Optimizing the Read and Write Barriers for Orthogonal Persistence

Antony Hosking[1] **Nathaniel Nystrom**[1] **Quintin Cutts**[2] **Kumar Brahnmath**[1]

[1]Department of Computer Sciences
Purdue University
West Lafayette, IN 47907-1398, USA
{hosking,nystrom,brahnmat}@cs.purdue.edu

[2]Department of Computing Science
University of Glasgow
Glasgow G12 8QQ, Scotland
quintin@dcs.gla.ac.uk

Abstract

Persistent programming languages manage volatile memory as a cache for stable storage, imposing a *read barrier* on operations that access the cache, and a *write barrier* on updates to the cache. The read barrier checks the cache residency of the target object while the write barrier marks the target as dirty in the cache to support a write-back policy that defers updates to stable storage until eviction or stabilization. These barriers may also subsume additional functionality, such as negotiation of locks on shared objects to support concurrency control. Compilers for persistent programming languages generate barrier code to protect all accesses to possibly persistent objects. Orthogonal persistence imposes this cost on every object access, since all objects are potentially persistent, at significant overhead to execution. We have designed a new suite of compiler optimizations, focusing on partial redundancy elimination of pointer-based access expressions, that significantly reduce this impact. These are implemented in an analysis and optimization framework for Java bytecodes, in support of orthogonal persistence for Java. In experiments with the traversal portions of the OO7 benchmark suite our optimizations reduce the number of read and write barriers executed by an average of 83% and 25%, respectively.

1 Introduction

A *persistent system* [Atkinson and Morrison 1995] treats permanent storage as a stable extension of volatile memory, in which objects may be dynamically allocated, but which persists from one program invocation to the next. A persistent programming language and object store together preserve *object identity*: every object has a unique 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.

The language principles of *transparency* and *orthogonality* have been repeatedly articulated [Atkinson and Morrison 1995; Moss and Hosking 1996] as important in the design of persistent programming languages, enabling the full power of the persistence abstraction. Transparency means that from the programmer's perspective access to persistent objects does not require writing explicit code to transfer them between stable store and main memory. Thus, a program that manipulates persistent (or potentially persistent) objects looks similar to a program concerned only with transient objects. Instead, the language's compiler and/or run-time system contrive automatically to cache persistent objects in volatile memory on demand for manipulation by the program. This is somewhat reminiscent of virtual memory: cache misses in a persistent system are called *object faults* and trigger retrieval of the missing object from stable storage into volatile memory.

Treating persistence as *orthogonal* to type encourages the view that a language can be extended to support persistence with minimal disturbance of its existing syntax and store semantics. Thus, programmers need add little to their understanding of the language in order to begin writing persistent programs. A common way to achieve orthogonal persistence is by treating persistent storage as a stable extension of the dynamic allocation heap. This allows a uniform and transparent treatment of both transient and persistent data; persistence is orthogonal to the way in which objects are defined (i.e., their types), allocated, and manipulated in the heap.

While there are a number of techniques for object faulting based on hardware support for memory mapping that are transparent to the compiler [Lamb

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

et al. 1991; Singhal et al. 1992; Wilson and Kakkad 1992; White and DeWitt 1994], the restrictions and performance degradation that such approaches impose are often unacceptable, resulting in a lack of control over explicit buffer management, location independence and true object identity [Kemper and Kossman 1995]. In the absence of such hardware support for object faulting, compilers for persistent programming languages must generate explicit code before each operation that may access a persistent object to check that it is resident in memory, and to fault it in if not. Similarly, to support efficient migration of updates back to stable storage, compilers must generate code along with every operation that updates a persistent object to signal that it eventually must be copied back to stable storage, either when replaced in the cache or during stabilization of the persistent store. These checks are generically termed the persistence *read barrier* and *write barrier*, respectively. In general they can subsume additional functionality, such as negotiation of locks on shared objects to control for concurrent access. As such, read and write barriers represent significant overhead to the execution of any persistent program.

The performance penalty is exacerbated for languages that provide orthogonal persistence, since they unify 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. Optimizations to remove redundant barriers have been postulated in the past but have never been fully specified and evaluated [Richardson 1990; Hosking and Moss 1990; 1991; Moss and Hosking 1995; Hosking 1995; 1997].

