PM3: An Orthogonally Persistent Systems Programming Language – Design, Implementation, Performance

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

PM3 is an orthogonally persistent extension of the Modula-3 systems programming language, supporting persistence by reachability from named persistent roots. We describe the design and implementation of the PM3 prototype, and show that its performance is competitive with its non-orthogonal counterparts by direct comparison with the SHORE/C++ language binding to the SHORE object store. Experimental results, using the traversal portions of the OO7 benchmark, reveal that the overheads of orthogonal persistence are not inherently more expensive than for non-orthogonal persistence, and justify our claim that orthogonal persistence deserves a level of acceptance similar to that now emerging for automatic memory management (i.e., "garbage collection"), even in performance-conscious settings. The consequence will be safer and more flexible persistent systems that do not compromise performance.

1 Introduction

PM3 is an extension of the Modula-3 systems programming language [Cardelli et al. 1991] that supports *orthogonal persistence* [Atkinson and Morrison 1995], which manifests itself as a model of persistence by reachability from designated persistent roots. Persistent storage is viewed as a transparent extension of the Modula-3 dynamic allocation heap; all heap-allocated data are potentially persistent. The merits of orthogonal persistence have been argued for many years, yet performance-conscious implementations of persistence have been lacking. Indeed, most implementations of orthogonally persistent programming languages have relied on an execution model that involves interpretation by a virtual machine, rather than compilation to native code. This trend continues today with Java.

In contrast, persistent extensions of systems programming languages have traditionally shunned orthogonal persistence as too expensive, or perhaps too difficult to implement. The primary reason for this is its implied reliance on garbage collection to effect *persistence by reachability*. Yet garbage collection is now gaining in acceptance, even in the systems programming realm. Evidence for this comes not just from the increased level of research activity related to garbage collection [ISMM 1998], but also from the commercial success in the C++ market of garbage collector vendors such as Chicago's Geodesic Systems. In this paper, we lay to rest the notion that orthogonal persistence is a luxury that a "real" systems programming language cannot afford.

We organize the remainder of the paper as follows. Section 2 more precisely defines what we mean by the term *orthogonal persistence* and outlines its advantages for programmers of large persistent applications, while also considering the performance problems it poses. Section 3 discusses related work in the area of persistent programming languages. In Section 4 we describe our design and implementation of PM3, followed in Section 5 by a description of our experimental framework for comparison of PM3 with SHORE/C++, our experimental results, and their detailed implications. Section 6 summarizes our conclusions and points towards future research directions.

2 Orthogonal persistence

Orthogonally persistent object systems [Atkinson and Morrison 1995] 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 will automatically be cached in volatile memory for manipulation by applications and updates propagated back

Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999

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 [1995] cite three design principles that are desirable in any persistent programming language design, enabling 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 types that can be applied to data of all lifetimes.

Finally, persistence designation gives every data item 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* [Jones 1996] by computing 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.

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 [Appel and Li 1991; Lamb et al. 1991; Singhal et al. 1992; Wilson and Kakkad 1992; White and DeWitt 1994], though the cost of fielding the resulting protection traps in some operating systems can be expensive [Hosking and Moss 1993]. In the absence of hardware-based solutions, or because of the performance shortcomings, barriers can be implemented in software. Typically, the language compiler or interpreter must arrange for appropriate checks to be performed explicitly 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* [Moss 1992] 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 by which they are referenced 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.[1]

3 Related work

The notion of orthogonal persistence has a long history [Atkinson and Buneman 1987], traced through the development of persistent programming languages such as PS-Algol [Atkinson et al. 1982; Atkinson et al. 1983; Atkinson et al. 1983] and Napier88 [Morrison et al. 1990; Dearle et al. 1990], and extensions to existing languages such as Smalltalk [Kaehler and Krasner 1983; Kaehler 1986; Straw et al. 1989; Hosking 1995] and Java [Atkinson et al. 1997; Atkinson et al. 1996]. It is important to note that all of these persistent languages rely on support for persistence from an underlying virtual machine, implemented as an abstract bytecode interpreter. While dynamic translation (i.e., "just-in-time" JIT compilation) can improve performance in these systems, neither performance nor features for systems programming were a design goal. On the other hand, abstraction of the execution engine as a virtual machine can more easily permit orthogonal persistence of active execution states (i.e., threads); certainly Napier88, Smalltalk and Tycoon [Matthes and Schmidt 1994] are noteworthy for this capability.

Performance-conscious persistent programming languages have historically almost exclusively been based upon C++, which at its outset was hostile to ideas of automatic storage management on the grounds that it compromised performance. Hence, most C++-based persistence extensions have adopted models of persistence that violate orthogonality in one or more dimensions. In E [Richardson and Carey 1987; 1990] and SHORE/C++ there is a distinction between database types and standard C++ types; only database types can persist. O++ [Agrawal and Gehani 1989; 1990] and Texas [Singhal et al. 1992; Wilson and Kakkad 1992], along with several commercial offerings [Lamb et al. 1991], 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 [Alagić 1997]. Until our own work [Hosking and Novianto 1997; Hosking and Chen 1999] 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 [Kakkad et al. 1998] to obtain accurate information on the location of references stored in the heap.

