

Swizzle barrier optimizations for orthogonal persistence in Java

Kumar Brahnmath¹

Nathaniel Nystrom¹

Antony Hosking¹

Quintin Cutts²

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

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

June 5, 1998

Abstract

Swizzling refers to the translation of object references from an external, persistent format to an internal, transient format used during application execution. *Eager* swizzling schemes translate all the references contained by objects as they are made resident. *Lazy* swizzling schemes defer translation of references until they are loaded from their container. Eager swizzling has the advantage of presenting a uniformly swizzled representation of references to the execution engine, at the cost of up-front translation of references that may never be used. Lazy swizzling avoids this cost, but requires a run-time check that we call a *swizzle barrier* to detect and convert unswizzled references as they are accessed. Lazy swizzling is most often used in situations where accesses are likely to be sparse and the up-front cost of eager swizzling is prohibitive. For example, large containers, such as arrays, may contain many thousands of references, only a fraction of which are ever actually accessed, let alone used to access their target. Thus, lazy swizzling of arrays makes sense even while other types of objects are eagerly swizzled, in which case every array access must be protected by a swizzle barrier. Many, if not most, of these barriers will occur in the bodies of loops that iterate through the elements of arrays. Here, we describe how to hoist loop-nested swizzle barriers into one inclusive barrier operation that can be performed outside the loop, and which swizzles the subset of array elements accessed in the loop body. Our approach to array swizzle barrier optimization is based on loop *induction variable analysis*. We have implemented this approach for the PJama prototype of orthogonal persistence for Java. In experiments with several benchmark applications our optimizations reduce the number of swizzle barriers executed by an average of 66%.

1 Introduction

Persistent programming languages manage volatile memory as a cache for stable storage and hide the details of stable storage underneath the abstraction of persistence [Atkinson and Morrison 1995]. Orthogonal persistence presents this abstraction for all objects uniformly, regardless of their *type*. Object references in stable storage are represented as some sort of *persistent identifier* (PID). To minimize the

costs of frequent translation from PIDs to in-memory pointers, persistent systems may convert inter-object references from their PID format to a more efficient internal representation that caches the resulting translation for future use. Conversion of PIDs in this way is termed *swizzling* [Moss 1992]. Thus, swizzling has two costs associated with it: *time* required for translation and *space* for caching the translation.

Persistent systems often choose to swizzle *eagerly* and pay the swizzling overhead up-front by translating all PIDs in an object when that object is fetched into volatile memory. In this case compiled code will never see unswizzled references. However, for container objects, such as large arrays, which contain a large number of references that are only sparsely accessed, eager swizzling may prove unnecessarily expensive. Swizzling arrays *lazily* avoids the up-front overhead by deferring conversion of array elements until they are accessed. In this case, since array elements are not always swizzled, every access to an array element requires a run-time check that we call a *swizzle barrier* to detect and convert unswizzled references as they are accessed. Subsequent accesses continue to incur the cost of the barrier.

Orthogonal persistence induces additional barriers on object accesses. Since a given reference may point to a transient object, a resident persistent object or a non-resident persistent object, every access must also check that the target of the access is resident. A *read barrier* checks to see if the object is resident, and retrieves it from stable storage if not. Similarly, updates require a *write barrier* to mark the object as modified for subsequent transfer back to stable storage. Our related paper [Hosking et al.] considers the issue of optimizing read and write barriers via partial redundancy elimination over access expressions.

Our focus in this paper is the removal of redundant array swizzle barriers. Since arrays are typically accessed in loops, many swizzle barriers occur in the bodies of these loops. Our optimization approach is to expose and hoist swizzle barriers out of loop bodies in the form of a single operation that swizzles the entire range of references at once, before the loop is entered. The transformation is driven by *induction variable analysis* to determine the upper and lower bounds of the loop index variable as well as a closed form expression for the value of the induction variable at each iteration of the loop.

2 Analysis and optimization

This section describes our analysis and optimization framework for array swizzle barrier optimizations, adopting standard terminology and notations used in the specification of the Java programming language to specify the analysis and optimization problem, and giving sufficient background to understand the approach.

