OPTIMIZING ORTHOGONAL PERSISTENCE FOR JAVA

A Thesis

Submitted to the Faculty

of

Purdue University

by

Kumar Jagadeeshwaraiah Brahnmath

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

May 1998

REVISED May 15, 1998

To Rachel

ACKNOWLEDGMENTS

Thanks to my advisor Antony Hosking, who was responsible for getting me interested in persistent systems and who has been a cheerful guide and teacher throughout my time at Purdue. Thanks to my committee members Jens Palsberg and Aditya Mathur. Thanks to Nate Nystrom for providing invaluable assistance in understanding BLOAT. Thanks to all the members of the lab: Gustavo, Mike, Aria, Anshul and Raghu. Thanks to the PJama group at Glasgow and Sunlabs. Thanks to my wife Rachel for her loving support and thanks to my parents and Giri, who gave me support from afar. Thanks to Lyle, Judy, Eli and Dani for everything they have done for me.

TABLE OF CONTENTS

Page

LIST OF TABLES vii

LIST OF FIGURES viii

ABSTRACT ix

1 INTRODUCTION 1

1.1 Orthogonal persistence 2

1.2 Overheads of persistence 3

1.3 Persistence optimizations 4

1.4 Contributions 5

1.5 Overview 5

2 OPTIMIZING ORTHOGONAL PERSISTENCE 6

2.1 Read barriers 6

2.2 Write barriers 7

2.3 Swizzle barriers 7

2.3.1 Swizzle policies 7

2.3.2 Detecting and handling object faults 8

2.4 Barrier redundancy 10

2.5 Barrier optimizations 11

2.5.1 Read and write barrier optimization 11

2.5.2 Range swizzle optimization 12

Page

3 ANALYSIS AND OPTIMIZATION 14
3.1 Background 14
3.1.1 Induction variables 14
3.1.2 Terminology 14
3.1.3 Sequence variables 16
3.1.4 Control flow graphs 17
3.1.5 Loops and loop inversion 18
3.1.6 SSA form 19
3.1.7 Demand-driven SSA graph 20
3.2 PRE over access path expressions 21
3.2.1 Terminology and notation 22
3.2.2 Barrier optimizations 24
3.3 Type-based alias analysis 25
3.4 Partial redundancy elimination 26
3.4.1 Java constraints on optimization 29
3.5 Demand-driven induction variable analysis (DIVA) 29
3.5.1 Cycles in SSA graphs 29
3.5.2 Detecting sequences 30
3.5.3 Tarjan’s algorithm 30
3.6 Range swizzle optimizations with DIVA 31
3.6.1 Classifying sequences 32
3.6.2 Well-behaved loops 33
3.6.3 Hoisting swizzle barriers 34

4 IMPLEMENTATION 35
4.1 The PJama persistent system 35
4.1.1 Architecture of PJama 35
4.1.2 Swizzling in PJama 37

Page

4.1.3 Array swizzling 38
4.2 Implementation of persistence optimizations 38
4.2.1 Bytecode-to-bytecode class transformation 39
4.2.2 Read and write barrier optimization for PJama 40
4.2.3 Range swizzle optimization for PJama 42
4.2.4 Cache management 42

5 EXPERIMENTS 44
5.1 Read and write barrier optimization 44
5.1.1 Benchmarks 44
5.1.2 Metrics 46
5.1.3 Results 46
5.2 Range swizzle optimization 47
5.2.1 Benchmarks 47
5.2.2 Results 48
5.3 Conclusions 48
5.4 Future work 49
5.4.1 Persistence-enabled optimizations 50
5.4.2 Persistence-enabling optimizations 50

BIBLIOGRAPHY 51

LIST OF TABLES

Table Page

2.1 Barrier expressions 11
3.1 Access expressions 23
3.2 Intermediate representation for access expressions 25
3.3 *FieldTypeDecl*(AP_1, AP_2) 26
3.4 Classification of SCCs based on frequency of operations 33
4.1 Bytecodes requiring barriers 40
4.2 New read and write barrier bytecodes 41
4.3 New swizzle barrier bytecodes 43
5.1 Small OO7 database configuration 45
5.2 Results of read barrier optimizations 47
5.3 Results of write barrier optimizations 48
5.4 Results of range swizzle optimizations 49

LIST OF FIGURES

Figure Page

2.1 Edge Marking 9
2.2 Node Marking 10
2.3 A loop with swizzle barriers 13
2.4 After range swizzle optimization 13
3.1 Basic sequence variable 15
3.2 Basic loop counter 15
3.3 Sequence variable lattice. 18
3.4 Loop representation in the CFG 19
3.5 Loop inversion and Hoisting 20
3.6 Loop representation in the SSA form 21
3.7 Demand-driven SSA graph 22
3.8 PRE for arithmetic expressions 27
3.9 PRE for access expressions 27
3.10 PRE for barrier expressions 28
3.11 SCC classification 31
3.12 SCC classification(cont.). 32
3.13 SCC classification (cont.). 33
3.14 A well-behaved loop 34
4.1 PJama's Object Cache Architecture 36

ABSTRACT

Brahnmath, Kumar Jagadeeshwaraiah. M.S., Purdue University, May 1998. Optimizing Orthogonal Persistence for Java. Major Professor: Antony Hosking.

Persistent programming languages provide object persistence across program invocations, by treating volatile memory as a cache for stable storage. Orthogonal persistence allows any object to be potentially persistent, without any restriction on its type. Adding orthogonal persistence to a language environment presents several performance-related challenges. This work is aimed at reducing the various overheads associated with orthogonal persistence. The costs being targeted are read barriers, write barriers and swizzle barriers. A read barrier checks the cache residency of the target object while a write barrier marks the target as dirty in the cache. Many of these read and write barriers are redundant, and applying partial redundancy elimination of pointer-based access path expressions can be very beneficial in eliminating them. Swizzling is the translation of an object reference from an external, persistent format to an internal, transient format; a swizzle barrier checks and makes sure that a reference is swizzled. Swizzle barriers have the added overhead that they are usually associated with iteration over container objects like arrays. Hence, there is a need for an additional optimization to merge multiple swizzle barriers into one inclusive barrier. By induction variable analysis, the bounds of a variable being used to loop through an array can be determined. We exploit this information to develop a range swizzle optimization technique to reduce the overhead of swizzle barriers. We have implemented our analysis and optimization framework for an orthogonally persistent Java system. In experiments performed on several benchmarks and applications, our optimizations eliminated on average 83 percent of read barriers, 25 percent of write barriers and 66 percent of swizzle barriers.

1 INTRODUCTION

Thesis statement:

> *The overheads of orthogonal persistence can be reduced significantly by program analysis and optimization.*

The traditional model of system development has been to write application programs in general-purpose programming languages, which create and manipulate data structures, interacting with the users of the system. Long-term storage for non-transient data is usually provided either via raw operating system files or a database management system (DBMS). The data types provided by the language and those created by the programmer usually do not translate directly into the file system or the DBMS world. Thus programmers must provide routines to perform this translation or utilize existing middle-ware which provides that functionality. In both cases, an *artificial barrier* is erected between the program in volatile memory and the data it needs, stored away in the labyrinth of the file system or DBMS. Hand-coded translation is error-prone since *type safety* and *integrity* cannot be guaranteed, thus decreasing system reliability and complicating extension and maintenance. With the growing use of object-oriented languages, there is also the added complexity of translating from the language object model to the DBMS (e.g. relational) data model and vice versa. This is difficult since there is no natural mapping between the two. All these complexities lead to an *impedance mismatch* [Copeland and Maier 1984] between application programming languages and the data storage subsystem.

Persistent programming languages have been designed with the explicit goal of erasing these barriers and provide a natural, safe interface to data, regardless of where it is located and *orthogonal* to its type. The Java™ programming language, with its regular structure and safety features, provides a good platform for persistence [Moss and Hosking 1996].

One such prototype is the system implemented at the University of Glasgow to support orthogonal persistence for Java [Atkinson et al. 1996]. Other efforts are also underway [Wileden et al. 1996; Garthwaite and Nettles 1996]. The key challenge now is to minimize the overheads of persistence, thereby making it a viable medium for system development. This thesis demonstrates a significant reduction of the overheads of orthogonal persistence by a combination of program analysis and optimization techniques.

1.1 Orthogonal persistence

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 complex 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 of explicitly programmed reads and writes, the language's compiler and/or run-time system contrive to automatically 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. Any object created dynamically can be made persistent by just referring to it from an already persistent object; this is usually termed *persistence by reachability.* 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.

1.2 Overheads of persistence

There are a number of techniques for object faulting based on hardware support for memory mapping, which is transparent to the compiler [Lamb et al. 1991; Singhal et al. 1992; Wilson and Kakkad 1992; White and DeWitt 1994]. However the restrictions 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], not to mention a performance penalty [Hosking and Moss 1993]. 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. Since many of these barriers are applied to the same objects repeatedly, there is a significant amount of redundancy that can be exploited to reduce the overhead.

The identifier used to store and retrieve an object from stable storage is called its *persistent identifier* (PID). The representation of object identifiers in virtual memory is usually different from their PID form. This means that a conversion has to take place before a persistent object can be accessed by a program. *Swizzling* is the process of converting an object reference from its external, persistent format to an internal, transient format and caching it for future use [Moss 1992]. The motivation for swizzling is that object access can be achieved through fast internal addressing as opposed to slow persistent identifier (PID) translation. However swizzling has two inherent costs: the time required for the first PID translation, and the space required to cache the translated identifier (which is implementation dependent). A *swizzle barrier* checks to see if a reference is swizzled, and swizzles it, if not already. This barrier is inserted when a reference is not guaranteed to be always in swizzled state. They typically occur in the body of loops traversing container objects like arrays.

