Partial Redundancy Elimination for Access Path Expressions

Antony L. Hosking **Nathaniel Nystrom** **David Whitlock**
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907-1398, USA

Quintin Cutts
Department of Computing Science
University of Glasgow
Glasgow G12 8QQ, Scotland

Amer Diwan
Department of Computer Science
Stanford University
Stanford, CA 94305-9030, USA

Abstract

Pointer traversals pose significant overhead to the execution of object-oriented programs, since every access to an object's state requires a pointer dereference. Eliminating redundant pointer traversals reduces both instructions executed as well as redundant memory accesses to relieve pressure on the memory subsystem. We describe an approach to elimination of redundant access expressions that combines partial redundancy elimination (PRE) with type-based alias analysis (TBAA). To explore the potential of this approach we have implemented an optimization framework for Java class files incorporating TBAA-based PRE over pointer *access expressions*. The framework is implemented as a classfile-to-classfile transformer; optimized classes can then be run in any standard Java execution environment. Our experiments demonstrate improvements in the execution of optimized code for several Java benchmarks running in diverse execution environments: the standard interpreted JDK virtual machine, a virtual machine using "just-in-time" compilation, and native binaries compiled off-line ("way-ahead-of-time"). We isolate the impact of access path PRE using TBAA, and demonstrate that Java's requirement of precise exceptions can noticeably impact code-motion optimizations like PRE.

1 Introduction

Pointer traversals pose significant overhead to the execution of object-oriented programs, since every access to an object's state requires a pointer dereference. Objects can refer to other objects, forming graph structures, and they can be modified, with such modifications visible in future accesses. Just as common subexpressions often appear in numerical code, common access expressions are likewise often encountered in object-oriented code. Where two such expressions redundantly compute the same value it is desirable to avoid repeated computation of that value by caching the result of the first computation in a temporary variable, and reusing it from the temporary at the later occurrence of the expression. Eliminating redundant computations in this way certainly eliminates redundant CPU overhead. Perhaps just as important for modern machine architectures, eliminating redundant access expressions also has the effect of eliminating redundant memory references, which are often the source of large performance penalties incurred in the memory subsystem.

In this paper we evaluate an approach to elimination of common access expressions that combines partial redundancy elimination (PRE) [Morel and Renvoise 1979] with type-based alias analysis (TBAA) [Diwan et al. 1998]. To explore the potential of this approach we have built an optimization framework for Java class files incorporating TBAA-based PRE for access expressions, and measured its impact on the performance of several benchmark programs. An interesting aspect of the optimization framework is that it operates entirely as a bytecode-to-bytecode translator, sourcing and targeting Java class files. Our experiments compare the execution of optimized and unoptimized classes for several SPARC-based execution environments: the interpreted JDK reference virtual machine, the Solaris 2.6 virtual machine with "just-in-time" (JIT) compilation, and native binaries compiled off-line ("way-ahead-of-time") to C and thence to native code using the Solaris C compiler.

We have measured both the static and dynamic impact of bytecode-level PRE optimization for a set of Java benchmark applications, including static code size, bytecode execution counts, native-instruction execution counts, and elapsed time. The results demonstrate general improvement on all measures for all execution environments, although some benchmarks have degraded performance in certain environments.

The remainder of the paper is organized as follows. Section 2 introduces the approach to elimination of redundant access path expressions based on partial-redundancy elimination and type-based alias analysis. Section 3 describes our implementation of the analysis and optimization framework that supports transformation of Java class files using TBAA-based PRE. Section 4 describes our experimental methodology for evaluation of TBAA-based PRE for several Java benchmark applications. The experimental results are reported and interpreted in Section 5. We conclude with a discussion of related work and speculate on directions for future work.

This research is supported in part by the National Science Foundation under Grant No. CCR-9711673, and by gifts from Sun Microsystems, Inc.
Authors' email addresses: {hosking, nystrom, whitlock}@cs.purdue.edu, quintin@dcs.gla.ac.uk, diwan@cs.stanford.edu

Table 1: Access expressions

Notation	Name	Variable accessed
$p.\mathtt{f}$	Field access	Field `f` of class instance to which p refers
$p[i]$	Array access	Component with subscript i of array to which p refers

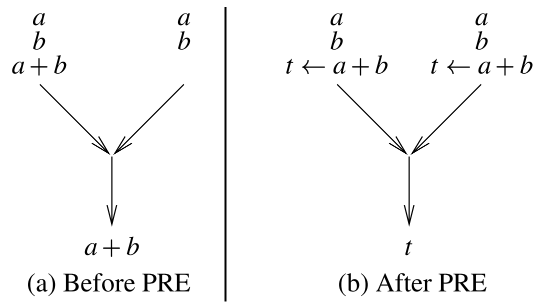


Figure 1: PRE for arithmetic expressions

2 PRE for Access Expressions

Our analysis and optimization framework revolves around PRE over object access expressions. We adopt standard terminology and notations used in the specification of the Java programming language to frame the analysis and optimization problem.

2.1 Terminology and notation

The following definitions paraphrase the Java specification [Gosling et al. 1996]. An *object* in Java is either a *class instance* or an array. Reference values in Java are *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. For example, the Java expression $a.b[i].c$ is an access path. Without loss of generality, our notation assumes that distinct fields within an object have different names (i.e., fields that override inherited fields of the same name from superclasses are trivially renamed).

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, including T. Thus, an access path p can legally access variables of type $Subtypes(Type(p))$.

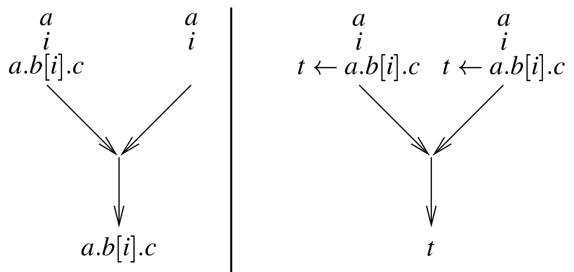
2.2 Partial redundancy elimination

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

Consider the Java access expression $a.b[i].c$, which translates to Java bytecode of the form:

```
aload a      ; load local variable a
getfield b   ; load field b of a
iload i      ; load local variable i
aaload       ; index array b
getfield c   ; load field c of b[i]
```

[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.



(a) Before PRE (b) After PRE

Figure 2: PRE for access expressions

Traversing the 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 lexically distinct access path in the program. Consider the example in Figure 2. 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 $a.b[i].c$ on the code path between the first evaluation of the expression and the second. In other words, if there are explicit stores to a or i (local variables cannot be aliased in Java) or potential aliases to any one of $a.b$, $a.b[i]$ or $a.b[i].c$ through which those locations *may* be modified between the first and second evaluation of the expression, then that second evaluation cannot be treated as redundant.

