BYTECODE-LEVEL ANALYSIS AND OPTIMIZATION OF JAVA CLASSES

A Thesis

Submitted to the Faculty

of

Purdue University

by

Nathaniel John Nystrom

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

August 1998

For my parents.

ACKNOWLEDGMENTS

Thanks to my parents and to my brother Mark for their support. Thanks also go to my advisor Tony Hosking for letting me work for him, and to the other members of my thesis committee, Jens Palsberg and Aditya Mathur. I would also like to thank Steve Lennon and Quintin Cutts at the University of Glasgow for their input and bug reports and the rest of the PJama group at Glasgow and SunLabs. Thank you to Kumar Brahnmath for using BLOAT and for finding many of the bugs. Finally, thanks to Aria, Gustavo, Raghu, Mike, Anshul, and Shaogang.

TABLE OF CONTENTS

Page

Page

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABSTRACT

Nystrom, Nathaniel John. M.S., Purdue University, August 1998. Bytecode-Level Analysis and Optimization of Java Classes. Major Professor: Antony Hosking.

The Java virtual machine specification provides the interface between Java compilers and Java execution environments. Its standard class file format is a convenient target for optimization of Java applications, even in environments where source code for both libraries and application is unavailable. Java bytecode can be optimized independently of the source-language compiler and virtual machine implementation. To explore the potential of bytecode-to-bytecode optimization frameworks, we have built a Java class file optimization tool called BLOAT and measured its impact on the performance of several benchmark programs. Our results demonstrate significant improvement in the execution of Java classes optimized by BLOAT, especially on an interpreted virtual machine, but indicate that more aggressive optimizations, particularly those enabled by interprocedural analysis will provide more benefit. We also consider execution in more performance-conscious environments such as just-in-time and off-line compilation.

1 INTRODUCTION

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 interoperate with multiple VM implementations, enabling cross-platform delivery of applications. 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. 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, and independently of the 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, 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.

1.1 Optimization framework

To explore the potential of bytecode optimization we have implemented a framework for analysis and optimization of standard Java class files. We use this framework to evaluate bytecode-level partial redundancy elimination (PRE) [Morel and Renvoise 1979] of both arithmetic expressions and access path expressions [Larus and Hilfinger 1988] and the consequent impact of these optimizations on execution in three execution environments: the interpreted VM of the standard Java Development Kit, the Solaris 2.6 SPARC JIT, and the

Toba system for translating Java classes into C [Proebsting et al. 1997]. PRE automatically removes global common subexpressions and moves invariant computations out of loops. While PRE over arithmetic expressions is certainly valuable, PRE over access path expressions has significant potential for further improvements since it eliminates redundant memory references, which are often the source of large performance penalties incurred in the memory subsystem of modern architectures.

1.2 Measurements

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 individual benchmarks see performance degradation in specific environments. Naturally, absolute improvements are more dramatic for execution environments that are able to exploit the bytecode-level transformations performed by PRE. In particular, substitution of cheaper bytecodes for more expensive equivalents and elimination of array load, getstatic, and getfield bytecodes through PRE of access path expressions has biggest impact in environments where the bytecodes are interpreted.

1.3 Overview

The rest of this thesis is organized as follows. In Chapter 2 we introduce some definitions and describe the static single assignment (SSA) program representation, partial redundancy elimination (PRE), and type-based alias analysis (TBAA). Chapter 3 describes the basic framework for bytecode-to-bytecode analysis and optimization and the implementation of the analyses and optimizations introduced earlier for Java bytecode. Chapter 4 outlines the experimental methodology we used to evaluate the impact of BLOAT's optimizations, followed by presentation of the results of those experiments. We conclude with a discussion of related work and directions for future work.

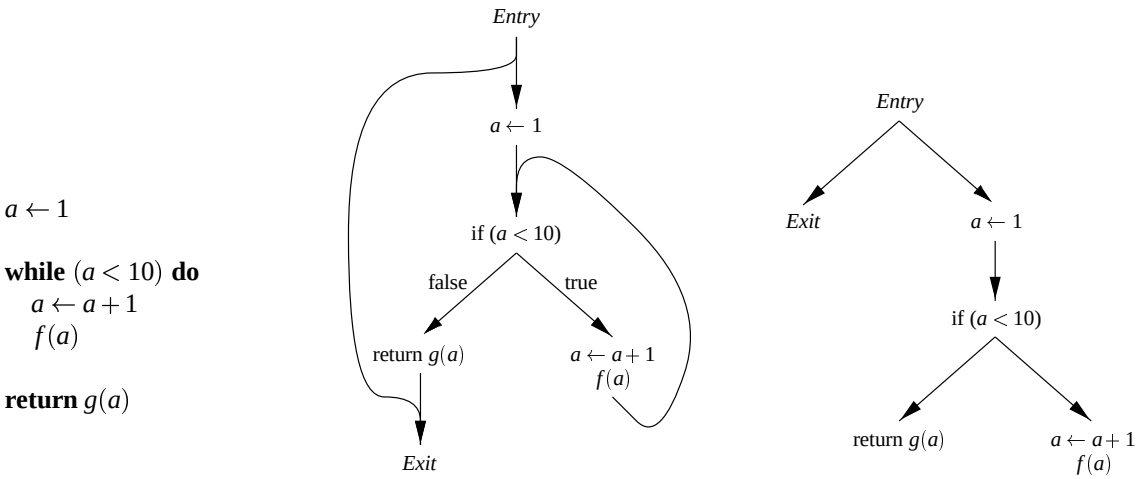
2 BACKGROUND

2.1 Control flow graphs

The instructions of a program can be divided into a set of *basic blocks*, where the flow of control enters a block only at its first instruction and exits only at its last instruction. A *control flow graph* (CFG) is a directed graph where the nodes are the basic blocks of the program and the edges represent branches from one block to another. The graph also contains two additional nodes: an *entry* node and an *exit* node. There is an edge from entry node to any block at which the program can be entered, and there is an edge from any block at which the program can be exited to the exit node. To represent the possibility that the program is not run, there is an additional edge from the entry node to the exit node. A program and its corresponding control flow graph are shown in Figure 2.1. For node, x, $Succ(x)$ is the set of all successors of x; that is, $\{y \mid (x \to y) \text{ is an edge}\}$. $Pred(x)$ is the set of all predecessors of x.

2.1.1 Dominators

We say x *dominates* y, written $x \preceq y$ [Purdom and Moore 1972; Lengauer and Tarjan 1979], if all paths from the entry node to y contain x. We say x *strictly dominates* y, or $x \prec y$, if x dominates y and $x \neq y$. The *immediate dominator* of x, denoted $idom(x)$, is the closest strict dominator of x; that is the strict dominator that is not dominated by any other dominator of x. The entry node has no immediate dominator. All other nodes have a single immediate dominator. We can thus define the *dominator tree* as the tree rooted at the entry node where the parent of a node is its immediate dominator. We denote the children of node x in the dominator tree by $DomChildren(x)$. Figure 2.1c shows the dominator tree



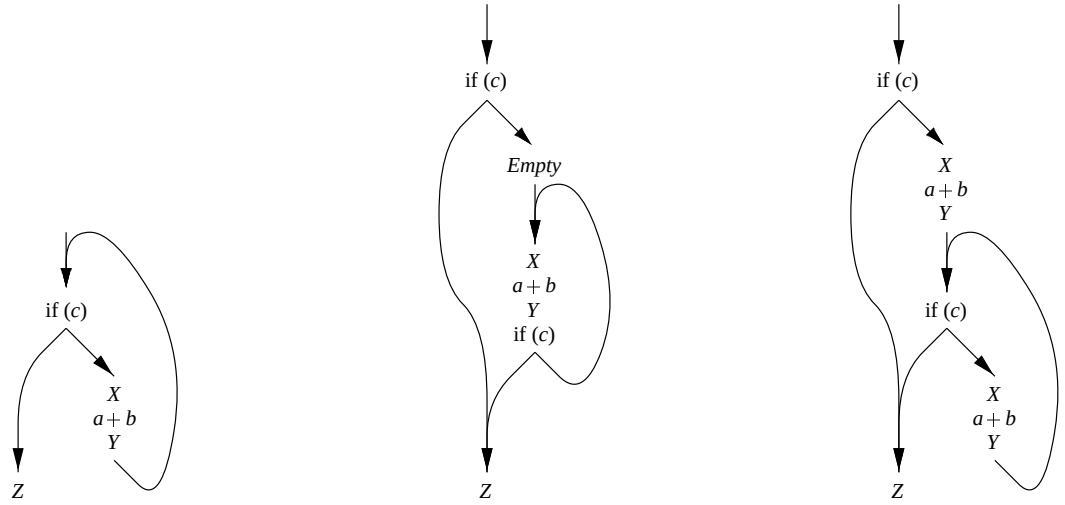
(a) A program (b) Its control flow graph (c) Its dominator tree

Figure 2.1: An example program

for the CFG in Figure 2.1b. For a CFG with V nodes and E edges, the dominator tree can be constructed in $O(E\alpha(E,V))$ time using the algorithm of Lengauer and Tarjan [1979], where α is the inverse of Ackermann's function [Ackermann 1928].

2.1.2 Loops

Transformations that move conditionally executed code are safe only if that code executes under exactly the same conditions after the transformation. Thus, transformations that hoist loop-invariant code of certain loops must move it to a position where it will be executed only if the loop is executed. Certain loop transformations restructure programs to provide safe places to hoist code. These require first that the loops in the control flow graph be identified. A *loop* is a strongly connected component of the CFG. The *loop header* is the block within the loop that dominates all other blocks in the loop. Many loop transformations apply only to loops that are *reducible*. A loop is said to be reducible if its only entry is at the loop header. Havlak [1997] presents an algorithm that identifies loop headers and classifies them as either reducible or irreducible.



(a) Original loop (b) Loop inversion (c) Loop peeling

Figure 2.2: Loop transformations

Loop inversion

Loop inversion [Wolfe 1996; Muchnick 1997] is a transformation that provides a place to insert code above a loop so that the code executes only if the loop is executed at least once. Intuitively, the transformation amounts to converting a while loop into a do-while or repeat loop. For example, in the program in Figure 2.2a we wish to hoist the expression $a+b$ out of the loop. However, the program only evaluates $a+b$ if the loop is entered at least once, so it cannot simply be moved out of the loop. Inverting the loop produces the program in Figure 2.2b in which $a+b$ can be safely hoisted into the empty preheader block. Loop inversion requires a unique loop entry point and thus cannot be used on irreducible loops.

Loop peeling

If the expression for hoisting out of the loop has side effects we can only evaluate the expression in the same context in which it originally occurred. For example, if the expression might throw an exception we must guarantee that any other expression that could also throw an exception is evaluated first. Loop peeling [Wolfe 1996] pulls out the first iteration of the loop by copying the loop body and fixing up the edges in the new

(a) A program (b) Its SSA form

Figure 2.3: A program and its SSA form

control flow graph. To fix the edges, the copied back edges are replaced with edges to the original loop header and edges from outside the loop are made to point to the new loop header. Figure 2.2c demonstrates peeling of the loop in Figure 2.2a. Like inversion, peeling cannot be used on irreducible loops. Peeling subsumes loop inversion, but can lead to exponential growth in the size of the program. Thus, in practice only the innermost loops of a program are peeled since they are the most frequently executed. We can also restrict the transformation to only peel loops which contain code that has side effects and can be hoisted.

2.2 Static single assignment form

Static single assignment (SSA) is a program representation useful for performing optimizations [Cytron et al. 1991]. SSA provides a compact representation of the use-definition relationships among program variables. In SSA form, each use of a variable in the program has only one definition. To distinguish between the definitions of a variable v, we use the *SSA name* for the variable, denoted with a subscript, e.g. v_i. If multiple definitions reach a use, a special definition node, called a ϕ-function, is inserted at the point in the program's control flow graph where the values merge. The ϕ-node has operands for each path into the merge point and serves as a definition for any uses dominated by the merge point. For example, in Figure 2.3, we see a simple program and its SSA form. 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.

2.2.1 Construction

The SSA construction algorithm consists of two phases:

1. Insert ϕ-nodes at the control flow merge points.
2. Transform the program to uniquely identify the definition of each variable occurrence.

