Partial redundancy elimination for access path expressions

SP&E

Antony L. Hosking[1,*,†], Nathaniel Nystrom[1], David Whitlock[1],
Quintin Cutts[2] and Amer Diwan[3]

[1]*Department of Computer Sciences, Purdue University, West Lafayette, IN 47907-1398, U.S.A.*
[2]*Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, U.K.*
[3]*Department of Computer Science, University of Colorado, Boulder, CO 80309-0430, U.S.A.*

SUMMARY

Pointer traversals pose significant overhead to the execution of object-oriented programs, since every access to an object's state requires a pointer dereference. Eliminating redundant pointer traversals reduces both instructions executed as well as redundant memory accesses to relieve pressure on the memory subsystem. We describe an approach to elimination of redundant access expressions that combines partial redundancy elimination (PRE) with type-based alias analysis (TBAA). To explore the potential of this approach we have implemented an optimization framework for Java class files incorporating TBAA-based PRE over pointer access expressions. The framework is implemented as a class-file-to-class-file transformer; optimized classes can then be run in any standard Java execution environment. Our experiments demonstrate improvements in the execution of optimized code for several Java benchmarks running in diverse execution environments: the standard interpreted JDK virtual machine, a virtual machine using 'just-in-time' compilation, and native binaries compiled off-line ('way-ahead-of-time'). Overall, however, our experience is of mixed success with the optimizations, mainly because of the isolation between our optimizer and the underlying execution environments which prevents more effective cooperation between them. We isolate the impact of access path PRE using TBAA, and demonstrate that Java's requirement of precise exceptions can noticeably impact code-motion optimizations like PRE. Copyright © 2001 John Wiley & Sons, Ltd.

KEY WORDS: object-oriented languages; object representation; code generation; language preprocessors; incremental compilers; run-time environments; virtual machines; optimization; cache memories; performance evaluation

INTRODUCTION

Pointer traversals pose significant overhead to the execution of object-oriented programs, since every access to an object's state requires a pointer dereference. Objects can refer to other objects, forming

graph structures, and they can be modified, with such modifications visible in future accesses. Just as common subexpressions often appear in numerical code, common access expressions are likewise often encountered in object-oriented code. Where two expressions redundantly compute the same value it is desirable to avoid the redundant computation by caching the result of the first occurrence in a temporary variable, and reusing this result from the temporary in place of the later occurrence of the expression. Eliminating redundant computations in this way certainly eliminates redundant CPU overhead. Perhaps just as important for modern machine architectures, eliminating redundant access expressions also has the effect of eliminating redundant memory references, which might otherwise incur large performance penalties in the memory subsystem.

In this paper we evaluate an approach to eliminating common access expressions by combining partial redundancy elimination (PRE) [1] with type-based alias analysis (TBAA) [2]. To explore its potential we have built an optimization framework for Java class files incorporating TBAA-based PRE for access expressions, and measured its impact on the performance of several benchmark programs. An interesting aspect of the optimization framework is that it operates entirely as a bytecode-to-bytecode translator, sourcing and targeting Java class files. Our experiments compare the execution of optimized and unoptimized classes for several SPARC-based execution environments: a Java virtual machine that interprets bytecodes, a Java virtual machine that performs 'just-in-time' (JIT) compilation of bytecodes to native code, and native binaries compiled off-line ('way-ahead-of-time') to C and thence to optimized native code using the Solaris C compiler.

We have measured the dynamic execution impact of bytecode-level TBAA-based PRE for a set of Java benchmark applications. Our measurements include per-bytecode execution counts, elapsed time, data cache reads, data cache misses, and instruction cache stalls. The benchmarks exhibit significant improvement in bytecode counts cache behavior due to optimization for all three execution environments. The impact on actual elapsed time is less significant; in some cases our 'optimizations' actually result in slower elapsed times. We hypothesize that this experience is due to in the following.

PARTIAL REDUNDANCY ELIMINATION FOR ACCESS EXPRESSIONS

Our analysis and optimization framework revolves around PRE over object access expressions. We adopt standard terminology and notation used in the specification of the Java programming language [3] to frame the analysis and optimization problem, as follows.

Terminology and notation

The following definitions paraphrase the Java specification [3]. 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, cast, type comparison (`instanceof`), and equality. There may be many references to the same object. Objects have mutable state, stored in the variable fields of class instances or the variable elements of arrays. Two variables may refer to the same object: the state of the object can be modified through the reference stored in one variable and then the altered state observed through the other. *Access expressions* refer to the variables that comprise an object's state. A *field access expression* refers to a field of some class instance, while

Table I. Access expressions.

Notation	Name	Variable accessed
p.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

an *array access expression* refers to a component of an array. Table I summarizes the two kinds of access expressions in Java. We adopt the term *access path* [2,4] to mean a non-empty sequence of accesses. For example, the Java expression $a.b[i].c$ is an access path. Without loss of generality, our notation assumes that distinct fields within an object have different names (i.e., fields that override inherited fields of the same name from superclasses are trivially renamed).

A variable is a storage location and has an associated type, sometimes called its *compile-time* type. Given an access path p, then the compile-time type of p, written $Type(p)$, is simply the compile-time type of the variable it accesses. A variable always contains a value that is *assignment compatible* with its type. A value of compile-time class type S is assignment compatible with class type T if S and T are the same class or S is a subclass of T. A similar rule holds for array variables: a value of compile-time array type $S[\,]$ is assignment compatible with array type $T[\,]$ if type S is assignable to type T. Interface types also yield rules on assignability: an interface type S is assignable to an interface type T only if T is the same interface as S or a super-interface 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[‡]. We write *Subtypes*(T) to denote all subtypes of type T, including T. Thus, an access path p can legally access variables of type *Subtypes*($Type(p)$).

Partial redundancy elimination

Our approach to optimization of access expressions is based on application of *partial redundancy elimination* (PRE) [1]. To our knowledge this is the first time PRE has been applied to access paths. PRE is a powerful global optimization technique that subsumes the more standard common subexpression elimination (CSE). PRE eliminates computations that are only partially redundant; that is, redundant only on some, but not all, paths to some later re-computation. By inserting evaluations on those paths where the computation does not occur, the later re-evaluation can be eliminated and replaced with a use of the pre-computed value. This is illustrated in Figure 1. In Figure 1(a), both a and b are available along each path to the merge point, where expression $a + b$ is evaluated. However, this evaluation is partially redundant since $a + b$ is available on one path to the merge but not both.

[‡]The term 'subtype' is not used at all in the official Java language specification [3], presumably to avoid confusing the type hierarchy induced by the subtype relation with class and interface hierarchies.
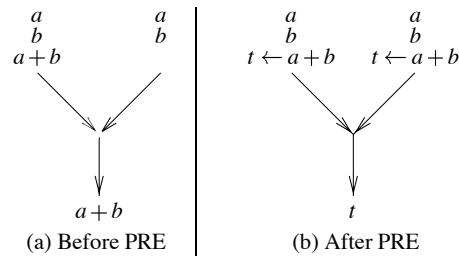
Figure 1. PRE for arithmetic expressions.

By hoisting the second evaluation of $a + b$ into the path where it was not originally available, as in Figure 1(b), $a + b$ need only be evaluated once along any path through the program, rather than twice as before.

