Reachability-based orthogonal persistence for C, C++ and other intransigents

Antony L. Hosking[*]
hosking@cs.purdue.edu

Aria P. Novianto
novianto@cs.purdue.edu

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907-1398, USA

August 4, 1997

Abstract

We describe how reachability-based orthogonal persistence can be supported even in uncooperative implementations of languages such as C and C++, where there is no support for accurate discovery of transient roots. Such ambiguous transient roots preclude the usual copying approach to promotion of objects from transient to persistent by reachability from well-known persistent roots [Atkinson et al. 1983]. Our approach extends Bartlett's mostly-copying garbage collector [Bartlett 1988; 1989] to manage both transient objects and resident persistent objects, and to perform the reachability closure necessary for stabilization in a mostly-copying fashion. The only requirement, necessary anyway for persistence, is accurate discovery of pointers in heap-allocated objects. Such support can be obtained through direct compiler assistance, extracted from debugging information, or provided explicitly by the programmer. We also consider how the garbage collector can inform the buffer manager of persistent pages that are likely candidates for removal.
Keywords: mostly-copying garbage collection, persistence by reachability, buffer management

1 Introduction

Orthogonal persistence [Atkinson and Morrison 1995] is widely preferred as the ideal model for persistent languages. Orthogonality typically means that persistence is a property independent of type. It is also usually taken to mean that any allocated *instance* of a type has the potential to persist; in other words, programmers are not required to indicate persistence at allocation time. Rather, objects persist by virtue of their being reachable from some set of persistent roots. Typically, persistent stores allow certain persistent objects to be named, and these bindings themselves persist as gateway objects into the persistent store; all objects reachable from these persistently-named objects must themselves persist. Thus, persistence is determined by reachability closure similarly to garbage collection [Atkinson et al. 1983].

In order to retain the ability to reclaim transient storage independently of persistent storage, most orthogonal persistence implementations call for allocation of new objects in a separately garbage-collected transient region of the allocation heap. Only when the persistent heap is stabilized are transient 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 the transient space into the persistent space. However, objects can only be copied if all pointers to those objects can accurately be determined, and updated to reflect the relocation of the object. In environments where such information is unavailable objects cannot be moved. Thus, all previous implementations of persistence for languages such as C and C++ (of which we are aware) break orthogonality, and require the programmer to distinguish transient and persistent objects whether by type or upon allocation.[1] Here, we show that

[*] See also: http://www.cs.purdue.edu/people/hosking. This work is supported by a gift from Sun Microsystems, Inc.

[1] Detlefs et al. 1988; Agrawal and Gehani 1989; Agrawal and Gehani 1990; Richardson and Carey 1990; Schuh et al. 1991; Lamb et al. 1991; Singhal et al. 1992; Richardson et al. 1993; White and DeWitt 1994.

reachability-based orthogonal persistence for such languages and environments is indeed possible using an approach based on mostly-copying garbage collection.

The casual reader is referred to the background material of Atkinson and Morrison [1995] on orthogonal persistence, and to Wilson [1992] or Jones and Lins [1996] on garbage collection.

2 Mostly-copying garbage collection

Mostly-copying garbage collection [Bartlett 1988; 1989] represents a hybrid of conservative [Boehm and Weiser 1988] and copying [Cheney 1970] collection. It is suitable for use in environments lacking accurate information on the layout of the register, static or stack areas; objects that *appear* to have references, termed *ambiguous roots*, from these areas are treated conservatively and are not moved. The collector does assume that all pointers in heap-allocated objects can be found accurately; objects accessible only from other heap objects can thus be moved during garbage collection. Finding heap pointers accurately can be achieved using information describing the layout of heap objects, generated either automatically by the compiler (whether directly, or indirectly through extraction from debugging information [Singhal et al. 1992; Wilson and Kakkad 1992]) or provided explicitly by the programmer [Bartlett 1989] (though the latter approach may be error-prone).

