ACM SIGPLAN Notices, Volume 34, Number 10, October 1999

Mostly-copying reachability-based orthogonal persistence

Antony L. Hosking
hosking@cs.purdue.edu

Jiawan Chen
chenj@cs.purdue.edu

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907-1398
U.S.A.

Abstract

We describe how reachability-based orthogonal persistence can be supported even in uncooperative implementations of languages such as C++ and Modula-3, and without modification to the compiler. Our scheme extends Bartlett's mostly-copying garbage collector to manage both transient objects and resident persistent objects, and to compute the reachability closure necessary for stabilization of the persistent heap. It has been implemented in our prototype of reachability-based persistence for Modula-3, yielding performance competitive with that of comparable, but non-orthogonal, persistent variants of C++. Experimental results, using the OO7 object database benchmarks, reveal that the mostly-copying approach offers a straightforward path to efficient orthogonal persistence in these uncooperative environments. The results also characterize the performance of persistence implementations based on virtual memory protection primitives.

1 Introduction

Incorporating *orthogonal persistence* [11] into a programming language yields a flexible object model that encourages abstraction, modularity and reuse in the construction of libraries and applications that manipulate persistent data. In fact, one can write code with little thought given to the persistence or transience of the data it allocates and manipulates.

Despite the attractions of orthogonal persistence, systems-oriented programming languages have typically shunned it as too expensive, or too difficult to implement. The primary reason is the implied reliance on garbage collection to effect *persistence by reachability*. Yet garbage collection is now gaining in acceptance, even in the systems programming realm [59], and one can expect a similar trend for orthogonal persistence. The remaining issue then is difficulty of implementation. Unfortunately, the state of the art in production-quality optimizing compilers for systems programming languages does not include support for accurate location of roots for garbage collection and orthogonal persistence, despite noble attempts [33]. Thus, we must resort to techniques that treat roots conservatively. In this paper we describe and evaluate a new approach to orthogonal persistence for such uncooperative language environments, demonstrating simplicity of implementation and effectiveness of outcome.

2 Orthogonal persistence

Orthogonally persistent object systems [11] provide an abstraction of permanent data storage that hides the underlying storage hierarchy of the hardware platform (fast access volatile storage, slower access stable secondary storage, even slower access tertiary storage, etc.). This abstraction is achieved by binding a programming language to an object store, such that persistent objects are automatically cached in volatile memory for manipulation by applications and updates propagated back to stable storage in a fault-tolerant manner to guard against crashes. The resulting *persistent programming language* and object store together preserve *object identity*: every object has a unique persistent identifier (in essence an address, possibly abstract, in the store), objects can refer to other objects, forming graph structures, and they can be modified, with such modifications visible in future accesses using the same unique object identifier.

In defining *orthogonal* persistence Atkinson and Morrison [11] stipulate three design principles for persistent programming languages that enable the full power of the persistence abstraction:

1. *Persistence independence*: the language should allow the programmer to write code independently of the persistence (or potential persistence) of the data that code manipulates. From the programmer's perspective access to persistent objects is *transparent*, with no need to write explicit code to transfer objects between stable and volatile storage.

2. *Data type orthogonality*: persistence should be a property independent of type. Thus, an object's type should not dictate its longevity.

3. *Persistence designation*: the way in which persistent objects are identified should be orthogonal to all other elements of discourse in the language. Neither the method nor scope of its allocation, nor the type system (e.g., the class inheritance hierarchy), should affect an object's longevity.

The advantages that accrue through application of these principles to the design of persistent programming languages are many. Persistence independence allows programmers to focus on the important problem of writing correct code, regardless of the longevity of the data that code manipulates. Moreover, the code will function equally well for both transient and persistent data.

Data type orthogonality allows full use of data abstraction throughout an application, since a type can be applied in any programming context. This permits the development of programming systems based on rich libraries of useful abstract data types that can be applied to data of all lifetimes.

Finally, persistence designation gives every allocated instance of a type the right to the full range of persistence without requiring that its precise longevity be specified in advance. Again, this aids programming modularity since the producer of data need not be concerned with the ultimate degree of longevity to which a consumer might subject that data. In sum, orthogonal persistence promotes the programming virtues of modularity and abstraction; both are crucial to the construction of large persistent applications.

2.1 Practicalities

Complete persistence independence typically cannot be achieved, and even if it can, it may not be desirable, since one usually wants to offer a degree of control to the programmer. For example, in using a transaction mechanism one must generally specify at least the placement of transaction boundaries (begin/end). Nevertheless, a language design would not be transparent if it required different expression for the usual manipulation of persistent and non-persistent objects; i.e., for operations such as method invocation, field access, parameter passing, etc.

Similarly, perfect type orthogonality may not be achievable and may not even be desirable. For example, some data structures refer to strictly transient entities (e.g., open file channels or network sockets), whose saving to persistent storage is not even meaningful (they cannot generally be recovered after a crash or system shutdown). Whether thread stacks and code can persist is a trickier question. In many languages these objects are not entirely first class, and supporting persistence for them may also be challenging to implement. Thus, perfect type orthogonality, in the sense that any instance of any type can persist, is not so desirable as that any instance of any type *that needs to persist* can persist.

The principle of persistence designation means that any allocated *instance* of a type is potentially persistent, so that programmers are not required to indicate persistence at object allocation time. Languages in which the extent of an object can differ from its scope usually allocate objects on a heap, where they are retained as long as necessary. Deallocation of an object may be performed explicitly by the programmer, or automatically by the system when it detects that there are no outstanding references to the object. This can be determined by a *garbage collector* [81, 80, 51] which computes the transitive closure of all objects reachable (by following references) from some set of system roots. In systems that support garbage collection, persistence designation is most naturally determined by *reachability* from some set of known *persistent* roots.

Only when the heap is stabilized are new objects made persistent, and then only if they are reachable from other persistent objects or the persistent roots. Usually, this entails physically copying objects from non-persistent regions of the heap into persistent regions. However, copying an object requires exact knowledge of all the pointers to the object, so that they can be updated to reflect the object's relocation. Objects cannot be moved in environments where such accurate pointer information is unavailable. Thus, previous implementations of persistence for languages such as C++ break orthogonality, and require the programmer to distinguish transient and persistent objects whether by type or upon allocation. In this paper we show that reachability-based orthogonal persistence for such languages and environments is indeed possible, and efficient, using an approach based on mostly-copying garbage collection.

2.2 Performance

Orthogonal persistence exacerbates problems of performance by unifying the persistent and transient object address spaces such that *any* given reference may refer to either a persistent or transient object. Since every access (read or write) might be to a persistent object, they must all be protected by an appropriate *barrier*. Thus, the persistence *read barrier* ensures that an object is resident in memory, and faults it in if not, before any read operation can proceed. Similarly, the persistence *write barrier* supports efficient migration of updates back to stable storage, either when updated objects are replaced in volatile memory or during explicit stabilization of the persistent store, by maintaining a record of which objects in volatile memory are dirty. In general the read and write barriers can subsume additional functionality, such as negotiation of locks on shared objects for concurrency control.

The read and write barriers may be implemented in hardware or software. Hardware support for barriers, utilizing the memory management hardware of the CPU, is usually implemented via the virtual memory protection primitives of the underlying operating system [4, 58, 75, 82, 79], though the cost of fielding the resulting protection traps in some operating systems is notoriously expensive [45]. In the absence of hardware-based solutions, or because of the performance shortcomings, barriers can be implemented in software. Typically, the compiler (whether "way-ahead-of-time" or "just-in-time") or interpreter must arrange for appropriate checks to be performed before each operation that may access or update a persistent object. Alternatively, some languages (such as C++) support overloading of access operations to include the checks. These explicit software barriers can represent significant overhead to the execution of any persistent program, especially if written in an orthogonally persistent language where every access might be to a persistent object.

There are several approaches to mitigating these performance problems. *Pointer swizzling* [63] is a technique that allows accesses to resident persistent objects to proceed at volatile memory hardware speeds by arranging for references to resident persistent objects to be represented as direct virtual memory addresses, as opposed to the persistent identifier format used in stable storage. A read barrier may still be necessary to ensure that a given reference is in swizzled format before it can be directly used. Unnecessary software barriers can also be eliminated by taking advantage of language execution semantics and compile-time program analysis and optimization [67, 43, 42, 64, 38, 39, 48, 66, 65, 18, 17].

3 Related work

The notion of orthogonal persistence has a long history [7], traced through the development of prototype orthogonal persistent programming languages such as PS-Algol [5, 8, 6] and Napier88 [61, 28], and extensions to existing languages such as Smalltalk [55, 54, 77, 44, 40, 38] and Java [10, 52, 9, 53]. It is important to note that all of these prototypes rely on support for persistence from an underlying virtual machine, implemented as an interpreter of abstract machine instructions. While dynamic translation (i.e., "JIT" compilation) can improve performance in these systems, neither performance nor features for systems programming were an initial design goal. One advantage of an abstract execution model is that persistence of code and active execution states (i.e., threads) can be supported more easily. Napier88, Tycoon [60] and Smalltalk support both, while the PJama implementation of orthogonal persistence for Java supports persistent code but not threads (yet) [10, 52, 9, 53].

Performance-conscious persistent programming languages have historically been based almost exclusively upon C++, which at its outset was hostile to ideas of automatic storage management on the grounds that it compromised performance. Hence, C++-based persistence extensions have adopted models of persistence that violate orthogonality in one or more dimensions. In E [68, 69, 73, 70], Avalon/C++ [31] and SHORE/C++ [20] there is a distinction between database types and standard C++ types; only database types can persist. O++ [1, 2], Texas [75, 82] and Quickstore [79], along with prominent commercial offerings [58], adopt a different approach, requiring designation of persistence at allocation time. Indeed, the object database standard for C++ persistence defined by the Object Data Management Group (ODMG) is not orthogonal [3]. Until our own work [47, 41] we are unaware of any attempt to bring orthogonal persistence into the C++ domain. This is not to say that C++ itself will not succumb to orthogonal persistence. In fact, we are also exploring this possibility through extension of Texas with persistence by reachability, by marrying a garbage collector to Texas's portable run-time type descriptors [56] to obtain accurate information on the location of references stored in the heap.

Orthogonal persistence can be supported without redesign and reimplementation of the programming language if one is prepared instead to layer support for persistence into the operating system. Several experimental projects have taken this approach: support for persistence is targeted explicitly in Grasshopper [29, 71] and Mungi [34, 37], but the rudiments are there in other experimental operating systems such as Opal [24, 25], among others. Our interest here focuses on efficient support for orthogonal persistence on stock operating systems.

4 Mostly-copying garbage collection

Mostly-copying garbage collection [12, 13] is a hybrid of conservative [16] and copying [35, 26] collection. It is suitable for use in environments lacking accurate information on the location of references from the register, static or stack areas; objects that *appear* to be referenced from these areas are treated conservatively and not moved. Such references are called *ambiguous roots*, since they have a bit pattern that coincides with the range of valid heap references. In addition to the usual *tidy* language-level object references, which always refer to object headers, ambiguous roots also include *derived* references that arise out of pointer arithmetic introduced by compiler optimizations or explicitly by the programmer in languages that permit such expression.