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

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

Traversing the access path requires successively loading the pointer at each storage location along the path and traversing it to the next location in the sequence. Before applying PRE to access path expressions, one must first disambiguate memory references sufficiently to be able safely to assume that no memory location along the access path can be aliased (and so modified) by some lexically distinct access path in the program. Consider the example in Figure 2. The expression $a.b[i].c$ will be redundant at some subsequent reevaluation so long as no store occurs to any one of a, $a.b$, i, $a.b[i]$ or $a.b[i].c$ on the code path between the first evaluation of the expression and the second. In other words, if there are explicit stores to a or i (local variables cannot be aliased in Java) or potential aliases to any one of $a.b$, $a.b[i]$ or $a.b[i].c$ through which those locations *may* be modified, between the first and second evaluation of the expression, then that second evaluation cannot be treated as redundant.

Type-based alias analysis

Alias analysis refines the set of possible variables to which an access path may refer. Two distinct access paths are said to be possible *aliases* if they may refer to the same variable. Without alias analysis the optimizer must conservatively assume that all access paths are possible aliases of each other. In general, alias analysis in the presence of references is slow and requires the code for the entire program to work. *Type-based alias analysis* (TBAA) [2] 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

$$
\begin{array}{c}
\begin{array}{cc}
\begin{array}{c} a \\ i \\ a.b[i].c \end{array} & \begin{array}{c} a \\ i \end{array}
\end{array} \\
a.b[i].c \\
\text{(a) Before PRE}
\end{array}
\qquad
\begin{array}{c}
\begin{array}{cc}
\begin{array}{c} a \\ i \\ t \leftarrow a.b[i].c \end{array} & \begin{array}{c} a \\ i \\ t \leftarrow a.b[i].c \end{array}
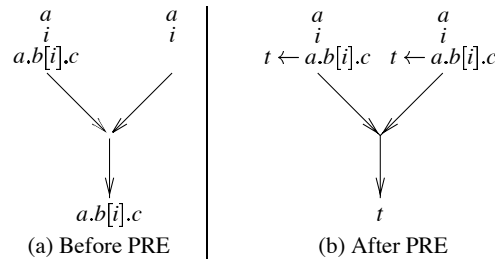\end{array} \\
t \\
\text{(b) After PRE}
\end{array}
$$

Figure 2. PRE for access expressions.

program be available. Rather, TBAA uses the type system to disambiguate memory references by refining the *type* of variables to which an access path may refer, since only type-compatible access paths can alias the same variable in a type-safe language such as Java. The compile-time type of an access path provides a simple way to do this: two access paths p and q may be aliases only if the relation *TypeDecl*(p, q) holds, as defined by

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

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

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

Analysing incomplete programs

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

Table II. *FieldTypeDecl*($\mathcal{AP}_1$, $\mathcal{AP}_2$).

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

Diwan *et al.* [2] further refine TBAA for *closed world* situations: those in which all the code that might execute in an application is available for analysis. The refinement enumerates all the assignments in a program to determine more accurately the types of variables to which a given access path may refer. An access path of type T may yield a reference to an object of a given subtype S only if there exist assignments of references of type S to variables of type T. Unlike *TypeDecl*, which always merges the compile-time type of an access path with all of its subtypes, Diwan's closed world refinement merges a type T with a subtype S only if there is at least one assignment of a reference of type S to a variable of type T somewhere in the code.

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

Java constraints on optimization

Java's thread and exception models impose several constraints on optimization. First, exceptions in Java are *precise*: when an exception is thrown all effects of statements prior to the throw-point must appear to have taken place, while the effects of statements after the throw-point must not. This imposes a significant constraint on code-motion optimizations such as PRE, since code that may throw exceptions (e.g. access expressions) may not be moved past other code with side effects (e.g. stores or exception throwing) [5]§. In regions of the code where program analysis can show that exceptions will not occur code motion is unconstrained. For example, the first access to an object via a given reference ensures that subsequent accesses via that reference cannot throw a null pointer exception.

---

§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 p. 205 of the Java language specification [3]).

Second, the thread model prevents movement of access expressions across (possible) synchronization points. Explicit synchronization points occur at **monitorenter/monitorexit** bytecodes. Also, without inter-procedural control-flow analysis every method invocation represents a possible synchronization point, since the callee, or a method invoked inside the callee, may be synchronized. Thus, calls and synchronization points are places at which PRE must assume all non-local variables may be modified, either inside the call or through the actions of other threads. Common access expressions cannot be considered redundant across these synchronization points. Naturally, one must also respect the `volatile` declaration modifier, which forces synchronization of the variable's state across threads on every access.

Finally, the Java memory model has recently come under heightened scrutiny since it appears to preclude many standard optimizations, including certain cases of PRE over access expressions. The model, as it stands, enforces a strict memory coherence semantics for all accesses to the same location, including accesses via potential aliases. Pugh's analysis [6] observes that reordering of potentially aliased load instructions is illegal under the current model, and implies that the model is inherently broken and needs careful revision. For loads of the same location via known aliases, PRE-style optimizations as described here certainly *are* legal. Our position is that code that depends on the current memory model for correctness is inherently buggy, and must be fixed to use explicit synchronization to effect correct behavior. Our optimization tools provide a switch that respects the current Java memory model when processing code that depends on it.

IMPLEMENTATION

The Java virtual machine (VM) specification [7] is intended as the interface between Java compilers and Java execution environments. Its standard class file format and instruction set permit multiple compilers to inter-operate with multiple VM implementations, enabling the cross-platform delivery of applications that is Java's hallmark. Conforming class files generated by *any* compiler will run in *any* Java VM implementation, no matter if that implementation interprets bytecodes, performs dynamic JIT translation to native code, or pre-compiles Java class files to native object files.

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

Our Java class file optimization tool is called BLOAT (Bytecode-Level Optimization and Analysis Tool). The analysis and optimization framework implemented in BLOAT is based on several recent developments in the field. Notably, we use control flow graphs and static single assignment (SSA) form as the basic intermediate representation [8,9,10]. On this foundation we have built several standard

optimizations such as dead-code elimination and copy/constant propagation, and SSA-based value numbering [11], as well as *type-based alias analysis* [2] and the SSA-based algorithm for PRE of Chow *et al*. [12].

SSA form

SSA form provides a concise representation of the use-definition relationships among the program variables. Efficient global optimizations can be constructed based on this form, including dead store elimination [8], constant propagation [13], value numbering [11,14–17], induction variable analysis [18] and global code motion [19]. Optimization algorithms based on SSA all exploit its sparse representation for improved speed and simpler coding of combined local and global optimizations.

SSA-based PRE

Prior to the work of Chow *et al*. [12], 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*. [12] removed this impediment with an approach (SSAPRE) that retains the SSA representation throughout PRE. The specific details of their algorithm are not relevant here, save to say that the algorithmic complexity is respectable: for a program of size n, SSAPRE's total time is $O(n(E + V))$, where E and V are the number of edges and nodes in the control flow graph, respectively.

Analysis

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

Two simple transformations are then applied to ease later analyses and optimizations. The first converts methods that initialize static arrays from the form emitted by the JDK javac compiler, comprising a straight-line sequence of array stores for every element of the array, into a form more amenable to later analysis, consisting of a loop that reads from a static string defined in the constant pool of the class. This transformation eliminates the unnecessarily large basic blocks emitted for static array initializers in such core classes as Character, significantly cutting the time for later analysis of these initializers. The second transformation identifies loops [20] and converts each 'while' loop into a 'repeat' loop preceded by an 'if' conditional. This provides a convenient place immediately after the 'if' to hoist loop-invariant code out of the loop body. Code that is loop-invariant apart from possibly throwing exceptions can thus be treated as invariant in the new loop body and will be eliminated by PRE.