It is worth noting that 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 [Dearle et al. 1994; Rosenberg et al. 1996] and Mungi [Elphinstone et al. 1997; Heiser et al. 1998], but the rudiments are there in other experimental operating systems such as Opal [Chase et al. 1994; Chase et al. 1992], among others. Of course, our interest here focuses on efficient support for orthogonal persistence on stock operating systems.

4 PM3: Orthogonally persistent Modula-3

To serve as a platform for research into compiler support for orthogonally persistent programming languages we have designed and implemented an extension of the Modula-3 programming language [Cardelli et al. 1991] that supports orthogonal persistence.

Modula-3 is a modern, portable, systems programming language in the Algol family, whose other representatives include Pascal, Ada, Modula-2, and Oberon. Modula-3 also adopts selected features from the BCPL family of languages (C and C++ are the current specimens) in order to provide support for low-level systems programming, while retaining a strong type system that avoids dangerous and machine-dependent features. Modula-3 also supports threads (lightweight processes in a single address space), exception handling and information-hiding features such as objects, interfaces, opaque types and generics. Provision 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.

Modula-3 is *strongly-typed*: every expression has a unique type, and assignability and type compatibility are

[1][Richardson 1990; Hosking and Moss 1990; 1991; Moss and Hosking 1995; Hosking 1995; 1997; Hosking et al. 1999; Nystrom 1998; Nystrom et al. 1998; Brahnmath 1998; Brahnmath et al. 1999]

```
INTERFACE Transaction;
EXCEPTION
  TransactionInProgress;
  TransactionNotInProgress;
TYPE
  T <: Public;
  Public = OBJECT METHODS
    begin()
      RAISES { TransactionInProgress };
      (* Starts (opens) a transaction.
         Raises TransactionInProgress if
         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 1: The Transaction interface

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.

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.[2] 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.

4.1 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.

[2]Modula-3 also supports *untraced* references to storage allocated in a separate heap that is not subject to garbage collection; untraced storage must be deallocated explicitly.

```
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 2: The Database interface

Persistence functionality is introduced by way of the new library interfaces *Transaction* and *Database*; their essentials are presented in Figures 1 and 2. They are similar to their namesakes from the ODMG standard [Cattell et al. 1997], 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 [Haines et al. 1994].

4.2 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 a loosely-coupled front-end to the GNU C compiler, and 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 PM3 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. Since PM3 persistence designation is by reachability, stabilization of the persistent store on transaction commit is driven by the garbage collector, on which we have focused the bulk of our effort so far. We have extended the existing incremental, generational, mostly-copying garbage collector [Bartlett 1988; 1989] 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. In short, heap objects are clustered into heap pages, which are some small multiple of the virtual memory page size. On the SPARC heap pages are 8K bytes. These are the unit of transfer between volatile memory and stable storage, and the unit of management of persistent data in the volatile heap. Pages are also currently the unit of locking for concurrency control, but we plan also to investigate object-level locking along the lines of Carey et al. [1994]. Stabilization copies newly-persistent objects from the transient pages of the heap into persistent pages, which are then committed to the object store. See Hosking and Chen [1999] for the precise details of the stabilization algorithm.

4.2.1 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 in the heap will trap and trigger a page fault: the heap page is unprotected, the data is read into it from the corresponding mapped database page, all references within the heap page are discovered and swizzled, the access is resumed and execution proceeds. 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 virtual memory addresses [Singhal et al. 1992; Wilson and Kakkad 1992].

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. Cross-database references are also permitted.

4.2.2 Persistent storage

The current PM3 implementation uses the University of Wisconsin's SHORE object repository [Carey et al. 1994] 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 [Kiesel et al. 1995; Baumann 1997] transactional page server that permits nested transactions, and which is implemented purely in Modula-3.

4.2.3 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 for the fingerprints of all the objects in the database; each object is stored with the key of its type's fingerprint entry in this index. This approach also means that we can avoid storing object methods (i.e., code) in the persistent store. 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.

5 Experiments

We compare the performance of the traversal portions of our PM3 implementation of the OO7 benchmark [Carey et al. 1993] against the SHORE/C++ implementation of OO7 distributed with SHORE. The traversal portions of

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)	2000
Manual size (bytes)	100000
Total composite parts	500
Total atomic parts	10000

Table 1: Small OO7 database configuration

OO7 are numbered T1 through T9, though we do not present results for all of them here.

5.1 The OO7 benchmark

The OO7 benchmarks [Carey et al. 1993] 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 *documentation* 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 bi-directional 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.

5.2 Hardware

Our experiments were run under Solaris 2.5.1 on a 170MHz Sun SPARCstation 5, with 64M bytes RAM. The processor implementation is the Fujitsu TurboSPARC, with direct-mapped instruction and data caches of 16K bytes apiece. 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 relating to variations in page mappings from one process incarnation to the next. The local disk is a SUN0535 SCSI disk of 535M bytes.