The performance penalty paid by these barriers 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].

1.3 Persistence optimizations

We seek to reduce the overheads of persistence by *program analysis* and *optimization*. To reduce the number of read and write barriers executed we avoid applying barriers to accesses where program analysis shows that the barrier is redundant. We use a combination of *type based alias analysis* (TBAA) [Diwan et al. 1998] and *partial redundancy elimination* (PRE) [Morel and Renvoise 1979] on access path expressions to identify redundant read and write barriers and eliminate them. To reduce the overhead of swizzle barriers on array accesses in loops, we apply *induction variable analysis* [Gerlek et al. 1995] to discover the

range of elements that are actually accessed and swizzle all those references in one swizzle operation outside the loop that traverses the array. This work aims at demonstrating the effectiveness of these techniques to reduce the fundamental overheads of orthogonal persistence.

1.4 Contributions

The major contributions of this work towards achieving the goal of reducing the overheads of orthogonal persistence are:

- Demonstrating the application of partial redundancy elimination over access path expressions to eliminate redundant read and write barriers.
- Developing range swizzle optimizations based on demand-driven induction variable analysis to reduce the swizzle barrier overhead.
- An implementation for the PJama prototype for orthogonal persistence in Java.
- Experimental evidence of the effectiveness of our analyses for the elimination of redundant read, write and swizzle barriers.

1.5 Overview

The rest of the thesis is organized as follows. Chapter 2 describes the overheads of orthogonal persistence in detail, defines the analysis and optimization problem and provides necessary background information. In chapter 3 we describe our program analysis and optimization framework. The application of partial redundancy elimination over access path expressions to eliminate redundant barriers is explained. The Demand-driven Induction Variable Analysis (DIVA) technique and its application for range swizzle optimizations is also presented. Chapter 4 describes implementation issues and the optimization framework for the PJama persistent system. Chapter 5 describes the experimental setup used to evaluate the impact of these optimizations and presents the results of those experiments. The thesis concludes with a summary of our findings and a discussion of possible future work.

2 OPTIMIZING ORTHOGONAL PERSISTENCE

Orthogonal persistence seeks to provide uniform and safe access to objects regardless of their transience or persistence. In achieving this goal, orthogonal persistence pays several performance penalties that are fundamental to its domain. This chapter describes these overheads and defines the notion of their *redundancy*. The read and write barrier overheads are described in detail and the goal of eliminating redundant barriers is motivated and explained. Swizzle barriers and their dynamic overhead are explored and the need for additional analysis to eliminate swizzle barriers is described.

2.1 Read barriers

In an orthogonally persistent system, a reference could point to a transient object, a resident persistent object or a non-resident persistent object. This means that every dereference has to make sure that the target object is resident. This is termed the *read barrier*. A read barrier checks to see if the object is resident and, if not, triggers an object fault. After the object has been made resident in memory the read access can proceed. Orthogonal persistence hides the actual location of objects under a layer of abstraction. This means that all objects (even transient objects) end up paying the cost of read barriers. If the performance of persistent programs is to approach that of their non-persistent counterparts, the costs of read barriers must be reduced significantly, enabling persistent programs to perform competitively when operating on objects that are entirely resident in memory.

2.2 Write barriers

Any modifications made to persistent data by a program become permanent only when some sort of *checkpoint* operation is invoked by the program. If the program has modified only a small subset of resident objects, writing all the objects back to stable storage would be very inefficient. The optimal procedure will save only those objects that have been modified. This means that every update to an object must mark that object as updated. This is termed the *write barrier*. Similar to the case of read barriers, the cost of write barriers is paid by all objects in an orthogonally persistent system.

2.3 Swizzle barriers

Persistent systems allow objects along with their inter-object references to persist from one program invocation to the next. A program running on such a system may refer to both resident and non-resident persistent objects. It may also refer to transient objects which may become persistent in the future. To more easily provide equal treatment to all objects, the persistent object references in a persistent object can be swizzled. If a program can potentially see a reference that has not yet been swizzled, a swizzle barrier would have to be inserted to protect that access.

2.3.1 Swizzle policies

There are several different swizzle policies that can be adopted by a persistent system. The two most widely used ones are *eager swizzling* and *lazy swizzling*.

Eager swizzling

When an object fault occurs, *eager swizzling* will swizzle all the object references in the faulted object. This is usually performed while the object is being copied into the program's virtual memory and involves identifying all the references in the object and swizzling them. The overhead of swizzling all the references can be profitable only if objects typically contain a limited number of references. The advantage of this scheme is

that no swizzle barrier need be invoked to check if a reference is swizzled or not, before using that reference to access the object it targets.

Lazy swizzling

Lazy swizzling converts references only as they are accessed; i.e. the swizzle operation is postponed to the last possible moment. This scheme avoids the overhead of eager swizzling, especially for objects that contain a large number of references. For example, consider an array of object references. Eager swizzling would swizzle every element in the array. This would not be very efficient in terms of space and time if the array is sparsely accessed. Lazy swizzling avoids the up front overhead of eager swizzling by converting reference elements only as they are accessed. Since array elements are not guaranteed to be swizzled, every access to an array element must be protected by a *swizzle barrier*. Subsequent accesses will continue to incur the cost of the swizzle barrier.

2.3.2 Detecting and handling object faults

If a program executing in a persistent system traverses a reference to a non-resident object then it must be made available to the program in memory. This is termed an *object fault*. For an object fault to occur, the system needs some mechanism to distinguish between references to resident and non-resident objects. These mechanisms may be divided into two categories, depending on the strategy they adopt [Hosking and Moss 1990]. For the purposes of this discussion we view the persistent heap as a *directed graph*: the objects are the *nodes* and the references between the objects are the *edges*.

Edge marking schemes take the approach of tagging the references between the objects. If tagged as *swizzled*, then a reference is a direct pointer to the corresponding object in memory; if *non-swizzled* then the reference consists of a persistent identifier (PID). Edge marking can be implemented easily by tagging pointers. When a marked link is traversed, an object fault occurs and the object is retrieved from *stable storage*. The marked edge is then unmarked. This process is illustrated in Figure 2.1(b). An apparent disadvantage of edge marking is that PIDs can be fetched from the pointer fields of objects, passed around,

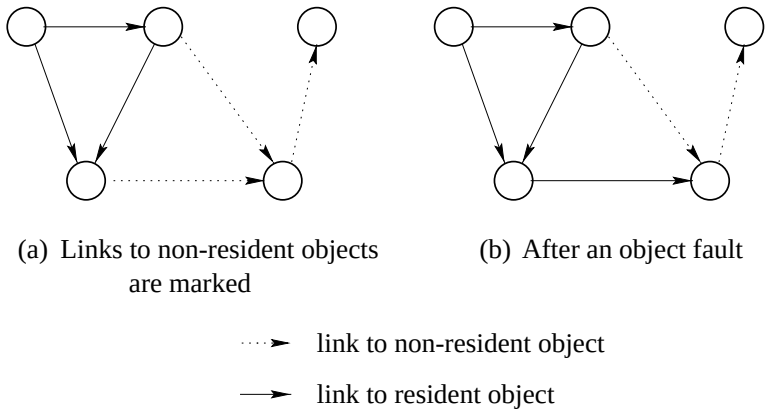


Figure 2.1: Edge Marking

and stored, without accessing the target object. When the target object finally is accessed the origin of the reference may no longer be known.

Node marking schemes require that all object references in resident objects be converted to pointers. In ObjectStore [Lamb et al. 1991] and Texas [Singhal et al. 1992] this is achieved by reserving (although not necessarily allocating) virtual pages for the objects referred to by the pointers, and protecting those pages to have the operating system trap all access to those pages. Another approach, is to have small proxy objects (we call them *fault blocks*) stand in for non-resident objects, as illustrated in Figure 2.2(b). A fault block contains the PID of the target object, and is distinguishable from an ordinary object. Whenever a reference is followed, if it refers to a fault block, then the target object is made resident. The fault block is changed to point to the now-resident object (see Figure 2.2(c)). We call the updated fault block an *indirect block*. If a reference to be followed refers to an indirect block then the target object can be located at the cost of an indirection. The indirection

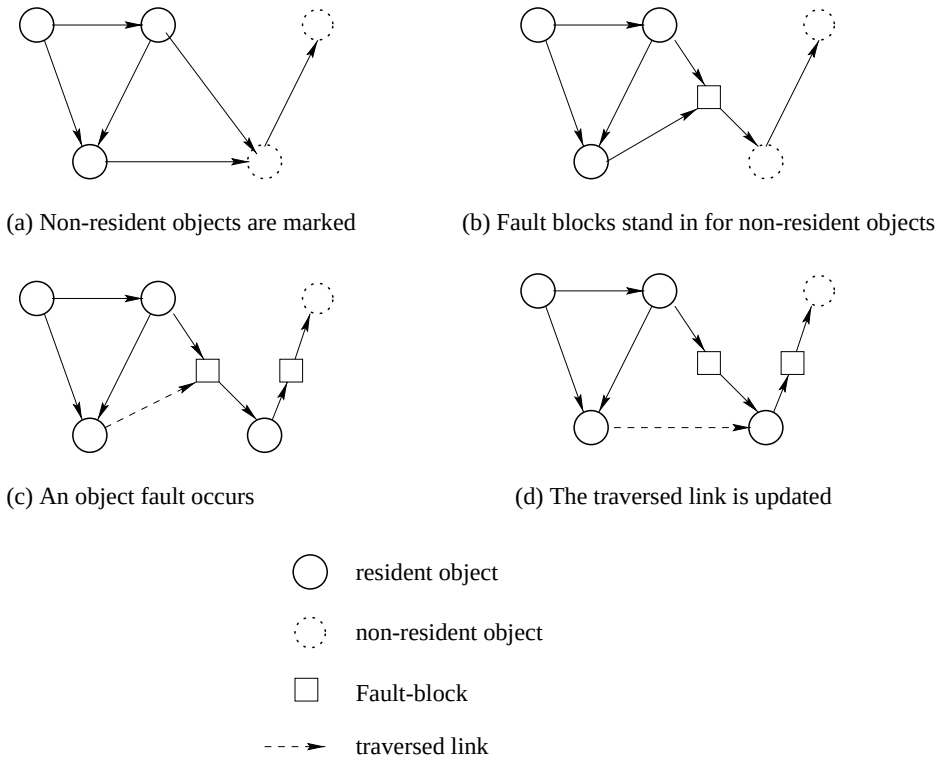


Figure 2.2: Node Marking

may be bypassed by updating the traversed link to point directly to the object instead of the fault block (see Figure 2.2(d)).

