Using inter-procedural side-effect information in JIT optimizations[⋆]

Anatole Le   Ondřej Lhoták   Laurie Hendren
{ale44,olhotak,hendren}@sable.mcgill.ca

Sable Research Group, McGill University, Montreal, Canada

Abstract. Inter-procedural analyses such as side-effect analysis can provide information useful for performing aggressive optimizations. We present a study of whether side-effect information improves performance in just-in-time (JIT) compilers, and if so, what level of analysis precision is needed.

We used SPARK, the inter-procedural analysis component of the SOOT Java analysis and optimization framework, to compute side-effect information and encode it in class files. We modified Jikes RVM, a research JIT, to make use of side-effect analysis in local common sub-expression elimination, heap SSA, redundant load elimination and loop-invariant code motion. On the SpecJVM98 benchmarks, we measured the static number of memory operations removed, the dynamic counts of memory reads eliminated, and the execution time.

Our results show that the use of side-effect analysis increases the number of static opportunities for load elimination by up to 98%, and reduces dynamic field read instructions by up to 27%. Side-effect information enabled speedups in the range of 1.08x to 1.20x for some benchmarks. Finally, among the different levels of precision of side-effect information, a simple side-effect analysis is usually sufficient to obtain most of these speedups.

1 Introduction

Over the past several years, just-in-time (JIT) compilers have enabled impressive improvements in the execution of Java code, mainly through local and intra-procedural optimizations, speculative inter-procedural optimizations, and efficient implementation techniques. However, JITs do not generally make use of whole-program analysis information, such as conservative call graphs, points-to information, or side-effect information, because it is too costly to compute it each time a program is executed. However, all non-trivial data types in Java are objects always accessed through indirect references (pointers), so one would expect optimizations using side-effect information to enable significant further improvements in performance of Java programs.

The purpose of the study presented in this paper is to answer two key questions. First, is side-effect information useful for the optimizations performed in a modern JIT, and can it significantly improve performance? Second, what level of precision of the side-effect information and the underlying analyses used to compute it is required to obtain these performance improvements?

To study these questions, we implemented a system consisting of an ahead-of-time inter-procedural side-effect analysis, whose result is communicated to a modified JIT containing optimizations that we adapted to take advantage of the side-effect information.

We implemented the side-effect analyses using the SPARK [15, 16] points-to analysis framework, a part of the SOOT [29] bytecode analysis, optimization, and annotation

[⋆] This work was supported, in part, by NSERC and FQRNT.

framework. The side-effect analysis makes use of points-to and call graph information computed by SPARK. The resulting side-effect information is encoded in class file attributes for use by the JIT using the annotation framework [21] included in SOOT.

We chose Jikes RVM [2] as the JIT for our study, and made several modifications to it. First, we added code to read in the side-effect information produced in our analysis. We then modified several analyses and optimizations to take advantage of the information, including local common subexpression elimination, heap array SSA construction, redundant load elimination, and loop-invariant code motion. Finally, we instrumented Jikes RVM both to count the static opportunities for performing optimizations, and to insert instrumentation code to measure the dynamic effects of the improved optimizations.

The contributions of this paper are the following:

– This is the first published presentation of the side-effect analysis that we have implemented in SOOT using points-to and call graph information computed by SPARK.
– To our knowledge, this is the first study of the run-time performance improvements obtainable by taking advantage of side-effect information in a range of optimizations in a Java JIT.
– We present empirical evidence that the availability of side-effect information in a Java JIT can enable significant performance improvements of up to 20%.
– We show that although precise analyses provide significantly more optimization opportunities when counted statically, most of the dynamic improvement is obtainable even with relatively simple, imprecise analyses. In particular, a side-effect analysis based on a call graph constructed using an inexpensive Class Hierarchy Analysis (CHA) already provides a very significant improvement over not having any side-effect information at all. This confirms what has been observed on other languages such as Modula-3 and C.

The remainder of this paper is organized as follows. Section 2 is devoted to our side-effect analysis in SOOT, the call graph and points-to analyses that it depends on, issues with encoding its result in class file attributes, and the precision variations with which we experimented. In Section 3, we describe how we modified the optimizations in Jikes RVM to take advantage of side-effect information. In Section 4, we present the benchmarks that we used, our experiments, and our empirical results. We discuss related work in Section 5, and we conclude with Section 6.