This paper presents a complete framework for such optimizations based on partial redundancy elimination over pointer expressions, describes its implementation for orthogonal persistence in Java, and provides experimental evidence of its effectiveness for the elimination of redundant read and write barriers.

2 Analysis and optimization

Our analysis and optimization framework revolves around partial redundancy elimination over pointer expressions that access persistent objects. We adopt standard terminology and notations used in the specification of the Java programming language to specify the analysis and optimization problem.

2.1 Terminology and notation

The following definitions paraphrase the Java specification [Gosling et al. 1996]. An *object* in Java is either a *class instance* or an array. Reference values in Java are either *pointers* to these objects or the null reference. Both objects and arrays are created by expressions that allocate and initialize storage for them. The operators on references to objects are field access, method invocation, casts, type comparison (`instanceof`), equality operators and the conditional operator. There may be many references to the same object. Objects have mutable state, stored in the variable fields of class instances or the variable elements of arrays. Two variables may refer to the same object: the state of the object can be modified through the reference stored in one variable and then the altered state observed through the other. *Access expressions* refer to the variables that comprise an object's state. A *field access expression* refers to a field of some class instance, while an *array access expression* refers to a component of an array. Table 1 summarizes the two kinds of access expressions in Java. We

Table 1: Access expressions

Notation	Name	Variable accessed
p.`f`	Field access	Field `f` of class instance referred to by p
$p[i]$	Array access	Component with subscript i of array referred to by p

adopt the term *access path* [Larus and Hilfinger 1988; Diwan et al. 1998] to mean a non-empty sequence of accesses, as specified by some access expression in the source program. For example, the Java access expression $a.b[i].c$ is an access path. Also, without loss of generality, our notation will assume that distinct fields within an object have different names.

A variable is a storage location and has an associated type, sometimes called its *compile-time* type. Given an access path p, then the compile-time type of p, written $Type(p)$, is simply the compile-time type of the variable it accesses. A variable always contains a value that is *assignment compatible* with its type. A value of compile-time class type S is assignment com-

patible with class type T if S and T are the same class or S is a subclass of T. A similar rule holds for array variables: a value of compile-time array type $S[]$ is assignment compatible with array type $T[]$ if type S is assignable to type T. Interface types also yield rules on assignability: an interface type S is assignable to an interface type T only if T is the same interface as S or a superinterface of S; a class type S is assignable to an interface type T if S implements T. Finally, array types, interface types and class types are all assignable to class type `Object`.

For our purposes we say that a type S is a *subtype* of a type T if S is assignable to T.[1] We write $\mathit{Subtypes}(T)$ to denote all subtypes of type T; i.e., $\mathit{Subtypes}(T)$ is the set of all types assignment compatible with T. Thus, an access path p can legally access objects of type $\mathit{Subtypes}(\mathit{Type}(p))$. Alias analysis refines the type of variables to which an access path may refer. If two distinct access paths refer to variables of the same type then they may be aliases for the same variable.

2.2 Read and write barriers

In an orthogonally persistent implementation of Java access expressions may refer to both persistent and transient objects. Thus, every field or array access must be protected by an appropriate barrier applied to the class instance or array being accessed. For example, in the absence of optimizations, the access path $a.b[i].c$ would require read barriers on the class instance referred to by a, the array referred to by b and the object referred to by the ith component of b. If the expression appears as the target of an assignment, then the object referred to by $a.b[i]$ would also require a write barrier.

Our goal is to avoid applying barriers to accesses where program analysis shows that the barrier is redundant. To do so, we must make them explicit in the access paths and then apply some definition of redundancy. Making barriers explicit means obtaining for the source code access expression an intermediate representation (IR) in which the barriers are exposed. Optimizations then operate on the IR to remove redundant barriers. Thus, we add barrier expressions to the original specification of access expressions given in Table 1. The specification for barrier expressions appears in Table 2. For each source code access expression Table 3 gives the form of the corresponding explicit-barrier IR.

[1] The term "subtype" is not used at all in the official Java language specification [Gosling et al. 1996], presumably to avoid confusing the type hierarchy induced by the subtype relation with class and interface hierarchies.