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 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 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

2.2 Barriers

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

Table 2: 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 $[i, j]$

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 to 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 Section 3, which presents our implementation for PJama. Cutts et al. [1998] consider the issue from the perspective of the run-time system.

Our goal is to avoid applying read, write and swizzle barriers to accesses where program analysis shows that the barrier is redundant. We eliminate redundant read and write barriers by partial redundancy elimination [Morel and Renvoise 1979] over access path expressions [Nystrom et al. ; Hosking et al.]. In this paper we describe an approach that eliminates loop-nested swizzle barriers based on analysis of loop induction variables. Before we can eliminate redundant barriers we must make them explicit in the access paths and then apply some definition of redundancy. Making barriers explicit means obtaining for the source code access expression an intermediate representation (IR) in which the barriers are exposed. Optimizations then operate on the IR to remove redundant barriers. Thus, we add barrier expressions to the original specification of access expressions given in Table 1. The specification for barrier expressions appears in Table 2. For each source code access expression Table 3 gives the form of the corresponding explicit-barrier IR.

Table 3: Intermediate representation for access expressions

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

2.3 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 those loop bodies. 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 1(a). Such repeated swizzle barriers are redundant and can be recognized and removed by program optimizations such as partial redundancy elimination over access expressions [Nystrom et al. ; Hosking et al.]. But not all swizzle barriers in the body of such a loop are redundant. For example, in Figure 1(a) analysis may find that the second swizzle barrier is redundant because it is applied to the same element $a[i]$ in both cases and can be removed.

The first swizzle barrier is not redundant and remains a serious overhead to execution of the loop. To remove that swizzle barrier it must be made redundant by performing a range swizzle barrier operation outside the loop. To do this, we need to determine the range of array elements being accessed, so we can swizzle just that range of references before entering the loop. To determine the access range, we must find loops that access arrays of references and determine the *bounds* of each 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 the operation $swizzleRange(a, l, u)$ outside the loop as shown in Figure 1(b). This enables the elimination of swizzle barriers on the components of array a from the body of the loop.

<pre> i ← 1 while i ≤ n do ... swizzle(a, i) e ← a[i].x ... swizzle(a, i) f ← a[i].y ... i ← i + 1 end </pre> (a) Before	<pre> i ← 1 swizzleRange(a, 1, n) while i ≤ n do ... e ← a[i].x ... f ← a[i].y ... i ← i + 1 end </pre> (b) After
--	---

Figure 1: Range swizzle optimization

2.4 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 may be linear, polynomial, geometric, wrap-around, periodic or monotonic [Gerlek et al. 1995]. Detecting linear sequence variables is the first step towards implementing array range swizzle optimizations.

2.5 Loops and loop inversion

A *loop* is a strongly connected component of the control flow graph. The loop *header* is the block within the loop that dominates all other blocks in the loop. When hoisting swizzle barriers out of loops, care must be taken to hoist the range swizzle barrier to a position where it will be executed only if the loop is executed. Several loop transformations performed on the control flow graph (CFG) provide safe places to hoist the barrier. The first inserts a new block called the *pre-header*, which has an out-edge only to the header, and whose in-edges are those that formerly entered the header from *outside* the loop. Similarly, a *post-body* block can be inserted, with an out-edge only to the header, and whose in-edges are those that formerly entered the header from *inside* the loop. The second transformation, known as *loop inversion*, amounts to converting each while loop into a do-while loop. For example, consider the loop in Figure 2(a) and its corresponding control-flow graph in Figure 2(b). Figure 2(c) shows the same loop after pre-header/post-body insertion and inversion. These provide a safe place to hoist the range swizzle check as shown in Figure 2(d).