2 Side-effect analysis in Soot

We implemented side-effect analysis in SOOT [29], a framework for analyzing, optimizing, and annotating Java bytecode. The side-effect analysis depends on two other interprocedural analyses, call graph construction and points-to analysis. We describe how we construct a call graph in Section 2.1. An important difference from most other work on call graph construction is that to obtain a conservative side-effect analysis, we need to ensure that our call graph includes all methods invoked, including those invoked implicitly by the Java VM. In Section 2.2, we briefly explain the output of SPARK, our points-to analysis framework [15, 16]. Section 2.3 explains how we put the information from these two analyses together and produce side-effect information. In Section 2.4, we briefly note some issues with encoding the side-effect analysis results in class file attributes to communicate them to the JIT. Finally, in Section 2.5, we describe how variations in the precision of the call graph and points-to analyses affect the side-effect information.

2.1 Call Graph Construction

To perform an inter-procedural analysis on a Java program, information about the possible targets of method calls is required. This information is approximated by a call graph, which maps each statement s to a set $cg(s)$ containing every method that may be called from s. Constructing a call graph for a Java program is complicated by the fact that most calls in Java are virtual, so the target method of the call depends on the run-time type of the receiver object.

In our study, we compared two different methods of computing call graphs. First, we computed call graphs using Class Hierarchy Analysis (CHA) [8], an inexpensive method which considers only the static type of each receiver object, and does not require any inter-procedural analysis. Second, we used a points-to analysis (discussed in the next section) to compute the run-time types of the objects that the receiver of each call site could point to, and we determined the target method that would be invoked for each run-time receiver type.

Several important, but subtle, details of the Java virtual machine (VM) complicate the construction of a conservative call graph suitable for side-effect analysis. In a Java program, methods may be invoked not only due to explicit invoke instructions, but also implicitly due to various events in the VM. Whenever a new class is first used, the VM implicitly calls its static initialization method. The set of events that may cause a static initialization method to be called is specified in [17, section 2.17.4]. In our analysis, we assume that any of these events could cause the corresponding static initialization method to be invoked. Each static initialization method is executed at most once in a given run of a Java program. Therefore, we use an intra-procedural flow-sensitive analysis to eliminate spurious calls to static initialization methods which must have already been called on every path from the beginning of the method. In addition, the standard class library often invokes methods using the `doPrivileged` methods of `java.security.AccessController`. Our analysis models these with calls of the `run` method of the argument passed to `doPrivileged`. Methods may also be invoked using reflection. In general, it is not possible to determine statically which methods will be invoked reflectively, and our analysis only issues a warning if it finds a reachable call to one of the reflection methods. However, calls to the `newInstance` method of `java.lang.Class` are so common that they merit special treatment. This method creates a new object and calls its constructor. In our analysis, we conservatively assume that any object could be created, and therefore any constructor with no parameters could be invoked.

To partially verify the correctness of the computed call graph, we instrumented the code to ensure that all methods that are executed at run time were included in the call graph and reachable from the entry points. To do this, we computed the set of methods that are not reachable from the entry points through the call graph, and modified them to abort the execution of the benchmark if they do get invoked at run time. Although this does not prove that every possible run-time call edge is included in the computed call graph, it does guarantee that every executed method is considered in call graph construction. To further check that our overall optimizations were conservative on the benchmarks studied, we verified that the benchmarks produced identical output in all configurations, including with the optimizations disabled.

2.2 Points-to Analysis

We use the SPARK [15, 16] points-to analysis framework to compute points-to information. For each *pointer* p in the program, it computes a set $pt(p)$ of *objects* to which it may point. The most common kind of *pointer* is a local variable of reference type in the Jimple representation of the code. Local variables appear in field read and write instructions as pointers to the object whose field is to be read or written, and in method invocation instructions as the receiver of the method call, which determines the method to be invoked. In addition, *pointers* are introduced to represent method arguments and return values, static fields, and special values needed in simulating the effects on pointers of native methods in the standard class library. Typically, an *object* is an allocation site; we model all run-time objects created at a given allocation site as a single entity. In addition, we must include special *objects* for run-time objects without an allocation site, such as objects created by the VM (the argument array to the main method, the main thread, the default class loader) and objects created using reflection. For some of these special *objects*, we may not know the exact run-time type. Therefore, we conservatively assume that their run-time type may be any subtype of their declared type.