After construction of the control flow graph both local and operand stack variables are converted to SSA form. This requires computation of the dominator tree and dominance frontier of the control flow graph [8]. We also remove *critical edges* in the graph by inserting empty basic blocks on such edges. Critical edge removal is required to provide a place to insert code during PRE and when translating back from SSA form.

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

Subroutines within a method are formed with the jsr and ret bytecodes. The jsr bytecode pushes the current program counter, a value of type returnAddress, onto the operand stack and branches to the subroutine. The ret bytecode loads a saved returnAddress from a local variable and resumes control at that code location. To permit verification of jsr subroutines the Java VM specification imposes a restriction that each jsr can have at most one corresponding ret. This allows each jsr to be tied to the ret that returns to it. Thus, for SSA construction we treat each subroutine such that, if a variable is not redefined within the subroutine, the use-definition information for the variable is propagated from each jsr site to its corresponding return site. This avoids unnecessarily merging information from multiple paths through the subroutine. Our SSA-based solution is equivalent to the 'variable splitting' approach implemented by Agesen *et al.* [21] in support of accurate garbage collection.

TBAA uses the compile-time type of every expression in the method, but local and operand stack variables in Java bytecode are not declared. Thus, after SSA construction we infer their types using an intra-procedural variation of the algorithm of Palsberg and Schwartzbach [22].

PRE operates by recognizing common subexpressions. Rather than basing equivalence of expressions purely on their lexical equivalence, we use SSA-based value numbering [11], assigning value numbers to every *first-order* expression. These are expressions for values that cannot be aliased, such as the contents of method local variables, constants, and non-access expressions over these. Using value numbering avoids the need for repetitive iteration of PRE interleaved with constant/copy propagation.

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

Optimization

After the analyses, BLOAT performs the following optimizations.

1. *Partial redundancy elimination.* BLOAT implements the SSA-based PRE algorithm of Chow *et al.* [12], extended to support TBAA-based PRE of access paths by treating alias definition points for a given access expression as redefining that expression. This forces reevaluation of the expression after the alias definition point. We also restrict PRE-induced code motion to respect the constraints on Java optimizations due to precise exceptions and threads, except that where analysis shows a given bytecode will never throw an exception we are free to move code with respect to that bytecode.
2. *Constant/copy propagation.* This is based on standard techniques for constant folding, algebraic simplification and copy propagation [9].
3. *Dead code elimination.* This is the standard SSA-based algorithm [8].

Code generation

Following the optimizations, SSA temporaries are mapped back to Java VM local variables, before generation of bytecode instructions from the (optimized) intermediate code trees. Liveness analysis and register coloring with coalescing [23,24] ensure a good allocation, packing as many SSA variables into the same physical local variable as possible. Priority is given to coalescing loop-nested local variables ahead of others. Peephole optimizations remove redundant load and store bytecodes for better utilization of the operand stack.

EXPERIMENTS

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

Platform

Our experiments were run under Solaris 2.6 on a Sun Ultra 2 Model 2200, with 256 Mb RAM, and dual 200 MHz UltraSPARC-I processors, each with 1 Mb external cache in addition to their on-chip instruction and data caches. The UltraSPARC-I data cache is a 16 kB write-through, non-allocating, direct-mapped cache with two 16-byte sub-blocks per line. It is virtually indexed and physically tagged. The 16 kB instruction cache is two-way set-associative, physically indexed and tagged, and organized into 512 32-byte lines.

Benchmarks

The benchmarks we used are summarized in Table III. They represent a spectrum from simple, single-class applications such as crypt, through to complex tools such as compilers.

Execution environments

We took measurements for three different Java execution environments: the Java Development Kit (JDK) Solaris Production Release version 1.1.8_09a running as a straightforward bytecode interpreter (JDK without JIT), the same JDK running with JIT translation to native code (JDK with JIT), and Toba version 1.1 (Toba) [26]. In each environment we ran all four optimization-level variants (base, pre, tbaa and loose) of the classes for each benchmark. Where Java source code for a benchmark was available, it was compiled using the JDK 1.1.8 javac compiler (without the -O optimization flag, since

Table III. Benchmarks.

Name	Description	Size*
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	22 317
Linpack	Standard Linpack benchmark	584
LZW	Lempel-Ziv-Welch file compression utility	314
Neural	Neural network simulation	1227
Tiger	Tiger compiler [25]	19 018

*Lines of source code (including comments).

in many cases this is known to generate erroneous code; our observations indicate that this flag has little impact anyway on the performance of our benchmarks).

JDK

The Solaris Production Release of the JDK version 1.1.8_09a Java virtual machine uses native Solaris threads and the bytecode interpreter loop is implemented in assembler. We optimized the class files of each benchmark against the JDK version 1.1.8 library classes [27] at each optimization level, to form the closure of optimized classes necessary to execute the benchmark in JDK. Similarly the unoptimized benchmark classes were run against the unoptimized library classes.

JIT

JIT refers to the Solaris Production Release of the JDK version 1.1.8_09a running with 'just-in-time' translation to native code. Running with JIT this VM translates a method's bytecodes to native SPARC instructions, along with the following optimizations:

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

Programmers are encouraged to perform the following optimizations by hand [28]:

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; and
11. pre-compute values known at compile time.

These suggestions hint at deficiencies in JIT translation 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.

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 [29]. We started with the same sets of classes as for JDK for execution in Toba. These class files were then compiled to native code using the SunPro C compiler version 5.0, with the -O compiler optimization flag. This optimization level performs basic local and global optimization, including induction variable elimination, local and global CSE, algebraic simplification, copy propagation, constant propagation, loop-invariant optimization, register allocation, basic block merging, tail recursion elimination, dead code elimination, tail call elimination and complex expression expansion. We use this level since it does not 'optimize references', nor 'trace the effect of pointer assignments', and thus will best reveal the impact of our own pointer optimizations.

Metrics

For each benchmark we took measurements for both the optimized and unoptimized classes, using dynamic metrics to expose the effects of optimization:

- bytecodes executed: dynamic per-bytecode execution frequencies, obtained via an instrumented version of the C-coded interpreter loop in the JDK source release;
- globally redundant (i.e. cross-activation) bytecode-level memory accesses: dynamic per-bytecode counts of all accesses that reload values from unmodified variables, also obtained via the instrumented JDK VM;
- counts of significant processor-level hardware events:

  — processor cycles to measure elapsed time,
  — instruction buffer stalls due to instruction cache misses,
  — data cache reads,
  — data cache read misses;

using a software library that allows user-level access to the UltraSPARC hardware execution counters[¶].

For these 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 variable instruction cache behavior from one invocation to the next, since each invocation may have quite different mappings from virtual to physical addresses that result in randomized instruction cache placement. Thus, the elapsed time and cache-related metrics were obtained for 10 separate runs and the results averaged. We ran each benchmark with sufficient heap space to eliminate the need for garbage collection, and verified that no collections occurred during benchmarking.

RESULTS

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

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

JDK

Figure 3(a) illustrates the anticipated increase in dynamic bytecode execution counts for many of the benchmarks, due to introduction of extra loads from and stores to temporaries introduced by PRE for partially-redundant expressions whose values are not used on all paths. As we shall see, execution environments that map method local variables to registers (e.g., JIT and Toba) do not suffer unduly from this overhead. For some benchmarks PRE removes sufficient bytecodes to offset the extra loads and stores.