Since we were uninterested in measuring network latencies both the SHORE server and the client were run on the same machine. This results in much improved client-server communication, with communication through shared memory where possible, and also more fully exposes the underlying overheads of the salient persistence mechanisms of interest to us.

5.3 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. For example, a SHORE object may be extended with a variable-sized heap, in which variable-sized components (e.g., strings, variable arrays, sets) of its value can be stored. The heap can also contain dynamic values that do not have independent identity; these may be linked together with *local references*, which are stored on disk as offsets from the start of the heap, but are swizzled in memory to actual memory addresses. SHORE also provides a variety of *bulk* types, including sets, lists and sequences.

The SHORE/C++ language binding allows methods for objects defined in the SHORE data language to be implemented in C++. An application, such as the SHORE/C++ implementation of the OO7 benchmark which we measure, is created as follows. First, one must write a description of the application's types in the SHORE data language (SDL), which the SDL compiler processes to create corresponding type objects as metadata in the SHORE repository. The SDL compiler also generates a set of C++ class declarations and special-purpose function definitions from the SDL types, in the form of a C++ header file. This header file is included in both the C++ source files that supply the implementation of the methods declared for each SDL type, and in source files that manipulate instances of those types. Some SDL types (e.g., integers) correspond directly to C++ types. Others, such as sets and object references (i.e., SHORE object identifiers) are represented in C++ using template classes (i.e., parameterized C++ types). C++ overloading features make SHORE object references appear to behave like ordinary C++ pointers, though with slower performance due to the software read and write barriers built into the overloaded operations.

Our PM3 implementation of OO7 is a direct transliteration of the SHORE/C++ implementation, but with the OO7 types implemented directly in Modula-3. Where the benchmark specifies the use of an index, we used a transparently persistent B+-tree coded in Modula-3. Moreover, 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 Modula-3 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++.

We took great care to match the SHORE/C++ implementation as closely as possible, including using the same C library random number generator and initializing it with the same seed so as to generate the same sequence of random numbers used to build the OO7 benchmark database and to drive the benchmark traversals.

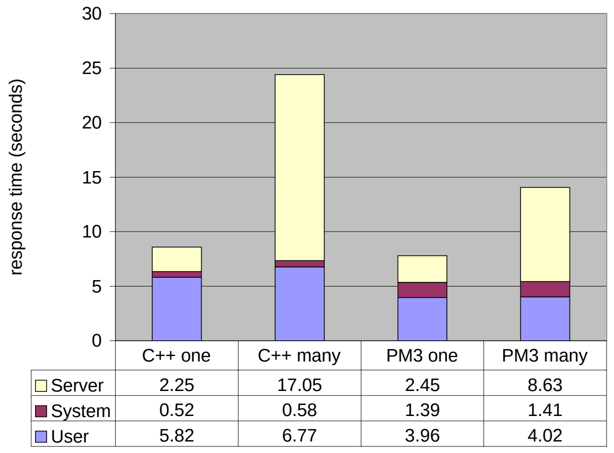
5.4 Results

The 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 [Carey et al. 1993] we obtain results for traversals running both "cold" and "hot". A cold traversal 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 traversal is then followed immediately by four successive iterations of the exact same query, with the results from the middle three taken as the hot measure. We ran the successive iterations in two modes: as a single transaction committing only after the last iteration (one), and as a sequence of chained transactions (many). The result for the last iteration is omitted 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 traversals. The reason for this is that PM3's incremental garbage collector induces random variable behavior from one hot iteration to the next, which would otherwise be obscured by averaging.

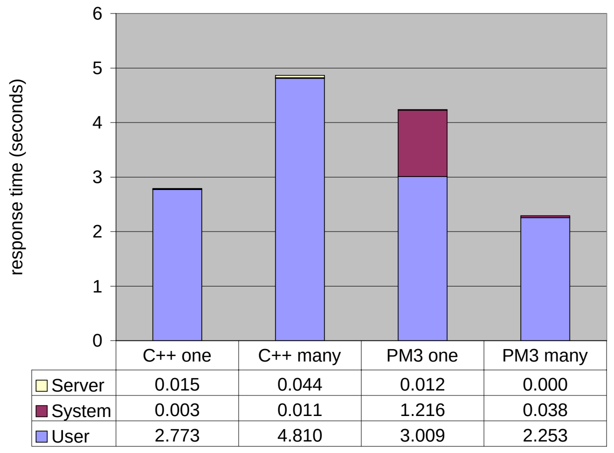
5.4.1 Traversal T1: Raw traversal speed

> *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.*



	C++ one	C++ many	PM3 one	PM3 many
🟨 Server	2.25	17.05	2.45	8.63
🟪 System	0.52	0.58	1.39	1.41
🟦 User	5.82	6.77	3.96	4.02

(a) Cold



	C++ one	C++ many	PM3 one	PM3 many
🟨 Server	0.015	0.044	0.012	0.000
🟪 System	0.003	0.011	1.216	0.038
🟦 User	2.773	4.810	3.009	2.253

(b) Hot: 3 iterations

Figure 3: Traversal T1

This is a test of raw pointer traversal speed. Figure 3(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 385 protection traps to fault 296 pages. The difference of 89 protection traps is due to the use of page protection primitives to implement barriers for PM3's incremental and generational garbage collector. Implementing barriers in PM3 with explicit checks instead would remove most of the system overhead for cold traversals, though they would add some to the user overhead; the compiler can attack this by optimizing away many checks if they are redundant [Cutts and Hosking 1997; Brahnmath 1998; Brahnmath et al. 1999].