For mostly-copying collection the heap is divided into a number of equal-sized heap *pages*.[2] The collector proceeds by copying all reachable objects, dividing the heap into two page spaces: *to-space*, containing objects copied by the garbage collector; and *from-space*, containing objects not yet copied by the collector. 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 objects to be copied by the collector in two possible ways: either by physically moving it to a page in to-space, or simply by resetting the space identifier of its page. The latter mechanism, called page *promotion*, is how ambiguous roots are handled.

The mostly-copying collector, sketched in Figure 1, operates in three phases. We assume that the spaces are abstracted as sets of pages, and that the operations $-$ and $+$ remove and insert an element in a set, respectively. The variables p, l and r range over heap pages, heap pointer locations and heap pointers (references), respectively. The auxiliary procedure *promote* removes a page from one space and adds it to another. The procedure *copy_scan* performs Cheney-style iterative scanning of the transitive closure of objects reachable from some stack of pages, *copyStack* [Cheney 1970]. We assume several additional auxiliary procedures:

page(r)**:** returns 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
copied(r)**:** returns true if the object (in from-space) referred to by r has been copied, and false otherwise
copy(r,s)**:** allocates a copy in space s of the object (in from-space) referred to by r; leaves a forwarding address behind
copy_address(r)**:** denotes the forwarding address from the original object (in from-space) referred to by r to its copy

The garbage collector (*gc*) begins with the to-space empty (line 22). It assumes a finite set of ambiguous roots (AR) from the registers, stack, and static areas. The first phase (line 23) promotes ambiguously-referenced pages to to-space. Note that promotion may retain unreferenced garbage objects that just happen to lie in those pages. After this phase, the only pages in to-space are those pinned by ambiguous roots. The second phase (line 25) copies reachable objects from from-space to to-space. The pages containing copied objects (initially just those pinned by the ambiguous roots in the first phase) are placed in a stack for processing. All pointer locations in copied pages are scanned, and each object in from-space reachable from a copied page in to-space is itself copied in turn to a to-space page, leaving behind a forwarding address; each pointer location

[2] Heap pages are not necessarily identified with virtual memory pages.

```
1  proc promote(p, var from, var to) ≡
2     from := from − p;
3     to := to + p.
4
5  proc copy_scan(move) ≡
6     while ¬copyStack.empty() do
7           p := copyStack.pop();
8           foreach l ∈ pointer_locations(p) do move(l); end
9     end.
10
11 proc mover(l) ≡
12    r := l↑;
13    if page(r) ∈ from-space then
14       if ¬copied(r) then
15          copy_address(r) := copy(r, to-space);
16          copyStack.push(page(copy_address(r)));
17       end;
18       l↑ := copy_address(r)
19    end.
20
21 proc gc() ≡
22    to-space := {};
23    foreach r ∈ AR do promote(page(r), from-space, to-space); end;
24    copyStack.init(to-space);
25    copy_scan(mover);
26    foreach p ∈ from-space do free(p) end;
27    from-space := to-space.
```

Figure 1: Mostly-copying garbage collection

is updated to refer to the forwarded to-space copy. 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 newly-copied to them, so eventually the stack becomes empty. At the end of this second phase there are no pointers from to-space to from-space, and the pages in from-space can be freed (line 26). The garbage collection is now complete and to-space becomes from-space for the subsequent collection (line 27).[3]

3 Mostly-copying persistence by reachability

We extend the implementation of the transient heap for persistence by assuming a new space of heap pages, *persistent-space*, in which resident persistent objects are cached. For now, we assume that only transient pages are deallocated by the garbage collector; that is, persistent-space pages are kept segregated from transient (from- and to-space pages). We assume also that references to *resident* persistent objects are *swizzled*[4] to direct memory pointers, and that those pointers may be stored in registers, the stack and static areas, as well as in the heap. Naturally, resident persistent objects must be treated as heap roots when garbage collecting the transient heap, to ensure that the transient objects to which cached persistent objects may have

[3] It is worth noting that the mostly-copying collector can be made both generational [Bartlett 1989] and incremental [DeTreville 1990]. Indeed, our implementation of mostly-copying persistence by reachability merges easily with the existing Modula-3 incremental/generational collector.