SPARK performs a flow-insensitive, context-insensitive, subset-based points-to analysis by propagating *objects* from their allocation sites through all *pointers* through which they may flow. SPARK has many parameters for experimenting with variations of the analysis that affect analysis efficiency and precision. In this study, we experimented with four points-to analysis variations. We explain the variations in more detail in Section 2.5.

2.3 Side-Effect Analysis

The side-effect analysis consists of two steps, which are discussed in this section. First, we compute a read and write set for each statement. Second, we use the read and write sets to compute dependencies between all pairs of statements within each method.

For each statement s, we compute sets $read(s)$ and $write(s)$ containing every static field sf read (written) by s, and a pair (o, f) for every field f of *object* o that may be read (written) by s. These sets also include fields read (written) by all code executed during execution of s, including any other methods that may be called, directly or transitively. The read and write sets are computed in two steps. In the first step, we compute only the direct read and write sets for each statement in the program, ignoring any code that may be called from the statement. The result of the points-to analysis is used to determine the possible objects being pointed to by the pointer in each field read or write instruction. In the second step, we continually aggregate the read and write sets of each method and propagate them to all call sites of the method, until a fixed-point is reached. During the propagation, the call graph is used to determine the call sites of each method.

Once the read and write sets for all statements have been computed, for each method, we compute an interference relation between all the read and write sets in the method. Two sets interfere if they have a non-empty intersection. From the interference relation on read and write sets, we construct four dependence relations between statements (read-read dependence, read-write dependence, write-read dependence, write-write dependence). For example, there is a read-write dependence between statements s_1 and s_2 if the read set of s_1 and the write set of s_2 interfere. It is the dependences between statements that we encode in class files for the JIT to use in performing optimizations.

2.4 Encoding Side-Effects in Class File Attributes

All of the analyses described in the preceding sections are performed on Jimple, the three-address intermediate representation (IR) used in SOOT. In order to communicate the analysis results to a JIT, we must convert them to refer to bytecode instructions during the translation of Jimple to bytecode. SOOT includes a universal tagging framework [21] that propagates analysis information through its various IRs, and encodes it in class file attributes. An important complication in this process is that one Jimple statement may be converted to multiple bytecode instructions. However, Jimple is low-level enough that whenever a Jimple instruction has side-effects, exactly one of the bytecode instructions generated for it has those side-effects. Therefore, for each type of Jimple instruction, we identify the relevant bytecode instruction to the tagging framework, and it attaches the side-effect information to that instruction.

Another complication in communicating the side-effect information is that some methods have a large number of statements with side-effects. Since the dependence relations may have size quadratic in the number of instructions with side-effects, a naive encoding of the dependence relations is sometimes unacceptably large. However, we have observed in those cases, many of the read and write sets in the method are identical. Therefore, we add a level of indirection. Instead of expressing the dependence relations in terms of statements, we enumerate all distinct read and write sets, and express the dependence relations between those sets. For each statement, we indicate which set it reads and writes. The resulting encoding has size $\Theta(m^2+n)$, where n is the number of statements, and m is the number of unique sets. In an earlier study [15, Sections 6.2.2 and 6.2.6], we observed that this encoding limits the annotation size to acceptable levels.

2.5 Analysis Variations

In this section, we briefly explain the differences between the analysis variations that we compare in our empirical study in Section 4. Figure 1 gives an overview of the relative precision of the variations, with precision increasing from bottom to top.

$$\begin{tikzcd}
 & \text{otf-fs} & \\
\text{otf-fb} \arrow[ur] & & \text{aot-fs} \arrow[ul] \\
 & \text{aot-fb} \arrow[ul] \arrow[ur] & \\
 & \text{CHA} \arrow[u] & \\
 & \text{none} \arrow[u] &
\end{tikzcd}$$

Fig. 1. Relative Precision of Analysis Variations

For the first variation, none, we compute no side-effect information at all, and rely only on the internal analysis in the Jikes RVM JIT for optimizations. This means that any method call in the code is conservatively assumed to read and write anything in the heap.