2.4 Barrier redundancy

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, or swizzle the reference) has not been undone (i.e., the barrier is *idempotent*). 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 Chapter 4, which presents our implementation for PJama. Cutts et al. [1998] consider the issue from the perspective of the run-time system.

2.5 Barrier optimizations

Our goal is to avoid applying barriers to accesses where program analysis shows that the barrier is redundant. To describe the problem of redundant barrier elimination, we define *barrier expressions* as shown in Table 2.1.

Table 2.1: Barrier expressions

Notation	Name	Description
$read(p)$	Read barrier	Apply read barrier to, and return, object referred to by p
$write(p)$	Write barrier	Apply write barrier to, and return, object referred to by p
$swizzle(p,i)$	Swizzle barrier	Apply swizzle barrier to the component of array p with subscript i
$swizzleRange(p,i,j)$	Range swizzle barrier	Apply swizzle barrier to components of array p with subscripts in the range [i, j]

2.5.1 Read and write barrier optimization

Given two read barrier expressions $read(p)$ and $read(q)$, if we can guarantee that p and q refer to the same object and that $read(p)$ dominates $read(q)$ then $read(q)$ is redundant and can be replaced simply by q. A similar replacement can also be applied to redundant write barriers. 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.* We use the type-based alias analysis framework of Diwan et al. [1998] to solve the problem of aliases and apply the partial redundancy elimination (PRE) technique of Morel and Renvoise [1979] to eliminate redundant read and write barriers.

2.5.2 Range swizzle optimization

Container objects such as arrays, are typically swizzled lazily, requiring the insertion of swizzle barriers. Since arrays are typically accessed in loops, these swizzle barriers end up in the bodies of loops. Often an array element like $a[i]$ is accessed repeatedly in the body of such a loop. Any such repeated reference must be protected by a swizzle barrier as shown in Figure 2.3. Such repeated swizzle barriers are redundant and can be recognized and removed by program analysis. But all swizzle barriers in the body of a loop cannot be entirely removed, just by applying the common access path definition of redundancy (as defined in Section 2.4). For example, in Figure 2.3, analysis may determine that the second swizzle barrier is redundant because it is applied to the same reference $a[i]$ in both cases and can be removed.[1] The first swizzle barrier cannot be removed, since it is not redundant and remains a serious overhead to the execution of the loop. To remove that swizzle barrier, it has to be made redundant by performing a range swizzle barrier operation outside the loop. To do this, we would like to determine the range of the array that is being accessed, so that we can swizzle that range of references before entering the loop. To determine the access range, we have to find loops that are accessing arrays of objects and determine the *bounds* of the loop. If the lower bound of a loop traversing array *a* is found to be *l*, and the upper bound is found to be *u*, then we can insert a range swizzle *swizzleRange*(a,l,u) outside the loop as shown in Figure 2.4. This enables the elimination

[1] It is difficult to treat swizzle barriers as expressions since they do not return any value. So, our current implementation of PRE leaves it to DIVA to eliminate such redundant swizzle barriers.

```
i ← 1

while (i ≤ n) do
   ...
   swizzle(a, i)
   e ← a[i].x
   ...
   swizzle(a, i)
   f ← a[i].y
   i ← i + 1
endwhile
```

Figure 2.3: A loop with swizzle barriers

```
i ← 1
swizzleRange(a, 1, n)
while (i ≤ n) do
   ...
   e ← a[i].x
   ...
   f ← a[i].y
   i ← i + 1
endwhile
```

Figure 2.4: After range swizzle optimization

of swizzle barriers on the components of array a in the body of the loop. We develop a Demand-driven Induction Variable Analysis (DIVA) technique which can perform these optimizations. This technique is described in the next chapter.

3 ANALYSIS AND OPTIMIZATION

3.1 Background

This section describes some terminology and background information which is necessary to understand our analysis and optimization framework. Induction variables and sequence expressions are defined and explained. The Control Flow Graph (CFG) and the Static Single Assignment representation of the CFG are explained with illustrative examples. We follow the terminology used in Gerlek et al. [1995].

3.1.1 Induction variables

Induction variables are program variables whose successive values form a definite pattern over some part of a program, usually a loop [Muchnick 1997]. They belong to a broader group of variables known as *sequence variables* where the pattern could be linear, polynomial, geometric, wrap-around, periodic or monotonic. Detecting such sequence variables is the first step towards implementing array swizzle optimizations. Once a sequence variable is found, the range of values it can assume is determined, thereby enabling swizzle checks to be hoisted out of loops.

3.1.2 Terminology

Basic sequence variables

Given a statement *s* within the body of a loop *l* assigning some arbitrary expression *e* to a scalar, integer variable *v* as in Figure 3.1, if expression *e* contains an occurrence of *v*, then *v* is a *basic sequence variable* in *l*, and *e* is the associated *sequence expression*.

l :
while $(condition)$ **do**
  ...
  $s : v \leftarrow e$
  ...
endwhile

Figure 3.1: Basic sequence variable

$h \leftarrow 0$
l :
while $(condition)$ **do**
  ...
  $h \leftarrow h + 1$
  ...
endwhile

Figure 3.2: Basic loop counter

Derived sequence variables

Given a statement s within the body of a loop l assigning some arbitrary expression e to a scalar, integer variable v as in Figure 3.1, if expression e does not contain an occurrence of v but does contain an occurrence of some sequence variable w, then v is a *derived sequence variable* in l.

Basic loop counter

Associated with each loop l is a *basic loop counter*, h, whose value is zero on the first iteration of the loop and is incremented by one at the end of each subsequent iteration, as shown in Figure 3.2. The goal is to assign closed form expressions in terms of h to sequences.

3.1.3 Sequence variables

Linear induction variables

Usually a loop's iterations are counted by an integer-valued variable that proceeds upward (or downward) by a constant amount with each iteration. Such sequence variables are termed *linear induction variables*. Often, additional variables, most notably subscript values follow a pattern similar to the loop-control variable's, although perhaps with different starting values, increments, and directions.

Polynomial induction variables

Usually induction variables that occur in typical programs are linear functions of a loop index, formed by the addition of loop-invariant values. However, when the term added to the induction variable is a linear induction variable, a *polynomial induction variable* is formed. A sequence variable whose expression contains an addition(or subtraction) of a polynomial induction variable, yields a polynomial of a correspondingly higher degree.

Geometric induction variables

In addition to linear and polynomial sequence variables, programs may also contain *geometric sequences*. These arise when the sequence variable is multiplied by some loop-invariant value. This multiplicative factor defines the base of the geometric term in the sequence expression.

Wrap-around variables

Wrap-around variables occur when a variable is assigned a value from outside the loop on the first iteration, and then takes on the pattern of another sequence variable (typically a linear induction variable) for the remainder of the iterations.

Periodic sequences

A sequence variable which keeps changing its value between the same two values forms a *periodic sequence*. Such variables are also referred to as *flip-flop* variables.

Monotonic sequences

Monotonic sequences arise when a variable is conditionally incremented by a known constant value. Depending on the constant, four classes can be distinguished: monotonically increasing, monotonically decreasing, monotonically strictly increasing, and monotonically strictly decreasing.

Sequence variable lattice

The range of sequence expressions can be expressed as a lattice, ordered by set containment, as in Figure 3.3. In this lattice, $\top$ represents "no expression", and $\bot$ represents "all expressions". The class containing wrap-around variables is represented below all other classes except $\bot$, since in the limit, a wrap-around variable can be cascaded through an infinite set of values, and therefore represent any sequence of n values in a loop. When certain sequences degenerate to simpler forms, the classification of the sequence variable can be strengthened. This will cause the classification to rise in the lattice.