2.3 Type-based alias analysis

Alias analysis refines the set of possible variables to which an access path may refer. Two distinct access paths are said to be possible *aliases* if they may refer to the same variable. Without alias analysis the optimizer must conservatively assume that all access paths are possible aliases of each other. In general, alias analysis in the presence of references is slow and requires the code for the entire program to work. *Type-based alias analysis* (TBAA) [Diwan et al. 1998] offers one possibility for overcoming these limitations. TBAA assumes a type-safe programming language such as Java, since it uses type declarations to disambiguate references. It works in linear time and does not require that the entire program be available. Rather, TBAA uses the type system to disambiguate memory references by refining the *type* of variables to which an access path may refer, since only type-compatible access paths can alias the same variable in a type-safe language such as Java. The compile-time type of an access path provides a simple way to do this: two access paths p and q may be aliases only if the relation $\mathit{TypeDecl}(p,q)$ holds, as defined by

$\mathit{TypeDecl}(\mathcal{AP}_1, \mathcal{AP}_2)$
$\mathit{Subtypes}(\mathit{Type}(\mathcal{AP}_1)) \cap \mathit{Subtypes}(\mathit{Type}(\mathcal{AP}_2)) \neq \emptyset$

Table 2: $\mathit{FieldTypeDecl}(\mathcal{AP}_1, \mathcal{AP}_2)$

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

A more precise alias analysis will distinguish accesses to fields that are the same type yet distinct. This more precise relation, $\mathit{FieldTypeDecl}(p,q)$, is defined by induction on the structure of p and q in Table 2. Again, two access paths p and q may be aliases only if the relation $\mathit{FieldTypeDecl}(p,q)$ holds. It distinguishes accesses such as $t.\mathtt{f}$ and $t.\mathtt{g}$ that *TypeDecl* misses. The cases in Table 2 determine that:

1. Identical access paths are always aliases
2. Two field accesses may be aliases if they access the same field of potentially the same object
3. Array accesses cannot alias field accesses and vice versa
4. Two array accesses may be aliases if they may access the same array (the subscript is ignored)
5. All other pairs of access expressions (when none of the above apply) are possible aliases if they have common subtypes

2.3.1 Analysing Incomplete Programs

Java dynamically links classes on demand as they are needed during execution. Moreover, Java permits dynamic loading of arbitrary named classes that are statically unknown. Also, code for native methods cannot easily be analysed. To maintain class compatibility, no class can make static assumptions about the code that implements another class. Thus, alias analysis must make conservative assumptions about the effects of statically unavailable code. Fortunately, both *TypeDecl* and *FieldTypeDecl* require only the compile-time types of access expressions to determine which of them may be aliases. Thus, they are applicable to compiled classes in isolation and optimizations that use the static alias information they derive will not violate dynamic class compatibility.

Diwan et al. [1998] further refine TBAA for *closed world* situations: those in which all the code that might execute in an application is available for analysis. The refinement enumerates all the assignments in a program to determine more accurately the types of variables to which a given access path may refer. An access path of type T may yield a reference to an object of a given subtype S only if there exist assignments of references of type S to variables of type T. Unlike *TypeDecl*, which always merges the compile-time type of an access path with all of its subtypes, Diwan's closed world refinement merges a type T with a subtype S only if there is at least one assignment of a reference of type S to a variable of type T somewhere in the code.

In general, Java's use of dynamic loading, not to mention the possibility of native methods hiding assignments from the analysis, precludes such closed world analysis. Of course, it is possible to adopt a closed world model for Java if one is prepared to restrict dynamic class loading only to classes that are known statically, and to support analysis (by hand or automatically) of the effects of native methods. Note that a closed world model will require re-analysis of the entire closure if any one class is changed to include a new assignment.

2.4 Java constraints on optimization

Java's thread and exception models impose several constraints on optimization. First, exceptions in Java are *precise*: when an exception is thrown all effects of statements prior to the throw-point must appear to have taken place, while the effects of statements after the throw-point must not. This imposes a significant constraint on code-motion optimizations such as PRE, since code with side-effects (including possible exceptions) cannot be moved relative to code that may throw an exception.[2] In regions of the code where program analysis can show that exceptions will not occur code motion is unconstrained. For example, the first access to an object via a given reference ensures that subsequent accesses via that reference cannot throw a null pointer exception.

Second, the thread model prevents movement of access expressions across (possible) synchronization points. Explicit synchronization points occur at monitorenter/monitorexit bytecodes. Also, without interprocedural control-flow analysis every method invocation represents a possible synchronization point, since the callee, or a method invoked inside the callee, may be synchronized. Thus, calls and synchronization points are places at which PRE must assume all non-local variables may be modified, either inside the call or through the actions of other threads. Common access expressions cannot be considered redundant across these synchronization points.

Naturally, one must also respect the `volatile` declaration modifier, which forces synchronization of the variable's state across threads on every access.

3 Implementation

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 file 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" translation to native code (JIT), or precompiles Java class files to native object files.

The bytecodes of the Java VM specification serve as a convenient target for optimization of Java applications. As the only constant in a sea of Java compilers and virtual machines, targeting the Java class files for analysis and optimization has several advantages. First, program improvements accrue even in the absence of source code for both libraries and applications, and independently of the source-language compiler and VM implementation. Second, Java class files retain enough high-level type information to enable many recently-developed type-based analyses and optimizations for object-oriented languages. Finally, analysing and optimizing bytecode can be performed off-line, permitting JIT compilers to focus on fast native code generation rather than expensive analysis. Indeed, off-line analysis may expose opportunities for fast low-level JIT optimizations. Thus, we have chosen to implement a framework for TBAA-based PRE over access expressions based on classfile-to-classfile transformation.

Our Java class file optimization tool is called BLOAT (Bytecode-Level Optimization and Analysis Tool). The analysis and optimization framework implemented in BLOAT is based on several recent developments in the field. Notably, we use control flow graphs and static single assignment (SSA) form as the basic intermediate representation [Cytron et al. 1991; Wolfe 1996; Briggs et al. 1998]. On this foundation we have built several standard optimizations such as dead-code elimination and copy/constant propagation, and SSA-based value numbering [Simpson 1996], as well as *type-based alias analysis* [Diwan et al. 1998] and the SSA-based algorithm for PRE of Chow et al. [1997].

3.1 SSA form

SSA form provides a concise representation of the use-definition relationships among the program variables. Efficient global optimizations can be constructed based on this form, including dead store elimination [Cytron et al. 1991], constant propagation [Wegman and Zadeck 1991], value numbering [Alpern et al. 1988; Rosen et al. 1988; Cooper and Simpson 1995; Simpson 1996; Briggs et al. 1997], induction variable analysis [Gerlek et al. 1995] and global code motion [Click 1995]. Optimization algorithms based on SSA all exploit its sparse representation for improved speed and simpler coding of combined local and global optimizations.