Table 2: Barrier expressions

Notation	Name	Description
$\mathit{read}(p)$	Read barrier	Apply read barrier to, and return, object referred to by p
$\mathit{write}(p)$	Write barrier	Apply write barrier to, and return, object referred to by p

Table 3: IR for access expressions

Source	Intermediate representation	
	Read access	Write access
p.`f`	$\mathit{read}(p)$.`f`	$\mathit{write}(\mathit{read}(p))$.`f`
$p[i]$	$\mathit{read}(p)[i]$	$\mathit{write}(\mathit{read}(p))[i]$

A barrier is redundant if we can guarantee that an earlier barrier of the same kind has already been applied to the same object, and that the earlier barrier's side-effect (e.g., to fault or dirty the object) has not been undone (i.e., the barrier is *idempotent* and *enduring*). This has implications for the interaction of barrier optimizations with the persistence run-time system, which must not undo the effect of a barrier while optimized code downstream of the barrier can still execute. Solving this problem requires a contract between the optimizer and the run-time system for each kind of barrier. The contract will depend on the specifics of the implementation so we defer discussion of this issue to Section 3, which presents our implementation for Java. A separate paper considers the issue from the perspective of the run-time system [Cutts et al. 1998].

Given two barrier expressions $\mathit{read}(p)$ and $\mathit{read}(q)$, if we can guarantee that p and q refer to the same object and that $\mathit{read}(p)$ dominates $\mathit{read}(q)$ then $\mathit{read}(q)$ is redundant and can be replaced simply by q. The crucial test here is that two access paths refer to the same object. This amounts to detection of common access expressions, and the optimization can be framed much like classical techniques for common subexpression

elimination. In the simplest case, two lexically identical access paths in the same scope must refer to the same object, so long as no component of the path has been modified between the first occurrence of the expression and the second. Unfortunately, the possibility of aliases means that an intervening assignment might change some component of the path through a lexically distinct access path. Showing that intervening assignments do not modify a given access path requires *alias analysis.*

2.3 Type-based alias analysis

Type-based alias analysis (TBAA) [Diwan et al. 1998] assumes a type-safe programming language such as Java, since it uses type declarations to disambiguate references. The compile-time type of an access path provides a simple way to do this: two access paths p and q may be aliases only if the relation $\mathit{TypeDecl}(p,q)$ holds, defined as:

$$\mathit{TypeDecl}(p,q) \equiv \\ \mathit{Subtypes}(\mathit{Type}(p)) \cap \mathit{Subtypes}(\mathit{Type}(q)) \neq \emptyset$$

A more precise alias analysis will distinguish accesses to fields that are the same type yet distinct. This more precise relation, $\mathit{FieldTypeDecl}(p,q)$, is defined by induction on the structure of p and q in Table 4. Again,

Table 4: $\mathit{FieldTypeDecl}(\mathcal{AP}_1, \mathcal{AP}_2)$

Case	$\mathcal{AP}_1$	$\mathcal{AP}_2$	$\mathit{FieldTypeDecl}(\mathcal{AP}_1, \mathcal{AP}_2)$
1	p	p	true
2	$p.\mathtt{f}$	$q.\mathtt{g}$	$(f = g) \wedge \mathit{FieldTypeDecl}(p,q)$
3	$p.\mathtt{f}$	$q[i]$	false
4	$p[i]$	$q[j]$	$\mathit{FieldTypeDecl}(p,q)$
5	p	q	$\mathit{TypeDecl}(p,q)$

two access paths p and q may be aliases only if the relation $\mathit{FieldTypeDecl}(p,q)$ holds. It distinguishes accesses such as $t.\mathtt{f}$ and $t.\mathtt{g}$ that *TypeDecl* misses. The cases in Table 4 determine that:

1. Identical access paths are always aliases
2. Two field accesses may be aliases if they access the same field of potentially the same object
3. Array accesses cannot alias field accesses and vice versa
4. Two array accesses are aliases if they may access the same array (the subscript is ignored)
5. All other pairs of access expressions are aliases if they have common subtypes

Diwan et al. [1998] further refine type-based alias analysis by enumerating all the assignments in a program to determine more accurately the types of objects an access path may reference: two variables may alias an object of a given type only if there are assignments of that type to both variables. This refines the *TypeDecl* relation, which merges the declared type of a variable with all of its subtypes, to only merge a type T with a subtype S if there actually exists an assignment of S to T in the program. Unfortunately, this requires having the complete program available for analysis at the time of optimization. In general, Java's use of dynamic loading, not to mention the possibility of native methods hiding assignments from the analysis, precludes a closed world analysis. Still, it may be possible to approximate closed world analysis in a persistent system that stores all classes pertaining to persistent data. Our plans for exploring this have been described elsewhere [Cutts and Hosking 1997].