To help identify the points in the CFG where control flow paths merge, we define the *dominance frontier* [Cytron et al. 1991] of a node x as

$$DF(x) = \{z \mid \exists y_1, y_2 \in Pred(z) \text{ such that } x \preceq y_1 \wedge x \not\preceq y_2\}$$

that is, z is in $DF(x)$ if x dominates some, but not all, predecessors of z. The dominance frontier of a node can be found in $O(E+V^2)$ time using the algorithm in Figure 2.4 [Cytron et al. 1991]. In practice, this algorithm is usually linear. The dominance frontier of a set of nodes S is

$$DF(S) = \bigcup_{x \in S} DF(x).$$

The *iterated dominance frontier* of S, denoted $DF^+(S)$ [Cytron et al. 1991], is the limit of the sequence

$$\begin{aligned} X_1 &= DF(S) \\ X_{i+1} &= DF(X_i \cup S) \end{aligned}$$

$DF^+(S)$ is often computed using the worklist driven algorithm in Figure 2.5 [Cytron et al. 1991].

Minimal SSA form [Cytron et al. 1991] places a ϕ for a variable at the beginning of each block in the iterated dominance frontier of the set of blocks containing a definition of the variable. This placement may introduce some unnecessary ϕ-nodes. If a ϕ for a variable

```
input:
   A CFG, G = (V,E), with entry node, entry
output:
   DF(x) for all x ∈ V

do
   computeDF(entry)
with
   procedure computeDF(x) begin
      DF(x) ← ∅

      for each y ∈ Succ(x) do
         if (idom(y) ≠ x) then
            DF(x) ← DF ∪ {y}

         for each z ∈ DomChildren(x) do
            computeDF(z)

            for each y ∈ DF(z) do
               if (idom(y) ≠ x) then
                  DF(x) ← DF(x) ∪ {y}
```

Figure 2.4: Computing $DF(x)$

```
input:
   A CFG, G = (V,E), with entry node, entry
   DF(x) for all x ∈ V
   A set of nodes, S ⊆ V
output:
   Output: DF+(S)

worklist ← S
inWorklist ← S

while (worklist ≠ ∅) do
   select and delete a node x from worklist

   for each y ∈ DF(x) do
      DF+(S) ← DF+(S) ∪ {y}

      if (y ∉ inWorklist) then
         inWorklist ← inWorklist ∪ {y}
         worklist ← worklist ∪ {y}
```

Figure 2.5: Computing $DF^+(S)$

```
input:
  A CFG, G = (V,E)
output:
  NonLocals, the set of non-local variables in G

NonLocals ← ∅

for each block b ∈ V do
  Killed ← ∅

  for each instruction, v ← x ⊗ y, in b do
    if (x ∈ Killed) then
      NonLocals ← NonLocals ∪ {x}
    if (y ∈ Killed) then
      NonLocals ← NonLocals ∪ {y}
    Killed ← Killed ∪ {v}
```

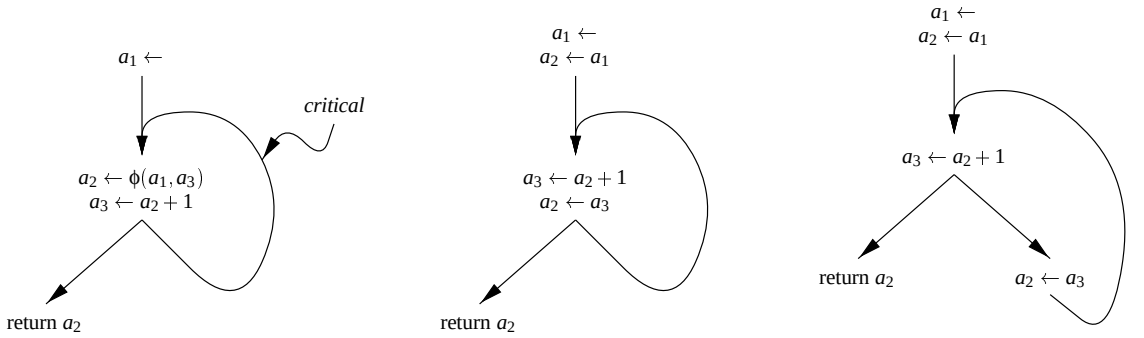
Figure 2.6: Detecting non-local variables

is placed outside the live range of a variable, it will not be used by any real occurrence of the variable and can be eliminated. *Pruned* SSA form [Cytron et al. 1991] only places ϕ-nodes for a variable within the variable's live range, but it requires that liveness analysis be performed first. A variation on this which does not require liveness analysis is *semi-pruned* form [Briggs et al. 1997]. This form inserts ϕ-nodes only for variables that are live in more than one basic block, thus ignoring many short-lived temporaries. Such variables can be detected with the algorithm in Figure 2.6 [Briggs et al. 1997]. The ϕ placement algorithm can then simply ignore any variables not in the *NonLocal* set.

After ϕ-nodes are inserted, SSA construction proceeds to transform the program so that each variable has the single-assignment property. To distinguish between different definitions of each variable, the SSA construction algorithm of Cytron et al. [1991] assigns a version number to each occurrence of the variable. If two occurrences have the same version number they share the same definition. We present the SSA renaming algorithm in Figure 2.7. We denote an occurrence of variable v by $\langle v \rangle$. The algorithm processes variable occurrences in a pre-order traversal of the dominator tree. The current version number for each variable is maintained using a stack. At an assignment or a ϕ-node, the definition is assigned a new version and the version number is pushed onto the stack. At a use of a variable the version number is fetched from the top of the stack. After processing a basic

```
input:
  A CFG, G, after ϕ-nodes are placed
output:
  The SSA form of G

do
  for each variable v do
    Stack(v) ← ∅
    Counter(v) ← 1

  renameBlock(entry)
with
  procedure renameBlock(block) begin
    for each variable v do
      TopOfStack(v) ← top(Stack(v))

    for each ϕ-node, ⟨v⟩ ← ϕ(...), in block do
      Version(⟨v⟩) ← Counter(v)
      push Counter(v) onto Stack(v)
      Counter(v) ← Counter(v) + 1

    for each instruction, ⟨v⟩ ← ⟨x⟩ ⊗ ⟨y⟩, in block do
      Version(⟨x⟩) ← top(Stack(x))
      Version(⟨y⟩) ← top(Stack(y))
      Version(⟨v⟩) ← Counter(v)
      push Counter(v) onto Stack(v)
      Counter(v) ← Counter(v) + 1

    for each succ ∈ Succ(block) do
      for each ϕ-node, ⟨v⟩ ← ϕ(...), in succ do
        ⟨v⟩ ← the block-operand of ϕ(...)
        Version(⟨v⟩) ← top(Stack(v))

    for each child ∈ DomChildren(block) do
      renameBlock(child)

    for each variable v do
      pop Stack(v) until top(Stack(v)) = TopOfStack(v)
```

Figure 2.7: SSA renaming



(a) A program with a critical edge

(b) Incorrect ϕ replacement without splitting critical edges

(c) Correct ϕ replacement with splitting critical edges

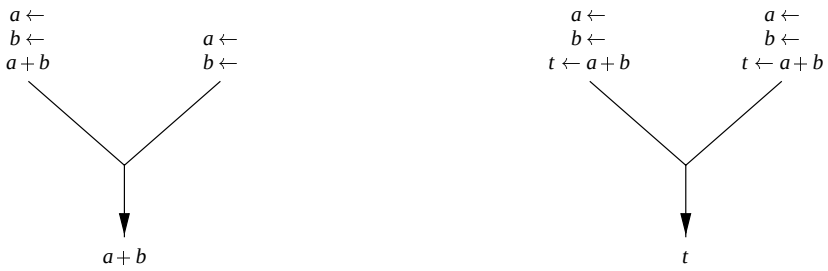
Figure 2.8: ϕ replacement with critical edges

block the stack is popped, so that its siblings in the dominator tree can use the version numbers live at the end of the parent block. Stoltz et al. [1994] introduce the factored use-def (FUD) chain representation, in which each use of a variable has a pointer back to its definition. This representation is more space efficient than version numbering. The construction of FUD chains is nearly identical to traditional SSA construction

2.2.2 Destruction

To translate from SSA form back into real code ϕ-nodes must be removed. A ϕ is removed by placing copy statements in the predecessors of the block containing the ϕ. To ensure that there is a safe place to insert the copies *critical edges* in the flow graph must first be removed. A critical edge is an edge from a block with more than one successor to a block with more than one predecessor. A critical edge can be removed by splitting it; that is, by placing an empty block along the edge. In Figure 2.8a we see an example program with a critical edge. If the edge is not split before ϕ replacement, the program in Figure 2.8b results. Note that the incorrect value of a_2 is returned because $a_2 \leftarrow a_3$ was evaluated before the loop exited. In Figure 2.8c the copies are inserted correctly.

Since optimizations potentially move code, we cannot assume that the renamed variable v_i maps back to its original name v. Again using the example program in Figure 2.8b, we see that the live range of a_2 overlaps that of a_3. Therefore, a_2 and a_3 must be assigned



(a) Before PRE (b) After PRE

Figure 2.9: Example of PRE

different names during code generation. A graph coloring algorithm [Chaitin 1982] can be used to assign a new name to each version of the variables in the program.

2.3 Partial redundancy elimination

Partial redundancy elimination (PRE) [Morel and Renvoise 1979] is a powerful global optimization technique that subsumes the more standard common subexpression elimination (CSE). Unlike CSE, PRE eliminates computations that are only partially redundant; that is, redundant on some, but not all, paths to some later reevaluation. By inserting evaluations on those paths where the computation does not occur, the later reevaluation is made fully redundant and can be eliminated and replaced instead with a use of the precomputed value. This is illustrated in Figure 2.9. In Figure 2.9a, both a and b are available along both paths to the merge point, where expression $a+b$ is evaluated. However, this evaluation is partially redundant since $a+b$ is available on one path to the merge but not both. By hoisting the second evaluation of $a+b$ into the path where it was not originally available, as in Figure 2.9b, $a+b$ need only be evaluated once along any path through the program, rather than twice as before.

2.3.1 SSAPRE

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. For a program with n lexically-distinct expressions, SSAPRE's total time is $O(n(E+V))$.

The algorithm makes separate passes over the program for each lexically-distinct, first-order expression. An expression is *first-order* if all of its subexpressions have no subexpressions and no side effects. All subexpressions of a first-order expression are thus local variables or constants. The algorithm assumes that each occurrence of the expression will be saved to or reloaded from a temporary t. Partially redundant occurrences of the expression are eliminated while simultaneously constructing the SSA form for t. The algorithm inserts ϕ-nodes for each *expression* on which it is performing PRE. These ϕ-nodes are denoted Φ to distinguish them from the ϕ-nodes for local *variables* in the program. Code is hoisted into non-redundant paths by inserting code at the predecessors of blocks containing Φ-nodes. We present a summary of the algorithm. For further details, consult Chow et al. [1997].

1. For each lexically-distinct, first-order expression in the program, insert a list of the occurrences of that expression, sorted by their pre-order positions in the CFG, onto a worklist.

2. If the worklist is empty, stop. Otherwise, select and remove from the worklist a list of occurrences, say for the expression E.

3. Place Φ-nodes for E. Since we cannot immediately determine if an occurrence of E is reloaded from or saved to the temporary t, we assume that all occurrences will be saved to t and thus insert Φ-nodes at the iterated dominance frontier of the set of blocks containing occurrences of E.

4. Build the SSA form for the occurrences of E. We mark with a version number each occurrence, including Φ-nodes and their operands, so that if two occurrences have the same version number, they evaluate to the same value. The renaming step is similar to the SSA renaming step presented in Section 2.2 in that it processes occurrences of

the expression in a pre-order traversal of the dominator tree, saving versions of the occurrences onto a renaming stack. A new version of the expression is created and pushed onto the stack when an occurrence is encountered whose local variables have different SSA versions than those of the expression on top of the renaming stack.

5. Determine which Φ-nodes are *down safe*; that is, the Φ-nodes that will be used at least once on all paths from the Φ-node to the exit node.
6. Determine at which Φ-nodes the expression will be available after code insertions are performed. Code can be inserted safely only at the end of the predecessor blocks of these Φ-nodes. Code will be inserted in a predecessor if the Φ will be available and if there is no occurrence with the same version as the Φ-operand that dominates the predecessor.
7. Determine which occurrences of *E* should be saved to a temporary and which should be reloaded. Determine at which Φ-operands code should be inserted to evaluate the expression on non-redundant paths.
8. Perform the code motion. If an occurrence of *E* is replaced by a temporary *t* and the parent of *t* is now a first-order expression, insert a list of the occurrences of the parent into the worklist.
9. Goto step 2.