[2] Of course an optimizing Java implementation *could* simulate precise exceptions, even while performing unrestricted code hoisting, by arranging to hide any such speculative execution from the user-visible state of the Java program (see page 205 of Gosling et al. [1996]).

3.2 SSA-based PRE

Prior to the work of Chow et al. [1997], PRE lacked an SSA-based formulation. As such, optimizers that used SSA were forced to convert to a bit-vector representation for PRE and back to SSA for subsequent SSA-based optimizations. Chow et al. [1997] removed this impediment with an approach (SSAPRE) that retains the SSA representation throughout PRE. The specific details of their algorithm are not relevant here, save to say that the algorithmic complexity is respectable: for a program of size n, SSAPRE's total time is $O(n(E+V))$, where E and V are the number of edges and nodes in the control flow graph, respectively.

3.3 Analysis

For each method in a class, BLOAT first builds a control flow graph over the bytecode instructions and then transforms each basic block into expression trees. The trees are constructed through a simulation of the operand stack.

Two simple transformations are then applied to ease later analyses and optimizations. The first converts methods that initialize static arrays from the form emitted by the JDK javac compiler, comprising a straight-line sequence of array stores for every element of the array, into a form more amenable to later analysis, consisting of a loop that reads from a static string defined in the constant pool of the class. This transformation eliminates the unnecessarily large basic blocks emitted for static array initializers in such core classes as Character, significantly cutting the time for later analysis of these initializers. The second transformation identifies loops [Havlak 1997] and converts each "while" loop into a "repeat" loop preceded by an "if" conditional. This provides a convenient place immediately after the "if" to hoist loop-invariant code out of the loop body. Code that is loop-invariant apart from possibly throwing exceptions can thus be treated as invariant in the new loop body and will be eliminated by PRE.

After construction of the control flow graph both local and operand stack variables are converted to SSA form. This requires computation of the dominator tree and dominance frontier of the control flow graph [Cytron et al. 1991]. We also remove *critical edges* in the graph by inserting empty basic blocks on such edges. Critical edge removal is required to provide a place to insert code during PRE and when translating back from SSA form.

Java bytecode has two forms of control flow which complicate SSA construction: exception handlers and method-local subroutines. To support exception handling, we must propagate local variable information from the protected area to the exception handler. We extend SSA to more easily distinguish all values of a variable that are live within the protected region.

Subroutines within a method are formed with the jsr and ret bytecodes. The jsr bytecode pushes the current program counter, a value of type returnAddress, onto the operand stack and branches to the subroutine. The ret bytecode loads a saved returnAddress from a local variable and resumes control at that code location. To permit verification of jsr subroutines the Java VM specification imposes a restriction that each jsr can have at most one corresponding ret. This allows each jsr to be tied to the ret that returns to it. Thus, for SSA construction we treat each subroutine such that, if a variable is not redefined within the subroutine, the use-definition information for the variable is propagated from each jsr site to its corresponding return site. This avoids unnecessarily merging information from multiple paths through the subroutine. Our SSA-based solution is essentially the same as the "variable splitting" approach proposed by Agesen et al. [1998] in support of accurate garbage collection.

TBAA uses the compile-time type of every expression in the method, but local and operand stack variables in Java bytecode are not declared. Thus, after SSA construction we infer their types using an intra-procedural variation of the algorithm of Palsberg and Schwartzbach [1994].

PRE operates by recognizing common subexpressions. Rather than basing equivalence of expressions purely on their lexical equivalence, we use the SSA-based value numbering approach of Simpson [1996]. We assign value numbers to every *first-order* expression. These are expressions for values that cannot be aliased, such as the contents of method local variables, constants, and non-access expressions over these. Using value numbering avoids the need for repetitive iteration of PRE interleaved with constant/copy propagation.

Finally, we identify alias definition points: those code locations where potentially-aliased variables may be modified. For example, an assignment to a (non-local) variable redefines every access expression that may alias that variable. Calls and monitor synchronization points redefine *all* access expressions.

3.4 Optimization

After the analyses, BLOAT performs the following optimizations:

1. *Partial redundancy elimination*. BLOAT implements the SSA-based PRE algorithm of Chow et al. [1997], extended to support TBAA-based PRE of access paths by treating alias definition points for a given access expression as redefining that expression. This forces reevaluation of the expression after the alias definition point. We also restrict PRE-induced code motion to respect the constraints on Java optimizations due to precise exceptions and threads, except that where analysis shows a given bytecode will never throw an exception we are free to move code with respect to that bytecode.

2. *Constant/copy propagation*. This is based on standard techniques for constant folding, algebraic simplification and copy propagation [Wolfe 1996].

3. *Dead code elimination*. This is the standard SSA-based algorithm [Cytron et al. 1991].

3.5 Code generation

Following the optimizations, SSA temporaries are mapped back to Java VM local variables, before generation of bytecode instructions from the (optimized) intermediate code trees. Liveness analysis and register coloring with coalescing [Chaitin 1982; Briggs et al. 1994] ensure a good allocation, packing as many SSA variables into the same physical local variable as possible. Priority is given to coalescing loop-nested local variables ahead of others. Peephole optimizations remove redundant load and store bytecodes for better utilization of the operand stack.

Table 3: Benchmarks

Name	Description	Size[a]
Crypt	Java implementation of the Unix crypt utility	650
Huffman	Huffman encoding	435
Idea	File encryption tool	2284
JLex	Scanner generator	7287
JTB	Abstract syntax tree builder	22317
Linpack	Standard Linpack benchmark	584
LZW	Lempel-Ziv-Welch file compression utility	314
Neural	Neural network simulation	1227
Tiger	Tiger compiler [Appel 1998]	19018

[a] Lines of source code (including comments).

4 Experiments

To evaluate PRE for access path expressions we took several Java programs as benchmarks, optimized them with BLOAT and compared the results of the optimization with their unoptimized counterparts, using several static and dynamic performance metrics. To isolate the effects of access path PRE we considered 3 successively more powerful levels of optimization: PRE over scalar expressions, TBAA-based PRE over both scalar and access expressions, and TBAA-based PRE that does not respect Java's precise exception requirements. Each optimization level subsumes all optimizations that are performed by lower level optimizations. In the following we will refer to results for the unoptimized code as base, and to the successive levels of PRE-based optimization as pre, tbaa and loose, respectively.

4.1 Platform

Our experiments were run under Solaris 2.5.1 on a Sun Ultra 2 Model 2200, with 256Mb RAM, and two 200MHz UltraSPARC-I processors, each with 1Mb external cache in addition to their on-chip instruction and data caches. The UltraSPARC-I data cache is a 16Kb write-through, non-allocating, direct-mapped cache with two 16-byte sub-blocks per line. It is virtually indexed and physically tagged. The 16Kb instruction cache is 2-way set-associative, physically indexed and tagged, and organized into 512 32-byte lines.

4.2 Benchmarks

The benchmarks we use are summarized in Table 3.