Mostly-copying garbage collectors do require that all pointers stored in heap-allocated objects are tidy and can be found accurately; objects accessible only from other heap objects can thus be moved during garbage collection. Accurately finding the source locations of heap pointers requires information describing the layout of heap objects. The compiler may generate such information directly (as does Persistent Modula-3) or it may be derived indirectly from compiler-generated debugging information [75, 82]. The alternative is information supplied manually by the programmer [13], though this approach may be error-prone.

For mostly-copying collection the heap is divided into a number of fixed-size *pages*, which are usually some fixed multiple of virtual memory pages. Aligning the heap pages appropriately gives each page a unique page number formed from the common high-order bits of the virtual addresses covered by the page. This permits efficient mapping of heap references to per-page information. Objects larger than a single heap page are allocated as a sequence of consecutive heap pages.

Mostly-copying garbage collection divides the heap into two page spaces, *current* and *previous*.[1] New objects are always allocated in the current space. When the heap is "full" (e.g., some allocation threshold is reached) the roles of the page spaces are reversed and the collector proceeds to move all reachable objects from the previous space into the current space. The pages in each space are not necessarily contiguous and pages from each space may be interleaved. Instead, each page has an associated *space identifier* to keep track of its status. This arrangement allows an object to be moved by the collector from previous space into current space in one of two ways: either by physically copying it to a current-space page, or simply by resetting the space identifier of its page. The latter mechanism is called page *promotion*. Objects that appear to be referenced by ambiguous roots can thus be "moved" between spaces by promoting their page; retaining the same virtual address preserves the integrity of the ambiguous reference. Large objects are also "moved" via promotion, to reduce the copying overhead of the collector.

The mostly-copying collector, sketched in Figure 1, operates in three phases. We assume that the spaces are abstracted as sets of

[1] A note about terminology: We use the terms *current* space and *previous* space, instead of the more conventional terms *to* space and *from* space, to emphasize that objects can move spaces by promotion as well as by copying.

```
1  **proc** $promote(p) \equiv$
2    $previous := previous \setminus \{p\};$
3    $current := current \cup \{p\}.$
4
5  **proc** $closure(move) \equiv$
6    **while** $\neg copyStack.empty()$ **do**
7          $p := copyStack.pop();$
8          **foreach** $l \in pointer\_locations(p)$ **do**
9            $move(l)$
10         **end**
11   **end.**
12
13 **proc** $mover(l) \equiv$
14   $r := l\uparrow;$
15   **if** $page(r) \in previous$ **then**
16     **if** $r' = nil$ **then**
17       $r' := copy(r);$
18       $copyStack.push(page(r'))$
19     **end**;
20     $l\uparrow := r'$
21   **end.**
22
23 **proc** $gc() \equiv$
24   $previous := current;\ current := \{\};$
25   **foreach** $r \in AR$ **where** $page(r) \in previous$ **do**
26     $promote(page(r));$
27     $copyStack.push(page(r))$
28   **end**;
29   $closure(mover);$
30   **foreach** $p \in previous$ **do** $free(p)$ **end.**
```

Figure 1: Mostly-copying garbage collection

pages. The variables p, l and r range over heap pages, heap pointer locations and heap pointers (references), respectively. The heap pointer stored at a given heap pointer location l is denoted $l\uparrow$. AR denotes the set of ambiguous roots; for simplicity we assume these are the only roots, without loss of generality. The auxiliary procedure *promote* removes a page from one space and adds it to another. The procedure *closure* performs iterative Cheney-style [26] copying and scanning of the transitive closure of objects reachable from current-space. We assume several additional auxiliary procedures:

page(r)**:** returns the the heap page to which heap pointer r refers

pointer_locations(p)**:** returns an accurate set of all locations in page p that contain non-nil heap pointers

copy(r)**:** allocates a current-space copy of the object referred to by r; the address of the copy is termed r's *forwarding address*, and denoted by r'

The garbage collector (*gc*) begins by condemning all the current pages of the heap, *flipping* their state from current-space to previous-space (line 24). All previous-space pages will be reclaimed at the end of collection, unless promoted in the interim. Thus, the collector's job is to evacuate all reachable objects, copying them from the condemned previous-space pages into current-space. We assume a finite set of ambiguous roots from the registers, stack, and static areas. To preserve these ambiguous roots, the collector must first promote the pages to which they refer (lines 25-28). Note that promotion may retain unreferenced garbage objects that just happen to lie in those pages. As the pages are promoted they are placed in the copy stack for later processing.

The second phase of collection (line 29) copies the transitive closure of reachable objects into current-space. It proceeds by popping pages from the copy stack and scanning their pointer locations to discover any that refer to previous-space objects, copying each such object and leaving behind a forwarding address, and updating the pointer locations to refer to the current-space copies. The pages to which objects are copied are themselves placed in the stack for processing. This is an iterative process that completes only when the copy stack is empty (i.e., there are no more objects whose locations need to be scanned for references to uncopied objects). Termination is guaranteed because the closure of reachable objects is finite: each iteration removes a page from the stack for processing and pages are added to the stack only when objects are copied to them, so eventually the stack becomes empty. At the end of this second phase there are no pointers from current-space to previous-space, and all pages in previous-space can be freed (line 30).

Mostly-copying collectors have both generational [13] and incremental [32] variations. Indeed, our implementation of mostly-copying persistence by reachability for Modula-3 merges directly with the existing incremental/generational collector.

5 Mostly-copying persistence

We extend the implementation of the volatile heap to include pages containing persistent objects mapped into volatile memory from some external address space (such as a persistent object store or object database). To keep garbage collection and heap stabilization simple, we assume that *resident* persistent objects are swizzled so that they can be addressed in the same way, and have the same format, as non-persistent objects. Naturally, the application may propagate swizzled references from the heap into registers, the stack and static areas.

When garbage collecting a volatile heap containing mapped persistent objects one must retain all (as-yet) non-persistent data reachable from those objects, even if such data is not otherwise reachable. The original garbage collector, as presented in Figure 1 must be modified to maintain this invariant. We defer discussion of this issue until Section 5.2, where we also consider how the mostly-copying collector can remove from the volatile heap unmodified persistent objects that are no longer reachable from the program's transient state. For now, we focus on the basic *stabilization* mechanism needed for reachability-based orthogonal persistence.

5.1 Stabilization

Stabilization refers to the flushing of new and modified persistent objects back to disk. When a persistent program invokes the *stabilize* operation (perhaps mediated by a transaction commit if the language offers transactional concurrency control) all modified persistent objects must be flushed to disk. Since a modified persistent ob-

```
1  proc $stabilizer(l)$ $\equiv$
2    $r := l\uparrow$;
3    if $page(r) \in current$ then
4      if $\neg page(r).persistent$ then
5        $page(r).persistent := true$;
6        $copyStack.push(page(r))$
7      end
8    else
9        if $r' = nil$ then
10         $r' := persistent\_copy(r)$;
11         $copyStack.push(page(r'))$
12       end;
13       $l\uparrow := r'$
14   end.
15
16 proc $stabilize()$ $\equiv$
17   $previous := current;\ current := \{\}$;
18   foreach $p \in previous$ where $p.persistent$ do
19     $promote(p)$;
20     $copyStack.push(p)$
21   end;
22   foreach $r \in AR$ where $page(r) \in previous$ do
23     $promote(page(r))$
24   end;
25   $closure(stabilizer)$;
26   foreach $p \in current$ do
27     if $p.persistent$ then $flush(p)$ end;
28     $copyStack.push(p)$
29   end;
30   $closure(mover)$;
31   foreach $p \in previous$ do $free(p)$ end.
```

Figure 2: Mostly-copying stabilization

ject may refer to newly-allocated objects, these must also be made persistent and stabilized. Forming the corresponding reachability closure is analogous to garbage collection, so we have modified the mostly-copying garbage collection algorithm to perform the necessary steps to stabilize the persistent heap. Again, this allows orthogonal persistence by reachability even for language environments in which there is no accurate way to recognize heap references from the registers, stacks and static areas. The only requirement is for accuracy in locating pointers stored in the heap.

We sketch the mostly-copying *stabilize* procedure in Figure 2 and illustrate each phase of its operation with an example in Figure 3. Figure 3(a) shows the initial heap configuration for the example. Stabilization begins (Figure 2, lines 18-21) by promoting all persistent pages (2 & 5 in the example) from previous-space to current-space (Figure 3(b)). The objects in these pages form the initial root set for persistence.

Like garbage collection, stabilization must be conservative with respect to ambiguous roots, so ambiguously-referenced pages are also promoted (line 23). In the example, this results in the promotion of page 1 (referenced by ambiguous root a1) and page 5 (by a2), as depicted in Figure 3(c).

The next phase (line 25) stabilizes the closure of objects reachable from persistent pages using the *stabilizer* routine. Two cases occur for each non-persistent object encountered in the closure: the object lies either in previous-space or in an ambiguously-referenced non-persistent page in current-space. If the former, then the object is simply copied to a current-space persistent page, as for objects F, H and J in Figure 3(d). If the latter, then the only way to make the object persist is to stabilize its containing page, since the object cannot be copied without violating conservative guarantees for ambiguous roots, as for page 1 in Figure 3(d). The result may be to stabilize objects on the page that then persist needlessly, such as object A in the example. The remaining previous-space objects (e.g., L, M and K) are not reachable from persistent storage so definitely need not persist.

The last task is to flush all the persistent pages (with whatever shadow paging or logging is necessary for rollback and recovery) and to complete garbage collection of the remaining transient objects, treating existing current-space pages as roots (lines 26-30 and Figure3(e)). Upon completion, the collector can safely reclaim the previous-space pages (Figure 3(f)). Note that persistent page 5 remains although it is no longer reachable from the transient state of the application. In the next section we modify the mostly-copying garbage collector to reclaim such transiently unreachable persistent pages.