2.6 SSA form

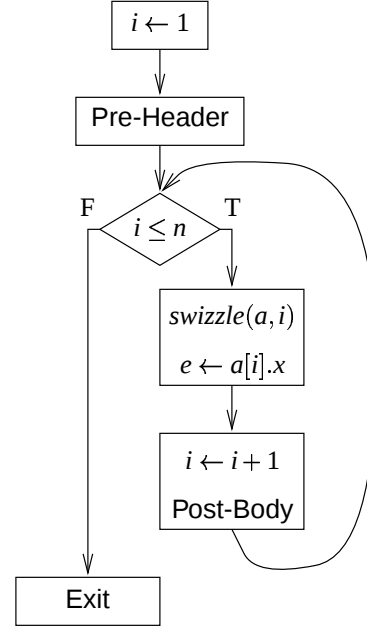
Static single assignment (SSA) form is an intermediate representation that provides a compact form of variable definition and use information. In this form, each use of a program variable has exactly one

```

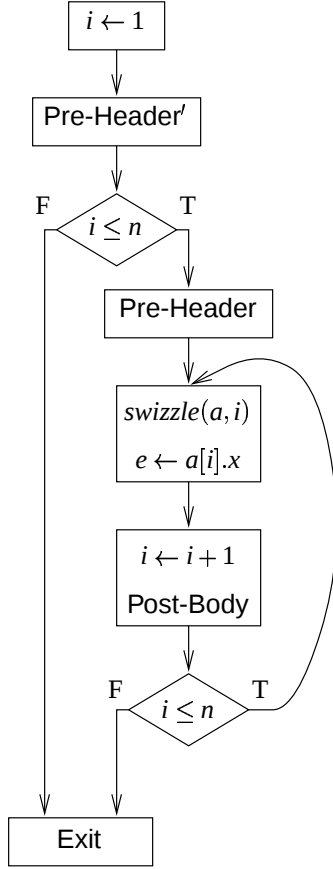
 $i \leftarrow 1$ 
while  $i \leq n$  do
  ...
   $\text{swizzle}(a, i)$ 
   $e \leftarrow a[i].x$ 
  ...
   $i \leftarrow i + 1$ 
end

```

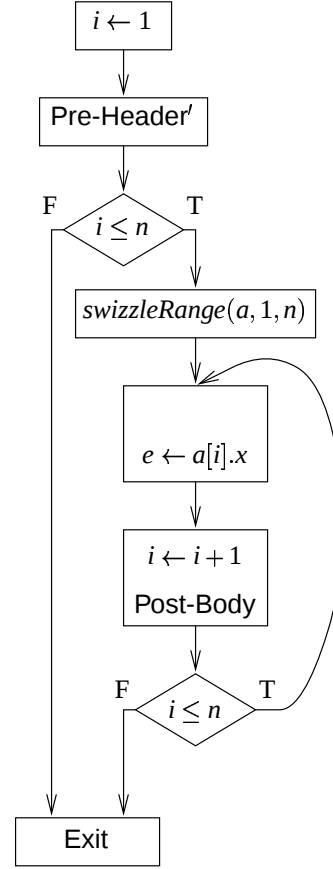
(a) A simple loop



(b) Its CFG



(c) Its CFG after inversion



(d) Its CFG after hoisting

Figure 2: Loop inversion and hoisting

$i \leftarrow 1$	$i_0 \leftarrow 1$
<u>do</u>	<u>do</u>
<u>if</u> $i > n$	$i_1 \leftarrow \phi(i_0, i_2)$
<u>exit</u>	<u>if</u> $i_1 > n_0$
<u>end</u>	<u>exit</u>
...	<u>end</u>
$i \leftarrow i + 1$	...
<u>end</u>	$i_2 \leftarrow i_1 + 1$
(a) A loop	<u>end</u>
	(b) Its SSA representation

Figure 3: Loop representation in SSA form

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 loop and its corresponding SSA form are shown in Figure 3. We use the SSA form of program representation in our induction variable analysis.