SHORE/C++ fetches 41 594 objects into the client-side cache for a total of approximately 3M bytes, compared to PM3's 296 objects (the heap pages) for approximately 2.4M bytes. This demonstrates the compactness of PM3's object representation compared to SHORE/C++.

Despite T1 being a read-only traversal, SHORE still imposes overhead for commits, as revealed in the results which include commit processing (many). The server overhead is higher for SHORE/C++ than PM3 since the cold commit requires a separate client-server communication request for each object in the client-side cache (41 594 versus 296); hot chained commits do not pay this overhead. We assume an explanation as follows: on first chained commit the client must communicate the state (clean or dirty) of any objects it is caching 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).

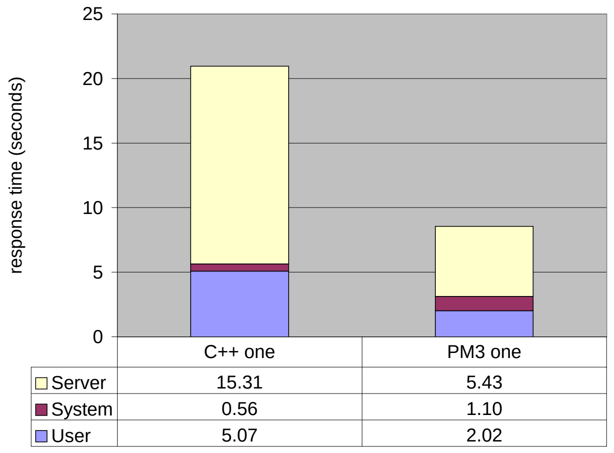
The hot T1 results appear in Figure 3(b). Here, the benefits of client caching are apparent for both SHORE/C++ and PM3. Run in a single transaction, sandwiched between the cold and last iteration, total elapsed time for

PM3 for the three hot iterations (PM3 one) is slower than for SHORE/C++ (C++ one). Indeed, there is noticeable system overhead to field protection traps (225 in fact) related to read barriers for the incremental garbage collector; each of these also results in some non-trivial user-charged garbage collector overhead, as well as contaminating the hardware caches and slowing down subsequent memory accesses. Inspection of the individual results for each of the three hot iterations reveals that response times for two of the three PM3 hot iterations are actually *faster* than the fastest SHORE/C++ hot iteration – 0.89s and 0.72s versus 0.93s, respectively – when PM3 is able to run with little or no garbage-collector overhead. Unfortunately, the last PM3 hot iteration includes a major garbage collection resulting in a response time of 2.6s. We could have turned off garbage collection for the experiments, but since the swizzling and faulting mechanisms are integrated with the garbage collector we felt it would be inappropriate to ignore its impact.

When the iterations are run as separate chaining transactions (many), the client caching is apparent for both SHORE/C++ and PM3 since they are able to cache all objects across the chaining commits into successive transactions. The hot commits impose negligible server-side overhead since the clients determine that no updates have occurred and restrict communication with the server only to signal the commit; nor are they subject to the communication overhead noted for cold commits. 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 from the previous chained commit, in which case it must communicate that fact to the server; there are simply more objects cached for SHORE/C++ than heap pages for PM3.

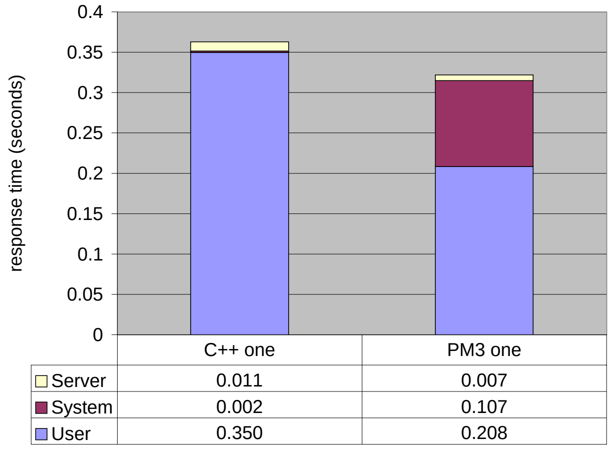
At first glance it might seem strange that the total elapsed time for the three PM3 many hot traversals, which include commits, is less than that for the PM3 one and SHORE/C++ hot traversals, which operate without commits. This is explained once again by considering garbage collection overhead. Since a heap stabilization involves a full heap garbage collection (to compute the reachability closure), commits leave the heap in a clean state for the next iteration so that it can proceed without additional traps and processing overhead due to incremental collection. With no updates occurring, no write protection traps are encountered, and the garbage collector can very quickly decide that heap stabilization is trivial; hence also is the commit. Indeed, none of the three PM3 many hot traversals is faster than the fastest PM3 one hot traversal.