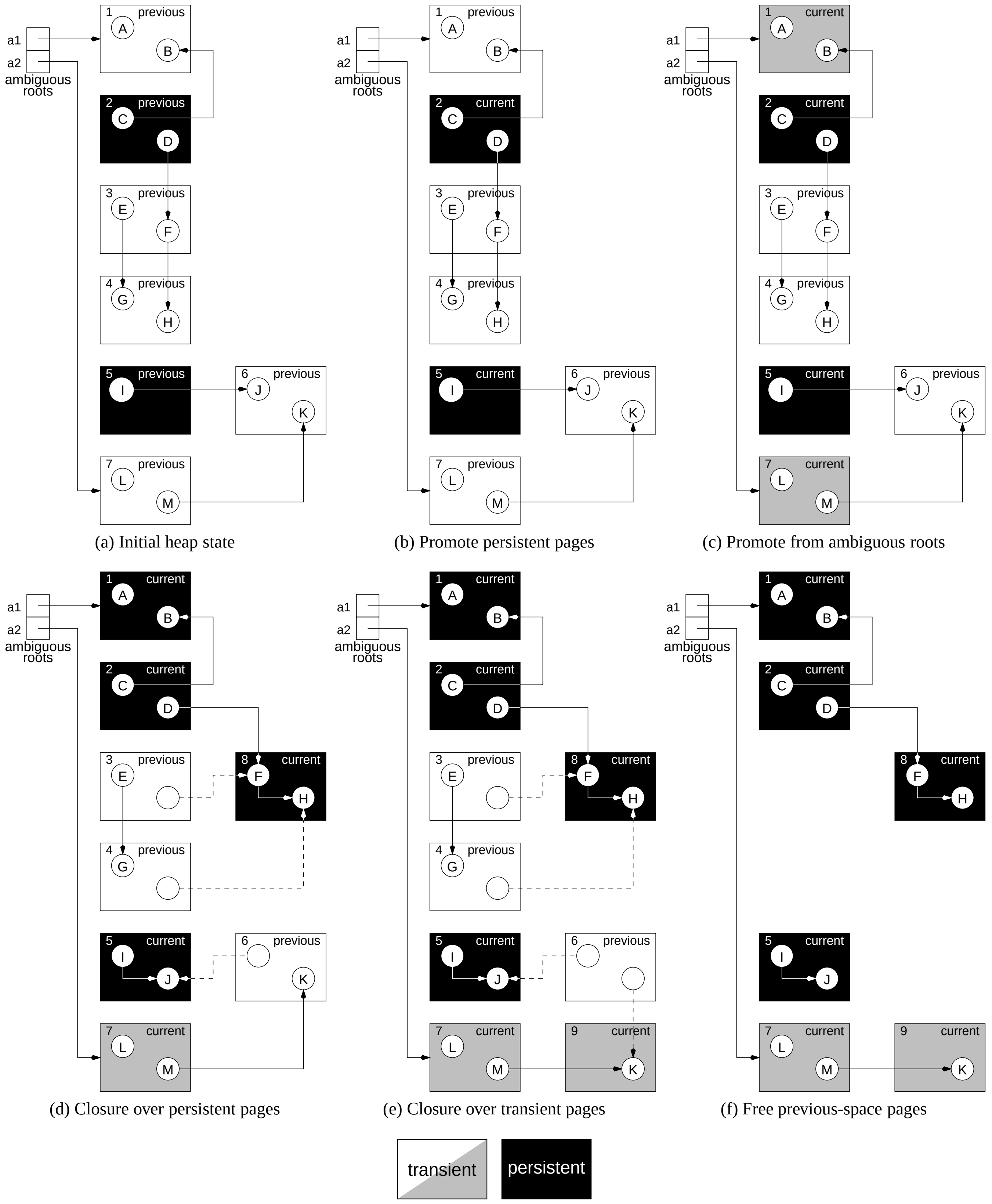
Note that we make no assumptions about the underlying persistent store, whether page- or object-based. Mostly-copying persistence is entirely compatible with both page-server and object-server approaches, despite its own page-based assumptions about the memory heap. Correctness and termination of mostly-copying stabilization can be inferred from the invariants stated here, similarly to mostly-copying garbage collection.

5.2 Revisions to the garbage collector

We say that a persistent page is *transiently unreachable* if it cannot be reached via pointers from transient store, though it may of course have undiscovered references from the persistent store. We are not obliged to retain such a page in volatile memory, since the application cannot immediately access it without first discovering references to it by faulting other persistent objects that refer to it. Thus, we modify the mostly-copying garbage collection algorithm to reclaim such pages as in Figure 4. Note that only unmodified persistent pages are reclaimed, to avoid triggering stabilization during garbage collection. Notice also that we always avoid the unnecessary overhead of physically copying already-persistent objects during collection or stabilization by simply promoting their page.

5.3 Storage architecture flexibility

As already mentioned, we make no assumptions about the underlying persistent storage architecture, be it page- or object-based. Similarly, we make no assumptions as to the underlying swizzling mechanisms, save to assume that directly-swizzled pointers to resident persistent objects can occur and may be stored into the registers, stacks, static areas, and heap memory. Mostly-copying reachability-based persistence is entirely flexible with respect to the implementation of these mechanisms. For example, our current implementation of Persistent Modula-3 uses mostly-copying



(a) Initial heap state

(b) Promote persistent pages

(c) Promote from ambiguous roots

(d) Closure over persistent pages

(e) Closure over transient pages

(f) Free previous-space pages

Figure 3: Mostly-copying stabilization

```
proc $mover'(l)$ $\equiv$
  $r := l\uparrow$;
  if $page(r) \in previous$ then
    if $page(r).persistent$ then
      $promote(page(r))$;
      $copyStack.push(page(r))$
    else
      if $r' = nil$ then
        $r' := copy(r)$;
        $copyStack.push(page(r'))$
      end;
      $l\uparrow := r'$
    end;
  end.

proc $gc'()$ $\equiv$
  $previous := current$; $current := \{\}$;
  foreach $p \in previous$ do
    if $p.persistent \wedge p.modified$ then
      $promote(p)$;
      $copyStack.push(p)$
    end
  end;
  foreach $r \in AR$ do
    $promote(page(r))$;
    $copyStack.push(p)$
  end;
  $closure(mover')$;
  foreach $p \in previous$ do $free(p)$; end.
```

Figure 4: Revised garbage collection

persistence with Texas-style "pointer-swizzling at page-fault time" [75, 82] as the underlying swizzling and faulting mechanism, above the SHORE object store [20]. We have designed Persistent Modula-3 for a move to compiler-inserted barriers in later versions, without needing to change the mostly-copying heap management mechanism. Just as easily, one might re-engineer the Texas persistent store to provide mostly-copying persistence by reachability for C++, based on the heap layout information for swizzling that Texas extracts from debugging information provided by the GNU C++ compiler [56].

6 PM3: Persistent Modula-3

To serve as a platform for research into compiler support for orthogonally-persistent programming languages we have designed and implemented PM3: a persistent extension of the Modula-3 [19] programming language.[2] Modula-3 is a systems programming language that supports threads, objects with single inheritance, and strong notions of type safety. Modula-3 is *strongly-typed*: every expression has a unique type, and assignability and type compatibility are defined in terms of a single syntactically specified subtype relation, written $<:$. There are specific subtype rules for ordinal types (integers, enumerations, and subranges), references and arrays. Modula3's support for garbage collection recognizes the high degree of safety afforded by automatic storage reclamation, which is achievable even in open runtime environments that allow interaction with non-Modula-3 code.

A *traced* reference type `REF` T refers to heap-allocated storage (of type T) that is automatically reclaimed by the garbage collector whenever there are no longer any references to it.[3] The type `REFANY` contains all references. The type `NULL` contains only the reference value `NIL`. Object types are also reference types. An *object* is either `NIL` or a reference to a data record paired with a set of procedures (*methods*) that will each accept the object as a first argument. Every object type has a supertype, *inherits* the supertype's representation and implementation, and optionally may extend them by providing additional fields and methods, or overriding the methods it inherits with different (but type-correct) implementations. This scheme is designed so that it is (physically) reasonable to interpret an object as an instance of one of its supertypes. That is, a subtype is guaranteed to have all the fields and methods defined by its supertype, but possibly more, and it may override its supertype's method implementations with its own.

6.1 Design

Persistence in PM3 is achieved by allowing traced references to refer not only to transient data, but also to persistent data. Allocated storage persists by virtue of its reachability by following traced references from the roots of named PM3 databases. The PM3 implementation is responsible for automatic caching of persistent data in memory, and for automatic mediation of accesses to cached data to enforce concurrency control.

Persistence functionality is introduced by way of the new library interfaces *Transaction* and *Database*; their essentials are presented in Figures 5 and 6. They are similar to their namesakes from the Object Data Management Group (ODMG) standard [23], with databases and transactions abstracted as Modula-3 objects. Each named database has a distinguished root, from which other persistent data can be reached. Databases can be shared by multiple users and operating system processes, with locking and concurrency control enforcing serializability of transactions. Unlike the ODMG transaction model, we do not necessarily enforce isolation between threads executing in the same virtual address space, though we do require that a thread execute in at most one transaction at any time, and that it enter a transaction before attempting to interact with a database. The design permits transactions to nest, though our current implementation does not. We are also exploring extended semantics for combining transactions and threads in PM3, along the lines of the Venari transaction model for ML [36].

[2]Source code for Persistent Modula-3 is available via anonymous FTP from ftp://ftp.cs.purdue.edu/pub/hosking/pm3. Further information appears on the World Wide Web at http://www.cs.purdue.edu/homes/hosking/pm3.html.

[3]Modula-3 also supports *untraced* references to storage allocated in a separate uncollected heap; untraced storage must be deallocated explicitly.

```
INTERFACE Transaction;
EXCEPTION
  TransactionInProgress;
  TransactionNotInProgress;
TYPE
  T <: Public;
  Public = OBJECT METHODS
    begin()
      RAISES { TransactionInProgress };
      (* Starts (opens) a transaction.
         Raises TransactionInProgress if nested
         nested transactions are not supported. *)
    commit()
      RAISES { TransactionNotInProgress };
      (* Commits and closes a transaction *)
    chain()
      RAISES { TransactionNotInProgress };
      (* Commits and reopens transaction;
         retains locks if possible *)
    abort()
      RAISES { TransactionNotInProgress };
      (* Aborts and closes a transaction *)
    checkpoint()
      RAISES { TransactionNotInProgress };
      (* Checkpoints updates, retains locks
         and leaves transaction open *)
    isOpen(): BOOLEAN;
      (* Returns true if this transaction
         is open, otherwise false *)
  END;
END Transaction.
```

Figure 5: The Transaction interface

```
INTERFACE Database;
FROM Transaction IMPORT
  TransactionInProgress,
  TransactionNotInProgress;
EXCEPTION
  DatabaseExists;
  DatabaseNotFound;
  DatabaseOpen;
PROCEDURE Create(name: TEXT)
  RAISES { DatabaseExists,
           TransactionInProgress };
PROCEDURE Open(name: TEXT): T
  RAISES { DatabaseNotFound, DatabaseOpen,
           TransactionInProgress };
TYPE
  T <: Public;
  Public = OBJECT METHODS
    getRoot(): REFANY
      RAISES { TransactionNotInProgress };
    setRoot(object: REFANY)
      RAISES { TransactionNotInProgress };
  END;
END Database.
```

Figure 6: The Database interface

6.2 Implementation

The current PM3 implementation is based on the Digital (now Compaq) Systems Research Center's version 3.6 Modula-3 compiler, runtime system and libraries (all written in Modula-3). The compiler is structured as a loosely-coupled front-end to the GNU C compiler, which generates efficient optimized native code. It also produces compact, executable type descriptors for heap-allocated data, in support of both garbage collection and persistence. The Persistent Modula-3 compiler is essentially unchanged from the original; it generates code that is *exactly* the same as that generated by the non-persistent Modula-3 compiler. Instead of explicit compiler-generated read and write barriers, our current implementation relies on the operating system's virtual memory primitives, triggering fault handling routines in the PM3 runtime system to retrieve objects, note updates, and obtain locks.