4.3 Execution environments

We took measurements for three different Java execution environments: the standard Java Development Kit (JDK) version 1.1.6, the Solaris 2.6 SPARC JDK with JIT version 1.1.3 (JIT) and Toba version 1.1 (Toba) [Proebsting et al. 1997]. In each environment we ran all four variants (base, pre, tbaa and loose) of the classes for each benchmark. Where Java source code for a benchmark was available, it was compiled using the standard JDK 1.1.6 javac compiler (without the -O optimization flag since in many cases this generates erroneous code; our observations indicate that this flag has little impact on the performance of our benchmarks).

4.3.1 JDK

JDK is the standard 1.1.6 Java virtual machine. It uses a portable threads package rather than the native Solaris threads and the bytecode interpreter loop is implemented in assembler. We optimized the class files of each benchmark against the JDK version 1.1.6 core Java classes [Gosling et al. 1996] at each optimization level, to form the closure of optimized classes necessary to execute the benchmark in JDK. Similarly the unoptimized benchmark classes were run against the unoptimized core classes.

4.3.2 JIT

JIT refers to the Solaris 2.6 SPARC JDK with JIT version 1.1.3. This VM translates a method's bytecodes to native SPARC instruction on first execution of the method, along with the following optimizations:

1. elimination of some array bounds checking
2. elimination of common subexpressions within blocks
3. elimination of empty methods
4. some register allocation for locals
5. no flow analysis
6. limited inlining

Interestingly, programmers are encouraged to perform the following optimizations by hand [SunSoft 1997]:

1. move loop invariants outside the loop
2. make loop tests as simple as possible
3. perform loops backwards
4. use only local variables inside loops
5. move constant conditionals outside loops

6. combine similar loops
7. nest the busiest loop, if loops are interchangeable
8. unroll loops, as a last resort
9. avoid conditional branches
10. cache values that are expensive to fetch or compute
11. pre-compute values known at compile time

These suggestions likely reveal deficiencies in the current JIT compiler which our optimizations may address prior to JIT execution.

We used the same sets of class files as for JDK for execution in the JIT environment.

4.3.3 Toba

Toba compiles Java class files to C, and thence to native code using the host system's C compiler. The Toba run-time system supports native Solaris threads, and garbage collection using the Boehm-Demers-Weiser conservative garbage collector [Boehm and Weiser 1988]. We started with the same sets of classes as for JDK for execution in Toba. These class files were then compiled to native code using the SunPro C compiler version 4.0, with the -O2 compiler optimization flag. C optimization level 2 performs basic local and global optimization, including induction variable elimination, algebraic simplification, copy propagation, constant propagation, loop-invariant optimization, register allocation, basic block merging, tail recursion elimination, dead code elimination, tail call elimination and complex expression expansion. We use this level since it does not "optimize references", nor "trace the effect of pointer assignments", and therefore will best reveal the impact of our pointer optimizations.

4.4 Metrics

For each benchmark we took measurements for both the optimized and unoptimized classes. We use both static and dynamic metrics to expose the effects of optimization:

- static code size: this is the size in bytes of the benchmark-specific (non-library) class files (excluding debug symbols) for JDK/JIT, and static executables for Toba
- bytecodes executed: dynamic per-bytecode execution frequencies, obtained via an instrumented version of the C-coded interpreter loop in the JDK source release
- globally redundant (i.e., cross-activation) bytecode-level memory accesses: dynamic per-bytecode counts of all accesses that reload values from unmodified variables, also obtained via the instrumented JDK VM
- native SPARC instructions executed: dynamic per-instruction execution frequencies using the Shade performance analysis toolkit [Cmelik and Keppel 1994]
- counts of significant performance-related events:
  - processor cycles to measure elapsed time
  - instruction buffer stalls due to instruction cache misses
  - data cache reads
  - data cache read misses

  using software[3] that allows user-level access to the UltraSPARC hardware execution counters

For the dynamic measurements each run consists of two iterations of the benchmark within a given execution environment. The first iteration is to prime the environment: loading class files, JIT-compiling them and warming the caches. The second iteration is the one measured.

The physically addressed instruction cache on the UltraSPARC means that programs can exhibit widely varying execution times from one invocation to the next, since each invocation may have quite different mappings from virtual to physical addresses that result in randomized instruction cache placement. Thus, the elapsed time and cache-related metrics were obtained for 10 separate runs and the results averaged. We ran each benchmark with sufficient heap space to eliminate the need for garbage collection, and verified that no collections occurred during benchmarking.

5 Results

Our presentation normalizes all optimization results with respect to the base metrics. Reporting the results in this way exposes the relative effects of the successive levels of optimization. Error bars in our graphs represent 90% confidence intervals; these display the variation in performance due to factors beyond our control, such as the varying virtual-to-physical page mappings. Grouped by benchmark, the adjacent columns in the graphs represent the transition to higher optimization levels from base, through pre and tbaa to loose,

In the following discussion we consider each execution environment in turn: JDK, then JIT, and lastly Toba.

5.1 JDK

Figure 3(a) illustrates the anticipated increase in dynamic bytecode execution counts for many of the benchmarks, mainly due to introduction of extra loads from and stores to temporaries introduced by PRE for partially-redundant expressions whose values are not used on all paths. As we shall see, execution environments that map method local variables to registers (e.g., JIT and Toba) do not suffer unduly from this overhead. The impact on elapsed time for interpretation by the JDK (Figure 3(b)) can be severe (notably for Huffman and Tiger), though for some benchmarks the extra load and store bytecodes are offset by elimination of partially-redundant code and replacement of many expensive long load/store bytecode forms with their cheaper short alternatives.