Our second variation, CHA, is to compute side-effects using a call graph, but without performing any points-to analysis. We construct the call graph using CHA, as described in Section 2.1. In this case, the side-effect information contains a list of all fields possibly read and written at each call site; the JIT takes advantage of the knowledge that no other fields will be accessed. However, this analysis does not distinguish between the same field of different objects.

The remaining variations all take advantage of points-to information of different levels of precision to distinguish different objects. We describe these differences only briefly, because although they do affect the analysis precision measured statically, we found their effect on the dynamic behaviour of real benchmarks to be negligible.

In a field-based analysis (fb), a single points-to set is used for each field regardless of which object it is a field of. On the other hand, a field-sensitive analysis (fs) computes a separate points-to set for each pair (*object*, *field*). Therefore, if an *object* is written to `b1.a` and a different *object* is written to `b2.a`, and if `b1` and `b2` are known to not be aliases, then a field-sensitive analysis determines that `b1.a` and `b2.a` point to different *objects*. In contrast, a field-based analysis does not make this distinction because it considers only the field `a`, and ignores the *objects* (`b1` and `b2`).

To propagate points-to sets inter-procedurally, a points-to analysis requires an approximation of the call graph, but we use the points-to information to build the call graph. We resolve this circular dependency by either building an imprecise initial CHA call graph only for the use of the points-to analysis (aot), or by constructing the call graph on-the-fly as the points-to analysis proceeds (otf): as points-to sets grow, we add edges to the call graph.

3 Optimizations enabled in Jikes RVM

The JIT compiler that we modified to make use of side-effect information is the Jikes Research Virtual Machine (RVM) [2]. Jikes RVM is an open source research platform for executing Java bytecode. It includes three levels of JIT optimizations (0, 1 and 2). We adapted three optimizations in Jikes RVM to make use of side-effect information: local common sub-expression elimination (CSE), redundant load elimination (RLE) and loop-invariant code motion (LICM). Sections 3.1 to 3.3 describe each of these optimizations and the changes that we made. Because side-effect information refers to the original bytecode of a method, bytecodes that come from an inlined method need to be treated specially. Section 3.4 describes how we dealt with this case.

3.1 Local common sub-expression elimination

The first optimization in Jikes RVM that we modified to make use of side-effect information is local CSE. This optimization is only performed within a basic block. The algorithm for performing CSE on fields is described in Figure 2(a). A cache is used to store the available field expressions. The algorithm iterates over all instructions in a basic block, and processes them. There are two parts in this process. The first is to try to replace each *getfield* or *getstatic* instruction encountered by an available expression. If one is available, it is assigned to a temporary variable and the *getfield* or *getstatic* instruction is replaced by a copy of the temporary. If none is available, a field expression is added to the cache for the *getfield* or *getstatic* instruction. For every *putfield* and *putstatic* instruction, an associated field expression is also added to the cache. The second part is to update the cache according to which expressions the current instruction kills. A call or synchronization instruction kills all expressions in the cache. A *putfield* or *putstatic* of some field X will remove any expression in the cache associated with field X.

In this algorithm, we used side-effect information to reduce the set of expressions killed (lines 13 and 15 in Figure 2(a)). When the current instruction is a field store or a call, we only remove from the cache entries that have a read-write or write-write dependence with the current instruction in the side-effect analysis.

(a)

```
1:  for each basic block bb do
2:    cache = createNewEmptyCache();
3:    for each instruction s in bb do
4:      if isVolatileFieldLoadOrStore( s ) then
5:        continue
6:      if isGetField( s ) or isGetStatic( s ) then
7:        if cache.availableExpression( s ) then
8:          T = findOrCreateTemporary( expression( s ) )
9:          replace s by copyTemporaryInstruction( T )
10:       else
11:         add expression( s ) to cache
12:     else if isPutField( s ) or isPutStatic( s ) then
13:       add expression( s ) to cache
14:     if isPutField( s ) or isPutStatic( s ) of field X then
15:       remove all expressions with field X from cache
          (excluding expression( s ))
16:     else if s is a call or synchronization then
17:       remove all expressions from cache
```