[4] Swizzling [Moss 1992; Wilson and Kakkad 1992] is the conversion of persistent object references from their persistent format to direct memory pointers. In object-oriented database systems the persistent format is typically some sort of *object identifier*, by which the object can be located in secondary storage. Accessing a resident object by identifier usually requires translation of the identifier to a memory address, with retrieval of the object if it is not yet resident. Allowing applications to cache pointers to resident objects can yield significant performance improvements by eliminating the need for identifier translation.

```
1  proc stabilizer(l) ≡
2    r := l↑;
3    if page(r) ∈ to-space then
4      promote(page(r), to-space, persistent-space);
5      copyStack.push(page(r));
6    elsif page(r) ∈ from-space then
7        if ¬copied(r) then
8          copy_address(r) := copy(r, persistent-space);
9          copyStack.push(page(copy_address(r)));
10       end;
11       l↑ := copy_address(r);
12   end.
13
14 proc stabilize() ≡
15   to-space := {};
16   foreach r ∈ AR do promote(page(r), from-space, to-space) end;
17   foreach r ∈ PR do
18     if page(r) ∈ to-space then promote(page(r), to-space, persistent-space);
19     elsif page(r) ∈ from-space then
20         if ¬copied(r) then copy_address(r) := copy(r, persistent-space) end;
21     end;
22   end;
23   copyStack.init(persistent-space);
24   copy_scan(stabilizer);
25   foreach p ∈ persistent-space do flush(p) end;
26   copyStack.init(to-space);
27   copy_scan(mover);
28   foreach p ∈ from-space do free(p) end;
29   from-space := to-space.
```

Figure 2: Mostly-copying stabilization

been made to refer are retained from one collection to the next. We defer completely specifying how the garbage collector must be extended to manage persistent pages until Section 4, where we also consider how the mostly-copying collector can be augmented to assist in buffer management by removing persistent pages when their cached objects are no longer reachable from the executing persistent program. For now, we focus on the basic mechanism need for reachability-based orthogonal persistence: *stabilization*.

3.1 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 checkpoint or commit if the language offers transactional concurrency control) all modified persistent objects must be flushed to disk. Since a persistent object may have been modified to refer to transient objects, they must also be made persistent and stabilized themselves, in turn making persistent any transient objects to which they refer, and so on. We also assume there are certain named root objects in the persistent store, and that the program may have changed one of these bindings, requiring that reachability also be determined from these named roots. Forming the necessary 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 recognise heap references from the registers, stacks and static areas. The only requirement is for accuracy in locating pointers stored in the heap.