The PM3 runtime system manages the volatile heap, supporting allocation of space for new and cached persistent data, and garbage collection to free unreachable space. We have extended the existing incremental, generational, mostly-copying garbage collector to manage both transient objects and resident persistent objects, and to compute the reachability closure for mostly-copying stabilization. Heap objects, whether persistent or transient, have the same size and layout as the original non-persistent Modula-3 implementation. Heap pages, currently sized and aligned at 8 Kbytes, are the unit of transfer between volatile memory and stable storage and the unit of locking for concurrency control. We plan also to investigate object-level transfer and locking along the lines of Carey et al [22].

6.2.1 Pointer swizzling

Each database is treated as a distinct virtual address space: an array of pages bounded by the address range of the hardware platform. Each database has a distinguished root object, at a known address in its address space. The runtime system simply maps pages from any number of open databases into the volatile heap as references to the (persistent) objects on those pages are *discovered*. Requesting the root object of a database is one way to discover a reference; another way is to fault in a page containing references to other persistent pages. Naturally, when a reference is discovered it must be swizzled to point to a mapped (though not necessarily resident) page in the volatile heap; mappings are created on demand as references are swizzled. All mapped but non-resident pages are protected from access using the virtual memory protection primitives. Thus, any access to a protected page will trap and trigger a fault: the fault handler unprotects the page, reads its data from the persistent store, and locates and swizzles all of its pointers. The access can then resume. As execution proceeds, volatile heap page frames are reserved in a "wave-front" just ahead of the most recently faulted and swizzled pages, guaranteeing that the application will only ever see object references as virtual memory addresses [75, 82].

We also track updates to persistent data by protecting heap pages from writes; on the first write to the page we set a dirty bit for it, unprotect the page and resume the write.

Note that at any point in time an application can address only as much persistent data as can be mapped into its virtual address space. Data from multiple databases can be mapped at the same time.

However, there is no restriction on the total volume of unmapped persistent data.

6.2.2 *Persistent storage*

The current PM3 implementation uses the University of Wisconsin's SHORE object repository [20] as a simple transactional page server. Each page is described in the SHORE data language (SDL) as a single SHORE text object, with simple read and write access implemented via the SHORE/C++ binding. Concurrency control and recovery support are inherited directly from SHORE, with the PM3 runtime system acquiring read locks on pages as they are faulted and write locks on first update. We also support interaction with a version of the GRAS3 [57, 14] transactional page server that permits nested transactions, and which is implemented purely in Modula-3.

6.2.3 *Types and metadata*

To ensure type safety each persistent object must also store some representation of its type. The type is used to locate pointers within the object when it is swizzled, and for run-time type checking. Rather than store a full type descriptor, we take advantage of Modula-3's implementation of structural type equivalence, which computes a characteristic 64-bit fingerprint for every type that can be mapped to its descriptor at run time. Every database contains an index containing the fingerprints of all the objects in the database; objects are stored with a reference to their fingerprint to encode their type. The type descriptors and the code for methods are compiled into the static area of each application program and thus need not be stored. Instead, objects are reunited with their methods as their contents are swizzled. The advantage of this is that we can continue to use traditional file-based program development tools such such compilers, assemblers, linkers and loaders. In the future, persistence-aware development tools that operate on code stored in the database will allow a tighter integration of code with data.

The type index is one example of metadata stored in every database. All metadata in PM3 is implemented as Modula-3 data structures, and stored transparently using the existing mechanisms for orthogonal persistence. This sleight of hand derives from our stabilization algorithm, which permits metadata to be treated just like other orthogonally persistent data. We believe PM3 to be unique among persistent programming languages in that it is implemented entirely in Modula-3, with explicit I/O only to read/write persistent pages from/to the page server.

7 Experiments

Our experiments use the OO7 benchmark [21]. We measure statistics for newly-stabilized persistent objects, and compare the performance of the traversal and insertion portions of our PM3 implementation of OO7 against the SHORE/C++ implementation of OO7 distributed with SHORE. The stabilization statistics yield an accurate picture of the effectiveness of mostly-copying stabilization. We report results for OO7 database generation, the OO7 insertion operation which allocates and inserts new objects into the database, and for sparse and dense OO7 traversals, both read-only and with updates (T1, T2 and T6). Additional results for indexed updates (T3) and updates to the manual (T9) are reported in our earlier paper [41].

Modules	1
Assembly levels	7
Subassemblies per complex assembly	3
Composite parts per base assembly	3
Composite parts per module	500
Atomic parts per composite part	20
Connections per atomic part	3
Document size (bytes)	2 000
Manual size (bytes)	100 000
Total composite parts	500
Total atomic parts	10 000

Table 1: Small OO7 database configuration

7.1 The OO7 benchmark

The OO7 benchmarks [21] are an accepted test of object-oriented database performance. They operate on a synthetic design database, consisting of a keyed set of *composite parts*. Associated with each composite part is a *document* object consisting of a small amount of text. Each composite part consists of a graph of *atomic parts* with one of the atomic parts designated as the *root* of the graph. Each atomic part has a set of attributes, and is connected via a bidirectional association to several other atomic parts. The connections are implemented by interposing a separate connection object between each pair of connected atomic parts. Composite parts are arranged in an *assembly* hierarchy; each assembly is either made up of composite parts (a *base* assembly) or other assemblies (a *complex* assembly). Each assembly hierarchy is called a *module*, and has an associated *manual* object consisting of a large amount of text. Our results are all obtained with the *small* OO7 database, configured as in Table 1.

7.2 Hardware

The experiments were run under Solaris 2.5.1 on a 170MHz Sun SPARCstation 5, with 64 Mbytes RAM. The processor implementation is the Fujitsu TurboSPARC, with direct-mapped instruction and data caches of 16 Kbytes each. Both caches are virtually-addressed, guaranteeing consistent performance regardless of the virtual-to-physical page mapping. This means that elapsed time measurements obtained on this platform are not subject to jitter due to variations in page mappings from one process incarnation to the next. The local disk is a SUN0535 SCSI disk of 535 Mbytes.

Since we were not interested in measuring network latencies both the SHORE server and the client were run on the same machine. This results in improved client-server communication, through shared memory where possible, and also more fully exposes the underlying overheads of the salient persistence mechanisms of interest to us.

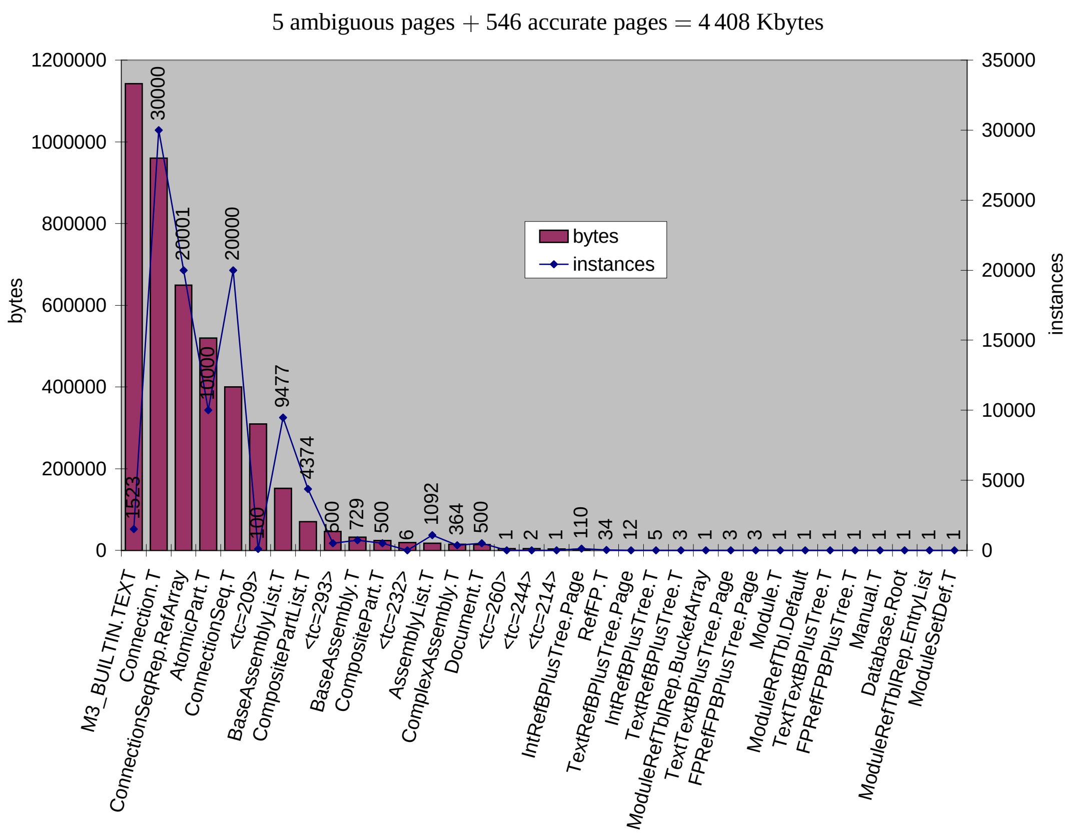


Figure 7: Stabilization statistics for database generation

7.3 Software

We use release 1.1.1 of SHORE as the underlying object store for PM3. SHORE objects are lighter-weight than a Unix file, but still more heavyweight than the typical fine-grained data structures coded in ordinary programming languages. Each persistent PM3 heap page is stored as a single SHORE object.

Our PM3 implementation of OO7 is a direct transliteration of the SHORE/C++ implementation distributed with SHORE, but with the OO7 schema specified using Modula-3 types, and the queries implemented directly in Modula-3. We took great care to match the reference SHORE/C++ (excluding known bugs), so as to ensure faithful reproduction of the benchmark, and directly comparable results. Where the benchmark requires the use of an index, we used a transparently persistent B+-tree coded in Modula-3. The PM3 compiler is based on the same GNU compiler version 2.7.2 used to compile SHORE/C++ programs. Thus, we can directly compare the performance of PM3 with the SHORE/C++ binding. Both versions of OO7 were compiled with optimization turned on (i.e., gcc -O2). The PM3 compiler was also invoked with a flag that disables runtime checks on indexing arrays out of bounds and to catch certain type errors, so as to give a fairer comparison with C++.