[3] See http://www.cs.msu.edu/~enbody/

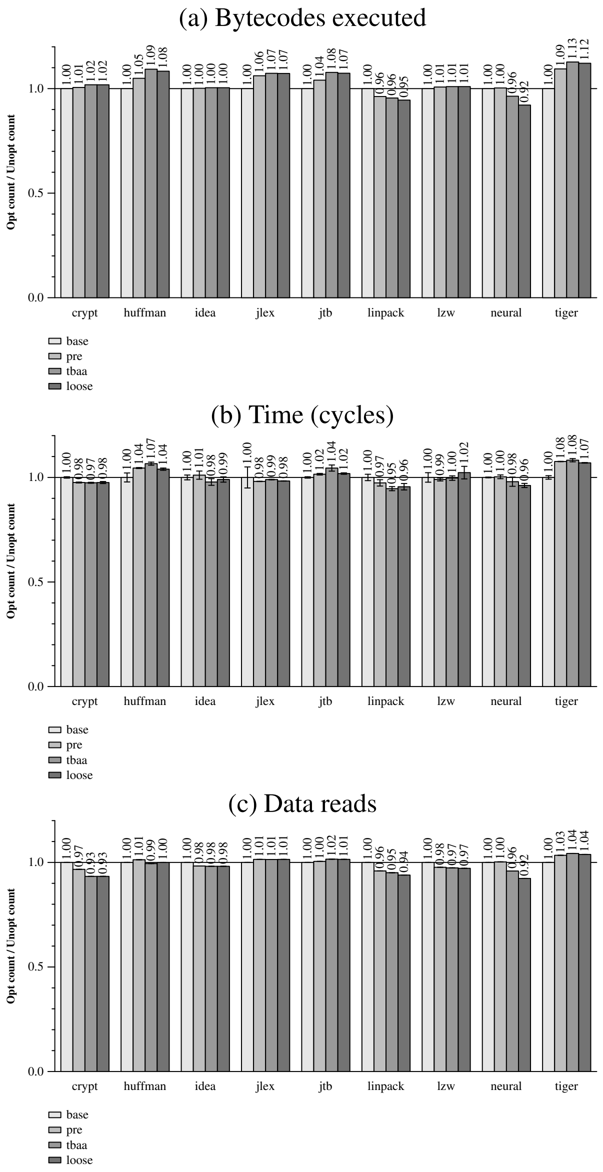


Figure 3: JDK metrics

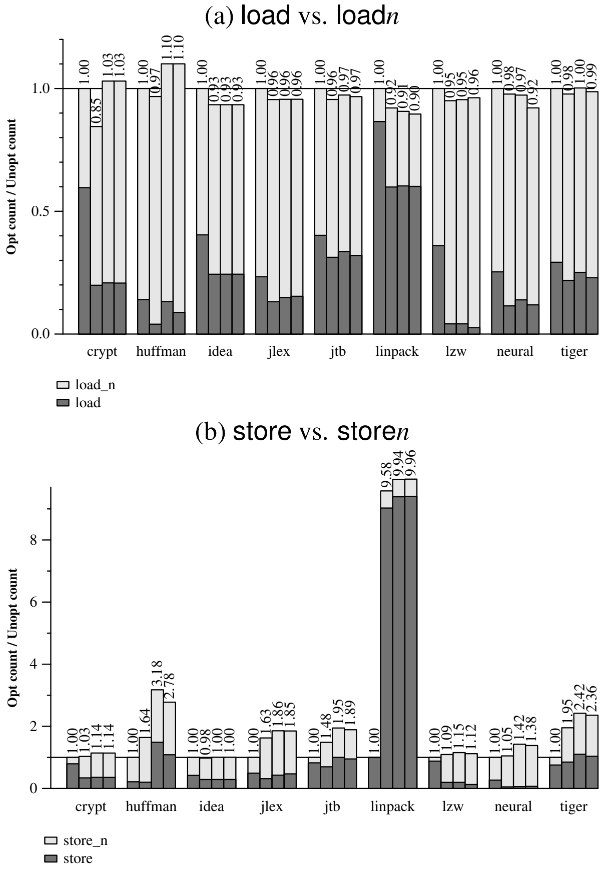


Figure 4: Replacing load/store with short forms

Figure 4 highlights these conversions. The effect is most notable with LZW where the frequency of the load bytecodes decreases from 11% to 1% of the total bytecodes executed and the frequency of load*n* increases from 20% to 28%. These effects result in less overhead in the interpreter's bytecode dispatch loop. The impact on data cache reads (remember, bytecodes are data) is revealed in Figure 3(c). The large increase in stores for Linpack is due to PRE's elimination of significant numbers of redundant arithmetic expressions.

The most dramatic effects of PRE over access paths are directly revealed in the results for the access bytecodes given in Figure 5. Here, we show the total number of access bytecodes performed, broken down into globally redundant versus non-redundant accesses. The non-redundant accesses are those that must always be performed. The globally redundant accesses represent opportunity for optimization; with perfect inter-procedural control flow and aliasing information all such accesses could be removed. It is remarkable how many of these are removed by our purely intra-procedural analysis. All benchmarks see a decrease, often significant, in the frequency of getfield bytecodes (Figure 5(a)) due to TBAA-based PRE over redundant access expressions (tbaa). The dramatic reduction for Neural represents a reduction of redundant getfield operations from 9% of total bytecodes executed to 5% of total bytecodes. Linpack's reductions are similar, but getfields represent just 0.02% of total bytecodes executed so the impact is minimal. Relaxing Java's precise exception requirement (loose) yields little benefit for getfield.

The array intensive benchmarks (Huffman, Linpack, and Neural) obtain noticeable reductions in arrayload frequency (Figure 5(b)). Interestingly, relaxing Java's precise exceptions gives significant improvement for both Linpack and Neural, because freedom from concern over precise delivery of array out of bounds exceptions, provides more opportunity for PRE-based code motion. The Huffman, Linpack, and Neural benchmarks, which have heavy array use (4%, 9%, and 11%, respectively), see an elimination of 4–7% of the arrayload bytecodes for TBAA-based PRE with precise exceptions (tbaa). Relaxing exception delivery (loose) sees reductions in arrayloads increase to a peak of 22% for Neural. Further improvement would accrue if array subscripts could be disambiguated via range analysis on the subscript expressions for use during array alias analysis. Few arrayload bytecodes are eliminated in any of