2.4 Partial redundancy elimination

Our approach to barrier optimization is based on application of *partial redundancy elimination* (PRE) [Morel and Renvoise 1979] to access expressions. To our knowledge this is the first time PRE has been applied to access paths. PRE is a powerful global optimization technique that subsumes the more standard common subexpression elimination (CSE). PRE eliminates computations that are only partially redundant; that is, redundant only on some, but not all, paths to some later re-computation. By inserting evaluations on those paths where the computation does not occur, the later re-evaluation can be eliminated and replaced instead with a use of the precomputed value. This is illustrated in Figure 1. In Figure 1a, both a and b are available along both paths to the merge point, where expression $a+b$ is evaluated. However, this evaluation is partially redundant since $a+b$ is available on one path to the merge but not both. By hoisting the second evaluation of $a+b$ into the path where it was not originally available, as in Figure 1b, $a+b$ need only be evaluated once along any path through the program, rather than twice as before.

Traversing an access path requires successively loading the pointer at each memory location along the path and dereferencing it to the next location in the se-

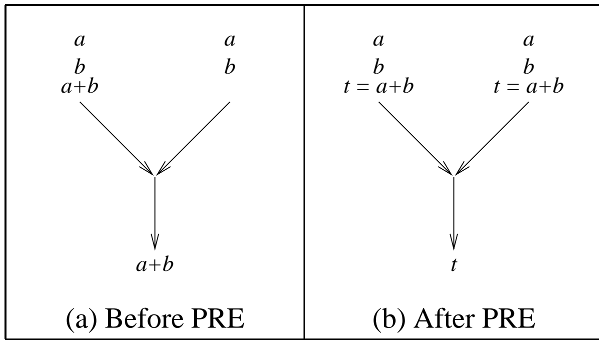

Figure 1: PRE for arithmetic expressions

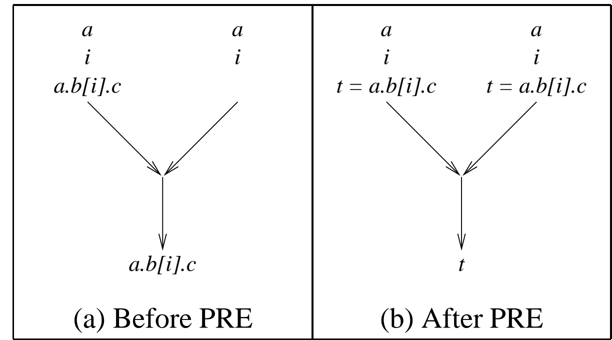

Figure 2: PRE for access expressions

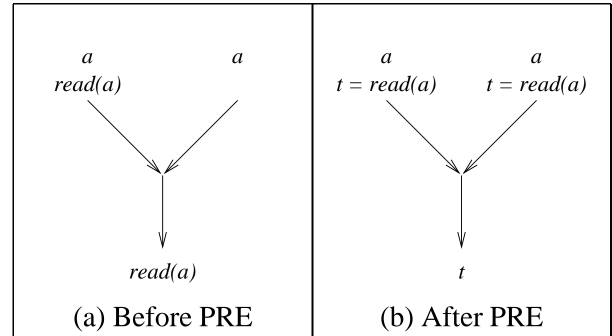

Figure 3: PRE for barrier expressions

quence. Before applying PRE to access path expressions, one must first disambiguate memory references sufficiently to be able safely to assume that no memory location along the access path can be aliased (and so modified) by some other distinct access path in the program. Consider the example in Figure 2. The expression $a.b[i].c$ will be redundant at some subsequent re-evaluation so long as no store occurs to any one of a, $a.b$, i, $a.b[i]$ or $a.b[i].c$ occurs on the code path between the first evaluation of the expression and the second. In other words, if there are potential aliases to any one of a, i, $a.b$, $a.b[i]$ or $a.b[i].c$ through which those locations *may* be modified between the first and second evaluation of the expression, then that second evaluation cannot be treated as redundant.[2] By exposing read and write barriers in the intermediate representation for access expressions partial redundancy elimination will optimize them in the same way as other expressions (Figure 3).