As in the original OO7 paper we henceforth omit reporting results for read-only traversals run as a sequence of chained transactions, and report only the cold and hot times for read-only traversals run as a single transaction; the effect of client caching across transaction boundaries is duplicated in every operation of the benchmark.



	C++ one	PM3 one
□ Server	15.31	5.43
■ System	0.56	1.10
■ User	5.07	2.02

(a) Cold



	C++ one	PM3 one
□ Server	0.011	0.007
■ System	0.002	0.107
■ User	0.350	0.208

(b) Hot: 3 iterations

Figure 4: Traversal T6

5.4.2 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.

Carey et al. [1993] intended this 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; one expects full swizzling to be penalized for expending swizzling effort to little or no benefit.[3] However, our elapsed time results do not tell the expected story. For the cold T6 traversal (Figure 4(a)) PM3 appears to pay little user-level swizzling penalty, though the system overhead to field the read barrier traps remains.

[3]One might suspect this to be the reason for ODI's withdrawal from the original OO7 study, since their faulting and swizzling strategy is similar to ours.

The truth of the matter turns out to be related to clustering. SHORE/C++ fetches 41 346 objects (3M bytes) versus PM3's 158 heap pages (1.2M bytes). 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 its promotion into persistent pages of objects discovered to be persistent during stabilization, via what amounts to breadth-first search [Cheney 1970], yields much better clustering [Schkolnick 1977]. Only an orthogonally persistent system has sufficient flexibility to place objects by reachability, instead of at allocation time, since placement is decoupled from allocation and deferred instead until commit time when the heap is stabilized via reachability.

The hot results (Figure 4(b)) again reveal the superiority of full swizzling for hot operations, with PM3 markedly outperforming SHORE/C++ on the user component. In this case, SHORE/C++ suffers from the overhead of having to issue 5468 paired pin/unpin operations for each access to an object in the cache; PM3 accesses incur no such overhead. Overall, PM3 only just edges out SHORE/C++ because of incremental garbage collection overheads, as revealed by the system component.

5.4.3 Traversal T2: Updates

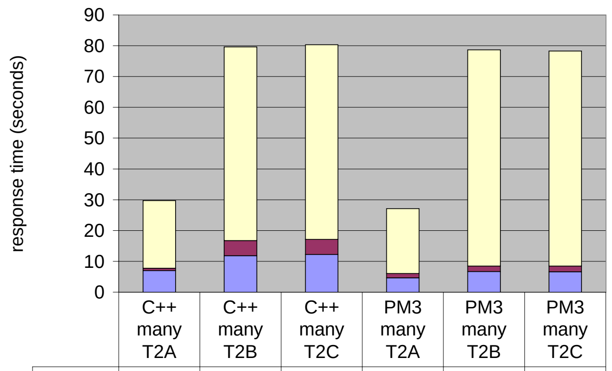
Repeat traversal T1, but update objects during the traversal. There are three types of update patterns in this traversal. In each, a single update to an atomic part consists of swapping its (x, y) attributes. The three types of updates are:

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.

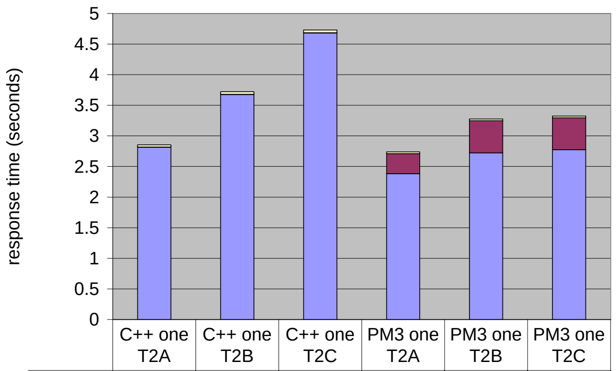
Since these are update traversals the cold traversals with commit are more interesting than without, as presented in Figure 5(a). Again, despite the overhead of trap-driven read barriers, PM3 exhibits superior cold performance. Both SHORE/C++ and PM3 display higher server overhead for the dense update T2B and T2C traversals than the sparse update T2A. Despite the compactness of the PM3 object representation it incurs slightly higher server overhead for the dense updates because of the need to consult a SHORE index for each updated page to map its PM3 page identifier to its corresponding SHORE identifier.

Figure 5(b) presents results for the hot T2 traversals without commits (one), showing the raw overhead to update the objects. The trap-based write barrier poses significant overhead to PM3 for the sparse update T2A traversal, with PM3 just edging out SHORE/C++. 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 PM3 is a definite winner since it pays the same trap overhead as for T2B, while SHORE/C++ incurs overhead on every update, even if to a part that has already been updated.



