, 24]. 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: one shifts the address right by k and uses the result as an index into the table. Whenever an object is modified, the corresponding card is *dirtied*.

A variant of this scheme uses the page protection mechanism 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]

One of the most attractive features of card marking is the simplicity of the write barrier. For this reason we have chosen to implement the card table as a byte array rather than a bit map.[3] By interpreting zero bytes as dirty entries and non-zero bytes as clean, a store can be recorded using just a shift, index, and byte store of zero. On the MIPS R2000 this comes to just four instructions: a load to get the base of the card table, a shift to determine the index, an add to

[2] An operating system could more efficiently supply the information needed in the page protection scheme if it offered appropriate calls to obtain the page dirty bits maintained by most memory management hardware [17].

[3] We first heard of this idea from Paul Wilson.

determine the byte entry's address, and a byte store of zero.[4]

At checkpoint time the dirty cards containing persistent objects are scanned, to perform unswizzling and determine any differences that must be logged. 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. For these reasons we must be able to locate the object headers within a card. These encode the formats of the objects and contain their persistent identifiers. To support object header location, we maintain a table of card offsets parallel to the dirty card table, indicating the location of the *last* (highest address) object header within each card.

Dirty cards are marked clean after they have been scanned. We reduce overheads by scanning all contiguous dirty cards as a batch, running from the first to the last. The size of the cards is an important factor influencing checkpoint costs, since 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.

5 Experiments

We draw on the benchmarks used by White and DeWitt [22] for comparison of the update schemes, who in turn based their benchmarks on those of the object operations benchmark, OO1 [3].

5.1 The benchmark database

We use the OO1 benchmark database, consisting 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 selected 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 362 physical segments, for a total size of 7.6M bytes. Each segment is at least 16K bytes in size, although some may be larger since Smalltalk objects larger than 16K bytes are allocated in their own private 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. 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 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 the parts that are the source of the connections. The B-tree index for the 20,000 parts consumes a total of 168,832 bytes.

[4] Note that on the MIPS R2000 a byte store is implemented in hardware as a read-modify-write instruction, possibly requiring more than one cycle for execution. It may be preferable to code the read-modify-write byte store operation explicitly, especially on more recent machines.

5.2 Benchmarks

We performed both the lookup and traversal portions of the OO1 benchmark, as well as adopting the update variant of the traversal measure used by White and DeWitt:

- **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. This benchmark is read-only with no checkpoints.
- **Traversal** finds all parts connected to a randomly chosen part, or to a part connected to it, and so on, up to seven hops (for a total of 3,280 parts, with possible duplicates). Similarly to the lookup, a null procedure is invoked for each part, taking as its arguments the *x*, *y*, and *type* fields of the part. This benchmark is read-only with no checkpoints.
- **Update** operates in the same way as the *traversal* measure, but in addition to the null procedure call it performs a simple update to each part object encountered, with some fixed probability. The update consists of incrementing the *x* and *y* 4-byte integer fields of the part. A checkpoint operation is performed at the end of the traversal to complete the transaction and commit the changes to disk.

OO1 calls for each measure to be 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 initialise 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 32M byte 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 time to run ten *hot* iterations of the traversal and update benchmarks, 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. To get a sense of the CPU overheads for noting updates in long-running transactions we also measured the time to run repeated hot iterations as a single transaction, varying the number of iterations per checkpoint.

5.3 Experimental setup

The client machine on which the benchmarks were run was a DECStation 3100 (MIPS R2000A CPU[5] clocked at 16.67MHz) running ULTRIX[6] 4.1.[7] The system has 24M bytes of main memory, 10% of which is used for operating system disk buffers. The log file is written locally to a relatively empty RZ56 drive (665M bytes SCSI, 24.3ms average access time, 16ms average seek time, 64K-byte data buffer), using ordinary buffered file I/O (as opposed to the raw disk device). The Smalltalk interpreter is coded in C and compiled with the GNU C compiler (**gcc**) version 2.3.3 at optimisation level 2. The benchmarks were run with the system in single user mode and the process's address space was locked in main memory to prevent paging. All checkpoints included a call to `fsync` to force the log data to the local disk before completing.

The database is accessed remotely via NFS. The server is a SPARCstation[8] 2 running SunOS[9] 4.1.2, with 32M bytes of main memory, and the database resides on a 1.3G byte 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[10] 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, unswizzling of modified persistent objects, allocation and unswizzling of newly created persistent objects, writing the log records to disk (including the `fsync`), and other checkpoint overheads.