The most dramatic effects of PRE over access paths are directly revealed in the results for the access bytecodes given in Figures 3(b)–(d). These show the total number of access bytecodes performed, broken down into globally redundant versus non-redundant accesses. The non-redundant accesses are those that must always be performed. The globally redundant accesses represent opportunity for optimization; with perfect foresight all such accesses could be removed. It is remarkable how many of these are removed by our purely intra-procedural analysis. All benchmarks see a decrease, often significant, in the frequency of getfield bytecodes (Figure 3(b)) due to TBAA-based PRE over redundant access expressions (tbaa). The dramatic reduction for Neural represents a reduction of

---

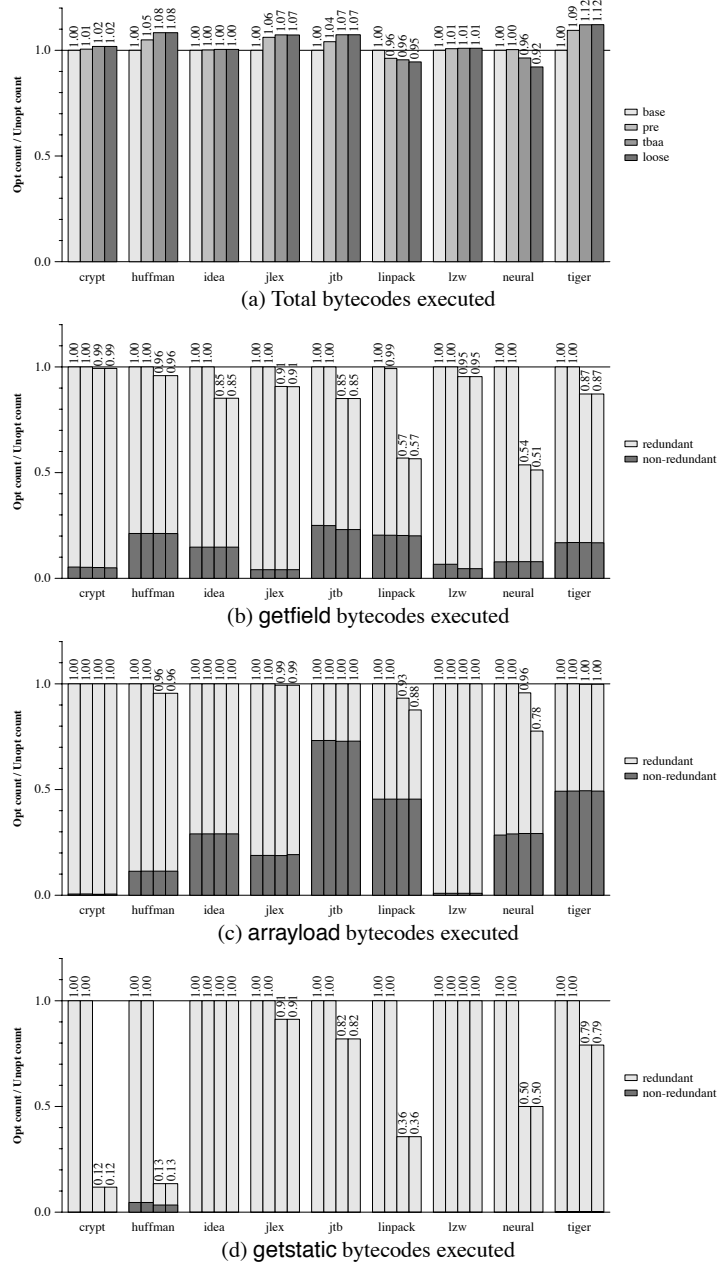[¶]See http://www.cs.msu.edu/~enbody/

Figure 3. Bytecodes executed (JDK).

redundant **getfield** operations from 9% of total bytecodes executed to 5% of total bytecodes. Linpack's reductions are similar, but **getfield** bytecodes represent just 0.02% of total bytecodes executed so the impact is minimal. Relaxing Java's precise exception requirement (**loose**) yields little benefit for **getfield**.

The array intensive benchmarks (Huffman, Linpack, and Neural) obtain noticeable reductions in **arrayload** frequency (Figure 3(c)). Interestingly, relaxing Java's precise exceptions gives significant improvement for both Linpack and Neural, because freedom from concern over precise delivery of array out of bounds exceptions provides more opportunity for PRE-based code motion. The Huffman, Linpack, and Neural benchmarks, which have heavy array use (4, 9 and 11%, respectively), see an elimination of 4–7% of the **arrayload** bytecodes for TBAA-based PRE with precise exceptions (**tbaa**). Relaxing exception delivery (**loose**) sees reductions in **arrayloads** increase to a peak of 22% for Neural. Further improvement would accrue if array subscripts could be disambiguated via range analysis on the subscript expressions for use during array alias analysis. Few **arrayload** bytecodes are eliminated in any of the other benchmarks, primarily because their array accesses are hidden inside method calls to library classes, etc.

The most dramatic gains are for **getstatic** accesses (Figure 3(d)), primarily because almost all such accesses are globally redundant. That we come close to the limit in eliminating almost all redundant accesses for Crypt and Huffman demonstrates the effectiveness of even simple alias analyses such as TBAA. The benchmarks where PRE does not eliminate many **getstatic** bytecodes do not have many repeat accesses to begin with.

For some benchmarks the extra **load** and **store** bytecodes are offset by elimination of partially-redundant code and replacement of many expensive long **load/store** bytecode forms with their cheaper short alternatives. Figure 4 highlights these conversions. The effect is most notable with LZW where the frequency of the **load** bytecodes decreases from 11% to 1% of the total bytecodes executed and the frequency of **load**n increases from 20% to 28%. These effects result in less overhead in the interpreter's bytecode dispatch loop. The impact on data cache reads (remember, bytecodes are data) is revealed in Figure 5(c). The large increase in stores for Linpack is due to PRE's elimination of significant numbers of redundant arithmetic expressions.

The positive impact on elapsed time for interpretation by the JDK (Figure 5(a)) is plain for Linpack, LZW and Neural. The other benchmarks see either slightly degraded performance for optimized code or no improvement. Certainly, the number of data reads (Figure 5(b)) is noticeably reduced with optimization for several benchmarks, though slightly increased for others. Along with the number of data reads, data cache read misses (Figure 5(c)) are markedly reduced for Linpack, LZW and Neural, hence their elapsed time improvement. Instruction fetch stalls (Figure 5(d)) reveal one aspect of code transformation that we cannot carefully control, since optimizations may inadvertently introduce increased instruction cache pressure and conflicts. Crypt and Linpack are severely affected by this secondary behavior, but the impact on execution time is outweighed by the primary effect of the optimizations.