2.5 Java constraints on optimization

Java's thread and exception models impose several constraints on optimization. Exceptions in Java are *precise*: when an exception is thrown all effects of statements prior to the throw-point must appear to have taken place, while the effects of statements after the throw-point must not. This imposes a significant constraint on code motion optimizations such as PRE, since code with side-effects cannot be moved relative to code that may throw an exception. The thread model prevents movement of access expressions across (possible) synchronization points. Without inter-procedural control-flow analysis this must include all method invocation sites, since the callee, or a method invoked inside the callee, may be synchronized. Fortunately, read and write barriers for orthogonal persistence do not have side-effects that are relevant to source-level program semantics, so their motion is unconstrained.

[2]Note that if a and i are local variables then they cannot be aliased.

3 Implementation

Our implementation uses bytecode-to-bytecode class transformation to apply type-based alias analysis and access path PRE to Java classes for execution on a modified version of the PJama [Atkinson et al. 1996; Daynès and Atkinson 1997] virtual machine.

3.1 Bytecode-level class transformation

The Java virtual machine (VM) specification [Lindholm and Yellin 1996] is intended as the interface between Java compilers and Java execution environments. Its standard class format and instruction set permit multiple compilers to inter-operate with multiple VM implementations, enabling the cross-platform delivery of applications that is Java's hallmark. Con-

forming class files generated by *any* compiler will run in *any* Java VM implementation, no matter if that implementation interprets bytecodes, performs dynamic "just-in-time" (JIT) translation to native code, or precompiles Java class files to native object files. Targeting compiled Java classes for analysis and optimization has several advantages. First, program improvements accrue even in the absence of source code, and independently of the compiler and VM implementation. Second, Java class files retain enough high-level type information to enable advanced optimizations. Finally, analyzing and optimizing bytecode can be performed off-line, permitting JIT compilers to focus on fast code generation rather than expensive analysis, while also exposing opportunities for fast low-level JIT optimizations.

We have implemented a bytecode-to-bytecode class transformer that performs PRE for access expressions in Java. Our implementation, called BLOAT (for *B*ytecode-*L*evel *O*ptimization and *A*nalysis *T*ool) takes compiled Java classes adhering to the Java VM specification and generates transformed classes as output. For each method, BLOAT first builds a control-flow graph, with an expression tree for each basic block, then infers the types of local variables and the operand stack at each point in the code [Palsberg and Schwartzbach 1994], constructs an intermediate representation based on static single-assignment (SSA) form [Cytron et al. 1991; Wolfe 1996; Briggs et al. 1997], performs SSA-based value numbering [Briggs et al. 1997] with TBAA, followed by SSA-based PRE [Chow et al. 1997], and finishes with generation of new Java bytecodes for the method. Note that BLOAT is a stand-alone tool that can be used to optimize Java classes independently of VM implementation.

3.2 Optimizations for PJama

PJama [Atkinson et al. 1996] is a prototype implementation of orthogonal persistence for Java being developed jointly by Sun Microsystems Laboratories and Glasgow University. The PJama VM is based on the Sun Java Development Kit (JDK) VM and conforms to the Java VM specification; it executes classes compiled to the standard bytecode instruction set and class file format. Persistence functionality is provided by an extended API, extensions to the VM for read and write barriers, and associated run-time support. In the current release of PJama, the read and write barriers are hidden inside the bytecodes that implement access expressions and method invocations; these are listed in Table 5.