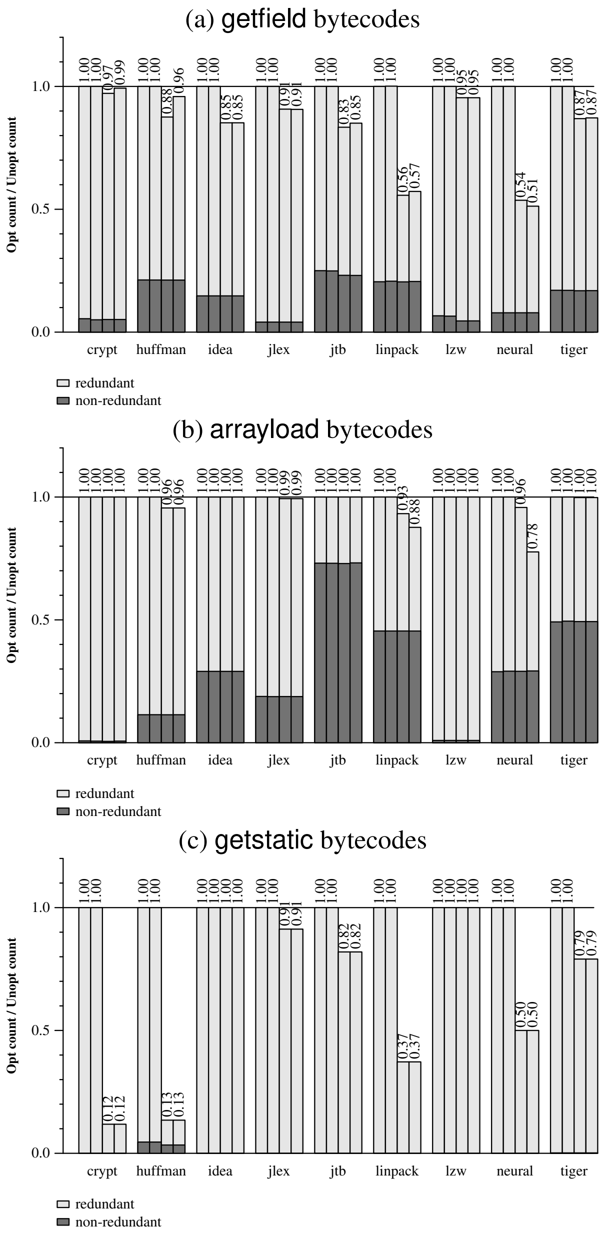


Figure 5: Access bytecodes executed

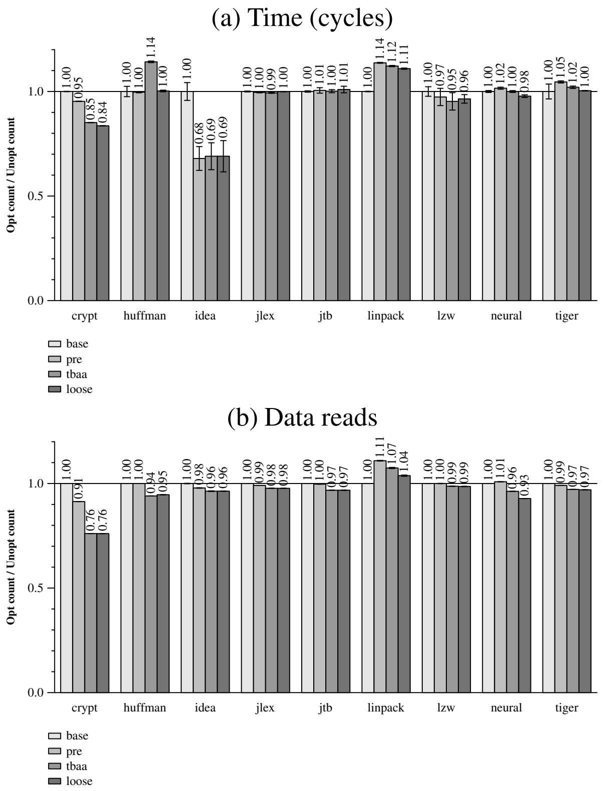


Figure 6: JIT metrics

the other benchmarks, primarily because they their array accesses are hidden inside method calls to library classes, etc.

The most dramatic gains are for getstatic accesses (Figure 5(c)), primarily because almost all such accesses are globally redundant. That we come close to the limit in eliminating almost all redundant accesses for Crypt and Huffman demonstrates the effectiveness of even simple alias analyses such as TBAA. The benchmarks where PRE doesn't eliminate many getstatics do not have many to begin with.

5.2 JIT

The JIT environment is not influenced by conversion of long bytecode forms to their short variants, since JIT eliminates the bytecode dispatch overhead that we were able to reduce for JDK. Nor, since JIT allocates local variables to registers, do the extra load and store bytecodes matter much since they are converted to register accesses. The only exception to this is Linpack, which we saw earlier suffers from the introduction of large numbers of temporaries and corresponding stores. Unfortunately, the corresponding increase in contention for register assignment of these temporaries causes most of them to remain in memory, with the loads turning into real memory accesses. This may simply be a shortcoming of the register allocation technique used by JIT. Thus, Linpack's elapsed time performance after PRE is disappointing (Figure 6(a)). Of the other benchmarks, only Crypt and Idea show marked improvement in elapsed time, although they all have fewer memory reads (Figure 6(b)). The marked improvement in Idea is a result of improved data read locality, resulting in many fewer data cache misses.

5.3 Toba

With Toba all benchmarks but Tiger show reductions in data reads (Figure 7(b)). Thus, our optimizations expose opportunities that the C compiler cannot exploit on its own at optimization level 2. These are reflected in reduced elapsed times (Figure 7(a)) for all but Huffman, Idea, LZW and Neural, which are unable to exploit the reduction in data reads in the face of an uncooperative instruction cache. This is an artefact of the hardware platform, and cannot be blamed on the optimizer, since it almost never increases code size, and actually is effective at reducing it.

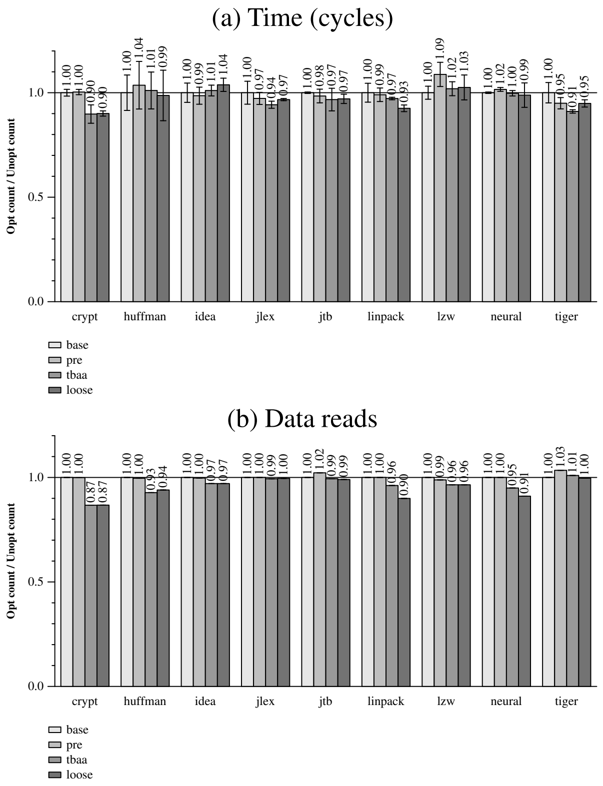


Figure 7: Toba metrics

6 Related work

The recent literature on alias analysis is extensive [Chase et al. 1990; Landi and Ryder 1992; Choi et al. 1993; Landi et al. 1993; Hummel et al. 1994; Deutsch 1994; Emami et al. 1994; Altucher and Landi 1995; Wilson and Lam 1995; Ruf 1995; Ghiya and Hendren 1996; Steensgaard 1996; Shapiro and Horwitz 1997; Debray et al. 1998; Ghiya and Hendren 1998; Hasti and Horwitz 1998; Jagannathan et al. 1998]. As in Diwan et al. [1998], our results are distinguished from prior work by comprehensive evaluation of TBAA with respect to a particular optimization, in this case PRE over access expressions, and metrics and upper bounds on redundant run-time memory accesses, as opposed to static measurements.