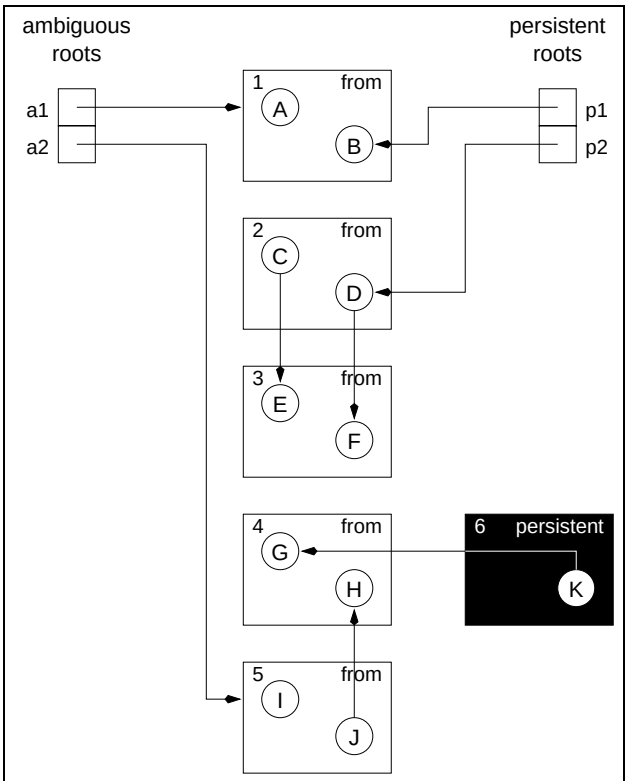

Figure 3: Initial state

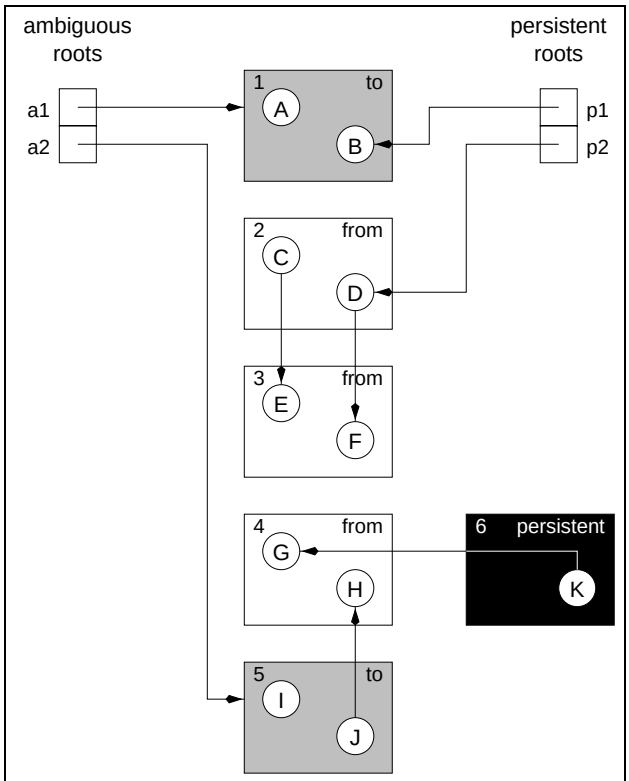

Figure 4: Trace ambiguous roots

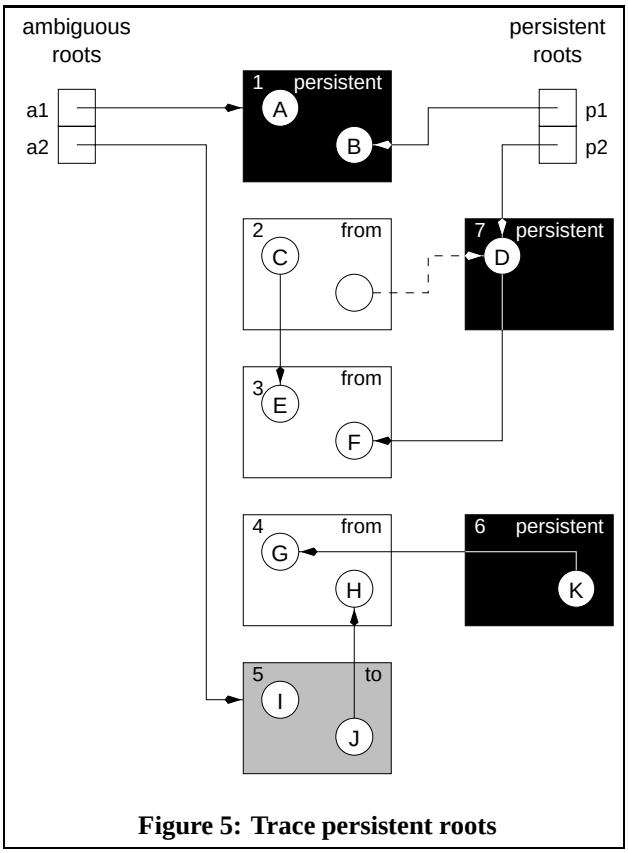

Figure 5: Trace persistent roots

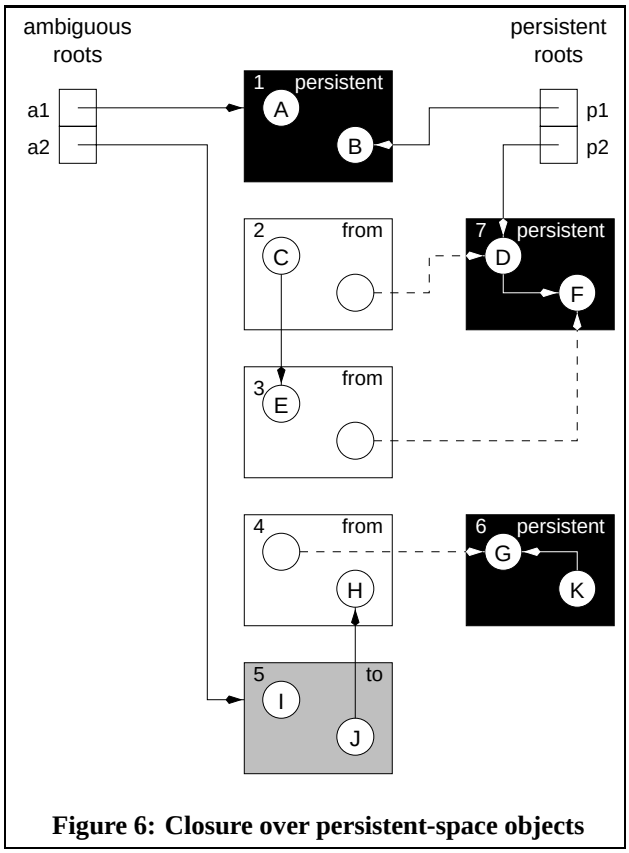

Figure 6: Closure over persistent-space objects

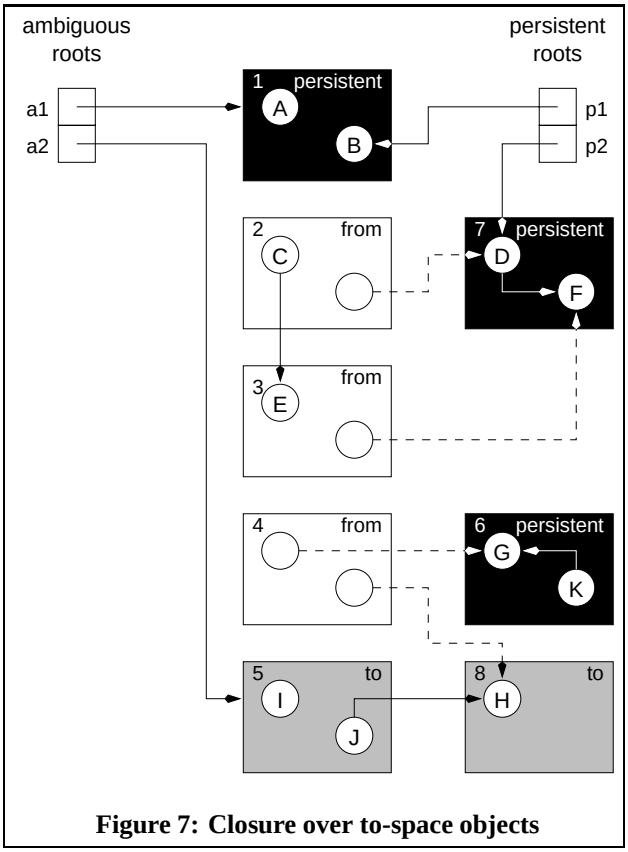

Figure 7: Closure over to-space objects

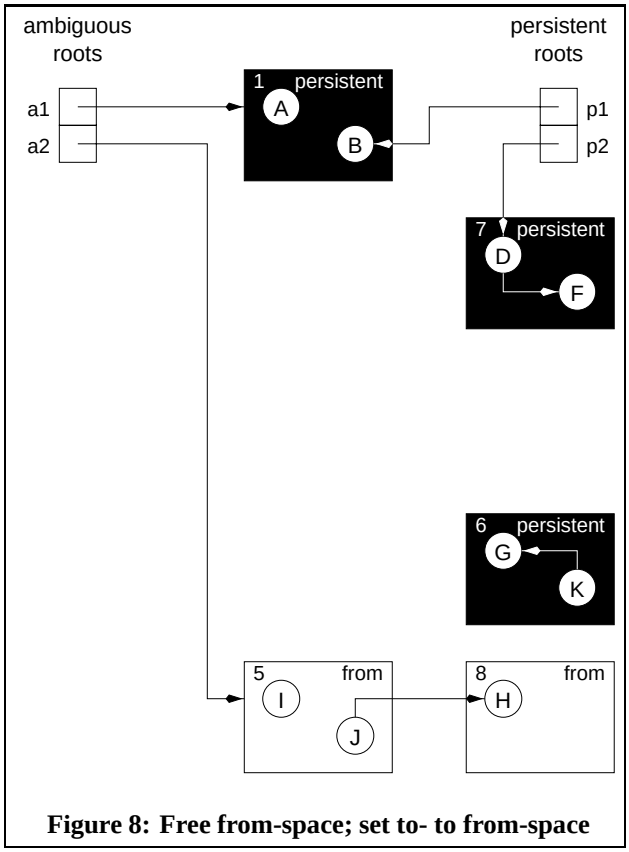

Figure 8: Free from-space; set to- to from-space

The mostly-copying *stabilize* procedure is sketched in Figure 2. We illustrate each phase of its operation with an example (Figures 3–8). Figure 3 shows the initial heap configuration for the example. As for mostly-copying garbage collection we begin by promoting from from-space to to-space all ambiguously-referenced pages (line 16). 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 4.

A second phase (lines 17–22) copies all transient objects reachable from the persistent roots (PR) into persistent-space, either logically by promoting the ambiguously-referenced to-space pages in which they lie, or physically by copying them to pages in persistent-space. Figure 5 illustrates this phase, with page 1 (referenced by p1) promoted to persistent-space, and object D (referenced by p2) copied to the new page 7 in persistent-space.