3.1.4 Control flow graphs

A *basic block* is a set of instructions in the program where the flow of control enters at the first instruction and exits only at the last instruction. A *control flow graph* (CFG) is a directed graph where the nodes are the basic blocks representing the program with the edges representing jumps from one block to another. The CFG also has an *entry* node and an *exit* node, with an edge from the entry node to any block at which the program can be entered and an edge from any block at which the program can be exited to the exit node. To take into account the possibility of the program not being run, there is an edge from the

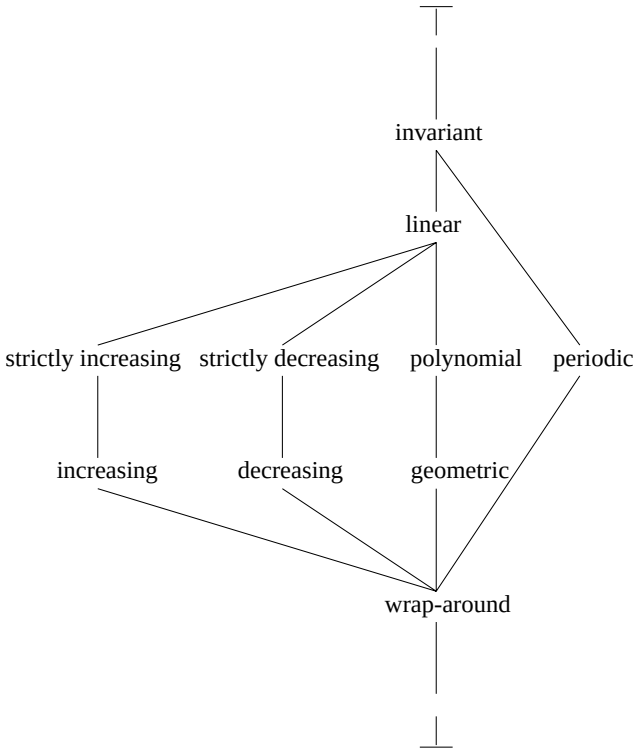
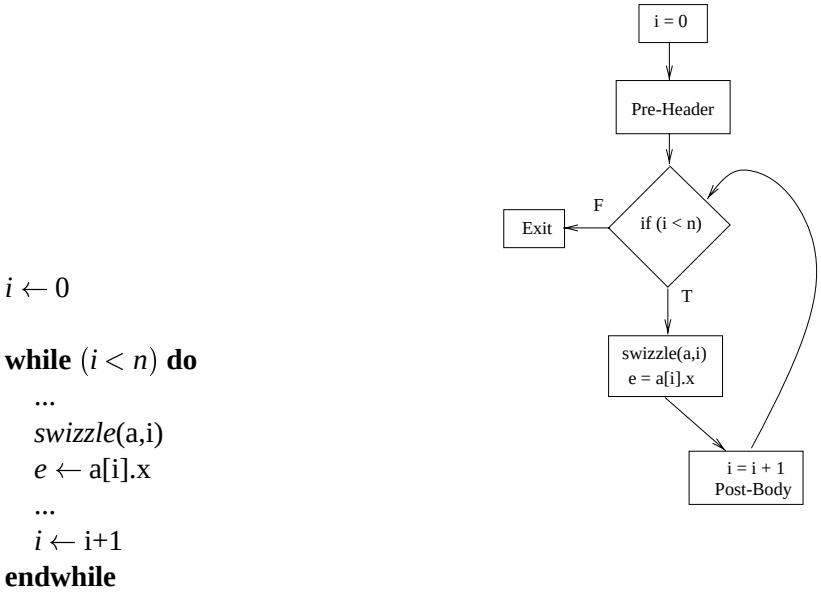


Figure 3.3: Sequence variable lattice.

entry node to the exit node. We say a node *x dominates* node *y*, if all paths from the entry node to *y* contain *x*.

3.1.5 Loops and loop inversion

A *loop* is a strongly connected component of the CFG. The *loop header* is the block within the loop that dominates all other blocks in the loop. When hoisting *loop invariant code* out of loops, care must be taken to hoist it to a position where it will be executed only if the loop is executed. Several loop transformations provide safe places to hoist such code. The first inserts a new block called the *pre-header*, which has an edge going out only to the header and all the edges which formerly entered the header from *outside* the loop instead enter the pre-header. Similarly, a *post-body* block can be inserted, which has an edge going out only to the header and all the edges which formerly entered the header from *inside* the loop instead enter the post-body block. The second transformation, *loop inversion*,



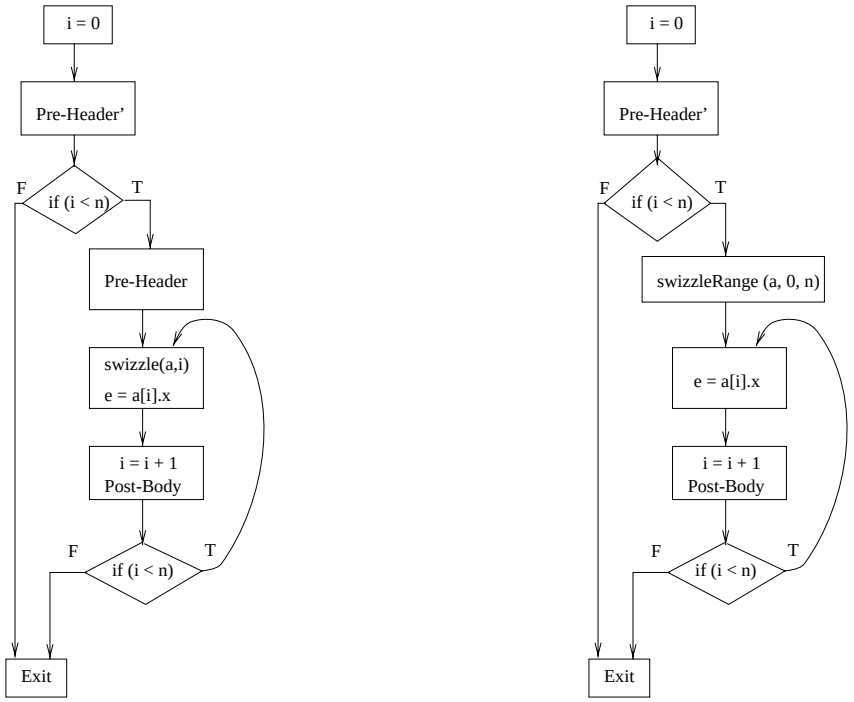
(a) A loop (b) Its CFG representation

Figure 3.4: Loop representation in the CFG

amounts to converting each `while` loop into a `do-while` loop. For example, looking at the loop in Figure 3.5(a), which is the inverted form of the loop in Figure 3.4(b), the loop invariant code can be hoisted out into the *Pre-Header* protected by the `if` statement. As previously explained in Section 2.5.2, in this loop, the *swizzle barrier* can be hoisted out as a *swizzleRange* as shown in Figure 3.5(b).

3.1.6 SSA form

Static single assignment (SSA) is a program representation which provides a compact form of variable definition and use information. In this form, each use of a program variable has exactly one corresponding reaching definition. Where distinct definitions of a variable merge at confluence points in the CFG, operators called ϕ-functions are introduced to merge each of the reaching definitions at that point. The ϕ-function in turn serves as a definition point. Unique definitions of a variable are represented by subscripting. A



(a) Inverted loop (b) After hoisting

Figure 3.5: Loop inversion and Hoisting

loop and its corresponding SSA form are shown in Figure 3.6. We use the SSA form of program representation in our induction variable analysis.

3.1.7 Demand-driven SSA graph

Our induction variable analysis framework is based on the demand-driven SSA representation of the CFG. Instead of the traditional *definition-use* chains [Aho et al. 1986], demand-driven SSA form uses *factored use-definition* (FUD) chains [Stoltz et al. 1994; Wolfe 1996]. In this format, uses and ϕ-functions have pointers to the corresponding definition of the variable. For the purpose of recognizing induction variables, merge operators that occur at loop headers need to be distinguished from those that occur as a result of forward branching. Within loop headers, merges of multiple definitions of a variable are

```
i ← 0

loop:
  if (i ≥ n) then
    break
  ...
  ...
  i ← i + 1
endloop
```

(a) A loop

```
i0 ← 0

loop:
  i1 ← φ(i0, i2)
  if (i1 ≥ n0) then
    break
  ...
  ...
  i2 ← i1 + 1
endloop
```

(b) Its SSA representation

Figure 3.6: Loop representation in the SSA form

handled by μ-functions instead of ϕ-functions. The semantics of the μ are essentially the same as the ϕ, with two differences:

- The arity of a μ-function is always two since pre-header and post-body blocks are added to each loop as described in Section 3.1.5.
- One of the reaching definitions at the μ will always be from within the body of the loop (the *internal ssalink*) and the other will always be from outside the loop (the *external ssalink*).

The *SSA graph* is an abstraction representing the operations within the SSA form of the program. The CFG and SSA graphs for the loop in Figure 3.6(b) are shown in Figure 3.7. The use-definition chain form, as opposed to the traditional definition-use chain form, finds the reaching definition at a given use by following the links from the use's node backward, against the data flow. On a recursive traversal of the SSA graph, each use *demands* the value of the earlier definition. We use this property in our demand-driven induction variable analysis.

3.2 PRE over access path expressions

Our analysis and optimization framework revolves around partial redundancy elimination over pointer expressions that access persistent objects [Hosking et al. 1998]. We adopt

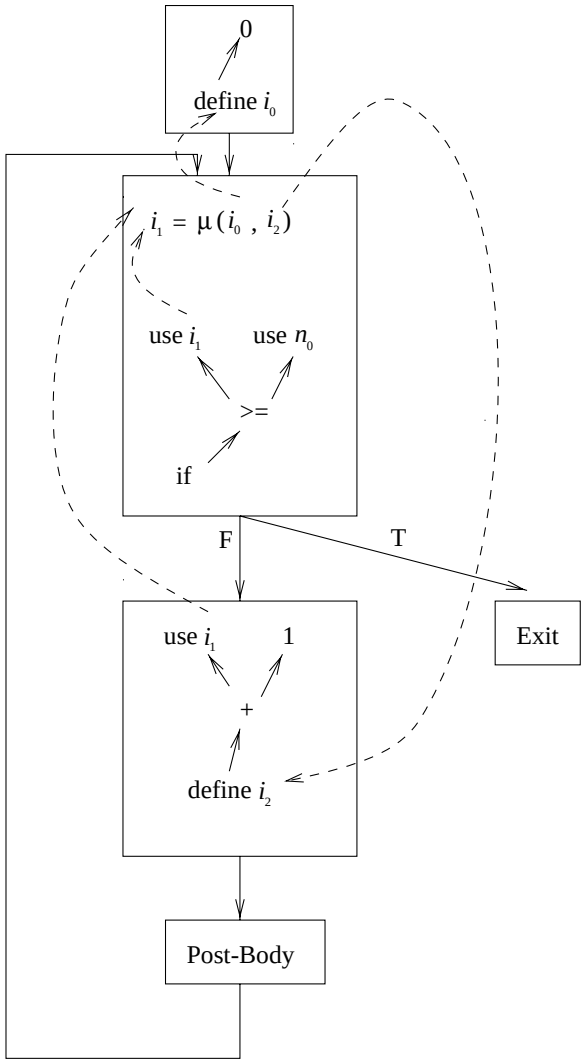


Figure 3.7: Demand-driven SSA graph

standard terminology and notations used in the specification of the Java programming language to specify the analysis and optimization problem.