2.4 Other optimizations

SSA form can also be used to construct other global optimizations, including dead code elimination [Cytron et al. 1991], constant propagation [Aho et al. 1986; Wegman and Zadeck 1991; Wolfe 1996], value numbering [Alpern et al. 1988; Rosen et al. 1988; Cooper and Simpson 1995; Simpson 1996; Briggs et al. 1997], and constant folding and algebraic simplification [Aho et al. 1986; Simpson 1996]. Optimization algorithms based on SSA all exploit its sparse representation for improved speed and simpler coding of combined local and global optimizations.

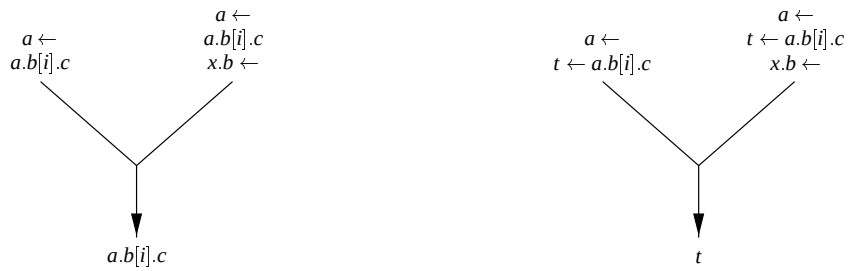
Constant propagation [Aho et al. 1986; Wegman and Zadeck 1991; Wolfe 1996] replaces uses of a variable defined by a constant, say $a \leftarrow 1$, with the constant expression, in this case 1. Copy propagation [Aho et al. 1986] replaces uses of a variable defined by another variable, say $a \leftarrow b$, with the source variable. After propagation, the original assignment can be eliminated. In SSA form, these optimizations are simply a matter of replacing all variables with the same version with the right hand side of the assignment.

Constant propagation can enable other optimizations such as constant folding, which replaces an arithmetic expression with constant operands, say $1+2$, with its value, in this case 3. Constant folding can also fold a conditional branch, such as $\mathsf{if}\ 0 < 1$, into an unconditional branch, potentially making code unreachable and subject to elimination. Algebraic simplification takes advantage of algebraic identities, such as $a+0=a$, to replace arithmetic expressions with simpler expressions. Both constant folding and algebraic simplification can be combined with value numbering for better results. Value numbering maps each expression in a program to a number such that if two expressions have the same number they must have the same value. An expression can then be considered constant if it has the same value number as a constant and if it has no side effects.

Dead code elimination [Cytron et al. 1991] eliminates code which is either unreachable or has no effect on the program's behavior. The standard SSA-based dead code elimination algorithm first marks as live any expression with a side effect. Any definition of a variable used in a live expression and any code which could affect the reachability of a live expression is also marked live, and so on, recursively. Any expressions not marked live can be eliminated from the program.

2.5 Type based alias analysis

An *access path* [Larus and Hilfinger 1988; Diwan et al. 1998] is a non-empty sequence of memory references, as specified by some pointer expression in the source program, for example, the Java expression $a.b[i].c$. 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. Since there could exist more than one pointer to the same object,



(a) Before PRE (b) After PRE

Figure 2.10: PRE for access paths

before eliminating redundant access path expressions, one must first disambiguate memory references sufficiently to be able to safely 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 2.10. The expression $a.b[i].c$ will be redundant at some subsequent reevaluation so long as no store to any of a, $a.b$, i, $a.b[i]$, or $a.b[i].c$ occurs on the code path between the first evaluation of the expression and the second. In other words, if there are potential aliases to any of the locations along the access path through which those locations *may* be modified between the first and second evaluation of the expression, then that second expression cannot be considered redundant. In the example, $x.b$ might alias the same location as $a.b$, so we cannot reuse $a.b$ along the right edge since the assignment to $x.b$ could change the value of $a.b$. If further analysis proved that $x.b$ did not alias the same location as $a.b$, then we could perform PRE to arrive at the program in Figure 2.10b.

2.5.1 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

Table 2.1: Access expressions

Notation	Name	Variable accessed
$p.\mathsf{f}$	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

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 2.1 summarizes the two kinds of access expressions in Java. Without loss of generality, our notation will assume that distinct fields within an object have different names.

A variable is a storage location and has an associated type, sometimes called its *compile-time* type. Given an access path p, then the compile-time type of p, written $\mathit{Type}(p)$, 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 $\mathit{Subtypes}(T)$ to denote all subtypes of type T. Thus, an access path p can legally access variables of type $\mathit{Subtypes}(\mathit{Type}(p))$.

[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.

2.5.2 TBAA

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. In a type-safe language such as Java, only type-compatible access paths can alias the same variable. 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

$$\mathit{TypeDecl}(AP_1, AP_2) \equiv \mathit{Subtypes}(\mathit{Type}(AP_1)) \cap \mathit{Subtypes}(\mathit{Type}(AP_2)) \neq \emptyset$$

A more precise alias analysis will distinguish accesses to fields that are the same type yet distinct. This more precise relation, $\mathit{FieldTypeDecl}(p,q)$, is defined by induction on the structure of p and q in Table 2.2. 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 2.2 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
4. Two array accesses may be aliases if they may access the same array (the subscript is ignored)

Table 2.2: *FieldTypeDecl*(AP_1, AP_2)

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

5. All other pairs of access expressions are possible aliases if they have common subtypes

2.5.3 Analyzing 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 analyzed. 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 may require re-analysis of the entire closure if any one class is changed to include a new assignment.

3 THE ANALYZER

To explore the potential of bytecode-to-bytecode optimization frameworks we have built a Java class file optimization tool called BLOAT (Bytecode-Level Optimization and Analysis Tool). The analysis and optimization framework implemented in BLOAT uses SSA form as the basic intermediate representation [Cytron et al. 1991; Stoltz et al. 1994; Briggs et al. 1997]. 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 partial redundancy elimination of Chow et al. [1997].

3.1 Design

BLOAT is intended to support not only bytecode optimization but also to provide a platform on which to build other Java class editing tools. As such, a generic interface was constructed to edit class files. This interface has been used as the basis for tools to insert profiling code, to strip debugging information from class files, and to instrument bytecode with an extended opcode set to support persistence [Brahnmath 1998]. While the initial implementation recognizes only class *files*, we plan to add facilities to edit classes from any source, notably classes residing within a persistent store [Cutts and Hosking 1997].

3.1.1 Java constraints on optimization

The Java virtual machine specification [Lindholm and Yellin 1996] enumerates the requirements for a Java class to be supported portably in multiple execution environments. Each Java class is compiled into a class file. This class file contains a description of the

names and types of the class's fields and methods, the bytecode for each method, and ancillary information. A method's bytecode is executed with an operand stack. The bytecode consists of approximately 200 opcodes for such operations as pushing constants on the stack, loading and saving to local variables, object and array access, arithmetic, control flow, and for synchronization and exception handling. When the Java virtual machine loads a class the class is passed through a bytecode verifier to ensure that it conforms to structural constraints and that the code for each method is safe. We assume that the classes we are optimizing verify successfully.

Each local variable and stack temporary is typed and different opcodes are used to manipulate values of different types. The precise types of the values on the stack and in local variables are not explicitly given, but can be computed using a simple data flow analysis over the lattice of object types augmented with primitive types. With one exception, the types depend only on the program point and not on the path taken to that point. This exception deals with verification of the jsr and ret opcodes and is described in detail in Section 3.2.3.

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.[1]

Second, the thread model prevents movement of access expressions across synchronization points. Threads act independently on their own working copy of memory and are synchronized with the monitorenter and monitorexit opcodes. 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

[1] 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]).

and synchronization points are places at which TBAA must assume all non-local variables may be modified, either inside the call or through the actions of other threads. When entering a synchronized region, the working copy of memory is reloaded from the JVM's master copy. When exiting the region, the working copy is committed back to the master copy. These semantics allow movement of access expressions within a synchronized region but force reevaluation of those expressions when entering and exiting the region. Common access expressions cannot be considered redundant across these synchronization points.

3.1.2 Class editing interface

The class editing interface provides access to the methods and fields of a Java class. The interface is similar to the Java core reflection API [JavaSoft 1997], which provides read-only access to the public interface of a class. We extend the reflection API to allow access to private and protected members, access to the bytecode of methods and to the class's constant pool. We also add the ability to edit these attributes and generate new class files incorporating the changes.

When the bytecode for a method is edited, it is first converted into a list of instructions and branch labels. We represent branch targets with labels rather than offsets from the branch instruction to allow code to be more easily inserted and removed between the branch and its target. Labels are also used to keep track of the instructions protected by exception handlers, and any line number or local variable debug information in the class file. In addition, similar opcodes, such as those for pushing integers stored in local variables onto the operand stack,[2] are consolidated. When changes made to the class are written out to a class file, branches are resolved and new constants are inserted into the class file's constant pool if necessary.

3.1.3 Control flow graph and expression trees

Once we have converted the bytecode into a list of instructions, we can perform our analyses and optimizations on the method. On top of the list of instructions and labels we

[2] iload, iload_0, iload_1, iload_2, iload_3.

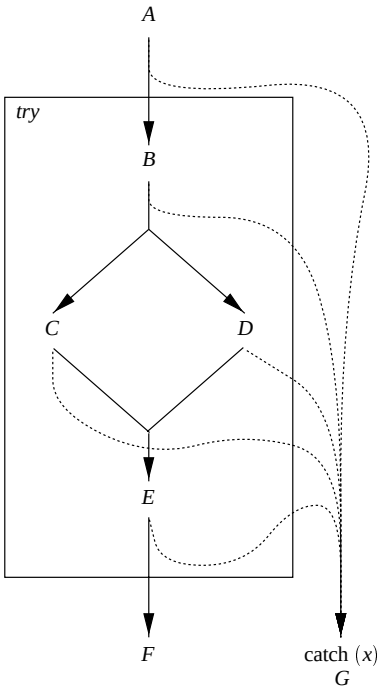


Figure 3.1: Exceptions and critical edges

construct a control flow graph. An expression tree is created for each basic block in the graph, encapsulating the operand stack behavior for the block.

The control flow graph is constructed with a recursive depth-first traversal of the instruction list, following branches to their corresponding targets within the list. Edges are also inserted from the blocks within and just before each protected region (i.e., try blocks) to its exception handler. This ensures that the dominance relation holds correctly from an exception throw-point to its handler, if any. 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 partial redundancy elimination and when translating back from SSA form. As shown in Figure 3.1, the edges inserted for exception handlers are often critical edges. These edges cannot be split without having to create a new exception handler for each such edge; instead they are given special treatment during PRE and SSA destruction.

The expression trees are constructed through a simulation of the operand stack. To preserve the evaluation order of the original bytecode and to save the stack across basic blocks, we insert saves and loads of expressions to stack variables. These variables are treated just

as local variables except that because of their effect on the stack, our optimizations must be more judicious when moving or eliminating them.

Our expression trees contain two node types: statements and expressions. Expressions are typed and can be nested. Statements cannot be nested and thus can only access the operand stack below the incoming height using stack variables. The bytecode verifier ensures that for all paths to a given point in the program, the operand stack is the same height and contains the same types.[3] Therefore, by only inserting, removing, or relocating statement nodes which do not contain stack variables, we can maintain this property throughout our transformations.

3.2 Implementation

Most of the analysis and optimization passes are performed on the CFG; however, some transformations operate directly on the instruction list before CFG construction and after conversion back to an instruction list. BLOAT implements the following analyses and optimizations:

1. Array initializer compaction
2. CFG construction
3. Loop peeling
4. Loop inversion
5. SSA construction
6. Type inference of locals
7. Value numbering
8. PRE of arithmetic and access path expressions
9. Constant/copy propagation
10. Constant folding and algebraic simplification

[3] Again, this is violated by the jsr and ret opcodes.

11. Dead code elimination
12. Liveness analysis
13. SSA destruction
14. Bytecode generation
15. Peephole optimizations

3.2.1 Array initializer compaction

The Sun JDK compiler, javac, generates code for array initializers using a straight-line sequence of array stores, as shown in Figure 3.2. For classes such as java.lang.Character, which have large static arrays, this array initialization code can be tens or even hundreds of kilobytes. In JDK versions prior to 1.1.5, many classes provided with the JDK which contained such initializers failed to pass bytecode verification because they violated the 64K limit on method size. Before CFG construction we translate such initializers into a loop which fills the array from a string inserted in the class's constant pool. This transformation eliminates the unnecessarily large basic blocks for such code and significantly reduces the time for later analysis of these initializers.