2.7 The demand-driven SSA graph

Our induction variable analysis framework is based on the demand-driven SSA representation of the CFG. Instead of the traditional *def-use* chains [Aho et al. 1986], demand-driven SSA form uses *factored use-def* (FUD) chains [Stolz 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 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 2.5.
- 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(b) are shown in Figure 4. The use-def chain form, as opposed to the traditional def-use chain form, finds the reaching definition at a given use by following the links from the use backward, against the data flow. On a recursive traversal of the SSA graph, each use is said to *demand* the value of the earlier definition. We use this property in our demand-driven induction variable analysis.

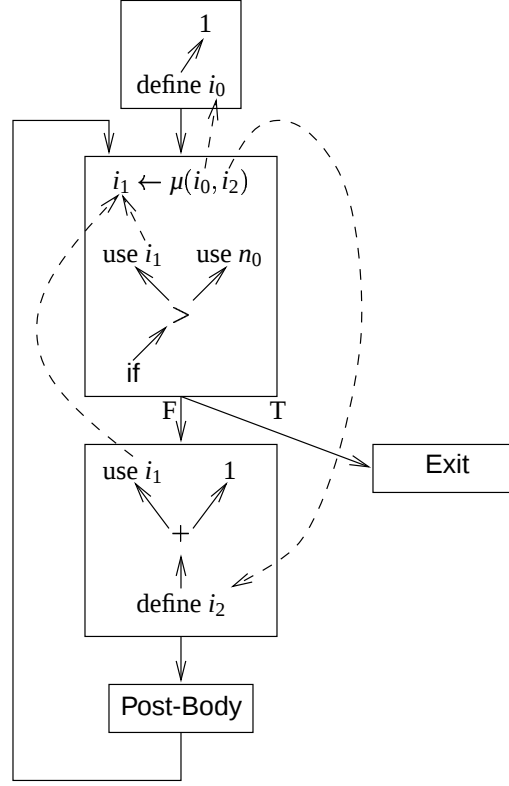


Figure 4: Demand-driven SSA graph

2.8 Demand-driven induction variable analysis

Demand-driven induction variable analysis (DIVA) is based on *factored use-def* (FUD) chains [Stolz et al. 1994; Wolfe 1996], a demand-driven representation of the popular SSA form. In this form, strongly connected components of the associated SSA graph correspond to sequences in the program [Gerlek et al. 1995].

Observe the SSA representation of i in Figure 3(b) and in Figure 4. 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.

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 .

```

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

```

Figure 5: A well-behaved loop

Each strongly connected component (SCC) corresponds to a loop-invariant value (viewed as a trivial sequence), a proper sequence form 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.

Here we consider only the class of *well-behaved loops* [Muchnick 1997]. With reference to the loop in Figure 5, 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 like *while* and *do-while* loops which follow the same semantics as the *for* loop in Figure 5 are also considered to be well-behaved. The induction variables of such well-behaved loops have a linear pattern.

Our goal is to reduce the number of swizzle barriers executed. As explained previously, we need to find the bounds of an induction variable that is being used to traverse a given loop in the program. A sequence variable can be identified as *linear* 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.

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, 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 2(d). 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.

3 Implementation

Our implementation uses bytecode-to-bytecode class transformation to apply the DIVA technique for range swizzle optimizations for execution on a modified version of the PJama [Atkinson et al. 1996] virtual machine.

3.1 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.

We have implemented a bytecode-to-bytecode class transformer that performs partial redundancy elimination over access expressions in Java. Our implementation, called BLOAT (for *Bytecode-Level Optimization and Analysis Tool*) takes compiled Java classes adhering to the Java VM specification and generates transformed classes as output. For each method, BLOAT first builds a control-flow graph, with an expression tree for each basic block, then infers the types of local variables and the operand stack at each point in the code [Palsberg and Schwartzbach 1994], constructs an intermediate representation based on static single-assignment (SSA) form [Cytron et al. 1991; Wolfe 1996; Briggs et al. 1997], performs SSA-based value numbering [Briggs et al. 1997] with TBAA, followed by SSA-based PRE [Chow et al. 1997], and finishes with generation of new Java bytecodes for the method. Note that BLOAT is a stand-alone tool that can be used to optimize Java classes independently of VM implementation. The DIVA technique has been added as a separate pass over the control-flow graph, just before the final code generation phase, to hoist array swizzle barriers out of loops.