Budimlic and Kennedy [1997] describe a bytecode-to-bytecode optimization approach very similar to ours. They recover and optimize an SSA-based representation of each class file, much as we do, performing dead code elimination and constant propagation on the SSA, local optimizations on the control flow graph (local CSE, copy propagation, and "register" allocation of locals), followed by peephole optimization. They do nothing like our PRE over access path expressions. Their performance results are similar to ours, showing significant improvements for JDK and JIT execution. In addition, they consider the effects of two new interprocedural optimizations: *object inlining* and *code duplication*. Similar in some respects to the well-known approaches of cloning and inlining, these optimizations yield factors of two to five in performance improvement, so are consistent with results reported elsewhere [Chambers and Ungar 1989; Chambers et al. 1989; Chambers and Ungar 1990; 1991; Chambers 1992; Dean et al. 1995; Dolby 1997; Dolby and Chien 1998]

Cierniak and Li [1997] describe another similar approach to optimization from Java class files, involving recovery of sufficient high-level program structure to enable essentially source-level transformations of data layouts to improve memory hierarchy utilization for a particular target machine. Their results are also convincing, with performance improvements in a JIT environment of up to a factor of two.

Our reading of Cierniak and Li [1997] and Budimlic and Kennedy [1997] is unable to determine to what extent they respect Java's precise exception semantics and its constraints on code motion. Still, both of these prior efforts are much more aggressive than us in the transformations they are willing to apply. We hope that TBAA-based PRE for access expressions will produce results as spectacular as theirs when combined with more aggressive inter-procedural analyses, such as they describe.

Added evidence for this comes from Diwan et al. [1998] in their work with elimination of common access expressions for Modula-3. Their results indicate that accesses are often only *partially*-redundant across calls, while their optimizer only eliminates fully redundant access expressions. Of course, our PRE-based approach eliminates partial redundancies by definition. Diwan's results for elimination of fully redundant accesses without inter-procedural analysis are broadly consistent with ours.

Several recent papers have focused on *register promotion* [Cooper and Lu 1997; Sastry and Ju 1998; Lo et al. 1998]: the identification of program regions in which memory-allocated values can be cached in registers. These techniques also address the issue of eliminating redundant loads and stores by selectively promoting values from memory into registers. Our approach differs in that we perform analysis and transformation at a higher level than these other approaches, with full knowledge of the types of the memory values being promoted. We are currently working to understand the precise relationship between our approach and these lower level techniques. Certainly, given the problems we have with loading and storing of temporaries in some benchmarks, it seems that our approach might benefit from the more selective placement of loads and stores that these promotion techniques employ.

7 Conclusions and Future Work

Our results reveal the promise of optimization of Java classes independently of the source-code compiler and the runtime execution engine. In particular, we have demonstrated improvements using TBAA-based PRE over access path expressions, with dramatic reductions in memory access operations. Applying interprocedural analyses and optimizations should yield even more significant gains as the context for PRE is expanded across procedure boundaries, especially since Java programming style promotes the use of many small methods whose intraprocedural context is severely limited.

Under some circumstances Java's precise exception model is overly constraining for code motion optimizations such as PRE. Relaxing the constraints can provide more opportunities for optimization. More evidence is needed whether precise exceptions are unnecessarily restrictive.

The implementation of further analyses and optimizations to BLOAT is under way and we are close to making the tool more widely available. One application domain we are now focusing on is analysis and optimization of Java programs in a persistent environment [Atkinson et al. 1996]. The structure access optimizations we have explored here prove particularly fruitful in a persistent setting, where loads and stores carry additional semantics, acting not just on virtual memory, but also on persistent storage [Cutts and Hosking 1997; Hosking et al. 1999; Cutts et al. 1999; Brahnmath et al. 1999].

References

AGESEN, O., DETLEFS, D., AND MOSS, J. E. B. 1998. Garbage collection and local variable type-precision and liveness in Java virtual machines. See PLDI [1998], 269–279.

ALPERN, B., WEGMAN, M. N., AND ZADECK, F. K. 1988. Detecting equality of values in programs. See POPL [1988], 1–11.

ALTUCHER, R. Z. AND LANDI, W. 1995. An extended form of must alias analysis for dynamic allocation. In *Conference Record of the ACM Symposium on Principles of Programming Languages* (Jan.). 74–84.

APPEL, A. W. 1998. *Modern Compiler Implementation in Java*. Cambridge University Press.

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.

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

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

BRIGGS, P., COOPER, K. D., HARVEY, T. J., AND SIMPSON, L. T. 1998. Practical improvements to the construction and destruction of static single assignment form. *Software: Practice and Experience 28,* 8 (July), 859–881.

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

BRIGGS, P., COOPER, K. D., AND TORCZON, L. 1994. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst. 16,* 3 (May), 428–455.

BUDIMLIC, Z. AND KENNEDY, K. 1997. Optimizing Java: Theory and practice. *Software: Practice and Experience 9,* 6 (June), 445–463.

CHAITIN, G. J. 1982. Register allocation and spilling via graph coloring. In Proceedings of the ACM Symposium on Compiler Construction (Boston, Massachusetts, June). *ACM SIGPLAN Notices 17,* 6 (June), 98–105.

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

CHAMBERS, C. AND UNGAR, D. 1989. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In Proceedings of the ACM Conference on Programming Language Design and Implementation (Portland, Oregon, June). *ACM SIGPLAN Notices 24,* 7 (July), 146–160.

CHAMBERS, C. AND UNGAR, D. 1990. Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs. See PLDI [1990], 150–164.

CHAMBERS, C. AND UNGAR, D. 1991. Making pure object oriented languages practical. In Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (Phoenix, Arizona, Oct.). *ACM SIGPLAN Notices 26,* 11 (Nov.), 1–15.

CHAMBERS, C., UNGAR, D., AND LEE, E. 1989. An efficient implementation of Self, a dynamically-typed object-oriented language based on prototypes. In Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (New Orleans, Louisiana, Oct.). *ACM SIGPLAN Notices 24,* 10 (Oct.), 49–70.

CHASE, D. R., WEGMAN, M., AND ZADECK, F. K. 1990. Analysis of pointers and structures. See PLDI [1990], 296–310.

CHOI, J.-D., BURKE, M., AND CARINI, P. 1993. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the ACM Symposium on Principles of Programming Languages* (Charleston, South Carolina, Jan.). 232–245.

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. See PLDI [1997], 273–286.

CIERNIAK, M. AND LI, W. 1997. Optimizing Java bytecodes. *Concurrency: Practice and Experience 9,* 6 (June), 427–444.

CLICK, C. 1995. Global code motion/global value numbering. See PLDI [1995], 246–257.

CMELIK, B. AND KEPPEL, D. 1994. Shade: A fast instruction-set simulator for execution profiling. In Proceedings of the ACM Conference on the Measurement and Modeling of Computer Systems (Nashville, Tennessee, May). *ACM ACM SIGMETRICS Performance Evaluation Review 22,* 1 (May), 128–137.

Conference Record of the ACM Symposium on Principles of Programming Languages 1996b. *Conference Record of the ACM Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida, Jan.).