JIT

The JIT environment is not influenced by conversion of long bytecode forms to their short variants, since JIT eliminates the bytecode dispatch overhead that we were able to reduce for JDK. Nor, since JIT allocates local variables to registers, do the extra **load** and **store** bytecodes matter much since they
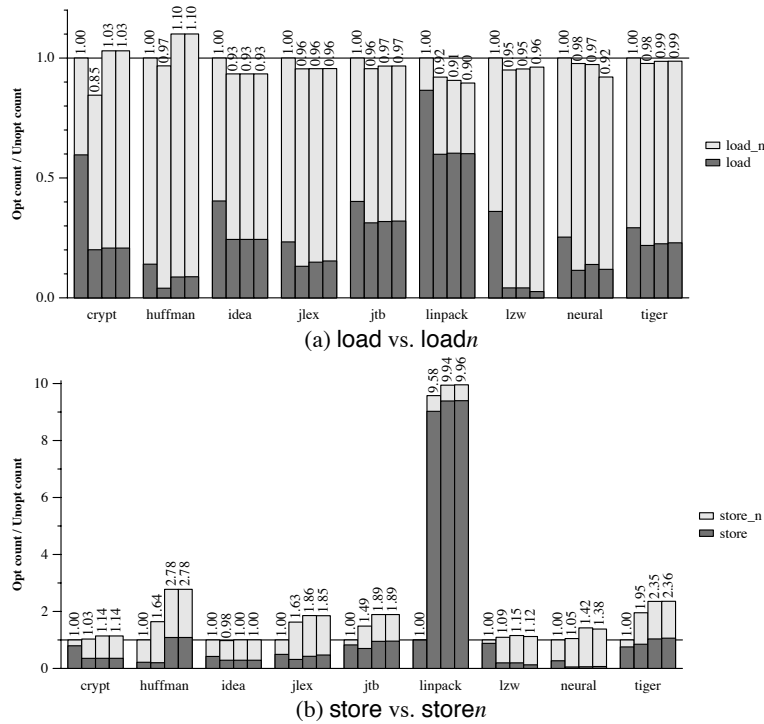
Figure 4. load/store bytecodes executed.

are converted to register accesses. The JIT's register allocator seems unfazed even by the large number of temporaries and associated loads and stores introduced with PRE for Linpack. Our experiences with earlier releases of the JDK with JIT were much less pleasant, since they implemented much less effective register allocation, and Linpack performed poorly as a result. Here, both Crypt and Linpack see improvements in elapsed times (Figure 6(a)); the remaining benchmarks are unaffected or slightly degraded. Again, Crypt and Linpack are the biggest winners in reductions in data reads (Figure 6(b)) and data cache read misses (Figure 6(c)).

Toba

With Toba all benchmarks show reductions in data reads (Figure 7(b)). Thus, our optimizations expose opportunities that the C compiler cannot exploit on its own. Unfortunately, these are not reflected in noticeably reduced elapsed times (Figure 7(a)), except for Crypt which sees a very significant reduction in data cache read misses (Figure 7(c)).
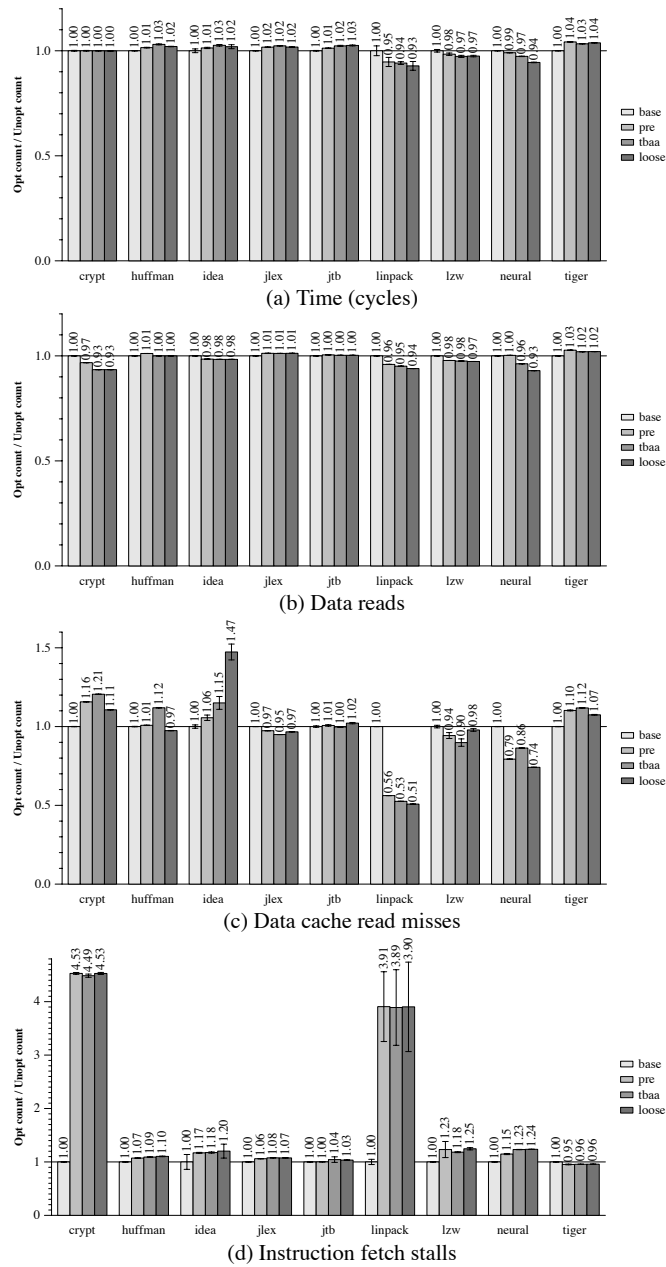
(a) Time (cycles)

(b) Data reads

(c) Data cache read misses

(d) Instruction fetch stalls

Figure 5. JDK metrics.

(a) Time (cycles)