3.2 Optimizations for PJama

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

In line with our optimization strategy, we have deleted the hidden swizzle barrier from the implementations of the original `aload` bytecode and extended the PJama VM with two new internal swizzle barrier bytecodes. As a class is loaded into the extended PJama VM its methods must now be edited to insert a swizzle barrier bytecode immediately before each occurrence of the `aload` bytecode. BLOAT supports this operation with a preprocessing (non-analyzing, non-optimizing) pass over the class to insert the swizzle barriers. The class can then go on to execute in the extended VM. Subsequent optimization by BLOAT can then occur at any convenient time. BLOAT also supports a “way-ahead-of-time” option to preprocess and optimize class files for later loading by the new PJama VM; this option is commonly

used to prepare the core Java classes for loading into a virgin PJama persistent store. The new barrier bytecodes are specified in Table 4.

Table 4: New swizzle barrier bytecodes

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

3.3 Cache management

As mentioned earlier, barrier optimizations require a contract with the persistence run-time system, which must not undo the effect of a barrier while optimized code can execute that assumes the barrier is still in effect. The contract with the PJama run-time system is simple: PJama must maintain the effect of both barriers for all objects directly referenced from a Java thread’s stack frames (both operand stacks and local variables). In other words, resident objects referenced directly from a thread stack must be *pinned* in the object cache whenever the thread is active. Thus, the PJama object cache manager must either avoid evicting pinned objects when it attempts to reclaim cache space, or arrange for them to be made resident before the pinning thread resumes execution. Dirty bits set on objects in the 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.

4 Experiments

Our experiments focus on revealing the gains to be had in eliminating loop-nested array swizzle checks, by counting the number eliminated for execution of several array-intensive benchmarks. Performance improvements as a result of the optimization are not directly measured here, though clearly for an interpreted VM any reduction in the number of bytecodes executed will have a noticeable impact on performance because of the corresponding reduction in bytecode dispatch overhead. We believe also that for JIT-compiled VM implementations a single range swizzle check can be translated to more efficient code than might otherwise obtain for the original loop-nested swizzle checks.

4.1 Metrics

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

4.2 Benchmarks

To best evaluate the impact of range swizzle optimizations using DIVA, we need a set of benchmarks that extensively use arrays of objects. With that objective the following applications were chosen:

- Linpack: The standard Linpack benchmark suite.
- Cholesky: Set of routines performing Cholesky decomposition.
- Neural: Back propagation on a multi-layered neural net.
- Inversion: Application performing a series of matrix inversions.

While these applications are not themselves inherently persistent, they might reasonably be used to perform computations over large data sets that might benefit from storage in a persistent environment.

4.3 Results

The results of range swizzle optimizations are given in Table 5. 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, 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.

Table 5: Results of range swizzle optimizations

Benchmark	aswizzles executed			aswizzleRanges executed		
	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

5 Conclusions

Our experiments show that on average 66% of swizzle barriers are eliminated at a small additional cost (0.9%) of introducing range swizzle barriers. These results show that range swizzle optimizations based

on DIVA can significantly reduce the array swizzle overhead of PJama. In general, our optimization can benefit any persistent Java system that implements lazy swizzling for arrays. We believe that DIVA optimization coupled with our read and write barrier optimizations [Hosking et al.] can have a significant positive impact on the performance of persistent Java systems. We also plan to integrate swizzle barrier optimizations into our PRE driven optimization framework by treating swizzles as expressions similar to read and write barriers. By treating all persistence barriers in a uniform manner we hope to build a general program analysis and optimization framework targeted at persistent systems [Cutts and Hosking 1997]. This will further enable us to exploit the strong connection and similarity between various persistence optimizations such as barrier elimination, concurrency control lock elimination, clustering, prefetching, and swizzling centered on program analysis and dynamic profiling.

Acknowledgements

We thank Laurent Daynès and Malcolm Atkinson for first suggesting that the array swizzle barriers of PJama might be a fruitful target for optimizations, with potential for significant impact on performance.