3.2.2 Loop transformations

After construction the CFG, we identify loops using Havlak [1997] and perform loop peeling [Wolfe 1996] and loop inversion [Wolfe 1996; Muchnick 1997] transformations. We restrict peeling to the innermost loops in the CFG to prevent the potential exponential growth in code size. Inversion is performed on all other loops, providing a convenient place immediately after the loop iteration condition is tested to hoist loop-invariant code out of the loop body. Code that is loop-invariant and does not throw exceptions can be recognized by PRE and hoisted out of inverted loops. Loop-invariant code which throws exceptions will be made redundant by loop peeling and can then be hoisted by PRE. The loop transformations are performed before SSA construction so that we do not have to maintain SSA form during the transformations.

```
static int[] a = new int[] {
    4, 3, 2, 1, 0
};
```

(a) Java code

```
 0  bipush 10      // push the array size
 2  newarray int   // create the array
 4  dup            // push a copy of the array
 5  iconst_0       // index 0
 6  bipush 4       // value 4
 8  iastore        // store into the array
 9  dup            // push a copy of the array
10  iconst_1       // index 1
11  bipush 3       // value 3
13  iastore        // store into the array
14  dup            // …
15  iconst_2
16  bipush 2
18  iastore
19  dup
20  iconst_3
21  bipush 1
23  iastore
24  dup
25  iconst_4
26  iconst_0
27  iastore
28  putstatic #5   // save the array in int a[]
31  return
```

(b) Bytecode

Figure 3.2: An array initializer

3.2.3 SSA construction

After the control flow graph is constructed we convert the local and stack variables to SSA form. This requires computation of the dominator tree and dominance frontier of the control flow graph [Cytron et al. 1991]. We use the algorithm of Lengauer and Tarjan [1979]. We also identify critical edges and split them to provide a place to insert code during PRE and translation back from SSA form. To reduce the number of ϕ-nodes inserted, we construct the semi-pruned form of SSA [Briggs et al. 1997].

Exceptions

Performing optimizations in the presence of exception handling requires the control flow graph be transformed to indicate the possible effects of explicit and implicit exception throws [Hennessy 1981]. If a variable is used after an exception is caught at an exception

handler, the value of the variable could be any of the values of the variable live within the protected region for that handler. An exception handler could be entered with any of the possible local variable states that occur within its protected region. Using standard SSA form to factor these values together requires inserting edges in the CFG to the exception handler from before and after each store of a local variable within the protected region.[4] In addition, if a new variable is introduced into the program, say to store the value of an expression eliminated by PRE, then the CFG must be adjusted to include an edge before and after the store to that variable. Adding these edges results in basic blocks containing only a store to a variable, which limits the effectiveness of local optimizations.

Rather than use traditional SSA form, we extend SSA to include a special ϕ-node, which we denote ϕ_c, to factor all values of a variable live within the protected region. These ϕ_c-nodes are inserted during the ϕ placement step of SSA construction at the beginning of each exception handler block. During renaming, when a block in a protected region is entered or when a version number is assigned to a variable within the protected region, an operand with that version number is added to the ϕ_c for the variable at the protected region's handler. For example, in Figure 3.3, a_1, a_2, and a_3 are live within the protected region and their definitions are used to define the operands of the ϕ_c in the handler.

When ϕ-nodes are replaced with copies before code generation, for each ϕ_c-operand, we locate the operand's definition and insert a copy from the operand to the ϕ_c target just after the definition. Note that if the definition is above the try block we could potentially have a long path from the definition to the beginning of the protected region where the ϕ_c target is not used but must be live. To alleviate this problem, before SSA construction we split the live range of variables entering the protected region by inserting just before the protected region copies from each variable defined at that point to itself (e.g., $a \leftarrow a$). This forces SSA to assign a new version to the variable immediately before entering the protected region and ϕ_c replacement will later insert its copies for these operands immediately before the protected region rather than higher up in the CFG.

[4]The values of stack variables need not be propagated along these edges since the operand stack is cleared when an exception is thrown.

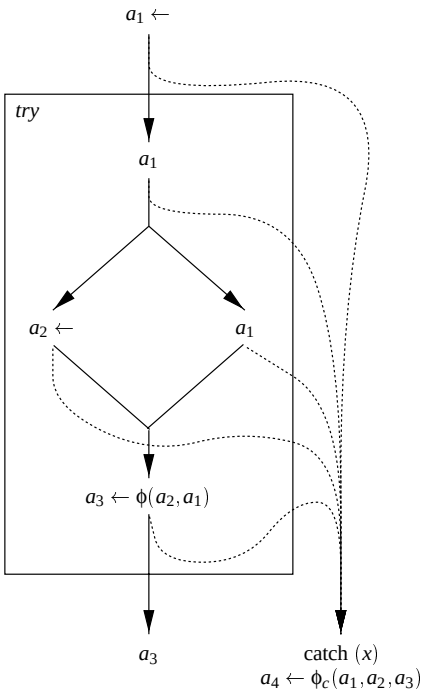


Figure 3.3: Exceptions and SSA

Subroutines

Java compilers translate finally blocks into method-local subroutines [Gosling et al. 1996; Lindholm and Yellin 1996]. Subroutines 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. As shown in the Java program in Figure 3.4, a finally block can be entered from several places within a method. Before every instruction that exits the try block, such as a return or a throw or simply falling off the end, a jsr to the finally block is placed. To permit verification of this construct, the Java VM specification allows any local variable that is not referenced between the jsr and the corresponding ret to retain its type across the subroutine. The consequence of this for SSA is that two variables with incompatible types could be factored together with a ϕ-node at the jsr target and then used again after the ret.

A simple solution to this problem is to inline the subroutine at each jsr. However, inlining could result in a substantial growth in code size. Rather than inlining, we modify

```
try {
    tryIt();
}
catch (Exception e) {
    handleIt();
}
finally {
    finishIt();
}
```

(a) Java code

```
 0  aload_0
 1  invokevirtual #7   // tryIt()
 4  goto 15            // jump over the handler
 7  pop                // pop the exception
 8  aload_0
 9  invokevirtual #6   // handleIt()
12  goto 15
15  jsr 25             // go to the finally block
18  return
19  astore_1           // save the exception to local 1
20  jsr 25             // go to the finally block
23  aload_1            // reload the exception ...
24  athrow             // ... and throw it
25  astore_2           // save the return address
26  aload_0
27  invokevirtual #5   // finishIt()
30  ret 2              // return from the subroutine
```

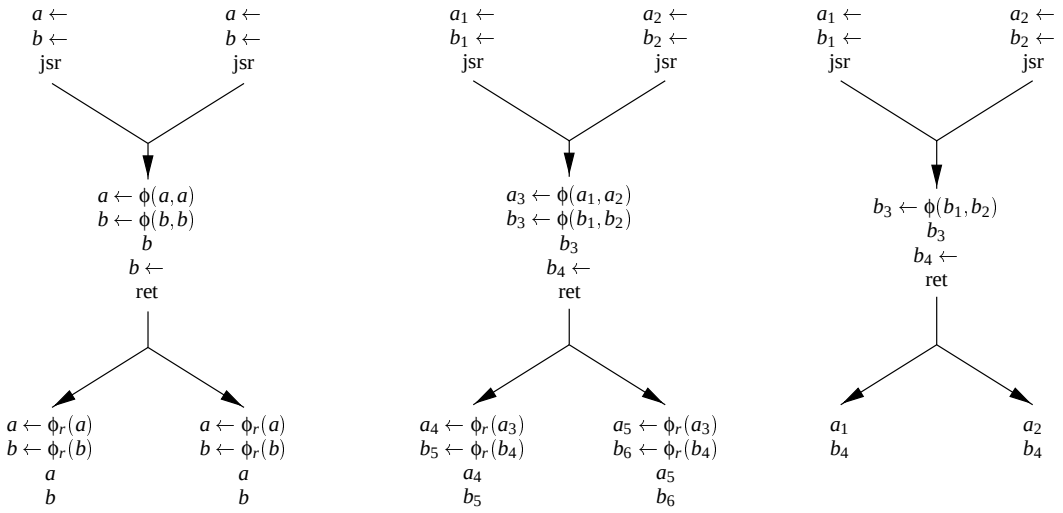
Exception table:

From	To	Target	Type
0	4	7	`java.lang.Exception`
0	15	19	any

(b) Bytecode

Figure 3.4: A Java finally block

the SSA construction algorithm so that if a variable is not redefined within a subroutine, the version number used on entry to the subroutine is propagated back through the ret to the instruction after the jsr. To achieve this, we insert a special ϕ-node, which we will denote by ϕ_r, at each jsr's return site, as illustrated by the example program in Figure 3.5a. Since the return site has only one incoming edge (because we removed critical edges), the ϕ_r has only one operand. The version number of this ϕ_r is used as usual to define the uses it dominates. If a variable has different versions at each of the jsr sites for a given subroutine, there will be a ϕ for the variable at the subroutine entry block. During the SSA renaming step, when we arrive at a ret, we locate the ϕ_r-nodes for the corresponding jsr instructions. If there is a ϕ at the subroutine entry and the version defined by that ϕ is the version on top of the renaming stack when we reach the ϕ_r, we assign the ϕ_r's operand the version of the entry ϕ's operand for the ϕ_r's jsr. Otherwise we assign the version at the ret to the ϕ_r's operand. Once the renaming step is complete, since the ϕ_r have only one operand, we can



(a) ϕ_r placement (b) After ϕ_r renaming (c) Final SSA form

Figure 3.5: ϕ_r example

rename the uses defined by each ϕ_r to use the version of the ϕ_r's operand. The ϕ_r-nodes can then be removed.

3.2.4 PRE of access expressions

To reduce the overhead of pointer traversals within Java we extend partial redundancy elimination to access path expressions. Doing so requires disambiguating memory references sufficiently using alias analysis to be able to detect redundancies among access paths. We combine the SSA-based PRE of Chow et al. [1997] with type based alias analysis [Diwan et al. 1998].

TBAA with Java bytecode

TBAA relies on the declared types of variables in the program. Unfortunately, local variables and operand stack slots in Java bytecode do not have declared types; therefore we must infer them. Since we are only interested in types used in access path expressions we limit ourselves to inference of reference types only.

We use a simplification of the type inference algorithm of Palsberg and Schwartzbach [1994]. Each expression in the program, say *x*, has an associated type variable, denoted

Table 3.1: Type Constraints

Expression	Example	Constraints
formal parameter	x	$\{\text{The declared type of } x\} \subseteq [\![x]\!]$
assignment	$x \leftarrow y$	$[\![y]\!] \subseteq [\![x]\!]$
ϕ-node	$x \leftarrow \phi(y,z,\ldots)$	$[\![y]\!] \subseteq [\![x]\!], [\![z]\!] \subseteq [\![x]\!], \ldots$
stack manipulation	$(x,y) \leftarrow \text{dup}(z)$	$[\![z]\!] \subseteq [\![x]\!], [\![z]\!] \subseteq [\![y]\!]$
exception handler	$x \leftarrow \text{catch}(C)$	$\{C\} \subseteq [\![x]\!]$
array reference	$x[i]$	$[\![x[i]]\!] \subseteq \{\text{The element type of } x\}$
virtual method call	$x.m(y)$	$[\![x.m(y)]\!] \subseteq \{\text{The return type of } m\}$
static method call	$C.m(y)$	$[\![C.m(y)]\!] \subseteq \{\text{The return type of } m\}$
cast	$(C)x$	$[\![(C)x]\!] \subseteq \{C\}$
string literal	"string"	$\{\text{java.lang.String}\} \subseteq [\![\text{"string"}]\!]$
field reference	$x.f$	$\{\text{The declared type of } f\} \subseteq [\![x.f]\!]$
static field reference	$C.f$	$\{\text{The declared type of } f\} \subseteq [\![C.f]\!]$
object allocation	new C	$\{C\} \subseteq [\![\text{new } C]\!]$
array allocation	new $T[i]$	$\{\text{Array of } T\} \subseteq [\![\text{new } T[i]]\!]$
multi-dimensional array allocation	new $T[i][j]$	$\{\text{Array of array of } T\} \subseteq [\![\text{new } T[i][j]]\!]$

$[\![x]\!]$. Constraints are derived relating the type variables with each other and with sets of types. The constraints for Java bytecode are shown in Table 3.1. The constraints are then solved using the algorithm in Figure 3.6, resulting in a set of types associated with each expression. The type of an expression is the common supertype of its associated set of types. Since we are only interested in the types for a single method, we remove those parts of the algorithm of Palsberg and Schwartzbach [1994] that insert and solve constraints for calls and instead use the declared return type and parameter types.