7.4 Results

The timing results were obtained from runs on the small OO7 benchmark database, which is small enough to fit entirely in main memory, including copies being cached in both the server and the client. We report the elapsed time in seconds broken down into three components: user and system CPU time in the client, plus other remaining elapsed time which we charge to interactions with the server for data transfer, concurrency control, etc. (identified in the figures as user, system and server, respectively).

As in the original specification of OO7 [21] we obtain results for the traversal operations running both cold and hot, by repeating each operation five times per run. We also ran the five iterations of each operation in two transaction modes: as a single transaction committing only after the last iteration (one), and as a sequence of chained transactions (many).

The cold first iteration, begins with no data cached anywhere in the client or the server, nor in the operating system's file system buffers (this is achieved by reading from a very large file in such a way as to flush the buffers of any useful data). The cold iteration is then followed immediately by four iterations of the exact same query, with the results from the middle three of the five iterations taken as the hot measure. The result for the last (fifth) iteration is discarded so that the overhead of commit processing is not included in the single-transaction hot times. In contrast to the original OO7 specification, we report the *sum* of the results for the three hot iterations instead of an average, so as to avoid obscuring any variations

Server interactions		SHORE/C++	PM3
Fetches	objects	438	1
	pages		35
	bytes	46 816	288 472
Updates		10 objects	19 pages
Creations		820 objects	11 pages
Index	insertions	430	22
	lookups	10	84

Table 2: Insert

in behavior from one hot iteration to the next that may arise from periodic invocations of the PM3 garbage collector.

7.4.1 *Database generation*

We generate the *small* database according to the configuration of Table 1. The stabilization statistics are of most interest since they characterize the effectiveness of mostly-copying stabilization over large numbers of newly-allocated objects, as illustrated in Figure 7. The figure displays the number of new instances of each type made persistent for database generation, ranked by volume in bytes. Types are identified by name, except for anonymous types which are tagged by their unique typecode, written <tc=*n*> for typecode *n*.

Note first that the bulk of the small OO7 database consists of Modula-3 texts (M3_BUILTIN.TEXT) for the manual and document objects, the connections (Connection.T) that capture the bidirectionality of the links between parts, sequences of references (ConnectionSeq.T and ConnectionSeqRep.RefArray) for the incoming and outgoing references of each atomic part, and the atomic parts themselves (AtomicPart.T). Most of the volume is accounted for by objects that are definitely part of the OO7 database; there is little or no data that is obviously inherently transient and persists only by falling on an ambiguously-referenced persistent page. The most likely source of such unnecessary overhead is in the text objects, yet the 500 documents (1 000 000 bytes) and one manual (100 000 bytes) account for most of that volume. Of the small remaining texts, many are allocated as index keys, so there is only a relatively small volume of ambiguously-persistent texts, if any. In sum, the conservatism of mostly-copying persistence appears to result in little additional burden, if any, in terms of the volume of data made persistent.

7.4.2 *Insert: Structural modification*

Create ten new composite parts, which includes creating 200 new atomic parts and ten new document objects, and insert them into the database by installing references to these composite parts into 10 randomly chosen base assembly objects.

The Insert operation measures the speed of structural modifications to the OO7 database, requiring allocation and installation of many new objects. Figure 8 records PM3 stabilization statistics for the Insert operation. Again, texts form the biggest fraction of new persistent objects, with the document objects accounting for 20 000 of the 21 088 text bytes stored, so the overhead for ambiguously-persistent texts is minimal. These results confirm the effectiveness of mostly-copying persistence.

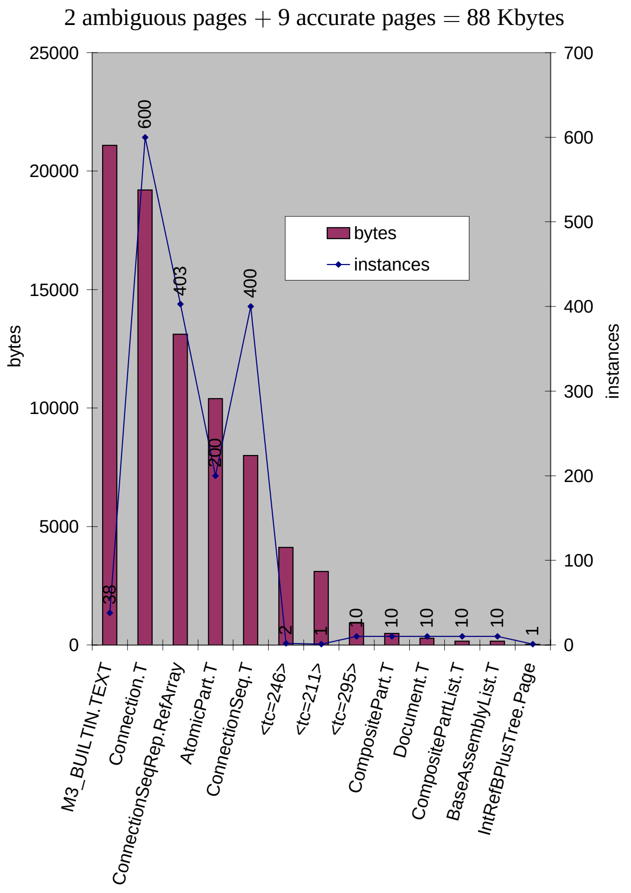
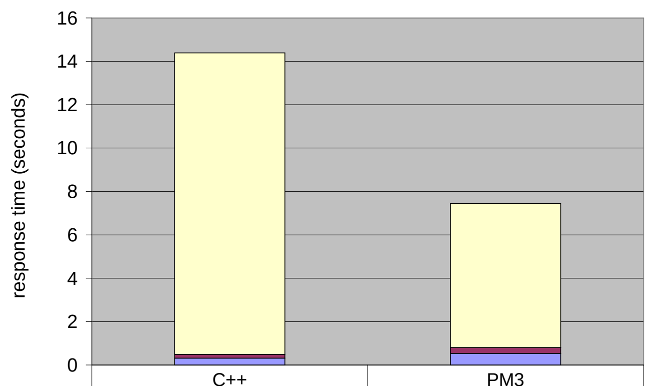


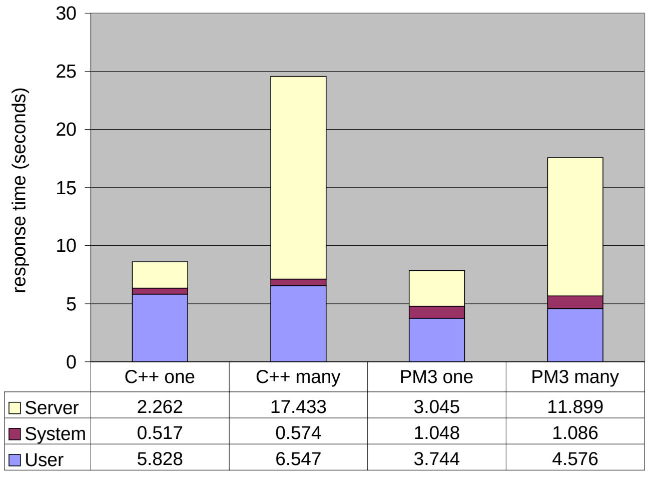
Figure 8: Stabilization statistics for Insert



	C++	PM3
□ Server	13.901	6.648
■ System	0.174	0.267
■ User	0.313	0.536

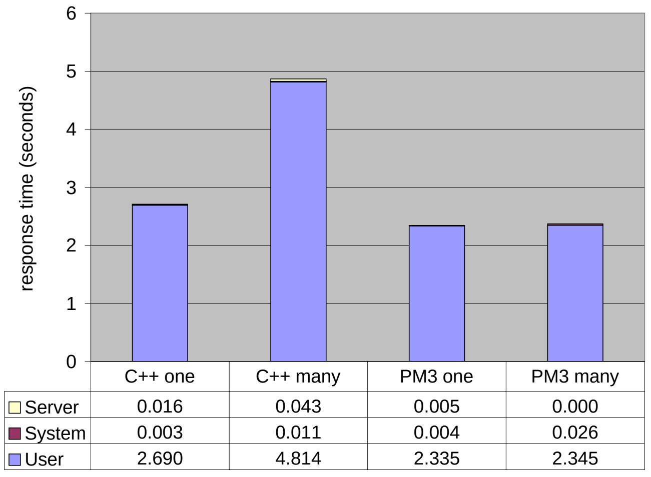
Figure 9: Insert

Figure 9 and Table 2 graph and tabulate the performance of SHORE/C++ against PM3 on a cold Insert with commit, showing elapsed times and statistics for the transaction. The biggest reason for PM3's better performance than SHORE/C++ is that the SHORE/C++ implementation of OO7 uses the builtin SHORE index support. These indexes are centralized on the server, so every index operation (10 lookups + 430 insertions) requires communication with the server at very high cost. In contrast, PM3 uses SHORE indexes only for mapping database pages keyed by their page number (84 lookups + 22 insertions), while the OO7 indexes are implemented natively as orthogonally persistent B+-trees so their pages can be cached and updated at the client.



	C++ one	C++ many	PM3 one	PM3 many
🟨 Server	2.262	17.433	3.045	11.899
🟪 System	0.517	0.574	1.048	1.086
🟦 User	5.828	6.547	3.744	4.576

(a) Cold



	C++ one	C++ many	PM3 one	PM3 many
🟨 Server	0.016	0.043	0.005	0.000
🟪 System	0.003	0.011	0.004	0.026
🟦 User	2.690	4.814	2.335	2.345

(b) Hot: 3 iterations

Figure 10: Traversal T1

7.4.3 Traversals

The OO7 traversal operations include both sparse and dense traversals of the assemblies that comprise the benchmark database, as well as sparse and dense updates of the atomic parts traversed.

7.4.3.1 Traversal T1: Raw traversal speed

Traverse the assembly hierarchy. As each base assembly is visited, visit each of its referenced unshared composite parts. As each composite part is visited, perform a depth-first search on its graph of atomic parts. Return a count of the number of atomic parts visited when done.

T1 is a test of raw pointer traversal speed. Figure 10(a) shows the cold T1 results. PM3 outperforms SHORE/C++ in both the traversal without commit (one) and the traversal with commit (many), despite the overhead for PM3 of the virtual memory page protection traps, as measured by the system CPU time, and the cost of swizzling as part of the user CPU time. PM3 fields 379 protection traps to fetch 379 pages, or approximately 3 Mbytes. SHORE/C++ fetches 41 594 objects, totaling 2.9 Mbytes.