	C++ many T2A	C++ many T2B	C++ many T2C	PM3 many T2A	PM3 many T2B	PM3 many T2C
🟨 Server	21.94	62.89	63.20	21.08	70.17	69.86
🟪 System	0.80	4.92	4.94	1.41	1.81	1.84
🟦 User	7.00	11.81	12.19	4.64	6.68	6.61

(a) Cold



	C++ one T2A	C++ one T2B	C++ one T2C	PM3 one T2A	PM3 one T2B	PM3 one T2C
🟨 Server	0.035	0.042	0.045	0.029	0.028	0.030
🟪 System	0.003	0.003	0.003	0.328	0.526	0.523
🟦 User	2.816	3.676	4.683	2.382	2.722	2.773

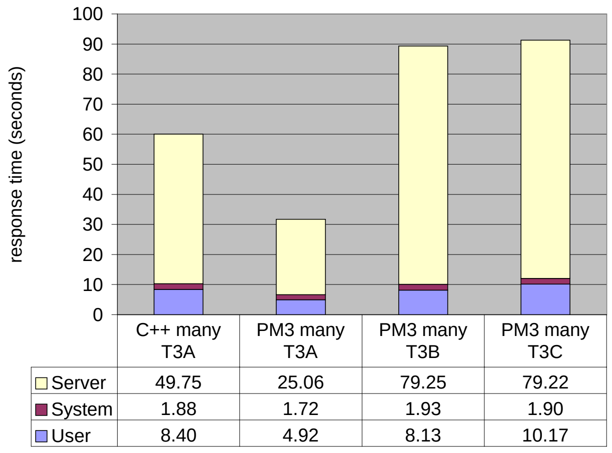
(b) Hot: 3 iterations

Figure 5: Traversal T2

5.4.4 Traversal T3: Indexed field updates

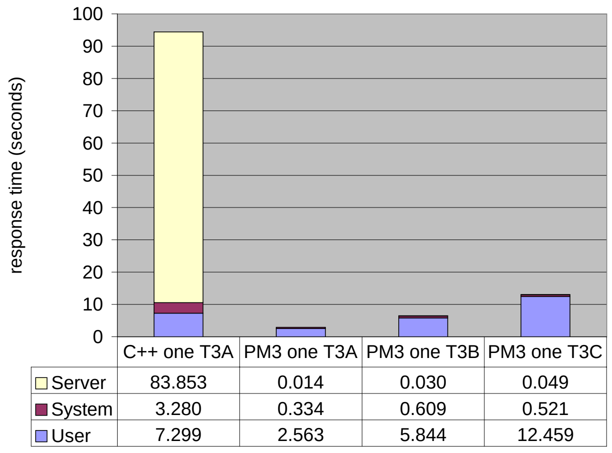
Repeat traversal T2, except that now the update is on the date field, which is indexed. The specific update is to increment the date if it is odd, and decrement the date if it is even.

Figure 6 gives results for T3. It turns out SHORE/C++ uses indexes that are centralized on the server so every indexed update requires interaction with the server, at very high cost. In fact, the overhead is so high that we were only able to run SHORE/C++ for the sparse T3A; for our configuration of SHORE the dense indexed updates result in the log overflowing and the transaction aborting. In contrast, our indexes for Modula-3 are implemented natively as orthogonally persistent B+-trees so their pages can be cached and updated at the client. PM3 wins on all 3 indexed traversals. Keeping the index at the server may permit more concurrency, so perhaps the comparison in this instance is not entirely fair. Nevertheless, the difference in performance is staggering.



	C++ many T3A	PM3 many T3A	PM3 many T3B	PM3 many T3C
Server	49.75	25.06	79.25	79.22
System	1.88	1.72	1.93	1.90
User	8.40	4.92	8.13	10.17

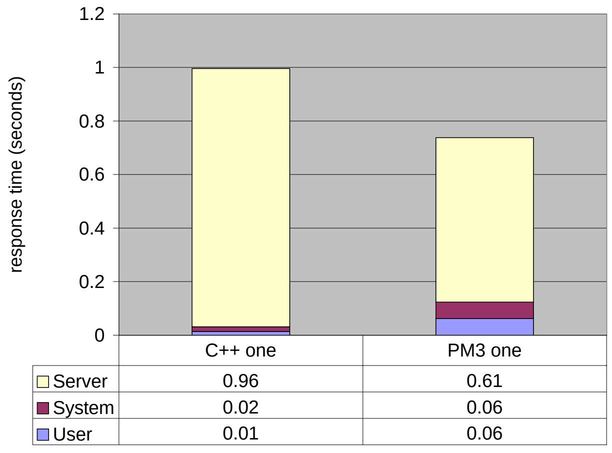
(a) Cold



	C++ one T3A	PM3 one T3A	PM3 one T3B	PM3 one T3C
Server	83.853	0.014	0.030	0.049
System	3.280	0.334	0.609	0.521
User	7.299	2.563	5.844	12.459