Our experiments included runs for the object marking, remembered set, page protection (the page size is 4096 bytes) and card schemes, with card sizes of 16, 64, 256, 1024 and 4096 bytes. For the update benchmarks we ran the experiments with update probabilities of 0.00, 0.05, 0.10, 0.15, 0.20, 0.50, and 1.00. The experiments were repeated several times for each configuration, and the results averaged (variance of the individual results from the mean was not significant). Each run is presented with the exact same database

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

[6] DECstation and ULTRIX are registered trademarks of Digital Equipment Corporation.

[7] The operating system had some official patches installed that fix bugs in the `mprotect` system call.

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

[9] SunOS is a trademark of Sun Microsystems.

[10] We thank Digital Equipment Corporation's Western Research Laboratory, and Jeff Mogul in particular, for giving us the high resolution timing board and the software necessary to support it.

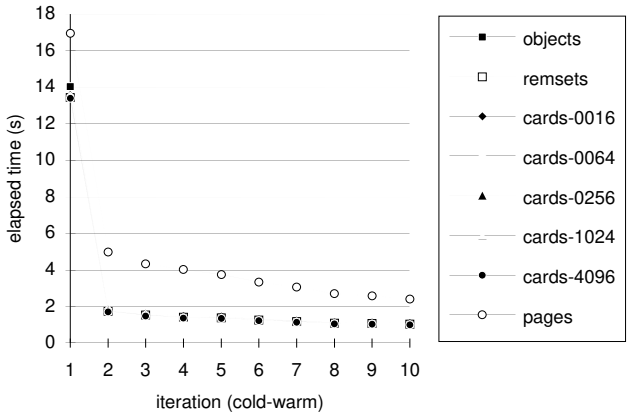

Figure 2: Lookup: 10 iterations (1 cold, 9 warm)

(no updates are ever propagated to the permanent database). Note also that the nth iteration within any given benchmark run will always access the same parts as the nth iteration within any other benchmark run, since the script that controls the benchmark runs presents the same sequence of random part identifiers to each run.

6 Results

We now report the results for each of the benchmarks. All times are reported in seconds, and exclude all initialisation time (i.e., all Smalltalk initialisation prior to beginning the benchmark). In each figure, the card schemes are identified with the card size in bytes, the page protection scheme is referred to as *pages*, the object marking scheme as *objects*, and the remembered set scheme as *remsets*.

6.1 Lookup

The behaviour of the lookup benchmark for the ten iterations (cold through warm) is illustrated in Figure 2, which gives the elapsed time for each iteration of the benchmark. We see the warming effect of the cache as the iterations proceed. As expected from a benchmark that performs no update there is little to distinguish the software update detection schemes. However, the page protection scheme incurs substantial extra overhead, caused by the need to protect and unprotect the pages in which newly faulted objects are cached. Recall that clean pages are write protected. Before a newly faulted object can be copied and swizzled into a clean page, the page must be unprotected. After swizzling has occurred, the page must be reprotected. We expect that schemes that perform page-at-a-time caching will lessen, but not eliminate, the impact of this page protection management overhead by performing more copying and swizzling work per object fault.

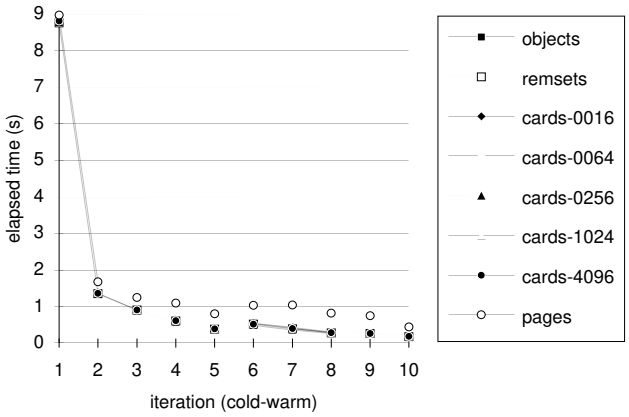

Figure 3: Traversal: 10 iterations (1 cold, 9 warm)