3.2.1 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 *pointers* to

these objects, as well as 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 3.1 summarizes the two kinds of access expressions in Java. We 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.

Table 3.1: 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

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 compatible 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 $Subtypes(T)$ to denote all subtypes of type T. Thus, an access path p can legally access objects of type $Subtypes(Type(p))$. 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.

3.2.2 Barrier optimizations

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 $a.b[i].c$ requires 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. Referring to the ith component of b must be protected by a swizzle barrier. If the expression appears as the target of an assignment, then the object referred to by $a.b[i]$ also requires 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 (as defined previously in Table 2.1) to the specification of access expressions given in Table 3.1. For each source code access expression Table 3.2 gives

[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.

the form of the corresponding explicit-barrier IR. Here we assume that arrays are always swizzled lazily.

Table 3.2: Intermediate representation for access expressions

Source	Intermediate representation	
	Read access	Write access
$p.\mathtt{f}$	$read(p).\mathtt{f}$	$write(read(p)).\mathtt{f}$
$p[i]$	$(t = read(p); swizzle(t,i); t[i])$	$write(read(p))[i]$

3.3 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 $TypeDecl(p,q)$ holds, defined as:

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

A more precise alias analysis will distinguish accesses to fields that are the same type yet distinct. This more precise relation, $FieldTypeDecl(p,q)$, is defined by induction on the structure of p and q in Table 3.3. Again, two access paths p and q may be aliases only if the relation $FieldTypeDecl(p,q)$ holds. It distinguishes accesses such as $t.\mathtt{f}$ and $t.\mathtt{g}$ that *TypeDecl* misses. The cases in Table 3.3 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
4. Two array accesses are aliases if they may access the same array (the subscript is ignored)
5. For all other pairs of access expressions they are aliases if they have common subtypes

Table 3.3: *FieldTypeDecl*(AP_1, AP_2)

Case	AP_1	AP_2	*FieldTypeDecl*(AP_1, AP_2)
1	p	p	true
2	$p.\mathtt{f}$	$q.\mathtt{g}$	$(f = g) \wedge$ *FieldTypeDecl*(p, q)
3	$p.\mathtt{f}$	$q[i]$	false
4	$p[i]$	$q[j]$	*FieldTypeDecl*(p, q)
5	p	q	*TypeDecl*(p, q)

[Diwan et al. 1998] further refines 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 in Cutts and Hosking [1997].

3.4 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 reevaluation can be eliminated and replaced instead with a use of the precomputed value. This is illustrated in Figure 3.8. In Figure 3.8a, 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 3.8b, $a+b$ need only be evaluated once along any path through the program, rather than twice as before.

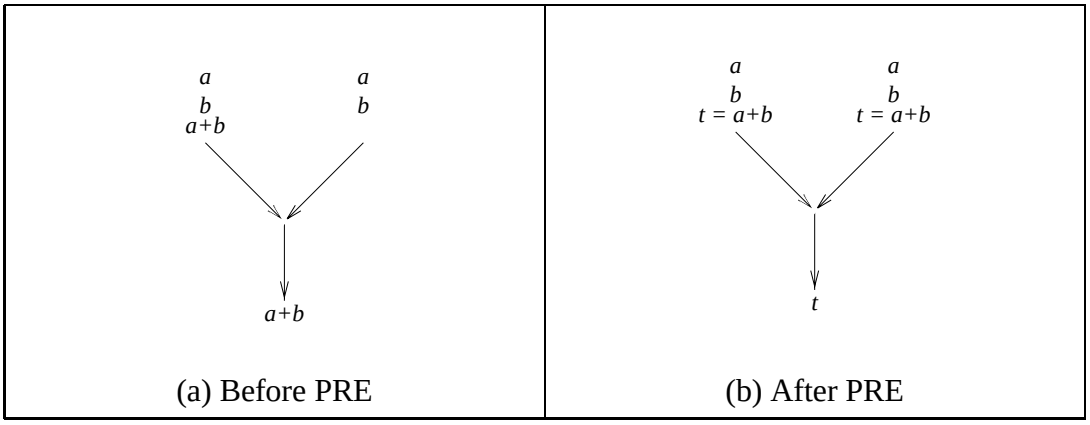


Figure 3.8: PRE for arithmetic expressions

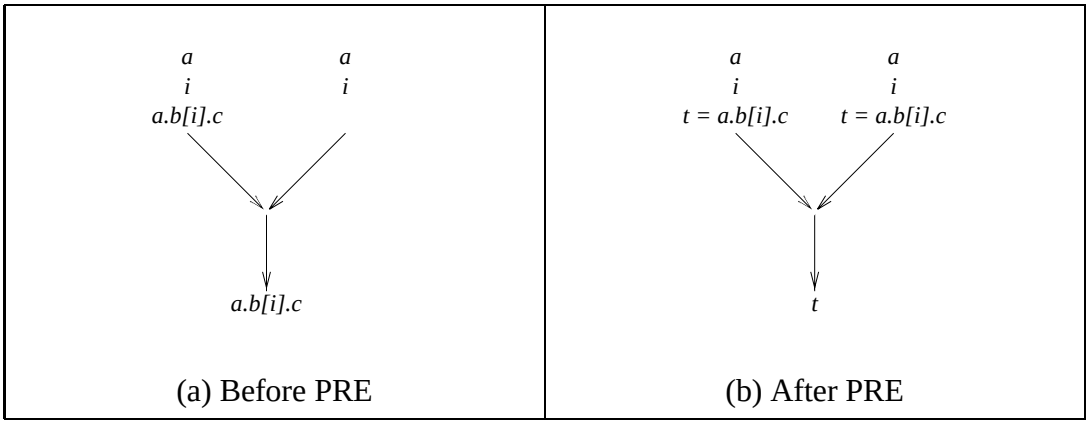


Figure 3.9: PRE for access expressions

Traversing an access path requires successively loading the pointer at each memory location along the path and traversing it to the next location in the sequence. 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 3.9. The expression $a.b[i].c$ will be redundant at some subsequent reevaluation 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* or *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. 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 3.10).

Swizzle barriers cannot be treated as expressions in our current implementation since they do not return any value. Thus they do not fit easily into this framework and PRE cannot currently eliminate them in the same manner as read and write barriers. But this is of minor consequence since only a small percentage of swizzle barriers are inherently redundant in typical programs. To eliminate a significant percentage of swizzle barriers we need to make them redundant by range swizzle optimizations using DIVA.

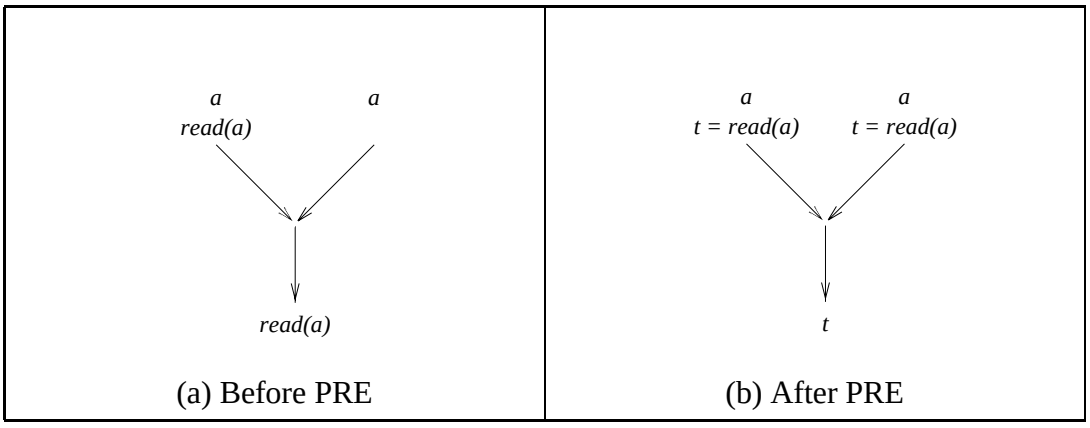


Figure 3.10: PRE for barrier expressions

3.4.1 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 and DIVA, 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, write and swizzle barriers for orthogonal persistence do not have side-effects that are relevant to source-level program semantics, so their motion is unconstrained.

3.5 Demand-driven induction variable analysis (DIVA)

We describe our induction variable analysis and optimization framework in this section. The technique presented here is based on Factored Use-Def (FUD) chains [Stoltz et al. 1994; Wolfe 1996], a demand-driven representation of the popular Static Single Assignment (SSA) form. In this form, strongly connected components of the associated SSA graph correspond to sequences in the program [Gerlek et al. 1995].