(b) Data reads
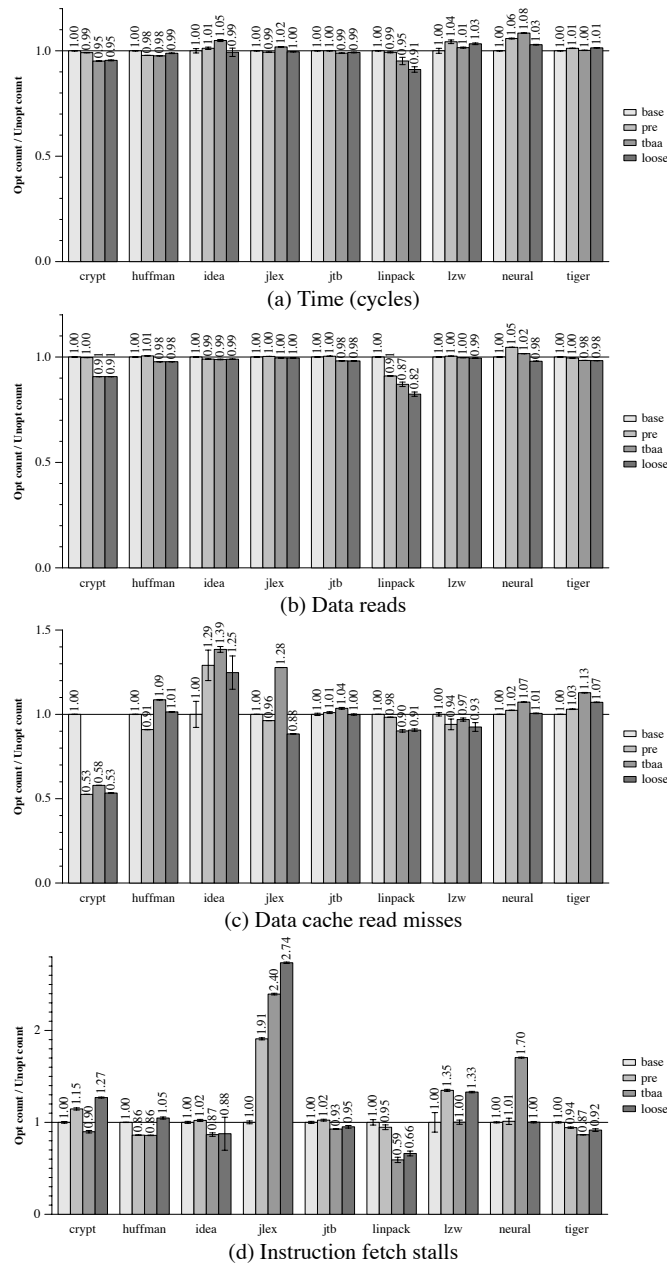
(c) Data cache read misses
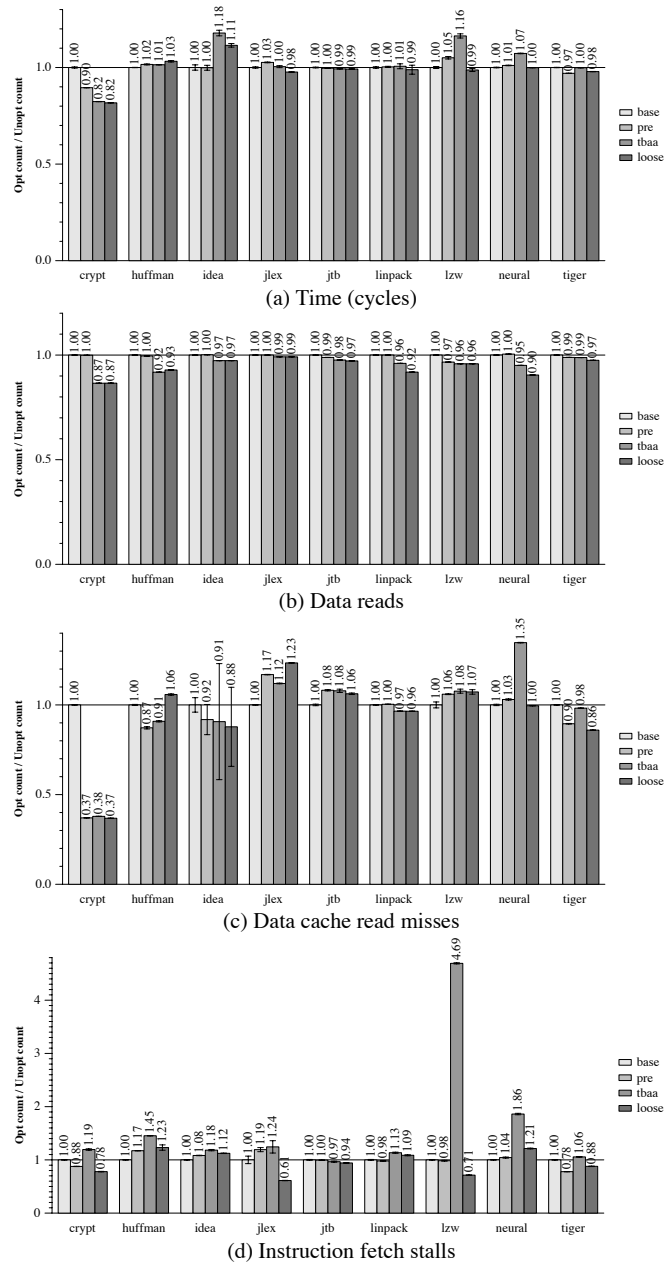
(d) Instruction fetch stalls

Figure 6. JIT metrics.

Figure 7. Toba metrics.

Summary of results

The dynamic bytecode counts (Figure 5) revealed significant reductions in the number of bytecodes needed for access expressions (getfield, arrayload, and getstatic). When executed on the JDK (without JIT), this bytecode mix reduced the number of data reads by up to 7%. Unfortunately, an increase in the total number of bytecodes for some benchmarks actually increased the number of reads needed to fetch these extra bytecodes for interpretation, so some benchmarks saw a degradation in elapsed time for the optimized code.

With JIT compilation, we see some elapsed time improvements, though it is difficult to isolate the impact of the underlying JIT compiler's optimizations from our own. In many cases, the JIT may be extracting the same optimizations from the unoptimized bytecode that we realize in the unoptimized bytecode. It is likely that the bytecode and JIT optimizers are in conflict. In future work, we plan to work with the source code of a state-of-the-art JIT compiler so as to turn off those optimizations that might interfere with our bytecode-level transformations. In sum, however, these results promise more success if control over the JIT can be obtained, as witnessed by the general improvement in data cache read counts in Figure 6(b).

Even more convincing results for data cache reads were obtained with Toba, as displayed in Figure 7(b). Again, the positive impact on execution times is not obvious, with some benchmarks exhibiting significantly degraded performance for bytecode that is optimized to respect precise exceptions. Here, our measurements also appear to be the victim of erratic cache behavior in the face of program reorganization resulting from our optimizing transformations.

As new architectures become more dependent on cache performance, we believe that reduction of data cache read counts will have a more significant impact on overall performance. Thus, we are optimistic that our techniques will come to yield more convincing bottom-line performance improvements.

RELATED WORK

The recent literature on alias analysis is extensive [30–46]. As in Diwan *et al.* [2], our results are distinguished from prior work by comprehensive evaluation of TBAA with respect to a particular optimization, in this case PRE over access expressions, and metrics and upper bounds on redundant run-time memory accesses, as opposed to static measurements.

Batory and Kim [47] 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 inter-procedural optimizations: *object inlining* and *code duplication*. Similar in some respects to the well-known approaches of cloning and inlining, these optimizations yield factors of two to five in performance improvement, so are consistent with results reported elsewhere [48–55].

Cierniak and Li [56] 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 [56] and Budimlic and Kennedy [47] is unable to determine to what extent they respect Java's precise exception semantics and its constraints on code motion. Still, both of these prior efforts are much more aggressive than us in the transformations they are willing to apply. We hope that TBAA-based PRE for access expressions will produce results as spectacular as theirs when combined with more aggressive inter-procedural analyses, such as they describe.

Added evidence for this comes from Diwan *et al.* [2] in their work with elimination of common access expressions for Modula-3. Their results indicate that accesses are often only *partially*-redundant across calls, while their optimizer only eliminates fully redundant access expressions. Of course, our PRE-based approach eliminates partial redundancies by definition. Diwan's results for elimination of fully redundant accesses without inter-procedural analysis are broadly consistent with ours.

Recently, Gupta *et al.* [5] have devised both static and dynamic analyses to optimize Java in the presence of exceptions. They demonstrate significant speedups for small numerical kernels (matrix multiplication and Cholesky factorization). The impact of their techniques for more general-purpose codes has not yet been determined. In this paper we have evaluated the impact of a limited set of optimizations that take advantage of knowledge of the data-flow impact on precise exceptions, as well as to establish a lower bound for PRE-style optimizations in which precise exception semantics are not enforced (with the loose results).

Several recent papers have focused on *register promotion* [57–59]: the identification of program regions in which memory-allocated values can be cached in registers. These techniques also address the issue of eliminating redundant loads and stores by selectively promoting values from memory into registers. Our approach differs in that we perform analysis and transformation at a higher level than these other approaches, with full knowledge of the types of the memory values being promoted. We are currently working to understand the precise relationship between our approach and these lower level techniques. Certainly, given the problems we have with loading and storing of temporaries in some benchmarks, it seems that our approach might benefit from the more selective placement of loads and stores that these promotion techniques employ.