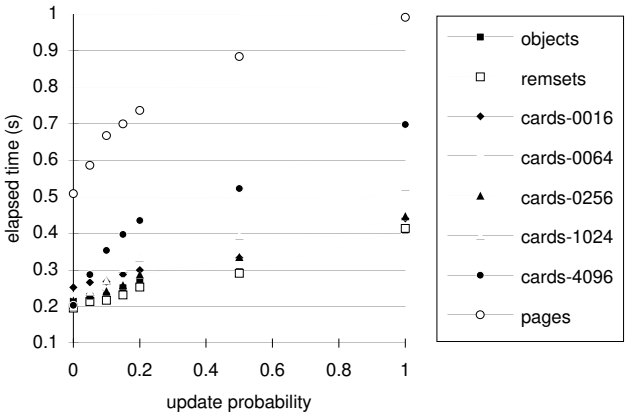

Figure 5: Warm update (tenth iteration)

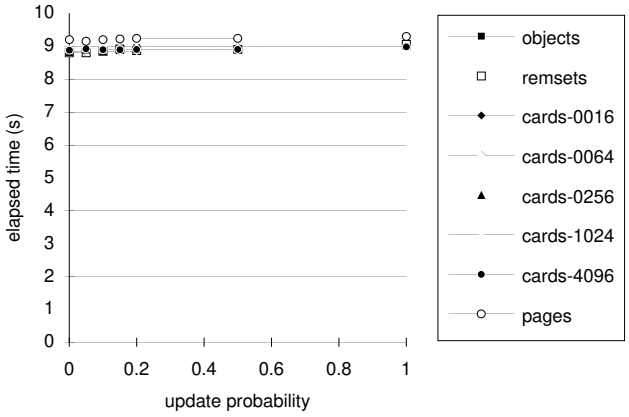

Figure 4: Cold update (first iteration)

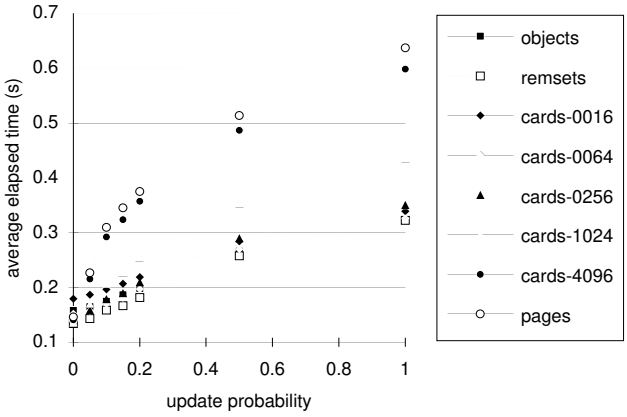

Figure 6: Hot update

6.2 Traversal

The results for the ten cold through warm iterations of the traversal benchmark are illustrated in Figure 3. The read-only traversal behaves similarly to the lookup, with the warming of the cache evident as the iterations proceed. Once again, the overheads for management of page protections are apparent.

6.3 Update

The update benchmark includes a checkpoint operation, so the results are naturally more interesting. Figure 4 presents the elapsed time for the first (cold) iteration at each of the update probabilities. There is little relative variation amongst the schemes since the cold times are dominated by I/O and swizzling costs. Nevertheless, the page protection approach is somewhat more expensive due to the overheads of page protection management.

Elapsed times for the tenth (warmest) iteration at each update probability are given in Figure 5. Here we begin to see the true differences among the schemes, becoming more pronounced as the update probability increases. Object marking and remembered sets are best overall, with remembered sets slightly better than object marking at the lower update probabilities. At the higher update probabilities object marking and remembered sets exhibit similar performance. The card schemes come close to the object-based schemes only at low update probabilities. The page protection approach is markedly worse than all other schemes across the whole range of update probabilities.

6.4 Hot update

The ten hot transactions traverse exactly the same parts as the last of the ten cold-warm iterations, by beginning each hot iteration at the same part. Thus, the hot iterations include no object faults or swizzling. Figure 6 summarises the average elapsed time for the ten hot iterations at each of the update probabilities. The results 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