Table 5: Bytecodes requiring barriers

Opcode	Barrier
arraylength	read on array operand
athrow *getfield* *instanceof*	read on object operand
Taload	read on array/object operand
Tastore	read *and* write on array/object operand
putfield	read *and* write on object operand
invokevirtual *invokespecial* *invokeinterface*	read on object operand

$T = b, s, i, l, f, d, c, a$

To optimize the persistence barriers they must first be exposed. Thus, we have deleted the hidden barrier code from the implementations of the original bytecodes and extended the PJama VM with two new internal barrier bytecodes. As a class is loaded into the extended PJama VM its methods must now be edited to insert the appropriate barrier bytecode immediately before each occurrence of the bytecodes listed in Table 5. BLOAT supports this operation with a preprocessing (non-analyzing, non-optimizing) pass over the class to insert the barriers. The class can then go on to execute in the extended VM. Subsequent optimization by BLOAT can then occur at any convenient time. BLOAT also supports a "way-ahead-of-time" option to preprocess and optimize class files for later loading by the new PJama VM; this option is commonly used to prepare the core Java classes for loading into a virgin PJama persistent store.

The new barrier bytecodes are specified in Table 6.[3] Rather than operating on the reference at the top of the stack, the new bytecodes take a stack offset so as to ease insertion of the barrier for the target of method invocation bytecodes, which is always located on the stack at some known offset below the other arguments

[3] The current PJama prototype distinguishes pointer stores from non-pointer stores in its implementation of the write barrier, for reasons having to do with details of its implementation of heap stabilization. To support this functionality we must insert and optimize *two* different write barrier bytecodes, one for pointers and one for non-pointers. BLOAT does in fact support this, but we consider them to be equivalent for this paper so as to demonstrate the full potential for optimization of write barriers.

Table 6: New barrier bytecodes

Operation	read barrier	write barrier
Format	*read* *index*	*write* *index*
Forms	*read* = 233 (0xe9)	*read* = 234 (0xea)
Stack	No change	No change
Description	The *index* is an unsigned byte between 0 and 255, inclusive. The operand stack word at offset *index* from the top of the stack must be of type `reference`. If that reference is not `null` then the object it references is checked for residency, and faulted in if it is not.	The *index* is an unsigned byte between 0 and 255, inclusive. The operand stack word at offset *index* from the top of the stack must be of type `reference`. If that reference is not `null` then the object it references is marked dirty in the object cache.

to the call. Thus, the initial preprocessing to insert barriers needs no expensive analysis.

As in Hosking [1997], we also exploit Java's object-oriented execution paradigm to avoid barriers on accesses to the object on which an instance method was invoked. Since the target—accessed via the `this` keyword inside the instance method—is made resident at the time of the call by the read barrier associated with the "invoke" bytecodes of Table 5, there is no need for barriers on accesses via `this`. The JDK compiler stores `this` in the first local variable of instance methods, allowing BLOAT to recognize such accesses. BLOAT also recognizes references to objects that are instantiated using the "new" bytecodes, so as to eliminate barriers on accesses to newly-allocated objects.

3.3 Cache management

As mentioned earlier, barrier optimizations require a contract with the persistence run-time system, which must not undo the effect of a barrier while optimized code can execute that assumes the barrier is still in effect. The contract with the PJama run-time system is simple: PJama must maintain the effect of both barriers for all objects directly referenced from a Java thread's stack frames (both operand stacks and local variables). In other words, resident objects referenced directly from a thread stack must be *pinned* in the object cache whenever the thread is active. Thus, the PJama object cache manager must either avoid evicting pinned objects when it attempts to reclaim cache space, or arrange for them to be made resident before the optimized thread resumes execution. Similarly, dirty bits set on objects in the cache that are directly referenced from a thread's stack must be maintained, even across stabilizations. Clearly, this contract has significant ramifications for the run-time system; Cutts et al. [1998] explore the issues in more detail.

It is possible to refine the compile-time/run-time contract if the compiler can provide more detailed information to the run-time system as to the barriers in effect for ranges of optimized code. Such information is similar to the static tables sometimes provided to the run-time system for exception handling and garbage collection [Diwan et al. 1992; Agesen et al. 1998].