CONCLUSIONS AND FUTURE WORK

Our results reveal the promise of optimization of Java classes independently of the source-code compiler and the runtime execution engine. In particular, we have demonstrated improvements using TBAA-based PRE over access path expressions, producing significant reductions in memory access operations. However, our results were mixed in their impact on elapsed execution time. Applying inter-procedural 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 intra-procedural context is severely limited.

Under some circumstances Java's precise exception model is overly constraining for code motion optimizations such as PRE. Relaxing the constraints can provide more opportunities for optimization, and noticeable improvements. Improved analysis to provide more accurate static information as to

what code fragments can never raise exceptions will allow optimizers to harvest these gains without compromising the precise exception model.

The implementation of further analyses and optimizations to BLOAT is under way, and the tool is now freely available for others to use [60]. One application domain we are now focusing on is analysis and optimization of Java programs in a persistent environment [61]. The structure access optimizations we have explored here prove particularly fruitful in a persistent setting, where loads and stores carry additional semantics, acting not just on virtual memory, but also on persistent storage [62–65].

REFERENCES

1. Morel E, Renvoise C. Global optimization by suppression of partial redundancies. *Communications of the ACM* 1979; **22**(2):96–103.
2. Diwan A, McKinley KS, Moss JEB. Type-based alias analysis. *Proceedings of the ACM Conference on Programming Language Design and Implementation*, Montréal, Canada, June 1998. *ACM SIGPLAN Notices* 1998; **33**(5):106–117.
3. Gosling J, Joy B, Steele G. *The Java Language Specification*. Addison-Wesley, 1996.
4. Larus JR, Hilfinger PN. Detecting conflicts between structure accesses. *Proceedings of the ACM Conference on Programming Language Design and Implementation*, Atlanta, Georgia, June 1988; 21–34.
5. Gupta M, Choi J-D, Hind M. Optimizing Java programs in the presence of exceptions. *Proceedings of the European Conference on Object-Oriented Programming*, Cannes, France, June 2000. (*Lecture Notes in Computer Science*, vol. 1850), Bertino E (ed). Springer-Verlag, 2000; 422–446.
6. Pugh W. Fixing the Java memory model. *ACM Java Grande Conference*, San Francisco, California, June 1999.
7. Lindholm T, Yellin F. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
8. Cytron R, Ferrante J, Rosen BK, Wegman MN, Zadeck FK. Efficiently computing static single assignment form and the program dependence graph. *ACM Trans. Program. Lang. Syst.* 1991; **13**(4):451–490.
9. Wolfe M. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.
10. Briggs P, Cooper KD, Harvey TJ, Simpson LT. Practical improvements to the construction and destruction of static single assignment form. *Software—Practice and Experience* 1998; **28**(8):859–881.
11. Simpson LT. Value-driven redundancy elimination. *PhD Thesis*, Rice University, Houston, TX, 1996.
12. Chow F, Chan S, Kennedy R, Liu S-M, Lo R, Tu P. A new algorithm for partial redundancy elimination based on SSA form. *Proceedings of the ACM Conference on Programming Language Design and Implementation*, Las Vegas, Nevada, June 1997. *ACM SIGPLAN Notices* 1997; **32**(5):273–286.
13. Wegman MN, Zadeck FK. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.* 1991; **13**(2):181–210.
14. Alpern B, Wegman MN, Zadeck FK. Detecting equality of values in programs. *Conference Record of the ACM Symposium on Principles of Programming Languages*, San Diego, California, January 1988; 1–11.
15. Rosen BK, Wegman MN, Zadeck FK. Global value numbers and redundant computations. *Conference Record of the ACM Symposium on Principles of Programming Languages*, San Diego, California, January 1988; 12–27.
16. Cooper K, Simpson LT. SCC-based value numbering. *Technical Report CRPC-TR95636-S*, Rice University, 1995.
17. Briggs P, Cooper KD, Simpson LT. Value numbering. *Software—Practice and Experience* 1997; **27**(6):701–724.
18. Gerlek MP, Stoltz E, Wolfe M. Beyond induction variables: detecting and classifying sequences using a demand-driven SSA form. *ACM Trans. Program. Lang. Syst.* 1994; **17**(1):85–122.
19. Click C. Global code motion/global value numbering. *Proceedings of the ACM Conference on Programming Language Design and Implementation*, La Jolla, California, June 1995. *ACM SIGPLAN Notices* 1995; **30**(6):246–257.
20. Havlak P. Nesting of reducible and irreducible loops. *ACM Trans. Program. Lang. Syst.* 1997; **19**(4):557–567.
21. Agesen O, Detlefs D, Moss JEB. Garbage collection and local variable type-precision and liveness in Java virtual machines. *Proceedings of the ACM Conference on Programming Language Design and Implementation*, Montréal, Canada, June 1998. *ACM SIGPLAN Notices* 1998; **33**(5):269–279.
22. Palsberg J, Schwartzbach MI. *Object-Oriented Type Systems*. Wiley, 1994.
23. Chaitin GJ. Register allocation and spilling via graph coloring. *Proceedings of the ACM Symposium on Compiler Construction*, Boston, Massachusetts, June 1982. *ACM SIGPLAN Notices* 1982; **17**(6):98–105.
24. Briggs P, Cooper KD, Torczon L. Improvements to graph coloring register allocation. *ACM Transactions Program. Lang. Syst.* 1994; **16**(3):428–455.