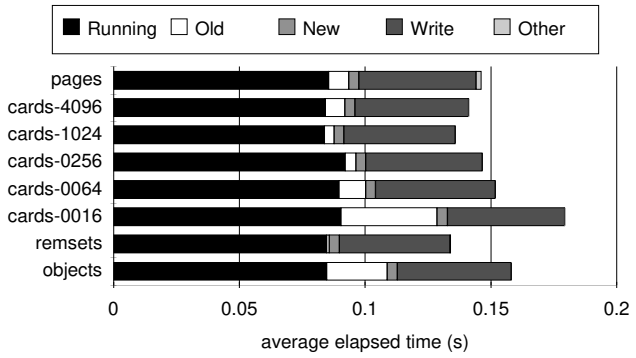

Figure 7: Hot update: $p = 0.0$

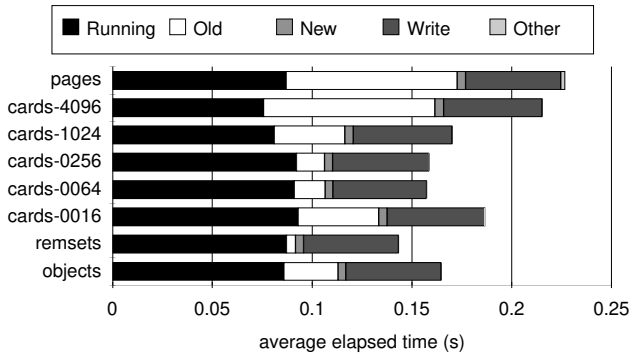

Figure 8: Hot update: $p = 0.05$

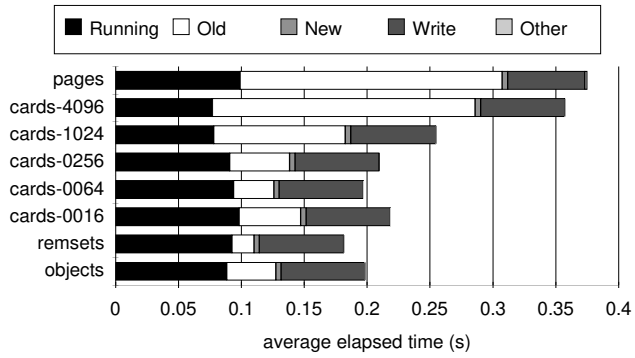

Figure 9: Hot update: $p = 0.2$

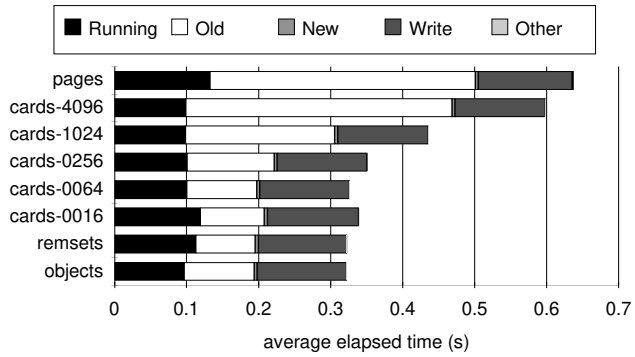

Figure 10: Hot update: $p = 1.0$

of objects occurs. Thus, the page protection scheme is no longer penalised 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 two schemes is explained by the need to manipulate the protection of dirty pages at checkpoint time.

For a better understanding of the behaviour of the hot runs Figures 7-10 show the breakdown of the average elapsed times at several update probabilities (p) for each phase of execution:

- *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.

The results show that the major differences among the schemes occur in the *running* and *old* components. Variation among the schemes in the *running* component is not all that great, with the differences being due not only to the intrinsic run-time overheads of the schemes associated with noting updates, but also to subtle underlying effects such as variations in hardware cache behaviour. This is the only possible explanation for variation among the card schemes, since the code for all the card schemes is exactly the same, barring the shift values.

The *old* component reflects the amount of scanning required to determine the differences between a cached object and its original in the client buffer pool, and has the most influence on total elapsed time, particularly at larger update probabilities. For the card-based schemes there is an evident tradeoff between the size of the card table and the card size. At the smaller update probabilities the cost of scanning the card table has 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

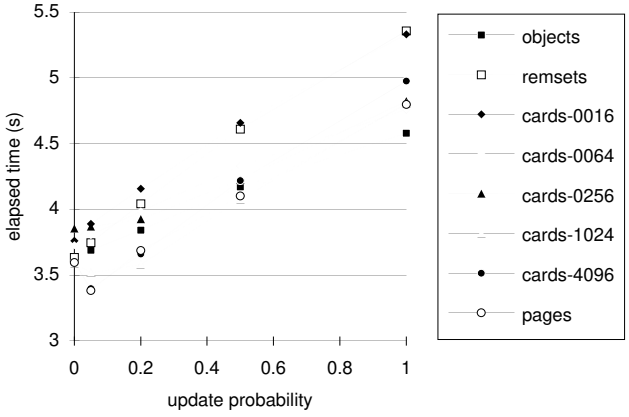

Figure 11: Hot update (50 iterations per checkpoint)