Phase three (line 24) computes the reachability closure over all objects in persistent-space. Again, ambiguously-referenced to-space pages are simply promoted, while from-space objects can physically be copied. At the end of this phase all objects are in persistent-space that should be, although promotion of ambiguously-referenced pages may have made certain objects persist that need not. Objects remaining in from-space or to-space are not reachable from persistent storage, and definitely need not persist. In the example (Figure 6) objects F and G have been copied into persistent-space pages 7 and 6, respectively.

Line 25 flushes all objects in persistent-space pages to the persistent store (with whatever shadow paging or logging is necessary for rollback and recovery). Note that we make no assumptions about the underlying persistent store, whether it is 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.

The last phases of stabilization merely finish off with a basic mostly-copying garbage collection to collect the remaining from-space pages. Line 27 copies remaining reachable transient from-space objects to to-space, as illustrated in Figure 7 for

object H. Finally, the now unreachable from-space pages can be freed (Figure 8). Note that persistent-space page 6 remains although it is no longer reachable from the application. In the next section we modify the mostly-copying collection and stabilization algorithms to recognize and reclaim such transiently unreachable persistent pages.

Correctness and termination of mostly-copying stabilization can be inferred from the invariants stated here, similarly to the way for non-persistent mostly-copying garbage collection.

4 Buffer management

We say that a persistent page is *transiently unreachable* if it cannot be reached via pointers from transient memory. Although we will keep a copy of such a page in persistent storage (modulo garbage collection of the persistent store itself), we are not obligated to keep it cached in memory. Such a page is a prime candidate for replacement, since the application cannot immediately begin accessing it again without first faulting some other persistent objects by which it can be reached. We now modify the mostly-copying collection and stabilization algorithms to recognize and reclaim such pages. In essence, we exploit information obtained via reachability analysis during garbage collection and stabilization as hints for persistent-space buffer management. The change is relatively straightforward, and entails placing persistent pages under the management of the mostly-copying collector. We relax the prior assumption that persistent pages are segregated from from- and to-space pages. Instead, persistent pages are in one of either from- or to-space at any given instant, just like transient pages. The modified algorithms for garbage collection and stabilization are given in Figures 9 and 10. Note that we avoid the unnecessary overhead of physically copying already-persistent objects during collection or stabilization by always simply promoting their page. Notice also that persistent pages are always flushed before being freed, to make sure updates to those pages are propagated back to persistent storage.