Value numbering

There is a cyclic dependency between copy propagation and PRE. PRE operates by recognizing lexically equivalent subexpressions. If a common expression on the right hand side of an assignment is eliminated and so replaced with a local variable, a copy will be created. Further copy or constant propagation could enable another round of PRE by replacing a variable within an expression with a constant or another variable and so make the expression redundant.

```
input:
   A method with set of expressions X
output:
   A set of types Types(e) for all e ∈ X

do
   for each expression, e ∈ X do
      if (there is a constraint for e, c, as defined in Table 3.1) then
         insert(c)
with
   procedure insert(constraint) begin
      if (constraint is a start constraint, C ∈ v) then
         propagate(C, v)
      else if (constraint is a propagation constraint, v ⊆ w) then
         add edge v → w
         for each C ∈ Types(v) do
            propagate(C, w)

   procedure propagate(C, v) begin
      if (C ∉ Types(v)) then
         Types(v) ← Types(v) ∪ C
         for each edge v → w do
            propagate(C, w)
```

Figure 3.6: Type inference algorithm

Using value numbering [Cooper and Simpson 1995; Simpson 1996; Briggs et al. 1997] avoids the need for repetitive iteration of PRE interleaved with constant/copy propagation. Value numbering assigns a number to each expression in a program such that if two expressions have the same number they must have the same value. Rather than basing equivalence of expressions purely on their lexical equivalence, we use the value numbering approach of Simpson [1996]. Then during SSAPRE renaming, rather than comparing the SSA version of variables within two expressions, we compare their value numbers.

Extending SSAPRE

To extend SSAPRE to recognize access paths, we identify *alias definition points*: those code locations where potentially aliased variables may be modified. These include not only explicit assignment to possible aliases, but also calls and monitor synchronization points. Because we do no interprocedural analysis, we cannot assume a call will not modify an expression through an alias. We include monitor synchronization points in order to satisfy

Java's thread model: at synchronization points any changes made to a variable in another thread must be made visible to the current thread.

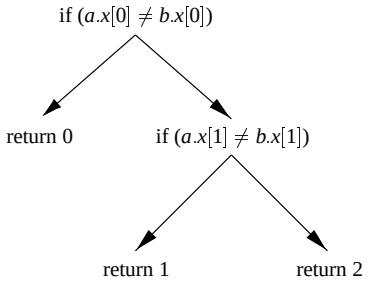
In addition, to prevent hoisting of code which can throw exceptions out of protected regions, we identify those edges in the control flow graph which go from a block not in a protected region to a block in the region. We call both these edges and alias definition points *kill points* since they kill any previous definition of expressions outside before the point.

Having identified kill points, we can perform SSAPRE as summarized in Section 2.3.1 by interleaving alias definition points with each pre-order list of occurrences of access path expressions and by interleaving protected region entry points with each list of occurrences of exception throwing expressions. During Φ placement for an expression, we place a Φ at any block in the iterated dominance frontier of the set of blocks containing either an occurrence of the expression or a kill point for the expression. During the renaming step, when a kill point for the expression is encountered, we kill the definition at the top of the renaming stack as if defining a new version of the expression.

PRE and Java bytecode

There are two special cases for performing PRE on access paths in Java. Fields that are declared *volatile* must be reloaded from the program's master copy of memory each time they are accessed. Therefore they cannot be eliminated at all. Fields declared *final*, however, cannot be redefined at all, much less through an alias, so we are free to move them across alias definition points.

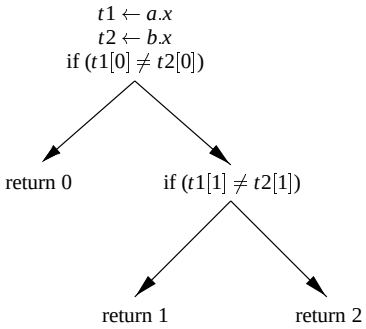
One difficulty encountered with PRE in Java bytecode is the addition of extra loads and stores. Because the bytecode is executed on an operand stack and we save and reload expressions from local variables, PRE can sometimes produce longer, and consequently slower, bytecode. Consider the code in Figure 3.7a. PRE transforms this to the code in Figure 3.7b. Because of the added overhead of the extra stores and reloads for *t1* and *t2*, the shortest path through the program, returning 0, is four instructions longer after PRE than before. The paths to `return 1` and to `return 2` are also longer than before PRE, by




```
 0  aload_1          // push a
 1  getfield #14     // get a.x
 4  iconst_0
 5  iaload           // load a.x[0]
 6  aload_2          // push b
 7  getfield #14     // get b.x
10  iconst_0
11  iaload           // load b.x[0]
12  if_icmpeq 17
15  iconst_0
16  ireturn
17  aload_1          // push a
18  getfield #14     // get a.x
21  iconst_1
22  iaload           // load a.x[1]
23  aload_2          // push b
24  getfield #14     // get b.x
27  iconst_1
28  iaload           // load b.x[1]
29  if_icmpeq 34
32  iconst_1
33  ireturn
34  iconst_2
35  ireturn
```

Path to return 0:
0, 1, 4, 5, 6, 7, 10, 11, 12, 15, 16

(a) Before PRE

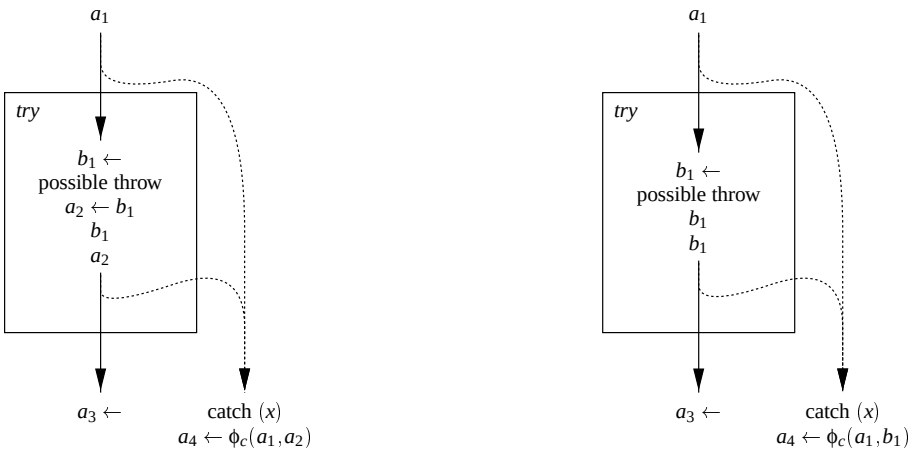



```
 0  aload_1          // push a
 1  getfield #14     // get a.x
 4  astore_3         // save a.x to t1
 5  aload_2          // push b
 6  getfield #14     // get b.x
 9  astore 4         // save b.x to t2
11  aload_3          // push t1
12  iconst_0
13  iaload           // load t1[0]
14  aload 4          // push t2
16  iconst_0
17  iaload           // load t2[0]
18  if_icmpeq 23
21  iconst_0
22  ireturn
23  aload_3          // push t1
24  iconst_1
25  iaload           // load t1[1]
26  aload 4          // push t2
28  iconst_1
29  iaload           // load t2[1]
30  if_icmpeq 35
33  iconst_1
34  ireturn
35  iconst_2
36  ireturn
```

Path to return 0:
0, 1, 4, 5, 6, 9, 11, 12, 13, 14, 16, 17, 18, 21, 22

(b) After PRE

Figure 3.7: PRE can produce longer bytecode



(a) Before copy propagation (b) After incorrect copy propagation

Figure 3.8: ϕ_c-nodes and copy propagation

two instructions. This problem could be remedied by more careful analysis of the cost of eliminating an expression and by attempting to cache the value of a redundant expression on the operand stack rather than in a local variable. JIT translation of the bytecode can eliminate these extra loads and stores since both local variables and operand stack slots are transformed into native variables subject to register allocation.

3.2.5 Constant and copy propagation

The constant/copy propagation algorithm is based on standard techniques [Aho et al. 1986; Wolfe 1996]. Because exceptions in Java are precise, we cannot propagate copies to ϕ_c-nodes. Doing so could result in an assignment to a variable before an exception is thrown that, before copy propagation, occurred after the exception was thrown. Consider Figure 3.8. In Figure 3.8a, there is a copy from b_1 to a_2 after the possible throw. After copy propagation, b_1 is an operand to the ϕ_c-node in Figure 3.8b. When the ϕ_c is removed, the copy expression $a_4 \leftarrow b_1$ will be placed at the definition of b_1, before the possible throw. In the original code, a_4 was equal to a_1 until the assignment after the throw.

3.2.6 Liveness analysis

Following optimizations, standard liveness analysis [Aho et al. 1986] is used to build an *interference graph*: an undirected graph $G = (V, E)$, where V is the set of variables in the program and E is a set of edges, such that if variables v and w are simultaneously live, there is an edge from v to w. A variable is *live* if it could be needed later, i.e., v is live at a program point p if there is a path in the CFG starting at p that uses v.

The interference graph is constructed by tracing backward through the CFG from each use of a variable, v, along all paths to its definition. If the definition of another variable, w, is encountered, an edge is added between v and w.

Interference graph construction is complicated by exception handling. In order to insert code for ϕ_c-nodes, we must ensure that the target of the ϕ_c is live throughout the protected region as well as after its definition. However, we do not want the ϕ_c target to conflict with its operands. Our solution is to make the ϕ_c target conflict with all variables that conflict with its operands. This solution could be overly conservative in that it could make the live range of the ϕ_c target unnecessarily long, introducing edges in the interference graph that do not represent an actual live range conflict in the program. This problem is solved, in part, by inserting self-copies just before entering the protected region, as described in Section 3.2.3. This prevents the live range of the ϕ_c target from extending above the protected region, but does not keep the live range from extending below the protected region. However, ϕ_c operand conflicts below the protected region occur rarely in practice because there is often a ϕ just below the protected region factoring the ϕ_c target back into the mainline of the program.

3.2.7 SSA destruction

After computing the interference graph, graph coloring with coalescing [Chaitin 1982; Briggs et al. 1994] maps different SSA versions of each local variable back to a single hard local variable. Our coloring algorithm favors placing commonly used variables, such as those nested within loops, in the lower four local variable indices, since these indices are accessed through a one-byte bytecode instruction rather than through a two-byte instruction

containing the local variable index. These lower four indices may also be candidates for special treatment as registers by naïve VM implementations.

The algorithm first attempts to coalesce nodes in the interference graph to eliminate copies. Each node in the graph is given a weight based on the number and position of the occurrences of the variables the node represents. Let $d(u)$ denote the loop-nesting depth of variable occurrence u. $d(u)$ is the number of surrounding loops containing u. The weight of node v is then given by

$$weight(v) = \sum_{u \in occurrences(v)} 10^{d(u)}$$

The weight represents the number of loads or stores of the variable assuming each loop body is executed 10 times.

Now let $deg(v)$ be the degree of node v. For all copies in the program, $v \leftarrow w$, including the copies that would be generated when ϕ-nodes are replaced, we select the copy with the maximum

$$\frac{weight(v)}{deg(v)} + \frac{weight(w)}{deg(w)}$$

where there is no edge from v to w in the interference graph. The nodes for v and w are then coalesced into one node, adjusting the edges so that v and w together conflict with the union of the nodes they conflicted with individually. We divide by the degree of the node to favor those nodes that have a shorter live range, so the coalesced node will conflict with a fewer number of other nodes. This process repeats until no copies can be found to consolidate.

Once nodes are coalesced, the graph is colored by first pre-coloring any formal parameters for the method, then repeatedly selecting the node v with the maximum $weight(v)$ and assigning it the lowest local variable index for which there is not a conflicting node already colored. In this stage, variables of type long and double have half their original weight since they take up two local variable indices rather than just one.

3.2.8 Code generation

Each expression tree is converted back to stack code by a pre-order traversal of the tree. Once in instruction list form, we perform peephole optimization of redundant loads and stores and for better utilization of the operand stack.

4 EXPERIMENTS

To evaluate the impact of our optimization framework 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.

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 are summarized in Table 4.1.

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 1.0 (Toba) [Proebsting et al. 1997; Toba 1998]. In each environment we ran both unoptimized and optimized versions of each benchmark. Where Java source code for a benchmark was available, it was compiled using the standard JDK 1.1.6

Table 4.1: 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).

javac compiler, without the -O optimization flag since in many cases this generates incorrect code. Informal observation indicates that this flag has little impact on the performance of our benchmarks.