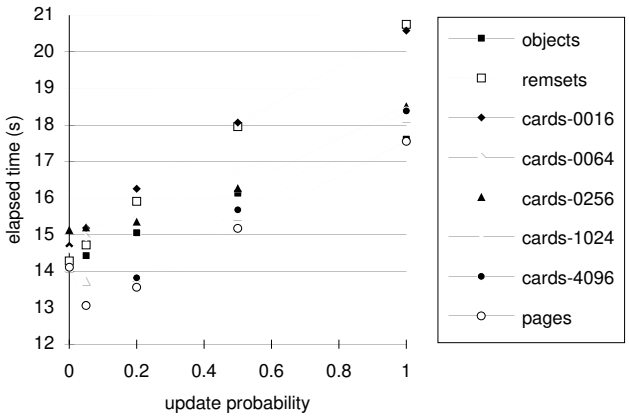

Figure 12: Hot update (200 iterations per checkpoint)

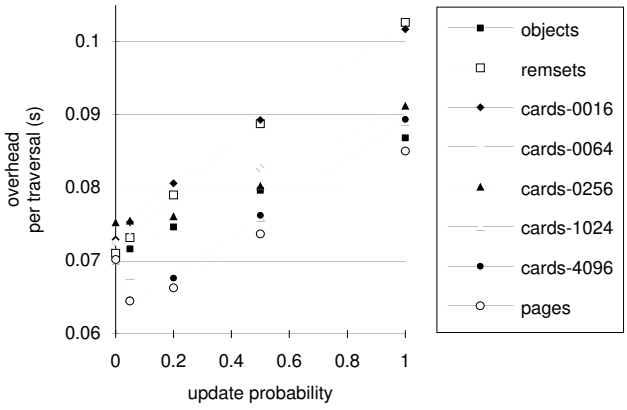

Figure 13: Long-running update: slope b

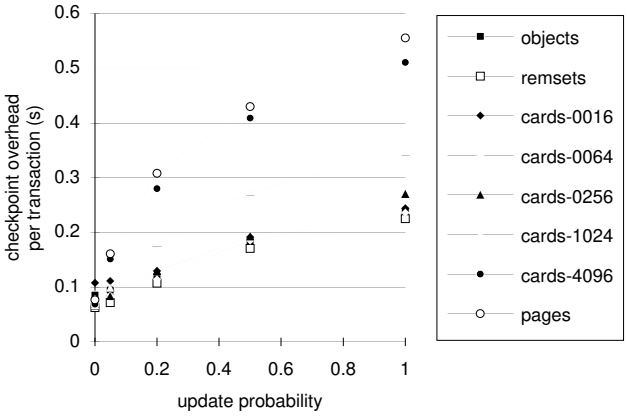

Figure 14: Long-running update: intercept a

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 evident for the object marking scheme, especially at low update probabilities.

6.5 Long-running transactions

The final set of results concerns the experiments in which multiple hot update traversals are performed as a single transaction. We measured the total elapsed time for 50, 100, 150, 200, 400, 600, 800 and 1000 iterations per checkpoint, at update probabilities 0.00, 0.05, 0.20, 0.50, and 1.00. The point of this was to try to obtain some estimate of the relative overheads incurred by each scheme in noting updates in long-running transactions.

Figure 11 plots the elapsed time for a transaction consisting of 50 iterations as update probability is varied. Similarly, Figure 12 shows the results for 200 iterations per checkpoint. These results illustrate how the relative importance of the per-traversal costs and per-checkpoint costs of each scheme varies with the length of the transaction. The longer the transaction the more important the cost of detecting and noting updates (cf. Figure 6). Most dramatically, the page protection scheme becomes more attractive as the length of the transaction increases. At 200 update traversals per transaction the page protection scheme is best at all update probabilities.

We have generalised these 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 slope b is a measure of the per-traversal costs of each scheme, while the y-axis intercept a approximates the checkpoint overhead per transaction. These measures are plotted in Figures 13 and 14.

The page protection scheme offers the least overhead per traversal of all the schemes (Figure 13), 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. Nevertheless, the software-mediated update detection schemes show only marginally worse per-traversal overheads, within 15% of the performance of the page protection approach. For larger card sizes the difference is even smaller, with per-traversal overheads comparable to the page protection scheme at low update probabilities. Curiously, at low update probabilities the larger card (smaller card table) schemes show improved performance with increasing update probability, even though higher update probabilities imply more updates, and hence more work. We are unable to provide an explanation for this within the current experimental setup, but can only point to underlying hardware cache effects or an artefact of the Smalltalk interpreter as potential causes. Further study, involving cache profiling and instrumentation of the interpreter may yield a definitive explanation. Meanwhile, the similarity of Figures 6 and 14 illustrates the dominance of per-checkpoint costs for short transactions.