Conference Record of the ACM Symposium on Principles of Programming Languages 1998a. *Conference Record of the ACM Symposium on Principles of Programming Languages* (San Diego, California, Jan.).

COOPER, K. AND LU, J. 1997. Register promotion in C programs. See PLDI [1997], 308–319.

COOPER, K. AND SIMPSON, L. T. 1995. SCC-based value numbering. Tech. Rep. CRPC-TR95636-S, Rice University. Oct.

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. 1999. Reconciling buffer management with persistence optimisations. See Morrison et al. [1999], 51–63.

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.

DEAN, J., CHAMBERS, C., AND GROVE, D. 1995. Selective specialization for object-oriented languages. See PLDI [1995], 93–102.

DEBRAY, S., MUTH, R., AND WEIPPERT, M. 1998. Alias analysis of executable code. See Conference Record of the ACM Symposium on Principles of Programming Languages [1998a], 12–24.

DEUTSCH, A. 1994. Interprocedural may-alias analysis for pointers: Beyond *k*-limiting. See PLDI [1994], 230–241.

DIWAN, A., MCKINLEY, K. S., AND MOSS, J. E. B. 1998. Type-based alias analysis. See PLDI [1998], 106–117.

DOLBY, J. 1997. Automatic inline allocation of objects. See PLDI [1997], 7–17.

DOLBY, J. AND CHIEN, A. A. 1998. An evaluation of automatic object inline allocation techniques. In Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (Vancouver, British Columbia, Oct.). *ACM SIGPLAN Notices 33,* 10 (Oct.), 1–20.

EMAMI, M., GHIYA, R., AND HENDREN, L. J. 1994. Context-sensitive interprocedural points-to analysis in the presence of function pointers. See PLDI [1994], 242–256.

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.

GHIYA, R. AND HENDREN, L. J. 1996. Is it a tree, a DAG, or a cyclic graph? a shape analysis for heap-directed pointers in c. See Conference Record of the ACM Symposium on Principles of Programming Languages [1996b], 1–15.

GHIYA, R. AND HENDREN, L. J. 1998. Putting pointer analysis to work. See Conference Record of the ACM Symposium on Principles of Programming Languages [1998a], 121–133.

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

GOSLING, J., YELLIN, F., AND THE JAVA TEAM. 1996. *The Java Application Programming Interface*. Vol. 1: Core Packages. Addison-Wesley.

HASTI, R. AND HORWITZ, S. 1998. Using static single assignment form to improve flow-insensitive pointer analysis. See PLDI [1998], 97–105.

HAVLAK, P. 1997. Nesting of reducible and irreducible loops. *ACM Trans. Program. Lang. Syst. 19,* 4 (July), 557–567.

HOSKING, A. L., NYSTROM, N., CUTTS, Q., AND BRAHNMATH, K. 1999. Optimizing the read and write barriers for orthogonal persistence. See Morrison et al. [1999], 149–159.

HUMMEL, J., HENDREN, L. J., AND NICOLAU, A. 1994. A general data dependence test for dynamic, pointer-based data structures. See PLDI [1994], 218–229.

JAGANNATHAN, S., THIEMANN, P., WEEKS, S., AND WRIGHT, A. 1998. Single and loving it: Must-alias analysis for higher-order languages. See Conference Record of the ACM Symposium on Principles of Programming Languages [1998a], 329–341.

LANDI, W. AND RYDER, B. G. 1992. A safe approximate algorithm for interprocedural pointer aliasing. In Proceedings of the ACM Conference on Programming Language Design and Implementation (San Francisco, California, June). *ACM SIGPLAN Notices 27,* 7 (July), 235–248.

LANDI, W., RYDER, B. G., AND ZHANG, S. 1993. Interprocedural modification side effect analysis with pointer aliasing. In Proceedings of the ACM Conference on Programming Language Design and Implementation (Albuquerque, New Mexico, June). *ACM SIGPLAN Notices 28,* 6 (June), 56–67.

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.

LO, R., CHOW, F., KENNEDY, R., LIU, S.-M., AND TU, P. 1998. Register promotion by sparse partial redundancy elimination of loads and stores. See PLDI [1998], 26–37.

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

MORRISON, R., JORDAN, M., AND ATKINSON, M., Eds. 1999. *Proceedings of the Eighth International Workshop on Persistent Object Systems* (Tiburon, California, August 1998). Advances in Persistent Object Systems. Morgan Kaufmann.

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

PLDI 1990. *Proceedings of the ACM Conference on Programming Language Design and Implementation* (White Plains, New York, June). *ACM SIGPLAN Notices 25,* 6 (June).

PLDI 1994. *Proceedings of the ACM Conference on Programming Language Design and Implementation* (Orlando, Florida, June). *ACM SIGPLAN Notices 29,* 6 (June).

PLDI 1995. *Proceedings of the ACM Conference on Programming Language Design and Implementation* (La Jolla, California, June). *ACM SIGPLAN Notices 30,* 6 (June).

PLDI 1997. *Proceedings of the ACM Conference on Programming Language Design and Implementation* (Las Vegas, Nevada, June). *ACM SIGPLAN Notices 32,* 5 (May).

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

POPL 1988. *Conference Record of the ACM Symposium on Principles of Programming Languages* (San Diego, California, Jan.).

PROEBSTING, T. A., TOWNSEND, G., BRIDGES, P., HARTMAN, J. H., NEWSHAM, T., AND WATTERSON, S. A. 1997. Toba: Java for applications – a way ahead of time (WAT) compiler. In *Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems* (Portland, Oregon, June). USENIX. See http://www.cs.arizona.edu/sumatra/toba.

ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. 1988. Global value numbers and redundant computations. See POPL [1988], 12–27.

RUF, E. 1995. Context-insensitive alias analysis reconsidered. See PLDI [1995], 13–22.

SASTRY, A. V. S. AND JU, R. D. C. 1998. A new algorithm for scalar register promotion based on ssa form. See PLDI [1998], 15–25.

SHAPIRO, M. AND HORWITZ, S. 1997. Fast and accurate flow-insensitive points-to analysis. In *Conference Record of the ACM Symposium on Principles of Programming Languages* (Paris, France, Jan.). 1–14.

SIMPSON, L. T. 1996. Value-driven redundancy elimination. Ph.D. thesis, Rice University, Houston, Texas.

STEENSGAARD, B. 1996. Points-to analysis in almost linear time. See Conference Record of the ACM Symposium on Principles of Programming Languages [1996b], 32–41.

SunSoft 1997. *Java On Solaris 2.6: A White Paper*. SunSoft.

WEGMAN, M. N. AND ZADECK, F. K. 1991. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst. 13,* 2 (Apr.), 181–210.

WILSON, R. P. AND LAM, M. S. 1995. Efficient context-sensitive pointer analysis for c programs. See PLDI [1995], 1–12.

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