25. Appel AW. *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.
26. Proebsting TA, Townsend G, Bridges P, Hartman JH, Newsham T, Watterson SA. Toba: Java for applications—A way ahead of time (WAT) compiler. *Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems*, Portland, Oregon, June 1997. USENIX, 1997. See http://www.cs.arizona.edu/sumatra/toba.
27. Gosling J, Yellin F, The Java Team. *The Java Application Programming Interface*, vol. 1: *Core Packages*. Addison-Wesley, 1996.
28. SunSoft. *Java On Solaris 2.6: A White Paper*, 1997.
29. Boehm H-J, Weiser M. Garbage collection in an uncooperative environment. *Software—Practice and Experience* 1988; **18**(9):807–820.
30. Chase DR, Wegman M, Zadeck FK. Analysis of pointers and structures. *Proceedings of the ACM Conference on Programming Language Design and Implementation*, White Plains, New York, June 1990. *ACM SIGPLAN Notices* 1990; **25**(6):296–310.
31. Landi W, Ryder BG. A safe approximate algorithm for interprocedural pointer aliasing. *Proceedings of the ACM Conference on Programming Language Design and Implementation*, San Francisco, California, June 1992. *ACM SIGPLAN Notices* 1992; **27**(7):235–248.
32. Choi J-D, Burke M, Carini P. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. *Conference Record of the ACM Symposium on Principles of Programming Languages*, Charleston, South Carolina, January 1993; 232–245.
33. Landi W, Ryder BG, Zhang S. Interprocedural modification side effect analysis with pointer aliasing. *Proceedings of the ACM Conference on Programming Language Design and Implementation*, Albuquerque, New Mexico, June 1993. *ACM SIGPLAN Notices* 1993; **28**(6):56–67.
34. Hummel J, Hendren LJ, Nicolau A. A general data dependence test for dynamic, pointer-based data structures. *Proceedings of the ACM Conference on Programming Language Design and Implementation*, Orlando, Florida, June, 1994. *ACM SIGPLAN Notices* 1994; **29**(6):218–229.
35. Deutsch A. Interprocedural may-alias analysis for pointers: Beyond *k*-limiting. *Proceedings of the ACM Conference on Programming Language Design and Implementation*, Orlando, Florida, June 1994. *ACM SIGPLAN Notices* 1994; **29**(6):230–241.
36. Emami M, Ghiya R, Hendren LJ. Context-sensitive interprocedural points-to analysis in the presence of function pointers. *Proceedings of the ACM Conference on Programming Language Design and Implementation*, Orlando, Florida, June 1994. *ACM SIGPLAN Notices* 1994; **29**(6):242–256.
37. Altucher RZ, Landi W. An extended form of must alias analysis for dynamic allocation. *Conference Record of the ACM Symposium on Principles of Programming Languages*, January 1995; 74–84.
38. Wilson RP, Lam MS. Efficient context-sensitive pointer analysis for c programs. *Proceedings of the ACM Conference on Programming Language Design and Implementation*, La Jolla, California, June 1995. *ACM SIGPLAN Notices* 1995; **30**(6):1–12.
39. Ruf E. Context-insensitive alias analysis reconsidered. *Proceedings of the ACM Conference on Programming Language Design and Implementation*, La Jolla, California, June 1995. *ACM SIGPLAN Notices* 1995; **30**(6):13–22.
40. Ghiya R, Hendren LJ. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. *Conference Record of the ACM Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, January 1996; 1–15.
41. Steensgaard B. Points-to analysis in almost linear time. *Conference Record of the ACM Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, January 1996; 32–41.
42. Shapiro M, Horwitz S. Fast and accurate flow-insensitive points-to analysis. *Conference Record of the ACM Symposium on Principles of Programming Languages*, Paris, France, January 1997; 1–14.
43. Debray S, Muth R, Weippert M. Alias analysis of executable code. *Conference Record of the ACM Symposium on Principles of Programming Languages*, San Diego, California, January 1998; 12–24.
44. Ghiya R, Hendren LJ. Putting pointer analysis to work. *Conference Record of the ACM Symposium on Principles of Programming Languages*, San Diego, California, January 1998; 121–133.
45. Hasti R, Horwitz S. Using static single assignment form to improve flow-insensitive pointer analysis. *Proceedings of the ACM Conference on Programming Language Design and Implementation*, Montréal, Canada, June 1998. *ACM SIGPLAN Notices* 1998; **33**(5):97–105.
46. Jagannathan S, Thiemann P, Weeks S, Wright A. Single and loving it: Must-alias analysis for higher-order languages. *Conference Record of the ACM Symposium on Principles of Programming Languages*, San Diego, California, January 1998; 329–341.
47. Budimlic Z, Kennedy K. Optimizing Java: Theory and practice. *Software—Practice and Experience* 1997; **9**(6):445–463.
48. Chambers C, Ungar D. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. *Proceedings of the ACM Conference on Programming Language Design and Implementation*, Portland, Oregon, June 1989. *ACM SIGPLAN Notices* 1989; **24**(7):146–160.

49. Chambers C, Ungar D, Lee E. An efficient implementation of Self, a dynamically-typed object-oriented language based on prototypes. *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, New Orleans, Louisiana, October 1989. *ACM SIGPLAN Notices* 1989; **24**(10):49–70.

50. Chambers C, Ungar D. Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs. *Proceedings of the ACM Conference on Programming Language Design and Implementation*, White Plains, New York, June 1990. *ACM SIGPLAN Notices* 1990; **25**(6):150–164.

51. Chambers C, Ungar D. Making pure object oriented languages practical. *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Phoenix, Arizona, October 1991. *ACM SIGPLAN Notices* 1991; **26**(11):1–15.

52. Chambers C. The design and implementation of the SELF compiler, an optimizing compiler for object-oriented programming languages. *PhD Thesis*, Stanford University, 1992.

53. Dean J, Chambers C, Grove D. Selective specialization for object-oriented languages. *Proceedings of the ACM Conference on Programming Language Design and Implementation*, La Jolla, California, June 1995. *ACM SIGPLAN Notices* 1995; **30**(6):93–102.

54. Dolby J. Automatic inline allocation of objects. *Proceedings of the ACM Conference on Programming Language Design and Implementation*, Las Vegas, Nevada, June 1997. *ACM SIGPLAN Notices* 1997; **32**(5):7–17.

55. Dolby J, Chien AA. An evaluation of automatic object inline allocation techniques. *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Vancouver, British Columbia, October 1998. *ACM SIGPLAN Notices* 1998; **33**(10):1–20.

56. Cierniak M, Li W. Optimizing Java bytecodes. *Concurrency—Practice and Experience* 1997; **9**(6):427–444.

57. Cooper K, Lu J. Register promotion in C programs. *Proceedings of the ACM Conference on Programming Language Design and Implementation*, Las Vegas, Nevada, June 1997. *ACM SIGPLAN Notices* 1997; **32**(5):308–319.

58. Sastry AVS, Ju RDC. A new algorithm for scalar register promotion based on SSA form. *Proceedings of the ACM Conference on Programming Language Design and Implementation*, Montréal, Canada, June 1998. *ACM SIGPLAN Notices* 1998; **33**(5):15–25.

59. Lo R, Chow F, Kennedy R, Liu S-M, Tu P. Register promotion by sparse partial redundancy elimination of loads and stores. *Proceedings of the ACM Conference on Programming Language Design and Implementation*, Montréal, Canada, June 1998. *ACM SIGPLAN Notices* 1998; **33**(5):26–37.

60. BLOAT: The bytecode-level optimization and analysis toolkit. ftp://ftp.cs.purdue.edu/pub/hosking/bloat.

61. Atkinson MP, Daynès L, Jordan MJ, Printezis T, Spence S. An orthogonally persistent Java. *ACM SIGMOD Record* 1996; **25**(4):68–75.

62. Cutts Q, Hosking AL. Analysing, profiling and optimising orthogonal persistence for Java. *Proceedings of the 2nd International Workshop on Persistence and Java* August 1997. Atkinson MP, Jordan MJ (eds). *Sun Microsystems Laboratories Technical Report 97-63*. 1997; 107–115.

63. Hosking AL, Nystrom N, Cutts Q, Brahnmath K. Optimizing the read and write barriers for orthogonal persistence. *Proceedings of the 8th International Workshop on Persistent Object Systems*, Tiburon, California, August 1998. *Advances in Persistent Object Systems*, Morrison R, Jordan M, Atkinson M (eds). Morgan Kaufmann, 1999; 149–159.

64. Cutts Q, Lennon S, Hosking AL. Reconciling buffer management with persistence optimisations. *Proceedings of the 8th International Workshop on Persistent Object Systems*, Tiburon, California, August 1998. *Advances in Persistent Object Systems*, Morrison R, Jordan M, Atkinson M (eds). Morgan Kaufmann, 1999; 51–63.

65. Brahnmath K, Nystrom N, Hosking AL, Cutts Q. Swizzle barrier optimizations for orthogonal persistence in Java. *Proceedings of the 3rd International Workshop on Persistence and Java*, Tiburon, California, August 1998. *Advances in Persistent Object Systems*, Morrison R, Jordan M, Atkinson M (eds). Morgan Kaufmann, 1999; 268–278.