6.6 The effect of compilation

Although these results are for an interpreted implementation of Smalltalk, we see no reason why they will not carry over to a compiled setting. Since compilation can only speed the running-time component of execution, checkpoint overheads will become relatively more important. Moreover, compiler optimisations may merge or eliminate the noting of updates at certain store sites. For example, control-flow information may reveal that multiple updates to the same location at several points along a given execution path can be noted just once, rather than at every update. Such optimisations will have the effect of reducing the per-update overheads of the software-based schemes, so that checkpoint overheads become the dominant factor influencing the choice of scheme.

7 Conclusions

We have described several schemes for the efficient generation of recovery information in persistent programming languages, and performed a comprehensive performance evaluation of the alternatives, using recognised benchmarks. There are several conclusions we draw from the benchmark results. First, the ranking of the schemes is quite evident, with approaches that record updates at smaller granularities having a significant advantage when the transactions are short and the update locality poor, since they greatly reduce the overheads of unswizzling and generation of differences for the log. Best overall is the remembered set scheme, since it provides a very concise summary of just those objects that have been modified.

For longer intervals between checkpoints, the run-time costs of update detection come into play, with the page protection scheme having 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. At high update probabilities, the remembered set scheme loses its appeal due to the relatively expensive overhead to manage the remembered set. 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 those of checkpointing.

The length of the interval between checkpoints is an important factor because of this tension between the run-time and checkpointing overheads of the various schemes. Long intervals between checkpoints are likely to result in correspondingly more updates, increasing the checkpoint latency. Only when the volume of modified data is small with respect to the length of time between checkpoints 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 realisations which must use expensive calls to the operating system seem to be limited in their effectiveness. Moreover, the large granularity of 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.

In conclusion, we offer three guidelines for the generation of recovery information in persistent programming languages:

- Avoid large granules of update detection, to minimise checkpoint overheads.
- Choose a checkpoint frequency corresponding to the rate of generation of new update information, so that checkpoint delays are tolerable. Long-running transactions that perform few updates need infrequent checkpoints.
- Make use of page protection mechanisms only where update locality is good and checkpoints are infrequent.

8 Acknowledgements

We gratefully acknowledge Carla Brodley and Nick Haines for their feedback on drafts of this paper. We also thank the anonymous referees for their useful and detailed comments.

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

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

[6] J. N. Gray. Notes on database operating systems. In R. Bayer et al., editors, *Operating Systems: An Advanced Course*, Lecture Notes in Computer Science. Springer-Verlag, 1978.

[7] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15:287–318, Dec. 1983.

[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. Object fault handling for persistent programming languages: A performance evaluation. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, Washington, DC, Sept. 1993. To appear.

[10] A. L. Hosking, J. E. B. Moss, and D. Stefanović. A comparative performance evaluation of write barrier implementations. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 92–109, Vancouver, Canada, Oct. 1992. *ACM SIGPLAN Not. 27*, 10 (Oct. 1992).

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

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

[13] J. E. B. Moss. Working with persistent objects: To swizzle or not to swizzle. *IEEE Trans. Softw. Eng.*, 18(8):657–673, Aug. 1992.

[14] J. E. B. Moss, B. Leban, and P. K. Chrysanthis. Finer grained concurrency control for the database cache. In *Proceedings of the Third International Conference on Data Engineering*, pages 96–103, Los Angeles, CA, Feb. 1987. IEEE.

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

[16] D. Schuh, M. Carey, and D. DeWitt. Persistence in E revisited—implementation experiences. In A. Dearle, G. M. Shaw, and S. B. Zdonik, editors, *Proceedings of the Fourth International Workshop on Persistent Object Systems*, pages 345–359, Martha's Vineyard, Massachusetts, Sept. 1990. Published as *Implementing Persistent Object Bases: Principles and Practice*, Morgan Kaufmann, 1990.

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

[18] 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.

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

[20] 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).

[21] J. S. M. Verhofstad. Recovery techniques for database systems. *ACM Comput. Surv.*, 10(2):167–195, June 1978.

[22] 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*, Vancouver, Canada, Aug. 1992.

[23] 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.

[24] P. R. Wilson and T. G. Moher. Design of the Opportunistic Garbage Collector. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 23–35, New Orleans, Louisiana, Oct. 1989. *ACM SIGPLAN Not. 24*, 10 (Oct. 1989).