4 Experiments

To evaluate the impact of our optimizations we applied them to the traversal portions of a Java implementation of the OO7 benchmarks [Carey et al. 1993], comparing the number of barriers required for execution of each benchmark for unoptimized code versus optimized code. The classes for the OO7 benchmarks, as well as the Java core classes used by OO7, were first edited by BLOAT to add the new persistence barrier bytecodes. Optimized classes were obtained from these using BLOAT's ahead-of-time optimization option. Also, in order to separate out the impact of exposed barrier PRE versus access expression PRE alone we optimized the original barrier-free classes, then edited them to add persistence barriers. Thus, we obtain results for three distinct configurations of the OO7 classes: unoptimized with barriers (none), access path optimizations without barrier optimizations (access), and access path optimizations with barrier optimizations (access+barrier). The dif-

ference between access and access+barrier reveals the advantage to be gained by exposing the barriers to optimization.

4.1 Benchmarks

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. Our results are all obtained with the *small* OO7 database, configured as in Table 7.

Table 7: Small OO7 database configuration

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
Total composite parts	500
Total atomic parts	10000

We used the following traversal operations of the OO7 benchmarks:

1 Raw traversal speed: traverse the assembly hierarchy; for each base assembly encountered visit each of its unshared composite parts; for each composite part encountered visit its entire graph of atomic parts using depth-first search; return a count of the number of atomic parts visited

2 Traversal with updates: repeat traversal 1 but update atomic parts during the traversal (as follows) by swapping two attributes; return the number of updates performed

- (a) Update one atomic part per composite part encountered
- (b) Update every atomic part encountered
- (c) Update each atomic part in a composite part four times

3 Traversal with indexed field updates: repeat traversal 2, except that the update is on an indexed attribute

6 Sparse traversal speed: traverse the assembly hierarchy; for each base assembly encountered visit each of its unshared composite parts; for each composite part encountered visit just the root atomic part; return the number of atomic parts visited

4.2 Metrics

For each combination of benchmark and optimization level we measure the number of barrier operations executed for the benchmark using an instrumented version of the VM that reports bytecode execution frequencies. We measured only warm executions of the benchmark operations, so as to eliminate the overhead of bytecodes executed for initialization of classes as they are dynamically loaded by the VM.

4.3 Results

The results are given in Table 8, revealing that on average 83% of read barriers, and 25% of write barriers, are removed by PRE over both access expressions and barrier expressions. Considering the write barrier results individually, one can immediately see the impact of optimization by comparing traversals 2b and 2c, which differ only in the number of times each part is updated. The four updates per part in 2c are performed in a tight loop, so the optimizer is able to hoist the write barrier out of the loop, resulting in the same number of write barriers executed as traversal 2b.

Applying PRE just to the access expressions before insertion of the barriers is much less effective, indicating the advantages to be gained from exposing them to the optimizer. In other words, simply adding PRE over access expressions to the original PJama implementation (in which the barriers are buried inside the access bytecodes) is less effective at reducing barrier overheads.

Table 8: Results

Traversal	Read barriers executed					Write barriers executed				
	PRE level				% removed	PRE level				% removed
	none	access	access +barrier	access +barrier +this		none	access	access +barrier	access +barrier +this	
1	10535707	7899406	3456590	2126883	80	495363	495363	404599	404599	18
2a	10588195	7951894	3500330	2168436	80	499737	499737	406786	406786	19
2b	10666927	8030626	3456590	2126883	80	582843	582843	448339	448339	23
2c	11191807	8555506	3456590	2170623	81	845283	845283	448339	448339	47
3a	10586008	7949707	3500330	2168436	80	497550	497550	406786	406786	18
3b	10623187	7986886	3456590	2126883	80	539103	539103	448339	448339	17
3c	11016847	8205586	3631550	2170623	80	670323	670323	448339	448339	33
6	3458575	1215934	47057	27363	99	14223	14223	10939	10939	23

5 Conclusions

Combining type-based alias analysis with partial redundancy elimination over access expressions, is a powerful technique for reducing the fundamental barrier overheads of orthogonal persistence. For the OO7 traversal benchmarks these optimizations remove a majority of read barriers and a significant fraction of write barriers. We believe these techniques will prove crucial to the achievement of respectable performance by persistent systems.