References

- 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., PRINTEZIS, T., AND SPENCE, S. 1996. An orthogonally persistent Java. *ACM SIGMOD Record* 25, 4 (Dec.), 68–75.
- ATKINSON, M. P. AND MORRISON, R. 1995. Orthogonally persistent object systems. *International Journal on Very Large Data Bases* 4, 3, 319–401.
- BRIGGS, P., COOPER, K. D., HARVEY, T. J., AND SIMPSON, L. T. 1997. Practical improvements to the construction and destruction of static single assignment form. Available at <http://www.cs.rice.edu/~harv/ssa.ps>.
- BRIGGS, P., COOPER, K. D., AND SIMPSON, L. T. 1997. Value numbering. *Software: Practice and Experience* 27, 6 (June), 701–724.
- CHOW, F., CHAN, S., KENNEDY, R., LIU, S.-M., LO, R., AND TU, P. 1997. A new algorithm for partial redundancy elimination based on SSA form. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (Las Vegas, Nevada, June). *ACM SIGPLAN Notices* 32, 5 (May), 273–286.
- CUTTS, Q. AND HOSKING, A. L. 1997. Analysing, profiling and optimising orthogonal persistence for Java. In *Proceedings of the Second International Workshop on Persistence and Java* (Half Moon Bay, California, Aug.), M. P. Atkinson and M. J. Jordan, Eds. Sun Microsystems Laboratories Technical Report 97-63, 107–115.
- 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. *ACM Trans. Program. Lang. Syst.* 13, 4 (Oct.), 451–490.
- DIWAN, A., MCKINLEY, K. S., AND MOSS, J. E. B. 1998. Type-based alias analysis. In *Proceedings of the ACM Conference on Programming Language Design and Implementation* (Montréal, Canada, June). *ACM SIGPLAN Notices* 33. To appear.
- GERLEK, M. P., STOLTZ, E., AND WOLFE, M. 1995. Beyond induction variables: detecting and classifying sequences using a demand-driven SSA form. *ACM Trans. Program. Lang. Syst.* 17, 1 (Jan.), 85–122.
- GOSLING, J., JOY, B., AND STEELE, G. 1996. *The Java Language Specification*. Addison-Wesley.
- HOSKING, A. L., NYSTROM, N., CUTTS, Q., AND BRAHNMATH, K. Optimizing the read and write barriers for orthogonal persistence. Submitted to the Eighth International Workshop on Persistent Object Systems (Tiburon, California, August 1998).

- LARUS, J. R. AND HILFINGER, P. N. 1988. Detecting conflicts between structure accesses. In Proceedings of the ACM Conference on Programming Language Design and Implementation (Atlanta, Georgia, June). *ACM SIGPLAN Notices*, 21–34.
- LINDHOLM, T. AND YELLIN, F. 1996. *The Java Virtual Machine Specification*. Addison-Wesley.
- MOREL, E. AND RENVOISE, C. 1979. Global optimization by suppression of partial redundancies. *Commun. ACM* 22, 2 (Feb.), 96–103.
- MOSS, J. E. B. 1992. Working with persistent objects: To swizzle or not to swizzle. *IEEE Trans. Softw. Eng.* 18, 8 (Aug.), 657–673.
- MUCHNICK, S. S. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann.
- NYSTROM, N., HOSKING, A. L., CUTTS, Q., AND DIWAN, A. Partial redundancy elimination for access path expressions. Submitted to the 1998 ACM Conference on Object-Oriented Programming Systems, Languages and Applications.
- PALSBERG, J. AND SCHWARTZBACH, M. I. 1994. *Object-Oriented Type Systems*. Wiley.
- STOLZ, 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* (Jan.). 43–52.
- TARJAN, R. E. 1972. Depth-first search and linear graph algorithms. *SIAM Journal of Computing* 1, 2 (June), 146–160.
- WOLFE, M. 1996. *High Performance Compilers for Parallel Computing*. Addison-Wesley.