4.3.1 JDK

JDK is the standard 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], 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 performs on-demand dynamic translation of Java bytecode to native SPARC instructions. It uses non-native threads and includes 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]. Since Toba only works with the JDK version 1.0.2 core classes, we optimized the benchmarks for execution in the Toba environment against the JDK version 1.0.2 core classes, to form the closure of optimized classes necessary to execute the benchmark in Toba. Similarly, the closure of unoptimized core classes was also formed for unoptimized benchmark execution.

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. Using this optimization level provides an opportunity to see optimizations that BLOAT misses.

4.4 Metrics

For each benchmark we took measurements for both the optimized and unoptimized classes. Our metrics include:

- 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 from an instrumented version of the JDK version 1.1
- native 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 [Enbody 1998] 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.

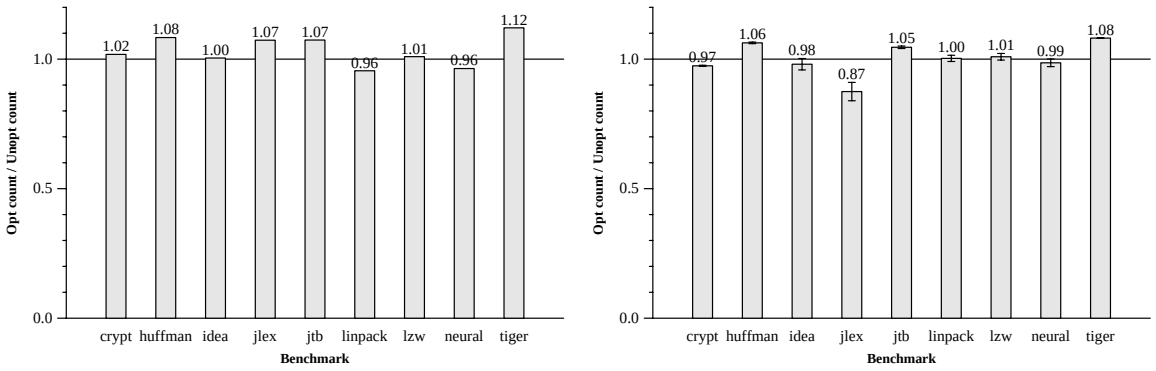
The physically addressed instruction cache on the UltraSPARC means that programs can exhibit widely varying execution times from one invocation to the next, since each invocation process will have different mappings from virtual to physical addresses resulting in randomized instruction cache placement. Thus, the elapsed time and cache-related metrics were obtained for 10 separate runs and the results averaged.

4.5 Results

The results are reported in Tables 4.2-4.10, one per benchmark. For each metric we give raw counts only for the unoptimized classes, and then only for the totals. All other counts are expressed as a percentage of this unoptimized total. Thus, if the total count for a metric obtained using the unoptimized classes is T, all other results for that metric c are reported as the percentage $100c/T$. This includes the breakdowns of total instruction counts (both bytecode and native). Reporting the results in this way enables the relative effect of optimization on specific instructions to be gauged more easily. We report only those native instructions whose execution frequencies change noticeably with optimization.

In the graphs accompanying the text, we show the relative change in each benchmark from the unoptimized execution to the optimized; that is, if the unoptimized count for given metric is c_1 and the optimized count is c_2, the graph shows c_2/c_1. The error bars in the graphs represent 90% confidence intervals.

As expected, we see an increase in dynamic bytecode execution counts for many of the benchmarks. As discussed earlier, these overheads are mainly due to introduction of extra



(a) Bytecodes executed (b) Time (cycles)

Figure 4.1: JDK metrics

loads from and stores to temporaries introduced by PRE for partially-redundant expressions whose values are not used on all paths. There are several interesting bytecode-level optimization effects:

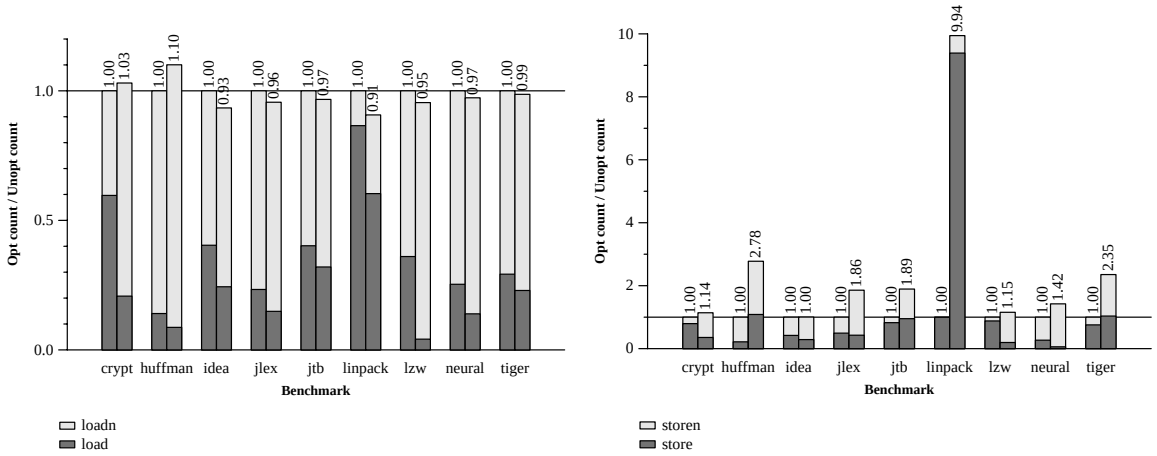
- Coloring of locals significantly reduces the number of extended length load and store bytecodes, replacing them with their shorter forms
- PRE over access paths can significantly reduce the number of arrayload, getfield and getstatic bytecodes
- Constant propagation via value numbering permits sometimes significant conversion of conditional jumps from two-operand (ifcmp) to single-operand (if).

As expected, we see significant variation in the instruction cache metric, resulting in variation in the number of cycles.

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

4.5.1 JDK

The JDK results are best appreciated by first considering the Java bytecode distribution for optimized and unoptimized classes. All the benchmarks, except linpack and lzw, see an increase in the number of bytecodes executed for optimized code, as shown in Figure 4.1a. The increase is significant for huffman, jlex, jtb, and tiger (7–12%) due mostly to insertion



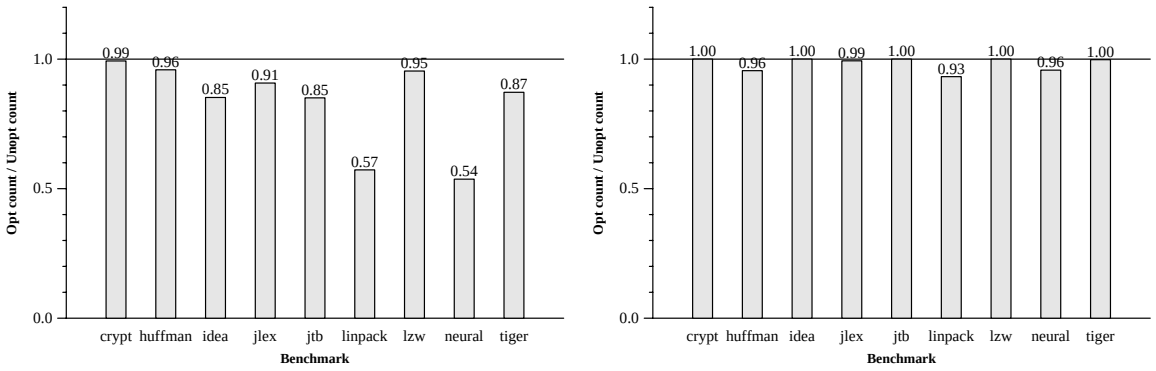
(a) load vs. load*n* (b) store vs. store*n*

Figure 4.2: Replacing load**s** and store**s** by shorter bytecodes

of additional load, store, and dup bytecodes to hold commonly used value in local variables and operand stack slots. Figure 4.1b demonstrates that, somewhat surprisingly, an increase in number of bytecodes executed does not always translate into a decline in performance for JDK, since the effects of coloring for allocation of local variables are strong, with many of the longer load and store bytecodes being replaced with their short forms. Figure 4.2 shows this change in the mix of bytecodes. The effect is most notable with lzw where the frequency of the load bytecodes decreases from 11% to 1% of the total bytecode count and the frequency of load*n* increases from 19% to 28%. This results in less overhead in the interpreter's bytecode dispatch loop. The large increase in stores for linpack is due to PRE's elimination of redundant arithmetic expressions. Those benchmarks which show an increase in the frequency of store versus store*n* bytecodes have formal parameters occupying several of the lower four local variable indices which prevent these indices from being allocated for other variables.

A similar, but less pronounced, effect results from conversion of two-operand ifcmp bytecodes to one-operand if bytecodes with constant propagation and folding.

As Figure 4.3a shows, all benchmarks see a decrease, often significant, in the frequency of getfield bytecodes due to elimination of redundant access path expressions. The jtb, neural, and tiger benchmarks, which have a high initial frequency of getfield bytecodes (12%, 9%, and 13%, respectively), show a 13–46% decrease in the number of getfield



(a) getfield bytecodes (b) arrayload bytecodes

Figure 4.3: Memory access bytecodes

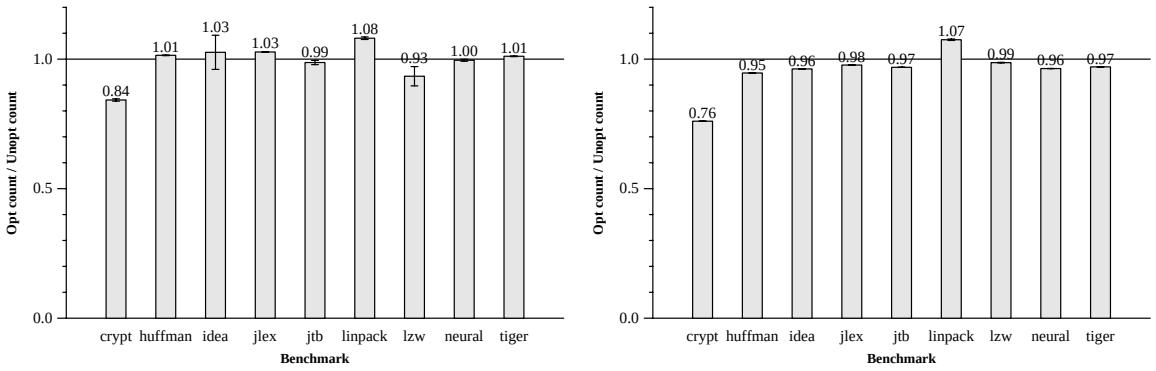
bytecodes executed. Linpack executes 43% fewer getfield bytecodes, but these bytecodes represent only 0.02% of the total bytecodes executed.

The relative change in arrayload bytecodes is shown in Figure 4.3b. The huffman, linpack, and neural benchmarks, which have heavy array use (9%, 4%, and 11%, respectively), see an elimination of 4–7% of the arrayload bytecodes. Few arrayload bytecodes are eliminated in any of the other benchmarks, primarily due to the restrictions on movement imposed by Java's precise exception model. Further improvement would accrue if array subscripts could be disambiguated via range analysis on the subscript expressions for use during array alias analysis.

For most benchmarks the increase in the number of bytecodes executed did not significantly impact execution time. The speedups we do see can be attributed mostly to our optimizations, with some uncontrollable effects due to perturbation in instruction cache behavior.

4.5.2 JIT

The JIT environment is not influenced by conversion of long bytecode forms to their shorter variants, since JIT eliminates the bytecode dispatch overhead that we were able to reduce significantly for JDK. Rather, the biggest impact on performance comes from changes in the number of expensive instructions such as loads and stores.