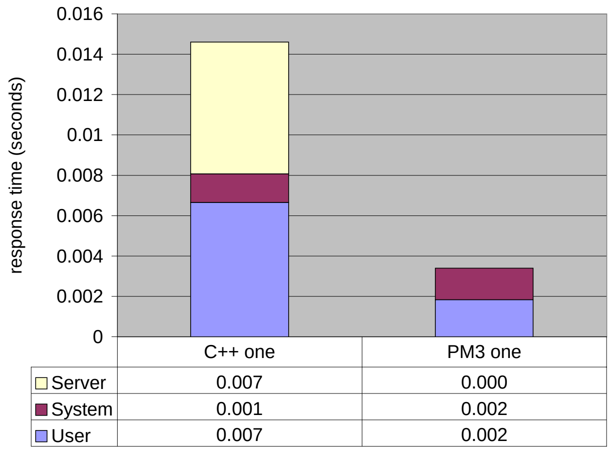
(b) Hot: 3 iterations

Figure 6: Traversal T3



	C++ one	PM3 one
Server	0.96	0.61
System	0.02	0.06
User	0.01	0.06

(a) Cold



	C++ one	PM3 one
Server	0.007	0.000
System	0.001	0.002
User	0.007	0.002

(b) Hot: 3 iterations

Figure 7: Traversal T9

5.4.5 Traversal T9: Operations on manual

Traversal T9 checks the manual object to see if the first and last character in the manual object are the same.

The results for the read-only T9 traversal presented in Figure 7 are for traversals without commits. The cold query results are somewhat inconclusive, mostly because the query accesses so little data in the small OO7 database (there is only one manual) as to be subject to spurious variations in system behavior. For example, the cold SHORE/C++ query incurs 17 virtual memory page faults requiring physical I/O to PM3's one, which accounts for a significant fraction of the server component for SHORE/C++ in Figure 7(a). The client CPU overheads seem more trustworthy, reflecting the high cost of PM3's trap-based read barrier and full swizzling when accessing so little data. For the hot iterations (Figure 7(b)), the high server component for SHORE/C++ results from its receiving a message from the server on each iteration. The cause of this anomaly can only be related to the difference in the underlying representation of the manual object. In PM3, a manual is simply a "large" heap object stored as a sequence of not-necessarily contiguous pages, although these pages are retrieved and mapped contiguously into the PM3 heap. For SHORE/C++ the manual is a large SHORE object, stored contiguously.

5.4.6 Traversals omitted

We have omitted several traversals in addition to those omitted in the original OO7 study Carey et al. [1993], notably traversals T8 (an operation on the manual) and CU (cached update). Unfortunately, we were unable to get the SHORE/C++ T8 traversal to run without crashing, and so could not obtain a comparison. In the case of CU, its goals are amply covered by the results for the other traversals as we have presented them here; nor does it contradict them. In all other cases, while the results for both SHORE/C++ and PM3 are available they do not provide further insights.

6 Conclusions and future work

We have demonstrated through implementation and experimentation that PM3, an orthogonally persistent systems programming language, can provide performance that is highly competitive with its non-orthogonal peers. Thus, the superior software engineering support that orthogonal persistence provides should not be withheld simply on the basis of prejudice against its reachability-based approach. In fact, there is no technical reason why more accepted systems programming languages such as C++ cannot be retrofitted with orthogonal persistence, as opposed to the non-orthogonal realizations of persistence currently imposed on them.

Our future work with PM3 will address the one remaining thorny issue in our results – the overhead of trap-based barrier implementations – by introducing explicit software barriers and using the compiler to optimize away any that are redundant. We also plan to explore the integration of buffer management with volatile heap management, disk garbage collection and extended transaction support.

Acknowledgments

This research is supported in part by the National Science Foundation under Grant No. CCR-9711673 and by gifts from Sun Microsystems, Inc. We thank Tony Printezis and the anonymous referees, whose comments spurred several improvements to this presentation.

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.

ALAGIĆ, S. 1997. 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.), 253–270.

APPEL, A. W. AND LI, K. 1991. Virtual memory primitives for user programs. In Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (Santa Clara, California, Apr.). *ACM SIGPLAN Notices 26,* 4 (Apr.), 96–107.

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

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

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

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

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

ATKINSON, M. P., JORDAN, M. J., DAYNÈS, L., AND SPENCE, S. 1997. Design issues for persistent Java: A type-safe object-oriented, orthogonally persistent system. See Connor and Nettles [1997], 33–47.

ATKINSON, M. P. AND MORRISON, R. 1995. Orthogonally persistent object systems. *International Journal on 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.

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

BRAHNMATH, K., NYSTROM, N., HOSKING, A. L., AND CUTTS, Q. 1999. Swizzle barrier optimizations for orthogonal persistence in Java. In *Proceedings of the Third International Workshop on Persistence and Java* (Tiburon, California, August 1998), R. Morrison, M. Jordan, and M. Atkinson, Eds. Advances in Persistent Object Systems. Morgan Kaufmann, 268–278.

BRAHNMATH, K. J. 1998. Optimizing orthogonal persistence for Java. M.S. thesis, Purdue University.

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

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. 1994. Shoring up persistent applications. See SIGMOD [1994], 383–394.

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

CAREY, M. J., FRANKLIN, M. J., AND ZAHARIOUDAKIS, M. 1994. Fine-grained sharing in a page server OODBMS. See SIGMOD [1994], 359–370.