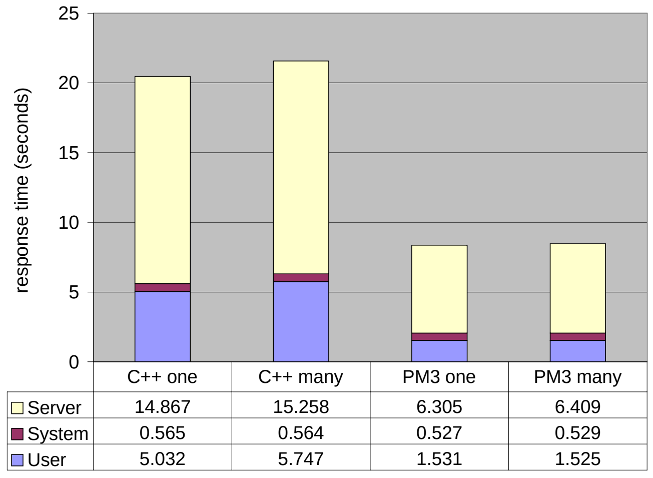
Despite the read-only nature of T1 there is noticeable overhead for commits, as revealed in the results that include commit processing (many). The server overhead is higher for SHORE/C++ than PM3 since the cold chained commit requires a separate client-server communication request for each object in the client-side cache (41 594 versus 379); hot chained commits do not pay this overhead. This is explained as follows: on first chained commit the SHORE client must communicate the state (clean or dirty) of any pages it is caching across the chained commit into the next transaction; subsequent chained commits need only update the server with any differences in status from the previous commit (in this case none).

The hot T1 results appear in Figure 10(b), showing the benefits of client caching, even across the chained commits (many). The hot commits impose negligible server-side overhead since the client determines that no updates have occurred and restricts communication with the server only to signal the commit. Again, PM3 outperforms SHORE/C++, as a result of its full swizzling of object references. There is significant client-side commit overhead for SHORE/C++, almost doubling the elapsed time. Again, the client must check each cached object to see if its status has changed since the previous commit; there are simply more objects cached for SHORE/C++ than heap pages for PM3.

7.4.3.2 Traversal T6: Sparse traversal speed

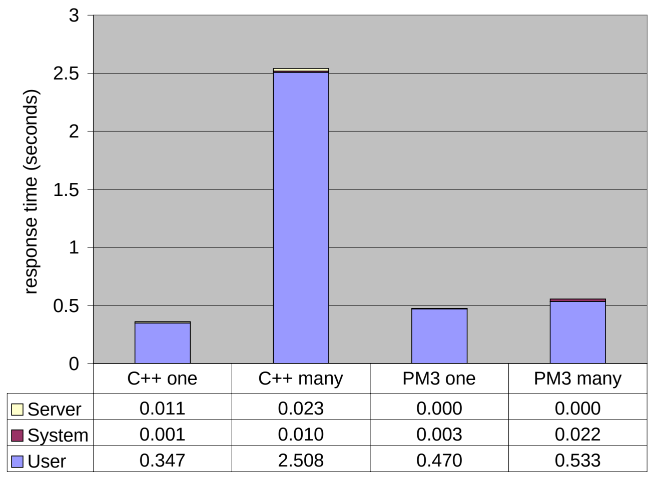
Traverse the assembly hierarchy. As each base assembly is visited, visit each of its referenced unshared composite parts. As each composite part is visited, visit the root atomic part. Return a count of the number of atomic parts visited when done.

The designers of OO7 intended the T6 traversal to provide insight into the costs and benefits of a full swizzling approach, since it is sparse and follows only a small fraction of swizzled references; full swizzling ought to be penalized for expending swizzling effort for little or no benefit. However, our elapsed time results do not tell the expected story. For the cold T6 traversal (Figure 11(a)) PM3 appears to pay moderately for the overhead of swizzling at a factor



	C++ one	C++ many	PM3 one	PM3 many
☐ Server	14.867	15.258	6.305	6.409
■ System	0.565	0.564	0.527	0.529
■ User	5.032	5.747	1.531	1.525

(a) Cold



	C++ one	C++ many	PM3 one	PM3 many
☐ Server	0.011	0.023	0.000	0.000
■ System	0.001	0.010	0.003	0.022
■ User	0.347	2.508	0.470	0.533

(b) Hot: 3 iterations

Figure 11: Traversal T6

of 3 in the user component for the cold T6 traversal over the hot traversal (Figure 11(b)).

Accounting for the difference in performance between SHORE/-C++ and PM3 turns out to be related to clustering. SHORE/C++ fetches 41 346 objects (2.9 Mbytes) versus PM3's 144 heap pages (1.1 Mbytes). That SHORE/C++ fetches almost as many objects and as much data for this sparse traversal as for the dense T1 traversal suggests extremely poor clustering. PM3 does much better because it uses reachability-based persistence, placing objects in persistent pages via Cheney-scanning [26] such that related objects are clustered together [72, 76]. Only an orthogonally persistent system has sufficient flexibility to place objects by reachability, since placement is decoupled from allocation and deferred until commit time.

The hot results with commit (many) shown in Figure 11(b) again reveal the superiority of full swizzling for hot operations, with PM3 markedly outperforming SHORE/C++ for hot commits. In this case, SHORE/C++ suffers from the overhead of having to issue 5 468 paired pin/unpin operations for each access to an object in the cache; PM3 accesses incur no such overhead.

7.4.3.3 Traversal T2: Updates

Repeat traversal T1, but update objects during the traversal. There are three types of update patterns in this traversal. In each, a single update to an atomic part consists of swapping its (x,y) attributes. The three types of updates are:

A *Update one atomic part per composite part.*
B *Update every atomic part as it is encountered.*
C *Update each atomic part in a composite part four times.*

When done, return the number of update operations that were actually performed.

All three of the T2 update traversals fetch and traverse exactly the same set of parts; they differ only in the density (T2A versus T2B/T2C) and frequency of update (T2B versus T2C). The comparable statistics for SHORE/C++ and PM3 T2 traversals appear in Table 3. The cold traversals without commit presented in Figure 12(a)(one) capture both the cost to fetch the data and to operate on it, but not the cost of committing the changes back to the server; the cold traversals with commit (Figure 12(a)(many)) includes all of these costs; the hot traversals without commit (Figure 12(b)(one)) isolate the pure CPU cost to perform the updates at the client without write barrier or commit overheads; while the hot traversals with commit (Figure 12(b)(many)) ignore the fetch costs, but record the CPU, write barrier and commit overheads.

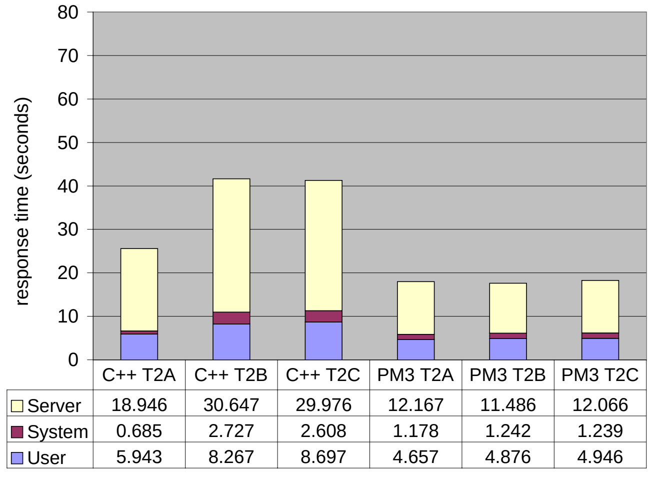
Hot access to cached objects is cheaper for PM3 than for SHORE/-C++, as revealed in Figure 12(b)(one), emphasizing the benefits of PM3's full swizzling approach over the overloaded SHORE/C++ access operations.

Once again, commit time as measured in Figure 12(b)(many) is proportional to the number of updated SHORE objects (i.e., atomic part objects for SHORE/C++ and pages for PM3). Thus PM3 wins due to the aggregation of many updated parts (9 820) into many fewer updated pages (171) for the dense updates T2B and T2C. For the sparse hot T2A update traversal with commit the difference in number of updated objects/pages is less significant and other overheads dominate. In this case (many) PM3 is penalized for using a SHORE index to map pages; SHORE checks each updated page to see if its integer index key has changed, which would require updating the index. These unnecessary checks are reflected in the server portion of elapsed time; they can be easily avoided, in which case we would expect PM3 to perform at least as well as SHORE/C++.

The hot T2 traversals with commit (Figure 12(b)(many)) also reveal the raw overhead to update the objects, including the cost of the write barrier. PM3's trap-based write barrier poses significant overhead (system) for the sparse update T2A traversal with commit. For the denser T2B traversals the overhead to PM3 of each trap is amortized over more updates, for significantly faster response than for SHORE/C++. With T2C the trap cost is also amortized over repeated updates to the same page. In contrast, SHORE/C++ incurs barrier overhead on every update, even if to a part that has already been updated.

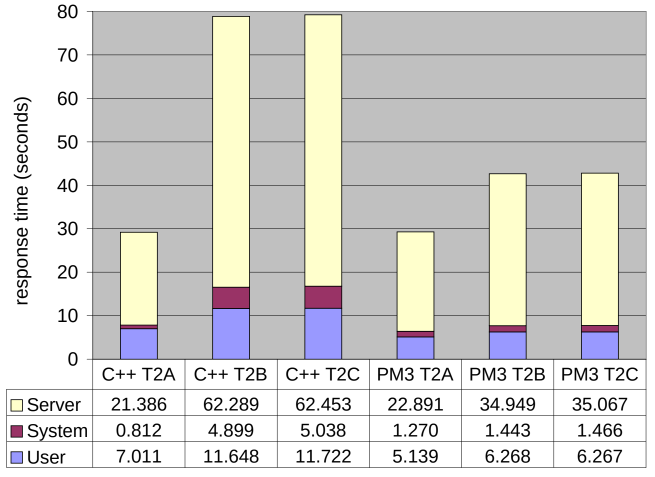
Server interactions		SHORE/C++ T2A	SHORE/C++ T2B	SHORE/C++ T2C	PM3 T2A	PM3 T2B	PM3 T2C
Fetches	objects	41 594	41 594	41 594	1	1	1
	pages				413	462	462
	bytes	3 053 136	3 053 136	3 053 136	3 403 192	3 806 952	3 806 952
Updates		491 objects	9 820 objects	9 820 objects	90 pages	171 pages	171 pages

Table 3: Traversal T2



	C++ T2A	C++ T2B	C++ T2C	PM3 T2A	PM3 T2B	PM3 T2C
□ Server	18.946	30.647	29.976	12.167	11.486	12.066
■ System	0.685	2.727	2.608	1.178	1.242	1.239
■ User	5.943	8.267	8.697	4.657	4.876	4.946