(a) Time (cycles) (b) Data reads

Figure 4.4: JIT metrics

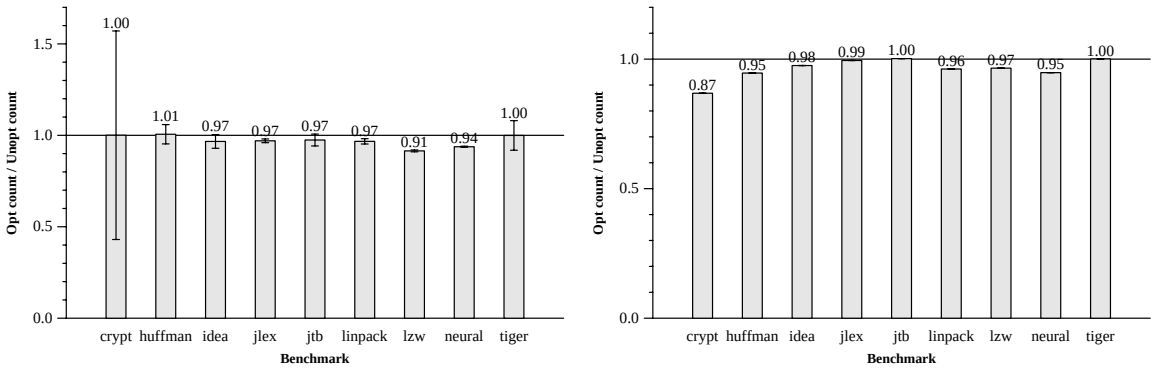
Crypt exhibits the most significant improvement with optimization for JIT execution. Total time (cycles), shown in Figure 4.4a, is reduced by 16%, with much of this coming from the 24% reduction in data reads and the 96% decrease in data read misses. The elimination of 29% of all stw instructions also plays a significant part.

Of the remaining benchmarks, all but linpack and neural reveal improvements of up to 5% in data reads, as shown in Figure 4.4b. Despite this improvement, most benchmarks show no execution time speedup as a result of this reduction, mostly because they suffer from a significant increase in instruction fetch stalls. This is likely to be an artifact of the particular instruction cache configuration for this machine, and we would expect to see execution time improvement for idea under a more favorable instruction cache regime. Linpack suffers from increases in both load and store instructions, despite good instruction cache locality.

4.5.3 Toba

There is little correlation between reduction due to optimization in static class file size and reduction in Toba native executable size. Nevertheless, optimization does result in reduced code size of up to 3% for all benchmarks except crypt, which exhibits a slight increase.

Figure 4.5b shows that all benchmarks show reductions in data reads, with crypt a clear winner at 13% fewer reads. Combined with much improved data cache miss rates, huffman,



(a) Time (cycles) (b) Data reads

Figure 4.5: Toba metrics

idea, linpack, lzw, and neural see significant speedups, shown in Figure 4.5a. Thus, our optimizations expose opportunities that the C compiler cannot exploit on its own.

Crypt is unable to exploit a reduction in data reads in the face of an uncooperative instruction cache, which stalls fetches from the instruction buffer almost 3 times more frequently as unoptimized code, and a doubling in the number of data read misses. Huffman suffers from the same problems as crypt, but to a lesser degree.

Table 4.2: Results for crypt

Metric	JDK		JIT		Toba	
	Unopt	Opt	Unopt	Opt	Unopt	Opt
Static code size	11864				43100	
%	100.00	100.00			100.00	100.15
Java bytecode	260627636					
% TOTAL	100.00	101.83				
arrayload	12.01	12.01				
getfield	0.45	0.44				
getstatic	5.36	0.63				
dup	0.02	4.73				
goto	0.08	0.49				
if	0.49	0.49				
ifcmp	0.74	0.68				
invoke	0.87	0.87				
load	15.26	5.31				
load*n*	10.33	21.05				
store	4.68	2.09				
store*n*	1.21	4.62				
Cycles	3567613744		585236746		440842396	
%	100.00	97.42	100.00	84.23	100.00	100.05
±%	0.48	0.23	0.10	0.52	3.45	56.99
Instruction fetch stalls	5984573		2097866		8951370	
%	100.00	87.60	100.00	128.76	100.00	276.62
±%	146.95	6.11	5.42	44.20	88.28	322.08
Data reads	881810632		143689032		93774629	
%	100.00	93.34	100.00	76.05	100.00	86.84
±%	0.00	0.00	0.00	0.00	0.00	0.00
Data read misses	8792878		21214232		1384580	
%	100.00	48.25	100.00	5.95	100.00	211.60
±%	0.21	0.39	0.02	0.35	17.05	161.15
SPARC instructions	2825007945		488834596		448208169	
% TOTAL	100.00	96.40	100.00	89.18	100.00	95.00
add	15.31	15.14	2.25	2.25	9.45	7.25
jmpl	9.35	9.52	1.63	1.63	1.99	1.99
or	1.45	1.45	9.74	11.63	7.27	7.32
orcc	1.80	1.83	0.41	0.41	3.20	0.46
sethi	0.48	0.48	4.53	2.01	4.22	4.15
sll	14.01	12.18	5.01	5.01	5.39	5.39
srl	1.71	1.71	9.55	7.44	3.39	3.39
subcc	1.86	1.86	9.01	8.98	13.52	16.24
bgu	0.00	0.00	0.01	0.01	2.70	0.52
ldub	13.94	12.13	0.00	0.00	0.07	0.07
lduw	16.76	16.45	28.98	21.94	20.65	17.90
stw	10.14	10.27	4.74	3.37	1.86	1.85

(Interval confidence is 90%)

Table 4.3: Results for huffman

Metric	JDK		JIT		Toba	
	Unopt	Opt	Unopt	Opt	Unopt	Opt
Static code size	7672				66364	
%	100.00	100.00			100.00	99.81
Java bytecode	27609468					
% TOTAL	100.00	108.33				
arrayload	3.80	3.63				
getfield	11.46	10.98				
getstatic	5.76	0.78				
dup	1.53	6.90				
goto	0.47	2.41				
if	3.35	3.37				
ifcmp	6.10	6.09				
invoke	6.75	6.75				
load	5.02	3.11				
load*n*	30.72	36.23				
store	0.52	2.59				
store*n*	1.87	4.03				
Cycles	672765181		301698428		357131368	
%	100.00	106.28	100.00	101.48	100.00	100.59
±%	2.28	0.28	0.17	0.14	0.07	5.25
Instruction fetch stalls	18092553		8733698		16284714	
%	100.00	158.05	100.00	138.58	100.00	136.08
±%	43.26	1.58	3.68	3.76	2.87	54.34
Data reads	161575042		44151040		36558181	
%	100.00	99.98	100.00	94.65	100.00	94.63
±%	0.00	0.00	0.00	0.00	0.00	0.01
Data read misses	8153108		2065790		2535091	
%	100.00	115.48	100.00	138.76	100.00	80.75
±%	0.25	0.25	0.61	0.85	17.85	15.15
SPARC instructions	571054371		246599706		224955140	
% TOTAL	100.00	101.16	100.00	99.14	100.00	98.22
add	12.03	12.42	6.32	6.31	10.01	9.99
jmpl	5.86	6.26	4.36	4.36	6.62	6.62
or	3.15	3.15	7.36	7.90	6.87	6.80
sethi	2.59	2.59	6.45	5.90	12.82	12.56
srl	0.92	0.86	2.11	1.85	0.32	0.32
sub	2.14	2.33	1.82	1.82	0.55	0.55
subcc	5.44	5.44	10.35	10.32	9.42	9.70
be	2.95	2.92	3.09	3.24	1.92	1.91
bne	2.80	2.74	3.33	3.19	3.72	3.72
lduw	17.08	17.01	16.41	15.45	15.29	14.42
stw	9.08	9.39	7.29	7.29	7.85	7.76

(Interval confidence is 90%)

Table 4.4: Results for idea

Metric	JDK		JIT		Toba	
	Unopt	Opt	Unopt	Opt	Unopt	Opt
Static code size	22107				148096	
%	100.00	100.00			100.00	97.64
Java bytecode	16348617					
% TOTAL	100.00	100.43				
arrayload	2.99	2.99				
getfield	2.40	2.05				
getstatic	0.00	0.00				
dup	0.31	2.84				
goto	0.48	1.30				
if	2.83	2.87				
ifcmp	3.15	3.07				
invoke	2.36	2.36				
load	15.67	9.45				
load*n*	23.12	26.78				
store	5.12	3.50				
store*n*	7.05	8.70				
Cycles	261303208		61395862		40287627	
%	100.00	98.04	100.00	102.63	100.00	96.66
±%	1.49	2.25	4.90	6.42	1.05	3.87
Instruction fetch stalls	11281294		7127667		6409337	
%	100.00	102.87	100.00	124.36	100.00	107.87
±%	31.06	55.19	41.60	46.10	6.86	19.49
Data reads	59121998		7848375		2358299	
%	100.00	98.16	100.00	96.22	100.00	97.53
±%	0.00	0.00	0.41	0.11	0.00	0.00
Data read misses	1909038		374480		116420	
%	100.00	61.36	100.00	138.50	100.00	91.68
±%	5.59	3.95	32.17	3.51	9.85	40.77
SPARC instructions	195788934		35154852		28379966	
% TOTAL	100.00	99.16	100.00	100.51	100.00	100.06
or	1.74	1.74	9.05	10.17	8.19	8.35
sethi	1.42	1.42	4.49	4.49	6.58	6.76
sll	12.62	12.03	1.80	1.80	2.77	2.77
be	1.42	1.39	2.99	2.99	2.59	2.77
bne	1.27	1.27	2.25	2.25	3.34	2.95
ldub	12.65	12.13	0.15	0.15	0.00	0.00
lduw	16.03	15.93	20.17	19.32	6.97	6.76
stw	9.97	9.95	7.07	6.61	0.46	0.46

(Interval confidence is 90%)

Table 4.5: Results for jlex

Metric	JDK		JIT		Toba	
	Unopt	Opt	Unopt	Opt	Unopt	Opt
Static code size	71961				473596	
%	100.00	100.00			100.00	94.71
Java bytecode	62750646					
% TOTAL	100.00	107.31				
arrayload	3.35	3.33				
getfield	13.68	12.42				
getstatic	0.00	0.00				
dup	0.90	7.11				
goto	0.36	0.72				
if	2.49	3.18				
ifcmp	9.00	8.31				
invoke	5.15	5.15				
load	9.20	5.87				
load*n*	30.24	31.83				
store	2.62	2.27				
store*n*	2.69	7.58				
Cycles	1664584041		781102454		934714659	
%	100.00	87.48	100.00	102.76	100.00	97.00
±%	0.08	4.05	0.21	0.15	2.45	1.05
Instruction fetch stalls	5894510		7800247		35382354	
%	100.00	217.26	100.00	196.58	100.00	72.83
±%	7.76	289.03	1.55	0.49	28.56	28.78
Data reads	331146695		118270464		128538823	
%	100.00	101.40	100.00	97.72	100.00	99.48
±%	0.00	0.00	0.00	0.00	0.00	0.00
Data read misses	21657098		1954337		2419713	
%	100.00	46.19	100.00	156.95	100.00	106.04
±%	4.15	0.24	0.09	0.13	1.86	3.52
SPARC instructions	1183017726		599372744		587556599	
% TOTAL	100.00	102.07	100.00	100.06	100.00	99.64
add	11.91	12.51	5.12	5.10	8.52	8.52
jmpl	6.87	7.26	6.47	6.47	6.52	6.52
or	2.16	2.16	7.30	7.82	7.12	7.19
sll	8.95	9.08	0.69	0.69	0.79	0.79
sub	2.33	2.57	1.73	1.74	0.96	0.96
subcc	5.47	5.43	7.84	7.83	8.64	8.51
ldub	9.00	9.15	0.00	0.00	0.00	0.00
lduw	17.10	17.33	18.03	17.58	21.84	21.73
stw	9.62	9.99	8.13	8.14	11.81	11.81

(Interval confidence is 90%)

Table 4.6: Results for jtb

Metric	JDK		JIT		Toba	
	Unopt	Opt	Unopt	Opt	Unopt	Opt
Static code size	368287				2755812	
%	100.00	100.00			100.00	100.07
Java bytecode	49058483					
% TOTAL	100.00	107.37				
arrayload	2.21	2.21				
getfield	11.59	9.86				
getstatic	0.08	0.07				
dup	3.38	8.74				
goto	1.95	2.20				
if	2.70	3.38				
ifcmp	6.67	5.99				
invoke	3.00	3.00				
load	14.95	11.90				
load*n*	22.23	24.05				
store	4.47	5.16				
store*n*	0.94	5.10				
Cycles	983063809		421146724		472530144	
%	100.00	104.58	100.00	98.69	100.00	97.44
±%	0.34	0.46	0.88	0.84	3.65	3.34
Instruction fetch stalls	24886398		34482858		66802677	
%	100.00	132.18	100.00	104.54	100.00	77.61
±%	10.22	9.40	10.16	9.55	27.83	21.35
Data reads	213196107		52127654		47121000	
%	100.00	101.48	100.00	96.84	100.00	100.20
±%	0.00	0.00	0.00	0.00	0.02	0.01
Data read misses	11518789		3909725		3185467	
%	100.00	109.97	100.00	99.75	100.00	112.86
±%	0.65	0.61	0.74	1.17	5.99	4.21
SPARC instructions	787770334		317604663		236446136	
% TOTAL	100.00	102.44	100.00	100.25	100.00	100.14
add	14.59	15.32	9.98	9.97	8.88	8.87
jmpl	6.89	7.35	3.31	3.31	6.65	6.65
or	2.08	2.09	5.76	6.44	7.01	7.34
sethi	1.62	1.63	4.28	4.29	11.63	11.36
sll	10.58	10.79	1.31	1.31	0.69	0.74
sub	2.38	2.68	1.29	1.29	0.69	0.69
subcc	5.71	5.67	13.44	13.44	9.11	8.81
be	2.38	2.27	2.45	2.44	1.94	1.93
bl	0.55	0.51	1.40	1.29	0.52	0.52
bne	3.68	3.68	7.17	7.18	4.01	3.90
ldub	10.74	10.96	0.09	0.09	0.29	0.29
lduw	14.60	14.79	15.00	14.47	18.71	18.83
stw	10.39	10.84	10.64	10.68	11.21	11.58