3.5.1 Cycles in SSA graphs

Observe the SSA representation of i in Figure 3.6(b) and in Figure 3.7. Beginning at the μ defining i_1, the *external ssalink* defines the value of i_1 on the first iteration of the loop. On subsequent iterations the value of i_1 is defined by the *internal ssalink* to the definition of i_2 at the statement $i_2 \leftarrow i_1 + 1$. This statement in turn obtains the value of i_1 from the μ above. Thus these edges form a cycle which represents the *flow* of i around the loop. The variable i is now identified as a sequence variable since it is defined as a function of itself on a previous iteration. Also, we can define the sequence expression for i as a linear

function of the basic loop counter, h. The variable i_2 is equal to $h+1$, which gives us the sequence expression.

3.5.2 Detecting sequences

Determining symbolic expressions for sequence variables is a two step process:

1. The sequence variables are found by partitioning a graph representation of the program in SSA form into *strongly connected components*.

2. The nodes in each component (sequence) are assigned symbolic expressions describing the sequence form, such as the closed forms in terms of the loop counter h.

Each strongly connected component (SCC) corresponds to a loop-invariant value (viewed as a trivial sequence), a proper sequence form (one of the types described in Section 3.1.3) or an unknown sequence form.

The sequence type and expression for a given component are dependent on the sequence types and expressions of those variables they use. Thus any given component will first *demand* the classification of any components it requires for its own classification. This demand-driven process is accomplished by using Tarjan's algorithm for detecting SCCs in directed graphs [Tarjan 1972]. This algorithm has the property that SCCs are visited only after visiting all descendant components in the graph; thus, a directed acyclic graph of components is formed and processed in postorder during a depth-first traversal. The working of the algorithm is described next.

3.5.3 Tarjan's algorithm

This section describes the process of detecting strongly connected components in the SSA graph. Each node, t, in the graph contains:

Type: the type of the operation

Lowlink: used within the algorithm

Status: one of (`notyet`, `onstack`, `done`)

```
do
  FindSCC(loop)
with
  procedure FindSCC(l) begin
    Number ← 0
    StackTop ← ∅

    for each t ∈ l.operations do
      t.Status ← notyet
    for each t ∈ l.operations do
      if (t.Status = notyet) then
        visitNode(t,l)
```

Figure 3.11: SCC classification

HasLeft, HasRight, HasSSA: true if the node has those fields.

The algorithm uses procedure *FindSCC*, called for each loop in the program, to visit each node in the loop. The procedure *visitNode* first visits all of a node's SSA graph successors and then processes that node. Depending on whether the node is a trivial component or the root of a SCC, one of two classification procedures are called. The procedure *visitDescendent* allows the treatment of a value from outside the current loop as invariant and also terminates the recursive depth-first traversal. These procedures are shown in Figures 3.11- 3.13.

3.6 Range swizzle optimizations with DIVA

Our goal is to reduce the number of swizzle barriers executed. As explained in Section 2.5.2, we need to find the bounds of an induction variable that is being used to traverse a given loop in the program. For this purpose, we have to classify sequences after their identification. The Classification proceeds as follows.

```
do
  visitNode(loop, node)
with
  procedure visitNode(l, t) begin
    t.Status ← onstack
    Number ← Number + 1
    low ← Number
    this ← Number
    pushStack(t)

    if (t.HasLeft) then
      low ← min(low, visitDescendent(t.Left, l))
    if (t.HasRight) then
      low ← min(low, visitDescendent(t.Right, l))
    if (t.HasSSA) then
      low ← min(low, visitDescendent(t.SSA, l))

    t.Lowlink ← low
    if (this ≠ low) then
      return
    if (StackTop = t ∧ t.Type ≠ μ) then
      ClassifyTrivial(t,l)
      popStack()
      t.Status ← done
    else
      Component ← ∅
    do
      StackTop ← popStack()
      StackTop.Status ← done
      Component ← Component ∪ StackTop
    while (StackTop ≠ t)
    ClassifySequence(Component, l)
```

Figure 3.12: SCC classification(cont.).

3.6.1 Classifying sequences

Once an SCC and its associated sequence variable have been identified, the sequence can be classified into one of linear, polynomial, geometric, wrap-around, periodic, monotonic or unknown. The conditions for determining the classification are shown in Table 3.4. If none of those conditions apply then the sequence is considered to be unknown and assigned class $\perp$. Also, the criteria for the linear, polynomial, and geometric classes are

```
do
   visitDescendent(loop, node)
with
   procedure visitDescendent(l, t) begin
      if (contains(l, t.Loop)) then
         return (Number)
      if (t.Status = notyet) then
         visitNode(t,l)
         return (t.Lowlink)
      else if (t.Status = onstack) then
         return (t.Lowlink)
      return (Number)
```

Figure 3.13: SCC classification (cont.).

sequence class	μ	ϕ	*arith*
linear	1	0	>0
polynomial	1	0	>0
geometric	1	0	>0
wrap-around	1	0	0
constant periodic	>1	0	0
nonconstant periodic	>1	0	>0
monotonic	1	>0	≥ 0

Table 3.4: Classification of SCCs based on frequency of operations

identical. They are distinguished by examining their operands. For example, linear sequences can be identified if the operations in the component consist of uses, definitions and additions or subtractions of loop-invariant values or other linear variables. The SCC defining a linear sequence will be a simple cycle, since the induction variable may only appear once on the right-hand side of the expression.

3.6.2 Well-behaved loops

For our optimizations we consider the class of *well-behaved loops* [Muchnick 1997]. With reference to the loop in Figure 3.14, a well-behaved loop is one in which exp_1 assigns a value to an integer-valued variable i, exp_2 compares i to a loop constant, exp_3 increments or decrements i by a loop constant, and *stmt* contains no assignments to i. Other loops

for $(exp_1; exp_2; exp_3)$ **do**
stmt

Figure 3.14: A well-behaved loop

like `while` and `do-while` loops which follow the same semantics as the `for` loop in Figure 3.14 are also considered to be well-behaved.

3.6.3 Hoisting swizzle barriers

To hoist out swizzle barriers from loops, all the strongly connected components in the program are determined. Trivial components which are loop-invariant are excluded. Components which represent well-behaved loops are recognized and the induction variable *i* is identified. As explained previously in Section 3.5.1, the external ssalink of the μ-function in the loop header provides the expression *init* which was assigned to *i* outside the loop. By recognizing the condition which terminates the loop, the expression *term* which is the last value assigned to *i* can be found. If the loop is traversing an array, a range swizzle instruction with the range [*init*, *term*] can be inserted into the pre-header as shown in Figure 3.5(b). Any swizzle barrier using *i* to swizzle a component of the array within the body of the loop is thus made redundant and can be removed from the program.

4 IMPLEMENTATION

4.1 The PJama persistent system

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 Virtual Machine (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.

PJama implements persistence by extending the standard Java VM with an *object cache*, which is a cache of persistent objects in virtual memory. The interpreter can access persistent objects (which include *class instances*, *methods* and *classes*) in the object cache through handles in the *resident object table* (ROT). The ROT is hashed on PID to speed up searches. The ROT handles are similar to Java's handles and these objects have the same format as objects in Java's garbage collected heap. Also persistent meta-data like classes have exactly the same format in virtual memory as their Java counterparts. The object cache implements object-faulting, update-tracking and object replacement.

4.1.1 Architecture of PJama

The PJama architecture consists of three main modules: the object cache manager, the Java VM and the buffer manager. Figure 4.1 shows a simplified view of the PJama system architecture. To illustrate the various possible combinations of inter-object references, the following objects are depicted in Figure 4.1:

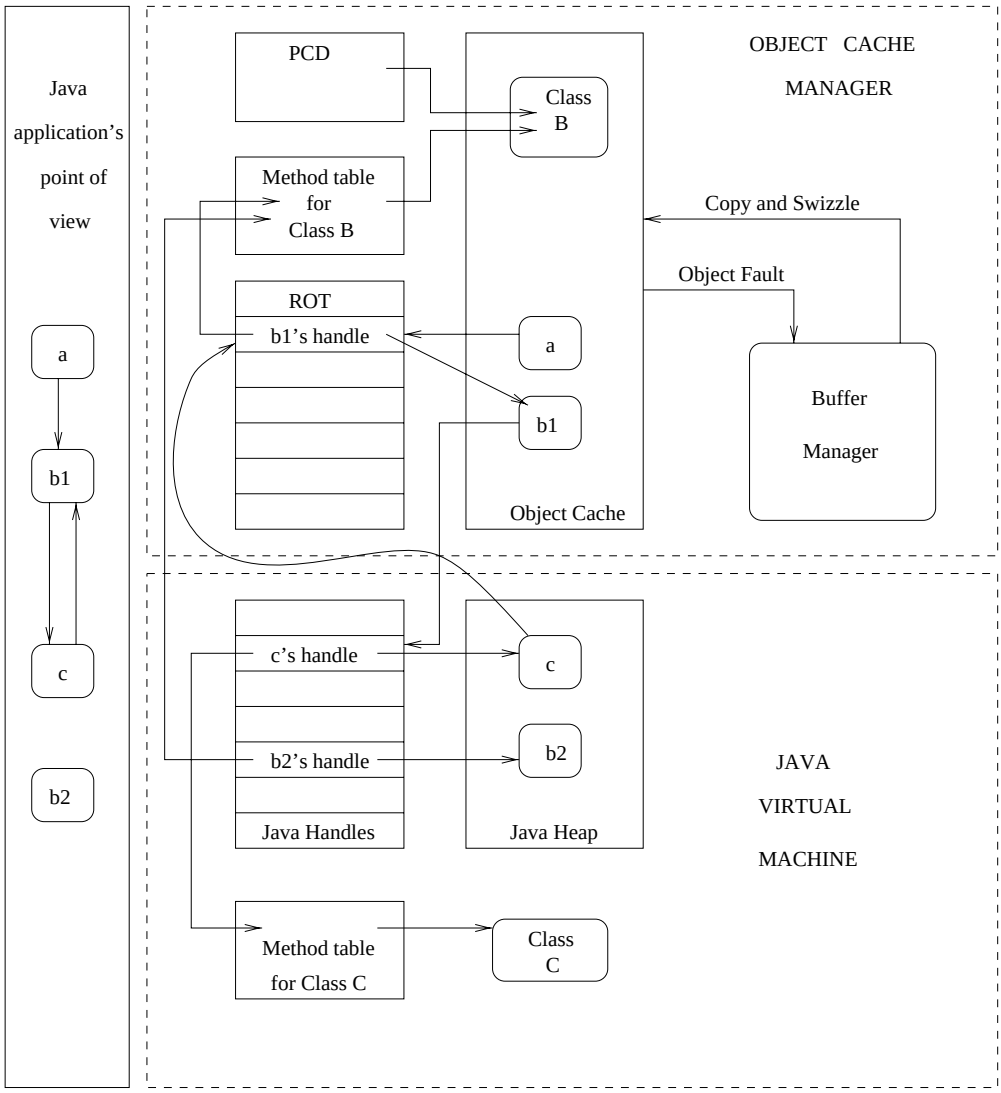


Figure 4.1: PJama's Object Cache Architecture

1. A transient instance of a transient class (the object `c`, instance of `C`) referring to object `b1`.

2. A persistent instance of a persistent class (the object `b1`, instance of `B`) referring to object `c`.

3. A transient instance of a persistent class (the object `b2`, instance of `B`).

4. A persistent instance referring to another persistent instance (the object `a` pointing to `b1`).

From the Java application's point of view all these objects appear to refer to one another in a natural manner. The underlying mechanisms for accessing persistent objects is totally hidden from the programmer. In essence, the operation of PJama consists of copying all the objects needed by the running application into a virtual memory area distinct from the buffer pool used by the stable store. This virtual memory area is called the *object cache*. The object cache implements an object-fault mechanism to load-on-demand any objects required by the application. This process is illustrated in Figure 4.1. The object cache also manages a memory-resident copy of the *persistent class directory* (PCD) of the store. When PJama is initialized with a valid store name, the first thing it does is to load the store's PCD into memory, and reroute to the PCD all searches for a class by the JVM.

When an object fault occurs, the object cache manager asks the buffer manager for a copy of the missing object. If the page containing the object is not found in the buffer pool, a page-fault is raised. After the page fault is serviced, the buffer manager provides the object cache manager with a pointer to the object, which is then used to copy it into the object cache. During this process of copying, any object references in the object just faulted in are converted from a persistent identifier (or PID) format into a virtual memory format(a pointer to a handle). This is called *pointer swizzling*.

4.1.2 Swizzling in PJama

PJama currently adopts an *eager* pointer swizzling strategy for objects. The current implementation of this strategy is similar to the node marking scheme described in Section 2.3.2. In PJama eager swizzling means that upon object fault-in, persistent identifier fields in that object are overwritten with pointers to handles that will eventually contain a virtual memory pointer to the actual location of the referenced object. Thus handles exist in one of several possible states including:

1. *Indirect*: The handle is an indirect block containing a virtual memory pointer to the resident target object.
2. *Fault*: The handle is a fault block, containing the target object's PID.

Since object references in the object cache can be in one of many states, every object access (i.e., dereference through a handle) must be protected by a *read barrier*. The read barrier makes sure that the target object is resident (i.e.,the handle is in the indirect state). If not, then an object fault is triggered to obtain the object identified by the PID stored in the handle, and the handle is converted from fault to indirect. One of the goals of this work is to eliminate redundant read barriers where the check will always succeed (i.e., where the handle is guaranteed to be indirect).

4.1.3 Array swizzling

When an object is faulted in and swizzled, fault blocks are allocated in the ROT for all persistent object references in that object. In PJama, the only exception to this *eager swizzling* approach is for arrays of references, whose contents are left in PID form to avoid the overhead of allocating a fault block for every element in the array. In this *lazy swizzling* approach, array references are swizzled upon the first use of that reference. As described in Section 2.3.1, lazy swizzling requires dynamic swizzle barriers. The only place where such swizzle barriers are needed in PJama is in the `aaload` instruction, which reads an object reference from an array and pushes the reference onto the Java stack. This instruction embodies a swizzle barrier in the current version of PJama. Our new approach is to expose this swizzle barrier by inserting a new internal swizzle bytecode before every occurence of the `aaload` bytecode. This allows DIVA to identify and hoist such swizzle bytecodes outside loops, thereby decreasing the iterative overhead of swizzle checks. The hoisted range swizzle is designed to swizzle a specified range of the array as determined by analysis.

4.2 Implementation of persistence optimizations

Persistence optimizations have been implemented using bytecode-to-bytecode class transformation that applies type-based alias analysis and access path PRE to Java classes.

This transformation removes redundant read and write barriers. Range swizzle optimizations are also performed by applying the DIVA technique. The optimized classes are targeted for execution on a modified version of the PJama [Atkinson et al. 1996] virtual machine.

4.2.1 Bytecode-to-bytecode 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. Conforming 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.

This bytecode-to-bytecode class transformation that performs PRE for access expressions in Java is implemented in BLOAT (for *B*ytecode-*L*evel *O*ptimization and *A*nalysis *T*ool)[Nystrom et al. 1998]. It 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; Stoltz et al. 1994; 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. In this thesis we extend BLOAT to recognize and eliminate redundant read and write barriers. Also, the DIVA technique has been added as a separate pass over the control-flow graph, just before the final code generation phase, to hoist swizzle checks out of loops.

Table 4.1: Bytecodes requiring barriers

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

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

4.2.2 Read and write barrier optimization for PJama

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 4.1. 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 read and write 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 4.1. 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 read and write barrier bytecodes are specified in Table 4.2.[1] 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 to the call. Thus, the initial preprocessing to insert barriers needs no expensive analysis.

Table 4.2: New read and write barrier bytecodes

Operation	read barrier	write barrier
Format	*read* *index*	*write* *index*
Forms	*read* = 233 (0xe9)	*write* = 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.

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

[1]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 work so as to demonstrate the full potential for optimization of write barriers.

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 4.1, 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.

4.2.3 Range swizzle optimization for PJama

In the current release of PJama the `aaload` instruction has a swizzle barrier built into it. In line with our optimization strategy, we elide the swizzle barrier from the `aaload` instruction and extend the PJama VM with two more internal swizzle bytecodes. The *aswizzle* bytecode swizzles a single reference element while the *aswizzleRange* bytecode swizzles a specified range of reference elements in an array. The details of the bytecodes can be found in Table 4.3. The same preprocessing step used to insert read and write barrier opcodes is used to insert an *aswizzle* bytecode before each `aaload` instruction. The *aswizzle* uses a copy of the operands of the `aaload`. After DIVA deduces loop bounds, an *aswizzleRange* can be inserted outside the loop and redundant *aswizzles* inside the loop can be safely eliminated.

4.2.4 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 all 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 pinning thread resumes execution. Dirty bits set on objects in the

Table 4.3: New swizzle barrier bytecodes

Operation	swizzle reference from array	swizzle range of references from array
Format	*aswizzle*	*aswizzleRange*
Forms	*aswizzle* = 236 (0xec)	*aswizzleRange* = 237 (0xed)
Stack	..., arrayref, index => ...	..., arrayref, start, end => ...
Description	The *arrayref* must be of type `reference` and must refer to an array whose components are of type `reference`. The *index* must be of type `int`. If the *arrayref* is not null, then the element at *index* is swizzled, if not already. *arrayref* and *index* are popped from the operand stack.	The *arrayref* must be of type `reference` and must refer to an array whose components are of type `reference`. The *start* and *end* must be of type `int`. If the *arrayref* is not null, then the elements within the intersection of [start,end] and [0,arraylength] are swizzled, if not already. *arrayref*, *start* and *end* are popped from the operand stack.

cache that are directly referenced from a thread's stack must be maintained, even across stabilizations. Similarly, reference elements in arrays that have been swizzled must remain swizzled. 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].

5 EXPERIMENTS

5.1 Read and write barrier optimization

To evaluate the impact of our read and write barrier optimizations we applied them to the traversal portions of a Java implementation of the OO7 benchmarks [Carey et al. 1993], comparing the number of read and write 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 read and write 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. We also wanted to separate out the effect of our this optimizations (see Section 4.2.2). Thus, we obtain results for four distinct configurations of the OO7 classes which are described in detail in Section 5.1.3.

5.1.1 Benchmarks

The OO7 benchmarks [Carey et al. 1993] are a comprehensive test of object-oriented database performance. They measure the speed of many different kinds of pointer traversals, including traversals over cached data, traversals over disk-resident data, sparse traversals, and dense traversals. The benchmarks also measure the efficiency of many different kinds of updates, including updates to indexed and unindexed object fields, repeated updates, sparse updates, updates of cached data, and the creation and deletion of objects. These operations are performed 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 5.1.

Table 5.1: 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

5.1.2 Metrics

For each combination of benchmark and optimization level we measure the number of read and write barriers 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.