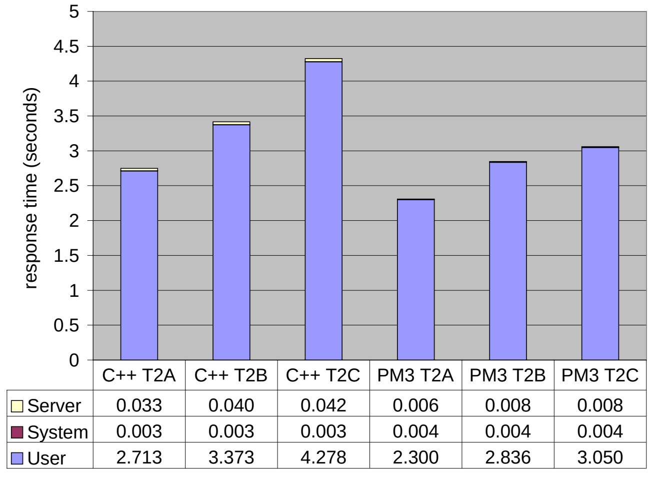
(one)



	C++ T2A	C++ T2B	C++ T2C	PM3 T2A	PM3 T2B	PM3 T2C
□ Server	21.386	62.289	62.453	22.891	34.949	35.067
■ System	0.812	4.899	5.038	1.270	1.443	1.466
■ User	7.011	11.648	11.722	5.139	6.268	6.267

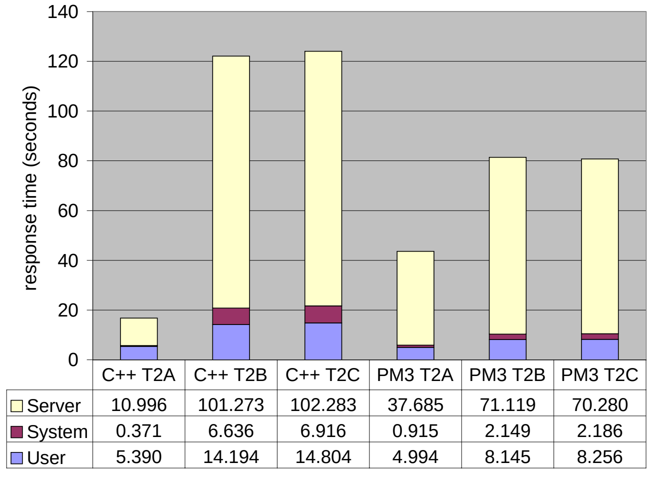
(many)

(a) Cold



	C++ T2A	C++ T2B	C++ T2C	PM3 T2A	PM3 T2B	PM3 T2C
□ Server	0.033	0.040	0.042	0.006	0.008	0.008
■ System	0.003	0.003	0.003	0.004	0.004	0.004
■ User	2.713	3.373	4.278	2.300	2.836	3.050

(one)



	C++ T2A	C++ T2B	C++ T2C	PM3 T2A	PM3 T2B	PM3 T2C
□ Server	10.996	101.273	102.283	37.685	71.119	70.280
■ System	0.371	6.636	6.916	0.915	2.149	2.186
■ User	5.390	14.194	14.804	4.994	8.145	8.256

(many)

(b) Hot: 3 iterations

Figure 12: Traversal T2

8 Conclusions

We believe PM3 to be the first successful and efficient extension of a systems programming language with reachability-based persistence. Our implementation approach, based on mostly-copying garbage collection, is robust and requires no cooperation from the compiler. Our experiments demonstrate that the creation of ambiguously-persistent data has minimal impact, at least for the OO7 benchmark operations. This does not mean to say that large amounts of ambiguously-persistent data can never cause problems; the pitfalls of conservatism are well-documented [78]. Techniques such as *black-listing* [15] can provide relief in such isolated cases.

We have also compared PM3's performance with that of the non-orthogonal SHORE/C++ binding, with highly favorable results. PM3 generally outperforms SHORE/C++ on the traversal and insertion operations of the OO7 benchmark. The expense of the trap-based barrier mechanism for PM3 appears to be tolerable, at least for the large fault and update granularity of 8 Kbyte pages. Nevertheless, with evidence that large granularities can adversely impact on fetch and commit times [44, 45, 46, 38], we plan also to explore software techniques for finer granularities by having the compiler attach explicit software barriers to object accesses, with compiler optimizations to remove those that are redundant. We also plan to explore the integration of mostly-copying persistence with buffer management, disk garbage collection and extended transaction support.

In summary, reachability-based orthogonal persistence is a desirable aim for any integration of programming languages with database systems, since it offers a more desirablethe cleanest model for programming access to persistent data. We have clearly demonstrated that implementing the model introduces no inherent inneffi-ciencies. The degree to which a given persistence model is orthogonal can now be claimed as the gold standard by which all persistent programming languages and systems should be judged.

Acknowledgments

We dedicate this work to the memory of Geoff Wyant. Norman Ramsey, Paul Wilson and Tony Printezis gave valuable feedback on early drafts of this paper. Aria Novianto participated in many of our early discussions on mostly-copying persistence.

This research is supported in part by the National Science Foundation under Grant No. CCR-9711673 and by gifts from Sun Microsystems, Inc.

References

[1] AGRAWAL, R., AND GEHANI, N. H. ODE (Object Database and Environment): The language and the data model. In *Proceedings of the ACM International Conference on Management of Data* (Portland, Oregon, May). *ACM SIGMOD Record 18*, 2 (June 1989), pp. 36–45.

[2] AGRAWAL, R., AND GEHANI, N. H. Rationale for the design of persistence and query processing facilities in the database language O++. In Hull et al. [49], pp. 25–40.

[3] ALAGIĆ, S. The ODMG object model: does it make sense? In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (Atlanta, Georgia, Oct.). *ACM SIGPLAN Notices 32*, 10 (Oct. 1997), pp. 253–270.

[4] APPEL, A. W., AND LI, K. 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, Apr.). *ACM SIGPLAN Notices 26*, 4 (Apr. 1991), pp. 96–107.

[5] ATKINSON, M., CHISOLM, K., AND COCKSHOTT, P. PS-Algol: an Algol with a persistent heap. *ACM SIGPLAN Notices 17*, 7 (July 1982), 24–31.

[6] ATKINSON, M. P., BAILEY, P. J., CHISHOLM, K. J., COCKSHOTT, P. W., AND MORRISON, R. An approach to persistent programming. *The Computer Journal 26*, 4 (Nov. 1983), 360–365.

[7] ATKINSON, M. P., AND BUNEMAN, O. P. Types and persistence in database programming languages. *ACM Comput. Surv. 19*, 2 (June 1987), 105–190.

[8] ATKINSON, M. P., CHISHOLM, K. J., COCKSHOTT, W. P., AND MARSHALL, R. M. Algorithms for a persistent heap. *Software: Practice and Experience 13*, 7 (Mar. 1983), 259–271.

[9] ATKINSON, M. P., DAYNÈS, L., JORDAN, M. J., PRINTEZIS, T., AND SPENCE, S. An orthogonally persistent Java. *ACM SIGMOD Record 25*, 4 (Dec. 1996), 68–75.

[10] ATKINSON, M. P., JORDAN, M. J., DAYNÈS, L., AND SPENCE, S. Design issues for persistent Java: A type-safe object-oriented, orthogonally persistent system. In Connor and Nettles [27], pp. 33–47.

[11] ATKINSON, M. P., AND MORRISON, R. Orthogonally persistent object systems. *International Journal on Very Large Data Bases 4*, 3 (1995), 319–401.

[12] BARTLETT, J. F. Compacting garbage collection with ambiguous roots. Research Report 88/2, Western Research Laboratory, Digital Equipment Corporation, Feb. 1988.

[13] BARTLETT, J. F. Mostly-copying garbage collection picks up generations and C++. Technical Note TN-12, Western Research Laboratory, Digital Equipment Corporation, Oct. 1989.

[14] BAUMANN, R. Client/server distribution in a structure-oriented database management system. Tech. Rep. AIB 97-14, RWTH Aachen, Germany, 1997.

[15] BOEHM, H.-J. Space efficient conservative garbage collection. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (Albuquerque, New Mexico, June). *ACM SIGPLAN Notices 28*, 6 (June 1993), pp. 197–206.

[16] BOEHM, H.-J., AND WEISER, M. Garbage collection in an uncooperative environment. *Software: Practice and Experience 18*, 9 (Sept. 1988), 807–820.

[17] BRAHNMATH, K., NYSTROM, N., HOSKING, A. L., AND CUTTS, Q. Swizzle barrier optimizations for orthogonal persistence in Java. In Morrison et al. [62], pp. 268–278.

[18] BRAHNMATH, K. J. Optimizing orthogonal persistence for Java. Master's thesis, Purdue University, May 1998.

[19] CARDELLI, L., DONAHUE, J., GLASSMAN, L., JORDAN, M., KALSOW, B., AND NELSON, G. Modula-3 language definition. In *Systems Programming with Modula-3*, G. Nelson, Ed. Prentice Hall, 1991, ch. 2, pp. 11–66.

[20] CAREY, M. J., DEWITT, D. J., FRANKLIN, M. J., HALL, N. E., MCAULIFFE, M. L., NAUGHTON, J. E., SCHUH, D. T., SOLOMON, M. H., TAN, C. K., TSATALOS, O. G., WHITE, S. J., AND ZWILLING, M. J. Shoring up persistent applications. In SIGMOD [74], pp. 383–394.

[21] CAREY, M. J., DEWITT, D. J., AND NAUGHTON, J. F. The OO7 benchmark. In *Proceedings of the ACM International Conference on Management of Data* (Washington, DC, May). *ACM SIGMOD Record 22*, 2 (June 1993), pp. 12–21.

[22] CAREY, M. J., FRANKLIN, M. J., AND ZAHARIOUDAKIS, M. Fine-grained sharing in a page server OODBMS. In SIGMOD [74], pp. 359–370.

[23] CATTELL, R. G. G., BARRY, D., BARTELS, D., BERLER, M., EASTMAN, J., GAMERMAN, S., JORDAN, D., SPRINGER, A., STRICKLAND, H., AND WADE, D., Eds. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann, 1997.

[24] CHASE, J. S., LEVY, H. M., FEELEY, M. J., AND LAZOWSKA, E. D. Sharing and protection in a single-address space operating system. *ACM Trans. Comput. Syst. 12*, 4 (Nov. 1994), 271–307.

[25] CHASE, J. S., LEVY, H. M., LAZOWSKA, E. D., AND BAKER-HARVEY, M. Lightweight shared objects in a 64-bit operating system. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (Vancouver, Canada, Oct.). *ACM SIGPLAN Notices 27*, 10 (Oct. 1992), pp. 397–413.

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

[27] CONNOR, R., AND NETTLES, S., Eds. *Proceedings of the Seventh International Workshop on Persistent Object Systems* (Cape May, New Jersey, May 1996). Persistent Object Systems: Principles and Practice, Morgan Kaufmann, 1997.

[28] DEARLE, A., CONNER, R., BROWN, F., AND MORRISON, R. Napier88 – a database programming language? In Hull et al. [49], pp. 179–195.

[29] DEARLE, A., DI BONA, R., FARROW, J., HENSKENS, F., LINDSTRÖM, A., ROSENBERG, J., AND VAUGHAN, F. Grasshopper: An orthogonally persistent operating system. *Computer Systems 7*, 3 (Summer 1994), 289–312.