Acknowledgments

We are indebted to Amer Diwan for first suggesting an optimization approach combining type-based alias analysis and partial redundancy elimination. We also thank the PJama team for their assistance in understanding PJama's internals, and for providing us with the source code to their implementation of the OO7 benchmarks. We especially thank Laurent Daynès for his assistance in fixing several bugs in their original OO7 implementation. Malcolm Atkinson's comments on an earlier draft also helped significantly to clarify and improve the presentation.

References

AGESEN, O., DETLEFS, D., AND MOSS, J. E. B. 1998. Garbage collection and local variable type-precision and liveness in Java virtual machines. See PLDI [1998]. To appear.

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. AND JORDAN, M. J., Eds. 1997. *Proceedings of the Second International Workshop on Persistence and Java* (Half Moon Bay, California, Aug.). Sun Microsystems Laboratories Technical Report 97-63.

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

BRIGGS, P., COOPER, K. D., HARVEY, T. J., AND SIMPSON, L. T. 1997. Practical improvements to the construction and destruction of static single assignment form. Available at http://www.cs.rice.edu/≈harv/ssa.ps.

BRIGGS, P., COOPER, K. D., AND SIMPSON, L. T. 1997. Value numbering. *Software: Practice and Experience 27,* 6 (June), 701–724.

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.

CHOW, F., CHAN, S., KENNEDY, R., LIU, S.-M., LO, R., AND TU, P. 1997. A new algorithm for partial redundancy elimination based on SSA form. In Proceedings of the ACM Conference on Programming Language Design and Implementation (Las Vegas, Nevada, June). *ACM SIGPLAN Notices 32,* 5 (May), 273–286.

CUTTS, Q. AND HOSKING, A. L. 1997. Analysing, profiling and optimising orthogonal persistence for Java. See Atkinson and Jordan [1997], 107–115.

CUTTS, Q., LENNON, S., AND HOSKING, A. L. 1998. Reconciling buffer management with persistence optimizations.

CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. 1991. Efficiently computing static single assignment form and the program dependence graph. *ACM Trans. Program. Lang. Syst. 13,* 4 (Oct.), 451–490.

DAYNÈS, L. AND ATKINSON, M. 1997. Main-memory management to support orthogonal persistence for Java. See Atkinson and Jordan [1997].

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.

DIWAN, A., MCKINLEY, K. S., AND MOSS, J. E. B. 1998. Type-based alias analysis. See PLDI [1998]. To appear.

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

GOSLING, J., JOY, B., AND STEELE, G. 1996. *The Java Language Specification.* Addison-Wesley.

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. In *Proceedings of the Seventh International Workshop on Persistent Object Systems* (Cape May, New Jersey, May 1996), R. Connor and S. Nettles, Eds. Persistent Object Systems: Principles and Practice. Morgan Kaufmann, 174–183.

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.

KEMPER, A. AND KOSSMAN, D. 1995. Adaptable pointer swizzling strategies in object bases: Design, realization, and quantitative analysis. *International Journal on Very Large Data Bases 4,* 3 (Aug.), 519–566.

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

LARUS, J. R. AND HILFINGER, P. N. 1988. Detecting conflicts between structure accesses. In Proceedings of the ACM Conference on Programming Language Design and Implementation (Atlanta, Georgia, June). *ACM SIGPLAN Notices*, 21–34.

LINDHOLM, T. AND YELLIN, F. 1996. *The Java Virtual Machine Specification.* Addison-Wesley.

MOREL, E. AND RENVOISE, C. 1979. Global optimization by suppression of partial redundancies. *Commun. ACM 22,* 2 (Feb.), 96–103.

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.

MOSS, J. E. B. AND HOSKING, A. L. 1996. Approaches to adding persistence to Java. In *Proceedings of the First International Workshop on Persistence and Java* (Drymen, Scotland, Sept.), M. P. Atkinson and M. J. Jordan, Eds. Sun Microsystems Laboratories Technical Report 96-58, 1–6.

PALSBERG, J. AND SCHWARTZBACH, M. I. 1994. *Object-Oriented Type Systems.* Wiley.

PLDI 1998. *Proceedings of the ACM Conference on Programming Language Design and Implementation* (Montréal, Canada, June). *ACM SIGPLAN Notices 33.*

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.

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. 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.

WOLFE, M. 1996. *High Performance Compilers for Parallel Computing.* Addison-Wesley.