There are lingering subtleties here, having to do with implementation of the page *flush* operation and its use during garbage collection when freeing transiently unreachable persistent pages (see Figure 9). We assume a page can only be flushed when all pointers in the page are to objects in persistent pages (so that the pointers can be unswizzled as necessary during the flush). To avoid flushing persistent pages for which this condition does not hold, the garbage collection reachability phase copies to to-space all objects reachable from persistent pages (lines 27&28), even from persistent pages that are transiently unreachable and remain in from-space. Thus, it is trivially possible to promote a transiently unreachable persistent page since it cannot contain pointers to from-space. We use this escape to avoid having to stabilize all objects transitively reachable from that page. In other words, we only flush persistent pages that are completely unconnected to the transient heap, having neither incoming pointers from, nor outgoing pointers to, transient (to-space) objects (lines 29–40).

5 Interaction with other persistence mechanisms

As mentioned earlier, we make no assumptions about the underlying persistent storage architecture, be it page- or object-server. 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, in our initial implementation of persistence for Modula-3 [Nelson 1991] we are using mostly-copying persistence with Texas-style "pointer-swizzling at page-fault time" [Singhal et al. 1992; Wilson and Kakkad 1992] as our underlying swizzling and faulting mechanism above the Mneme persistent object store [Moss and Sinofsky 1988; Moss 1990a; 1990b]. We have designed the system for a move to explicit object fault checks in later versions, without needing to change the mostly-copying heap management mechanism. Similarly, one can imagine re-engineering the Texas persistent store to use mostly-copying

```
1  proc mover'(l) ≡
2    r := l↑;
3    if page(r) ∈ from-space then
4      if page(r) ∈ persistent-space then
5        promote(page(r), from-space, to-space)
6      else
7          if ¬copied(r) then
8            copy_address(r) := copy(r, to-space);
9            copyStack.push(page(copy_address(r)));
10         end;
11         l↑:= copy_address(r);
12     end;
13   end.
14
15 proc gc'() ≡
16   to-space := {};
17   foreach r ∈ AR do promote(page(r), from-space, to-space) end;
18   foreach r ∈ PR do
19     if page(r) ∈ from-space then
20       if page(r) ∈ persistent-space then
21         promote(page(r), from-space, to-space);
22       elsif ¬copied(r) then
23           copy_address(r) := copy(r, to-space)
24       end;
25     end;
26   end;
27   copyStack.init(persistent-space ∪ to-space);          [Persistently-referenced objects must be retained]
28   copy_scan(mover');
29   foreach p ∈ from-space do
30     if p ∈ persistent-space then
31       if ∃l.l ∈ pointer_locations(p) where page(l↑) ∉ persistent-space then
32         promote(p, from-space, to-space);
33       else
34           flush(p);
35           free(p);
36       end;
37     else
38         free(p);
39     end;
40   end;
41   from-space := to-space.
```

Figure 9: Mostly-copying garbage collection over persistent pages

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.