[30] DEARLE, A., SHAW, G. M., AND ZDONIK, S. B., Eds. *Proceedings of the Fourth International Workshop on Persistent Object Systems* (Martha's Vineyard, Massachusetts, Sept. 1990). Implementing Persistent Object Bases: Principles and Practice, Morgan Kaufmann, 1991.

[31] DETLEFS, D. D., HERLIHY, M. P., AND WING, J. M. Inheritance of synchronization and recovery in Avalon/C++. *IEEE Computer 21*, 12 (Dec. 1988), 57–69.

[32] DETREVILLE, J. Experience with concurrent garbage collectors for Modula-2+. Tech. Rep. 64, Systems Research Center, Digital Equipment Corporation, Palo Alto, CA, Aug. 1990.

[33] DIWAN, A., MOSS, J. E. B., AND HUDSON, R. L. Compiler support for garbage collection in a statically typed language. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (San Francisco, California, June). *ACM SIGPLAN Notices 27*, 7 (July 1992), pp. 273–282.

[34] ELPHINSTONE, K., RUSSELL, S., HEISER, G., AND LIEDTKE, J. Supporting persistent object systems in a single address space. In Connor and Nettles [27], pp. 111–119.

[35] FENICHEL, R. R., AND YOCHELSON, J. C. A LISP garbage-collector for virtual-memory computer systems. *Commun. ACM 12*, 11 (Nov. 1969), 611–612.

[36] HAINES, N., KINDRED, D., MORRISETT, J. G., NETTLES, S. M., AND WING, J. M. Composing first-class transactions. *ACM Trans. Program. Lang. Syst. 16*, 6 (Nov. 1994), 1719–1736.

[37] HEISER, G., ELPHINSTONE, K., VOCHTELOO, J., RUSSELL, S., AND LIEDTKE, J. The Mungi single-address-space operating system. *Software: Practice and Experience 28*, 9 (July 1998), 901–928.

[38] HOSKING, A. L. *Lightweight Support for Fine-Grained Persistence on Stock Hardware*. PhD thesis, University of Massachusetts at Amherst, Feb. 1995. Available as Computer Science Technical Report 95-02.

[39] HOSKING, A. L. Residency check elimination for object-oriented persistent languages. In Connor and Nettles [27], pp. 174–183.

[40] HOSKING, A. L., BROWN, E., AND MOSS, J. E. B. Update logging for persistent programming languages: A comparative performance evaluation. In *Proceedings of the International Conference on Very Large Data Bases* (Dublin, Ireland, Aug.). Morgan Kaufmann, 1993, pp. 429–440.

[41] HOSKING, A. L., AND CHEN, J. PM3: An orthogonally persistent systems programming language – design implementation, performance. In *Proceedings of the International Conference on Very Large Data Bases* (Edinburgh, Scotland, Sept.). Morgan Kaufmann, 1999.

[42] HOSKING, A. L., AND MOSS, J. E. B. Compiler support for persistent programming. Tech. Rep. 91-25, Department of Computer Science, University of Massachusetts at Amherst, Mar. 1991.

[43] HOSKING, A. L., AND MOSS, J. E. B. Towards compile-time optimisations for persistence. In Dearle et al. [30], pp. 17–27.

[44] HOSKING, A. L., AND MOSS, J. E. B. 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, Sept.). *ACM SIGPLAN Notices 28*, 10 (Oct. 1993), pp. 288–303.

[45] HOSKING, A. L., AND MOSS, J. E. B. 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, Dec.). *ACM Operating Systems Review 27*, 5 (Dec. 1993), pp. 106–119.

[46] HOSKING, A. L., AND MOSS, J. E. B. Lightweight write detection and checkpointing for fine-grained persistence. Tech. Rep. 95-084, Department of Computer Sciences, Purdue University, Dec. 1995.

[47] HOSKING, A. L., AND NOVIANTO, A. P. Reachability-based orthogonal persistence for C, C++ and other intransigents. In *Proceedings of the OOPSLA Workshop on Memory Management and Garbage Collection* (Atlanta, Georgia, Oct.). 1997. http://www.dcs.gla.ac.uk/˜huw/oopsla97/gc/papers.html.

[48] HOSKING, A. L., NYSTROM, N., CUTTS, Q., AND BRAHNMATH, K. Optimizing the read and write barriers for orthogonal persistence. In *Proceedings of the Eighth International Workshop on Persistent Object Systems* (Tiburon, California, August 1998), R. Morrison, M. Jordan, and M. Atkinson, Eds. Advances in Persistent Object Systems. Morgan Kaufmann, 1999, pp. 149–159.

[49] HULL, R., MORRISON, R., AND STEMPLE, D., Eds. *Proceedings of the Second International Workshop on Database Programming Languages* (Salishan Lodge, Gleneden Beach, Oregon, June 1989). Morgan Kaufmann, 1990.

[50] *Proceedings of the ACM International Symposium on Memory Management* (Vancouver, Canada, Oct., 1998). *ACM SIGPLAN Notices 34*, 3 (Mar. 1999).

[51] JONES, R. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, May 1996. With a chapter by R. Lins.

[52] JORDAN, M. Early experiences with persistent Java. In *Proceedings of the First International Workshop on Persistence and Java* (Drymen, Scotland, Sept.), M. P. Atkinson and M. J. Jordan, Eds. Sun Microsystems Laboratories Technical Report 96-58, Nov. 1996.

[53] JORDAN, M., AND ATKINSON, M. Orthogonal persistence for Java – a mid-term report. In Morrison et al. [62].

[54] KAEHLER, T. Virtual memory on a narrow machine for an object-oriented language. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, Sept.). *ACM SIGPLAN Notices 21*, 11 (Nov. 1986), pp. 87–106.

[55] KAEHLER, T., AND KRASNER, G. LOOM – large object-oriented memory for Smalltalk-80 systems. In *Smalltalk-80: Bits of History, Words of Advice*, G. Krasner, Ed. Addison-Wesley, 1983, ch. 14, pp. 251–270.

[56] KAKKAD, S. V., JOHNSTONE, M. S., AND WILSON, P. R. Portable run-time type description for conventional compilers. In ISMM [50], pp. 146–153.

[57] KIESEL, N., SCHÜRR, A., AND WESTFECHTEL, B. GRAS, a graph-oriented (software) engineering database system. *Information Systems 20*, 1 (1995), 21–52.

[58] LAMB, C., LANDIS, G., ORENSTEIN, J., AND WEINREB, D. The ObjectStore database system. *Commun. ACM 34*, 10 (Oct. 1991), 50–63.

[59] LIM, T. F., PARDYAK, P., AND BERSHAD, B. N. A memory-efficient real-time non-copying garbage collector. In ISMM [50], pp. 118–129.

[60] MATTHES, F., AND SCHMIDT, J. W. Persistent threads. In *Proceedings of the International Conference on Very Large Data Bases* (Santiago, Chile, Sept.). Morgan Kaufmann, 1994, pp. 403–414.

[61] MORRISON, R., BROWN, A., CARRICK, R., CONNOR, R., DEARLE, A., AND ATKINSON, M. P. The Napier type system. In *Proceedings of the Third International Workshop on Persistent Object Systems* (Newcastle, New South Wales, Australia, Jan. 1989), J. Rosenberg and D. Koch, Eds. Workshops in Computing. Springer-Verlag, 1990, pp. 3–18.

[62] MORRISON, R., JORDAN, M., AND ATKINSON, M., Eds. *Proceedings of the Third International Workshop on Persistence and Java* (Tiburon, California, August 1998). Advances in Persistent Object Systems, Morgan Kaufmann, 1999.

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

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

[65] NYSTROM, N., HOSKING, A. L., WHITLOCK, D., CUTTS, Q., AND DIWAN, A. Partial redundancy elimination for access path expressions. Tech. Rep. 98-044, Department of Computer Sciences, Purdue University, Oct. 1998. Submitted for publication.

[66] NYSTROM, N. J. Bytecode level analysis and optimization of Java classes. Master's thesis, Purdue University, Aug. 1998.

[67] RICHARDSON, J. E. Compiled item faulting: A new technique for managing I/O in a persistent language. In Dearle et al. [30], pp. 3–16.

[68] RICHARDSON, J. E., AND CAREY, M. J. Programming constructs for database implementations in EXODUS. In *Proceedings of the ACM International Conference on Management of Data* (San Francisco, California, May). *ACM SIGMOD Record 16*, 3 (Dec. 1987), pp. 208–219.

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

[70] RICHARDSON, J. E., CAREY, M. J., AND SCHUH, D. T. The design of the E programming language. *ACM Trans. Program. Lang. Syst. 15*, 3 (July 1993), 494–534.

[71] ROSENBERG, J., DEARLE, A., HULSE, D., LINDSTRÖM, A., AND NORRIS, S. Operating system support for persistent and recoverable computations. *Commun. ACM 39*, 9 (Sept. 1996), 62–69.

[72] SCHKOLNICK, M. A clustering algorithm for hierarchical structures. *ACM Trans. Database Syst. 2*, 1 (Mar. 1977), 27–44.

[73] SCHUH, D., CAREY, M., AND DEWITT, D. Persistence in E revisited – implementation experiences. In Dearle et al. [30], pp. 345–359.

[74] *Proceedings of the ACM International Conference on Management of Data* (Minneapolis, Minnesota, May). *ACM SIGMOD Record 23*, 2 (June 1994).

[75] SINGHAL, V., KAKKAD, S. V., AND WILSON, P. R. Texas, an efficient, portable persistent store. In *Proceedings of the Fifth International Workshop on Persistent Object Systems* (San Miniato, Italy, Sept.), A. Albano and R. Morrison, Eds. Workshops in Computing. Springer-Verlag, 1992, pp. 11–33.

[76] STAMOS, J. W. Static grouping of small objects to enhance performance of a paged virtual memory. *ACM Trans. Database Syst. 2*, 2 (May 1984), 155–180.

[77] STRAW, A., MELLENDER, F., AND RIEGEL, S. Object management in a persistent Smalltalk system. *Software: Practice and Experience 19*, 8 (Aug. 1989), 719–737.

[78] WENTWORTH, E. P. Pitfalls of conservative garbage collection. *Software: Practice and Experience 20*, 7 (July 1990).

[79] WHITE, S. J., AND DEWITT, D. J. QuickStore: A high performance mapped object store. In SIGMOD [74], pp. 395–406.

[80] WILSON, P. R. Uniprocessor garbage collection techniques. *ACM Comput. Surv.*. To appear.

[81] WILSON, P. R. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management* (St. Malo, France, Sept.), Y. Bekkers and J. Cohen, Eds. No. 637 in Lecture Notes in Computer Science. Springer-Verlag, 1992.

[82] WILSON, P. R., AND KAKKAD, S. V. 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* (Paris, France, Sept.). IEEE Computer Society, 1992, pp. 364–377.