(Interval confidence is 90%)

Table 4.7: Results for linpack

Metric	JDK		JIT		Toba	
	Unopt	Opt	Unopt	Opt	Unopt	Opt
Static code size	3834				37516	
%	100.00	100.00			100.00	95.36
Java bytecode	9733541					
% TOTAL	100.00	95.51				
arrayload	9.13	8.51				
getfield	0.02	0.01				
getstatic	0.00	0.00				
dup	0.00	6.45				
goto	0.12	1.48				
if	0.43	0.49				
ifcmp	1.90	1.84				
invoke	0.12	0.12				
load	39.10	27.25				
load*n*	6.07	13.72				
store	0.72	6.77				
store*n*	0.00	0.40				
Cycles	138498459		15152824		14956369	
%	100.00	100.29	100.00	108.08	100.00	96.71
±%	1.04	1.20	0.10	0.45	1.47	1.50
Instruction fetch stalls	30907		58417		148240	
%	100.00	187.16	100.00	84.56	100.00	93.26
±%	15.12	13.52	8.93	15.83	68.75	77.78
Data reads	35186863		3298217		3688269	
%	100.00	95.09	100.00	107.49	100.00	96.19
±%	0.00	0.00	0.00	0.26	0.01	0.00
Data read misses	398002		223725		388542	
%	100.00	654.43	100.00	92.12	100.00	99.17
±%	0.07	0.02	0.05	0.08	0.38	3.92
SPARC instructions	107641162		12383949		15758277	
% TOTAL	100.00	94.81	100.00	107.56	100.00	97.31
add	17.45	16.13	16.51	9.86	14.46	14.14
jmpl	9.09	8.68	0.14	0.30	0.22	0.22
or	1.08	0.61	0.54	4.63	3.48	3.36
sll	14.49	13.63	10.50	10.28	5.73	5.73
srl	1.47	1.42	3.17	3.86	0.15	0.15
sub	3.25	3.01	0.32	0.39	0.33	0.33
subcc	1.83	1.77	12.27	12.41	18.78	17.89
tcc			9.91	9.79		
bcs	0.43	0.43	0.01	0.01	7.54	7.75
lddf	0.00	0.00	6.40	6.24	3.26	3.26
ldf	2.99	2.99	3.65	3.98	10.52	10.26
ldub	13.37	12.68	0.00	0.00	0.00	0.00
lduw	15.96	14.92	16.56	18.60	9.53	8.89
stf	1.46	1.46	0.83	1.16	5.37	5.37
stw	9.85	9.36	1.62	3.68	1.21	1.21

(Interval confidence is 90%)

Table 4.8: Results for lzw

Metric	JDK		JIT		Toba	
	Unopt	Opt	Unopt	Opt	Unopt	Opt
Static code size	4047				41452	
%	100.00	100.00			100.00	101.05
Java bytecode	28009228					
% TOTAL	100.00	100.96				
arrayload	5.98	5.98				
getfield	14.00	13.36				
getstatic	0.00	0.00				
dup	0.19	2.31				
goto	0.57	1.07				
if	5.18	6.17				
ifcmp	2.36	1.37				
invoke	7.01	7.01				
load	11.02	1.27				
load*n*	19.55	27.91				
store	5.94	1.32				
store*n*	0.81	6.47				
Cycles	544150042		193650000		151030549	
%	100.00	100.92	100.00	93.40	100.00	91.49
±%	1.66	1.27	0.97	3.98	2.40	0.56
Instruction fetch stalls	16697542		12299277		22230129	
%	100.00	156.95	100.00	112.29	100.00	48.69
±%	43.51	23.58	10.60	36.94	17.42	2.00
Data reads	144244856		26335439		15462242	
%	100.00	97.39	100.00	98.64	100.00	96.52
±%	0.00	0.00	0.04	0.12	0.00	0.00
Data read misses	6052882		5647087		2472050	
%	100.00	87.92	100.00	79.16	100.00	103.56
±%	3.68	4.14	0.95	2.20	3.65	2.49
SPARC instructions	440093811		84259742		68856767	
% TOTAL	100.00	98.61	100.00	100.19	100.00	96.78
or	2.58	2.60	9.63	9.80	7.43	5.70
sll	11.69	10.84	2.38	2.38	2.45	2.46
subcc	2.71	2.65	7.16	7.16	16.09	15.79
be	2.46	2.42	1.31	1.33	2.15	0.20
bne	2.54	2.48	3.84	3.82	4.58	6.52
ldub	11.70	10.89	0.06	0.06	1.96	1.96
lduw	18.87	18.80	30.19	29.76	20.07	19.29
stw	10.83	10.86	6.95	6.94	3.90	3.61

(Interval confidence is 90%)

Table 4.9: Results for neural

Metric	JDK		JIT		Toba	
	Unopt	Opt	Unopt	Opt	Unopt	Opt
Static code size	15522				143284	
%	100.00	100.00			100.00	97.08
Java bytecode	17898283					
% TOTAL	100.00	96.37				
arrayload	10.96	10.49				
getfield	8.73	4.68				
getstatic	0.46	0.23				
dup	1.00	2.04				
goto	0.47	3.02				
if	0.23	0.64				
ifcmp	3.94	3.53				
invoke	3.26	3.26				
load	8.87	4.87				
load*n*	26.14	29.20				
store	0.50	0.11				
store*n*	1.35	2.53				
Cycles	398062740		239905427		184637033	
%	100.00	98.58	100.00	99.57	100.00	93.76
±%	0.42	1.49	0.41	0.38	3.16	0.32
Instruction fetch stalls	7173133		3681935		5666500	
%	100.00	134.95	100.00	106.79	100.00	87.90
±%	10.77	39.44	14.21	8.50	65.27	7.18
Data reads	89151848		36563097		19406875	
%	100.00	95.90	100.00	96.37	100.00	94.79
±%	0.00	0.00	0.01	0.00	0.01	0.01
Data read misses	2154458		3537800		1745887	
%	100.00	105.76	100.00	109.72	100.00	80.82
±%	0.30	0.31	0.27	0.24	3.97	4.91
SPARC instructions	373903754		231568134		148125611	
% TOTAL	100.00	97.67	100.00	99.50	100.00	97.63
jmpl	6.00	5.83	3.31	3.31	4.95	4.95
or	3.98	3.93	6.61	6.66	7.89	7.32
sll	8.50	8.03	2.65	2.56	1.87	1.87
subcc	5.60	5.53	10.03	9.92	11.41	10.82
bcs	0.28	0.28	0.25	0.25	1.55	1.44
be	2.56	2.31	2.98	2.98	1.36	1.31
ldub	7.21	6.89	1.15	1.15	0.29	0.29
lduw	14.80	14.02	13.04	12.44	10.72	10.04
stw	9.01	8.64	4.78	4.77	5.20	5.37

(Interval confidence is 90%)

Table 4.10: Results for tiger

Metric	JDK		JIT		Toba	
	Unopt	Opt	Unopt	Opt	Unopt	Opt
Static code size	203584				1492852	
%	100.00	100.00			100.00	99.76
Java bytecode	47945748					
% TOTAL	100.00	112.08				
arrayload	1.31	1.31				
getfield	13.10	11.42				
getstatic	0.07	0.06				
dup	2.28	9.34				
goto	1.76	2.10				
if	4.34	4.78				
ifcmp	4.90	4.46				
invoke	4.44	4.44				
load	11.49	9.02				
load*n*	27.83	29.77				
store	3.96	5.43				
store*n*	1.28	6.91				
Cycles	937644876		434716771		236941544	
%	100.00	108.12	100.00	101.11	100.00	99.94
±%	1.18	0.07	0.14	0.14	11.36	8.10
Instruction fetch stalls	15767832		13236768		16400467	
%	100.00	160.43	100.00	166.02	100.00	122.41
±%	41.33	1.37	2.21	2.46	83.34	58.54
Data reads	221033626		62746891		21215154	
%	100.00	103.81	100.00	97.01	100.00	100.08
±%	0.00	0.00	0.00	0.01	0.00	0.00
Data read misses	9887470		3092349		1247393	
%	100.00	114.57	100.00	93.95	100.00	92.27
±%	0.21	0.11	0.08	0.26	1.08	20.29
SPARC instructions	780587970		332842011		137192464	
% TOTAL	100.00	104.50	100.00	100.47	100.00	99.99
add	12.85	13.99	5.48	5.46	9.50	9.48
jmpl	7.01	7.76	5.03	5.02	7.34	7.34
or	2.66	2.66	8.04	9.01	8.29	8.50
sethi	2.28	2.27	6.10	6.10	12.91	12.73
sll	10.45	11.03	1.38	1.38	1.11	1.11
sub	2.43	2.85	1.82	1.82	0.49	0.49
subcc	4.24	4.21	9.32	9.31	9.76	9.48
be	2.78	2.67	3.18	3.16	2.08	2.07
ldub	10.47	11.07	0.08	0.08	0.16	0.16
lduw	16.08	16.55	17.49	16.91	14.96	14.97
stw	9.38	10.10	7.13	7.15	8.48	8.65

(Interval confidence is 90%)

5 RELATED WORK

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 cloning and inlining approaches most prominently demonstrated in Self [Chambers and Ungar 1989; Chambers et al. 1989; Chambers and Ungar 1990; 1991; Chambers 1992], these optimizations yield factors of two to five in performance improvement. Such results are consistent with that earlier work on optimizations for Self.

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 interprocedural 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 interprocedural analysis are broadly consistent with ours.

6 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 substantial improvements based on PRE over access path expressions, with sometimes dramatic reductions in memory accesses. 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.

We also plan to explore the impact of Java's precise exception model and the associated constraints on code motion. Relaxing the constraints may provide more opportunities for optimization. If that is so, then a strong argument can be made that the precise exception model unnecessarily restricts optimizers from obtaining useful performance improvements for Java.

The implementation of further analyses and optimizations to BLOAT is under way and we expect to make the tool publicly available as soon as it becomes stable. 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 should 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].

BIBLIOGRAPHY

BIBLIOGRAPHY

ACKERMANN, W. 1928. Zum Hilbertschen Aufbau der reellen Zahlen. *Math. Ann. 99*, 118–133.

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

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

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

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. Submitted for publication.

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*, 146–160.

CHAMBERS, C. AND UNGAR, D. 1990. Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs. In Proceedings of the ACM Conference on Programming Language Design and Implementation (White Plains, New York, June). *ACM SIGPLAN Notices 25*, 6 (June), 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.

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.

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. In Proceedings of the ACM Conference on Programming Language Design and Implementation (La Jolla, California, June). *ACM SIGPLAN Notices 30,* 6 (June), 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* (May). 128–137.

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.

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.

ENBODY, R. 1998. *Perfmon User's Guide*. Michigan State University. http://web.cps.msu.edu/∼enbody/perfmon/index.html.

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.

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

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

HENNESSY, J. 1981. Program optimization and exception handling. In *Conference Record of the ACM Symposium on Principles of Programming Languages* (Williamsburg, Virginia, Jan.). 200–206.

JAVASOFT. 1997. *Java Core Reflection API and Specification*.

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.

LENGAUER, T. AND TARJAN, R. E. 1979. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst. 1,* 1 (July), 121–141.

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

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

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

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

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.

PURDOM, P. W. J. AND MOORE, E. F. 1972. Algorithm 430: Immediate predominators in a directed graph. *Communications of the ACM 15,* 8 (Aug.), 777–778.

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

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

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

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

TOBA. 1998. Toba: A Java-to-C translator. http://www.cs.arizona.edu/sumatra/toba.

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

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