6 Conclusions

We believe this to be the first presentation of algorithms for orthogonal persistence by reachability in uncooperative implementations of languages such as C and C++. The approach is being used in our implementation of persistence for Modula-3 based on the Digital Systems Research Center's reference compiler, which comes with a mostly-copying garbage collector for transient heap management. We have extended the SRC collector, which is also incremental and generational, to support

```
1  proc stabilizer'(l) ≡
2    r := l↑;
3    if page(r) ∉ persistent-space then
4      if page(r) ∈ to-space then
5        persistent-space := persistent-space + page(r);
6        copyStack.push(page(r));
7      else                                                    [page(r) ∈ from-space]
8          if ¬copied(r) then
9            copy_address(r) := copy(r, persistent-space);
10           copyStack.push(page(copy_address(r)));
11         end;
12         l↑:= copy_address(r);
13     end;
14   end.
15
16 proc stabilize'() ≡
17   to-space := {};
18   foreach r ∈ AR do promote(page(r), from-space, to-space) end;
19   foreach r ∈ PR do
20     if page(r) ∈ to-space then
21       persistent-space := persistent-space + page(r);
22     elsif page(r) ∈ persistent-space then
23         promote(page(r), from-space, to-space);
24     elsif ¬copied(r) then
25         copy_address(r) := copy(r, persistent-space);
26     end;
27   end;
28   copyStack.init(persistent-space);
29   copy_scan(stabilizer');
30   foreach p ∈ persistent-space do flush(p) end;
31   copyStack.init(to-space);
32   copy_scan(mover');
33   foreach p ∈ from-space do free(p) end;
34   from-space := to-space.
```

Figure 10: Mostly-copying stabilization, with collection over persistent pages

orthogonal persistence by reachability. We look forward to experimenting with our implementation in the very near future.[5] In particular, it will be interesting to determine just how much data is unnecessarily made persistent when entire ambiguously-referenced pages are promoted. While disk-oriented garbage collectors for persistent storage can reclaim such data, it is important that the burden placed on disk collection not be overly great, perhaps through application of techniques such as *black-listing* [Boehm 1993] to further winnow non-pointers from the ambiguous root set. Still, the performance effects of spurious retention of persistent data remain an open question for experimental study.

[5] We should have preliminary results by the time of the workshop

Acknowledgements

We are indebted to both Norman Ramsey and Paul Wilson for their comments on early versions of this paper.

References

AGRAWAL, R. AND GEHANI, N. H. 1989. 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), 36–45.

AGRAWAL, R. AND GEHANI, N. H. 1990. Rationale for the design of persistence and query processing facilities in the database language O++. See Hull et al. [1990], 25–40.

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

ATKINSON, M. P. AND MORRISON, R. 1995. Orthogonally persistent object systems. *Int. J. Very Large Data Bases 4*, 3, 319–401.

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

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

BOEHM, H.-J. 1993. 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), 197–206.

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

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

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

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

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

JONES, R. AND LINS, R. 1996. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley.

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

MOSS, J. E. B. 1990a. Addressing large distributed collections of persistent objects: The Mneme project's approach. See Hull et al. [1990], 269–285. Also available as COINS Technical Report 89-68, University of Massachusetts.

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

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

MOSS, J. E. B. AND SINOFSKY, S. 1988. Managing persistent data with Mneme: Designing a reliable, shared object interface. In *Proceedings of the International Workshop on Object Oriented Database Systems* (Bad Münster am Stein-Ebernburg, Germany, Sept.), K. R. Dittrich, Ed. Lecture Notes in Computer Science, vol. 334. Springer-Verlag, 298–316.

NELSON, G., Ed. 1991. *Systems Programming with Modula-3*. Prentice Hall.

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

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

SCHUH, D., CAREY, M., AND DEWITT, D. 1991. Persistence in E revisited—implementation experiences. In *Proceedings of the Fourth International Workshop on Persistent Object Systems* (Martha's Vineyard, Massachusetts, Sept. 1990), A. Dearle, G. M. Shaw, and S. B. Zdonik, Eds. Implementing Persistent Object Bases: Principles and Practice. Morgan Kaufmann, 345–359.

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

WHITE, S. J. AND DEWITT, D. J. 1994. QuickStore: A high performance mapped object store. In *Proceedings of the ACM International Conference on Management of Data* (Minneapolis, Minnesota, May). *ACM SIGMOD Record 23*, 2 (June), 395–406.

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

WILSON, P. R. 1992. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management* (St. Malo, France, Sept.), Y. Bekkers and J. Cohen, Eds. Number 637 in Lecture Notes in Computer Science. Springer-Verlag.

WILSON, P. R. AND KAKKAD, S. V. 1992. 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, 364–377.