5.1.3 Results

The results of read barrier optimizations are given in Table 5.2 and those of write barrier optimizations are given in Table 5.3. The four different configurations we measured are:

- none: unoptimized with barriers
- access: access path optimizations without barrier optimizations
- access+barrier: access path optimizations with barrier optimizations
- access+barrier+this: this, access path optimizations with barrier optimizations

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

The results reveal 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) cannot significantly reduce barrier overheads.

Table 5.2: Results of read barrier optimizations

	Read barriers executed				
	PRE level				
Traversal	none	access	access+barrier	access+barrier+this	% removed
1	10535707	7899406	3456590	2126883	80
2a	10588195	7951894	3500330	2168436	80
2b	10666927	8030626	3456590	2126883	80
2c	11191807	8555506	3456590	2170623	81
3a	10586008	7949707	3500330	2168436	80
3b	10623187	7986886	3456590	2126883	80
3c	11016847	8205586	3631550	2170623	80
6	3458575	1215934	47057	27363	99

5.2 Range swizzle optimization

5.2.1 Benchmarks

To evaluate the impact of range swizzle optimizations using DIVA, a set of benchmarks which use arrays of objects extensively had to be chosen. With that objective the following applications were chosen:

1. *Linpack*: The standard Linpack suite of applications.
2. *Cholesky*: Set of routines performing Cholesky Decomposition.
3. *Neural*: Back propagation on a multi-layered neural net.
4. *Inversion*: Application performing a series of matrix inversions.

Table 5.3: Results of write barrier optimizations

	Write barriers executed				
	PRE level				
Traversal	none	access	access+barrier	access+barrier+this	% removed
1	495363	495363	404599	404599	18
2a	499737	499737	406786	406786	19
2b	582843	582843	448339	448339	23
2c	845283	845283	448339	448339	47
3a	497550	497550	406786	406786	18
3b	539103	539103	448339	448339	17
3c	670323	670323	448339	448339	33
6	14223	14223	10939	10939	23

5.2.2 Results

The results of range swizzle optimizations are given in Table 5.4. The number of *aswizzle* bytecodes executed in classes that have had them inserted, are under the column heading decorated. The count of *aswizzle* bytecodes executed in classes that have been optimized after being decorated, are under the column heading optimized. The results reveal that DIVA optimizations remove on average 66% of *aswizzles* in the decorated code. Looking at Table 5.4, we observe that the number of new *aswizzleRanges* introduced is on average just 0.9% of *aswizzles* in the decorated code. This demonstrates the effectiveness of range swizzle optimizations to reduce the array swizzle overhead with negligible cost.

5.3 Conclusions

The results shown above demonstrate the effectiveness of the optimization techniques developed in this work. For the OO7 traversal benchmarks these optimizations remove a majority (83%) of read barriers and a significant fraction (25%) of write barriers. For

Table 5.4: Results of range swizzle optimizations

	aswizzles executed			aswizzleRanges executed		
Benchmark	decorated	optimized	% removed	decorated	optimized	% added
Linpack	75365	20217	73	0	304	0.4
Cholesky	921855	256994	72	0	14029	1.5
Neural	6491933	3397983	48	0	36832	0.6
Inversion	2309400	649710	71	0	26020	1.1

the range swizzle benchmarks a major portion (66%) of swizzle barriers are eliminated at negligible cost.

Thus, the major conclusions that can be drawn from this work are:

- 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.
- Demand-driven induction variable analysis is very effective in reducing the array swizzle overheads of orthogonal persistence.

Our results conclusively demonstrate that *the overhead of orthogonal persistence can be reduced significantly by program analysis and optimization*. We believe these techniques will prove crucial to the achievement of respectable performance by persistent systems in general and persistent Java systems in particular.

5.4 Future work

We anticipate several extensions of this work in the domain of orthogonal persistence. These opportunities can be broadly categorized as:

- Optimizations *enabled by* persistence: Analysis, compilation and execution in a persistent setting can open up new avenues for generic program improvement.
- Optimizations *enabling for* persistence: Further reduction of persistence-specific overheads, especially reduction of I/O by clustering and prefetching.

5.4.1 Persistence-enabled optimizations

When it comes to traditional language optimizations there is usually a tradeoff between the *speed* of optimization versus their *effectiveness*. For a language like Java with its dynamic, late-binding, object-oriented nature, several aggressive optimizations can be envisaged as in Self-91 [Chambers 1992]. But due to the fast turnaround time required by "just-in-time" (JIT) compilation, many of these analyses have to be toned down [Hölzle and Ungar 1996]. Yet a persistent environment can provide an appropriate setting for exploiting such aggressive analysis by taking them off-line [Cutts and Hosking 1997]. Also, analysis and optimization phases are greatly simplified when all code, data, profile information, etc., are retained within a persistent store. There are several other interesting opportunities like "whole-program" optimizations [Diwan et al. 1996; Diwan 1997] in a persistent setting and specialization of code with respect to stored data.

5.4.2 Persistence-enabling optimizations

Optimizing the I/O performance of persistent systems is a challenge that is yet to be addressed adequately. We believe that a combination of *clustering* and *prefetching*, driven by information provided by *dynamic profiling*, can lead to significant I/O performance gains. There is also a strong connection between these techniques and read, write and swizzle barrier optimizations presented in this work.

To summarize, future work will be driven by the unique setting provided by persistence for optimizations, both persistence-enabled and persistence-enabling. Our goal is to unify these approaches into a common framework of program analysis and execution profiling.

BIBLIOGRAPHY

BIBLIOGRAPHY

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.

AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley.

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

ATKINSON, M. P. AND MORRISON, R. 1995. Orthogonally persistent object systems. *Int. J. 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. To appear; 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. *ACM SIGMOD Record 22,* 2 (June), 12–21.

CHAMBERS, C. 1992. The design and implementation of the SELF compiler, an optimizing compiler for object-oriented programming languages. Ph.D. thesis, Stanford University.

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. *ACM SIGPLAN Notices 32,* 5 (June), 273–286.

COPELAND, G. AND MAIER, D. 1984. Making Smalltalk a database system. In Proceedings of the ACM International Conference on Management of Data (Boston, Massachusetts, June). *ACM SIGMOD Record 14,* 2, 316–325.

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.

CUTTS, Q., LENNON, S., AND HOSKING, A. L. 1998. Reconciling buffer management with persistence optimizations. Submitted to the Eighth International Workshop on Persistent Object Systems.

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. *Transactions on Programming Languages and Systems 13,* 4 (Oct.), 451–490.

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 Fancisco, California, June). *ACM SIGPLAN Notices 27,* 7 (July), 273–282.

DIWAN, A., MOSS, J. E. B., AND MCKINLEY, K. S. 1996. Simple and effective analysis of statically-typed object-oriented programs. In Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (San Jose, California, Oct.). *ACM SIGPLAN Notices 31,* 10 (Oct.), 292–305.

DIWAN, A. S. 1997. Understanding and improving the performance of modern programming languages. Ph.D. thesis, University of Massachusetts at Amherst.

GARTHWAITE, A. AND NETTLES, S. 1996. Transactions for Java. See PJ1 [1996].

GERLEK, M. P., STOLTZ, E., AND WOLFE, M. 1995. Beyond induction variables: detecting and classifying sequences using a demand-driven SSA form. *Transactions on Programming Languages and Systems 17,* 1 (Jan.), 85–122.

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

HÖLZLE, U. AND UNGAR, D. 1996. Reconciling responsiveness with performance in pure object-oriented languages. *Transactions on Programming Languages and Systems 18,* 4 (July), 355–400.

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], 17–127.

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.). *27,* 5 (Dec.), 106–119.

HOSKING, A. L., NYSTROM, N., CUTTS, Q., AND BRAHNMATH, K. J. 1998. Optimizing the read and write barrier for orthogonal persistence. Submitted to Eighth International Workshop on Persistent Object Systems: Design, Implementation and Use, Tiburon, California.

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

LAMB, C., LANDIS, G., ORENSTEIN, J., AND WEINREB, D. 1991. The objectstore database system. *Communications of the 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). 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. *Communications of the ACM 22,* 2 (Feb.), 96–103.

MOSS, J. E. B. 1992. Working with persistent objects: To swizzle or not to swizzle. *IEEE Transactions on Software Engineering 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.

MOSS, J. E. B. AND HOSKING, A. L. 1996. Approaches to adding persistence to java. See PJ1 [1996], 1–6.

MUCHNICK, S. S. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann.

NYSTROM, N., HOSKING, A. L., CUTTS, Q., AND DIWAN, A. 1998. Partial Redundancy Elimination for Access Path Expressions. Submitted to OOPSLA'98.

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

PJ1 1996. Proceedings of the First International Workshop on Persistence and Java. Tech. Rep. 96-58, Sun Microsystems Laboratories. Nov.

PLDI 1998. *Proceedings of the ACM Conference on Programming Language Design and Implementation* (Montreal, Canada, June). *ACM SIGPLAN Notices 33*, 5 (June).

RICHARDSON, J. E. 1990. Compiled item faulting: A new technique for managing I/O in a persistent language. See Dearle et al. [1990], 3–6.

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.

STOLTZ, E., GERLEK, M. P., AND WOLFE, M. 1994. Extended SSA with factored use-def chains to support optimization and parallelism. In *Proceedings of the 27th Annual Hawaii International Conference on System Sciences*. 43–52.

TARJAN, R. E. 1972. Depth-first search and linear graph algorithms. *SIAM Journal of Computing 1*, 2 (June), 146–160.

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.

WILEDEN, J. C., KAPLAN, A., MYRESTRAND, G. A., AND RIDGWAY, J. V. 1996. Our SPIN on persistent Java: The JavaSPIN approach. See PJ1 [1996].

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.). 364–377.

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