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

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

CHASE, J. S., LEVY, H. M., LAZOWSKA, E. D., AND BAKER-HARVEY, M. 1992. 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.), 397–413.

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

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

CUTTS, Q. AND HOSKING, A. L. 1997. Analysing, profiling and optimising orthogonal persistence for Java. In *Proceedings of the Second International Workshop on Persistence and Java* (Half Moon Bay, California, Aug.), M. P. Atkinson and M. J. Jordan, Eds. Sun Microsystems Laboratories Technical Report 97-63, 107–115.

DEARLE, A., CONNER, R., BROWN, F., AND MORRISON, R. 1990. Napier88—A database programming language? See Hull et al. [1990], 179–195.

Dearle, A., di Bona, R., Farrow, J., Henskens, F., Lindström, A., Rosenberg, J., and Vaughan, F. 1994. Grasshopper: An orthogonally persistent operating system. *Computer Systems 7,* 3 (Summer), 289–312.

Dearle, A., Shaw, G. M., and Zdonik, S. B., Eds. 1990. *Proceedings of the Fourth International Workshop on Persistent Object Systems* (Martha's Vineyard, Massachusetts, Sept.). Implementing Persistent Object Bases: Principles and Practice. Morgan Kaufmann, 1991.

Elphinstone, K., Russell, S., Heiser, G., and Liedtke, J. 1997. Supporting persistent object systems in a single address space. See Connor and Nettles [1997], 111–119.

Haines, N., Kindred, D., Morrisett, J. G., Nettles, S. M., and Wing, J. M. 1994. Composing first-class transactions. *ACM Trans. Program. Lang. Syst. 16,* 6 (Nov.), 1719–1736.

Heiser, G., Elphinstone, K., Vochteloo, J., Russell, S., and Liedtke, J. 1998. The Mungi single-address-space operating system. *Software: Practice and Experience 28,* 9 (July), 901–928.

Hosking, A. L. 1995. Lightweight support for fine-grained persistence on stock hardware. Ph.D. thesis, University of Massachusetts at Amherst. Available as Computer Science Technical Report 95-02.

Hosking, A. L. 1997. Residency check elimination for object-oriented persistent languages. See Connor and Nettles [1997], 174–183.

Hosking, A. L. and Chen, J. 1999. Mostly-copying reachability-based orthogonal persistence. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (Denver, Colorado, Nov.).

Hosking, A. L. and Moss, J. E. B. 1990. Towards compile-time optimisations for persistence. See Dearle et al. [1990], 17–27.

Hosking, A. L. and Moss, J. E. B. 1991. Compiler support for persistent programming. Tech. Rep. 91-25, Department of Computer Science, University of Massachusetts at Amherst. Mar.

Hosking, A. L. and Moss, J. E. B. 1993. 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.), 106–119.

Hosking, A. L. and Novianto, A. P. 1997. 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.). http://www.dcs.gla.ac.uk/~huw/oopsla97/gc/papers.html.

Hosking, A. L., Nystrom, N., Cutts, Q., and Brahnmath, K. 1999. 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, 149–159.

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.

ISMM 1998. *Proceedings of the ACM International Symposium on Memory Management* (Vancouver, Canada, Oct.). ACM.

Jones, R. 1996. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley. With a chapter by R. Lins.

Kaehler, T. 1986. 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.), 87–106.

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

Kakkad, S. V., Johnstone, M. S., and Wilson, P. R. 1998. Portable run-time type description for conventional compilers. See ISMM [1998], 146–153.

Kiesel, N., Schürr, A., and Westfechtel, B. 1995. GRAS, a graph-oriented (software) engineering database system. *Information Systems 20,* 1, 21–52.

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

Matthes, F. and Schmidt, J. W. 1994. Persistent threads. In *Proceedings of the International Conference on Very Large Data Bases* (Santiago, Chile, Sept.). Morgan Kaufmann, 403–414.

Morrison, R., Brown, A., Carrick, R., Connor, R., Dearle, A., and Atkinson, M. P. 1990. 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, 3–18.

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 Hosking, A. L. 1995. 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, 3–15.

Nystrom, N., Hosking, A. L., Whitlock, D., Cutts, Q., and Diwan, A. 1998. Partial redundancy elimination for access path expressions. Tech. Rep. 98-044, Department of Computer Sciences, Purdue University. Oct. Submitted for publication.

Nystrom, N. J. 1998. Bytecode level analysis and optimization of Java classes. M.S. thesis, Purdue University.

Richardson, J. E. 1990. Compiled item faulting: A new technique for managing I/O in a persistent language. See Dearle et al. [1990], 3–16.

Richardson, J. E. and Carey, M. J. 1987. 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.), 208–219.

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.

Rosenberg, J., Dearle, A., Hulse, D., Lindström, A., and Norris, S. 1996. Operating system support for persistent and recoverable computations. *Commun. ACM 39,* 9 (Sept.), 62–69.

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

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

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.

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

White, S. J. and DeWitt, D. J. 1994. QuickStore: A high performance mapped object store. See SIGMOD [1994], 395–406.

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.