(b)

```
1   A obj1 = new A();
2   A obj2 = new A();
3   i = obj1.x;
4   obj2.x = 10;
5   nothing();
6   j = obj1.x;
```

Fig. 2. Local common sub-expression (a) original algorithm (b) example

An example is shown in Figure 2(b). Without side-effect information, the compiler would conservatively assume that statement `obj2.x = 10` could write to memory location `obj1.x` and that the call to `nothing()` could write to any memory locations. In contrast, the side-effect analysis would specify that there is no dependence between these instructions, and thus enable the replacement of the load of `obj1.x` on line 6 by an available expression (line 3).

3.2 Redundant load elimination

The redundant load elimination algorithm relies on extended Array SSA (also known as Heap Array SSA or Heap SSA) [10] and Global Value Numbering [3]. We explain the general idea of the algorithm below. For a detailed description, please refer to [10].

The algorithm transforms the IR into heap SSA form. A heap array is created for each object field. The object reference is used as the index into this heap array. For example, line 2 of Figure 3(a) would be represented as "heap array X [`a`] = 2" meaning that a store is performed in heap array X at index `a` (the object reference).

After the transformation to heap SSA form is completed, global value numbers are computed. The global value numbering computes definitely-different (*DD*) and definitely-same (*DS*) relations for object references. The *DD* relation distinguishes two object references coming from different allocation sites, or when one is a method parameter and the other one is the result of a new statement. The *DS* relation returns true when two object references have the same value number (one is a copy of the other).

Once global value numbers are computed, index propagation is performed. The index propagation solution holds the available indices into heap arrays at each use of a heap array. Scalar replacement is performed using the sets of available indices. Note that in the algorithm, these sets actually contain value numbers of available indices. For simplicity, we consider sets of available indices.

For increasing the number of opportunities for load elimination, we used side-effect information during the heap SSA transformation and in the *DD* relation. During the heap

SSA construction, without side-effect information, each call instruction is annotated with a definition and a use of every heap array. With side-effect information we annotate a call with a definition of a heap array, say X, only if there is a write-read or write-write dependence between the call and the instruction using heap array X. Similarly we annotate a call with a use of a heap array if there is a read-read or read-write dependence. We also use side-effect information when the *DD* relation returns false. Two instructions having no data dependence is equivalent to *DD*(`a`, `b`) = true, where `a` and `b` are the object references used in the instructions.

In Figure 3(a), without side-effect information, since `a` and `b` are method parameters, *DD*(`a`, `b`) = false. Thus, only {`b`} is available after line 3. This allows the load of `b.x` on line 9 to be eliminated. Since it is conservatively assumed that calls can write to any memory location, the available index set after `nothing()` on line 10 is the empty set. Line 12 represents a merge point of the available index sets after line 7 and 10. The intersection of these two sets is the empty set. After the load of `a.x` on line 14, {`a`} is available. Since *DS*(`a`, `b`) = false, the load of `b.x` on line 15 cannot be eliminated.

(a)

```
1    int foo( A a, A b, int n ) {
2      a.x = 2;
3      b.x = 3;
4
5      int i;
6      if( n > 0 ) {
7        i = a.x;
8      } else {
9        i = b.x;
10       nothing();
11     }
12     // Merging point: a phi is
13     // placed here in heap SSA
14     int j = a.x;
15     int k = b.x;
16     return i + j + k;
17   }
18
19   public static void
20       main( String[] args ) {
21     foo( new A(), new A(), 1 );
22   }
```

(b)

```
1    int foo( A a, A b, int n ) {
2      t1 = 2;
3      a.x = t1;
4      t2 = 3;
5      b.x = t2;
6
7      int i;
8      if( n > 0 ) {
9        i = t1;
10     } else {
11       i = t2;
12       nothing();
13     }
14     // Merging point: a phi is
15     // placed here in heap SSA
16     int j = t1;
17     int k = t2;
18     return i + j + k;
19   }
20
21   public static void
22       main( String[] args ) {
23     foo( new A(), new A(), 1 );
24   }
```

Fig. 3. Redundant load elimination example (a) before (b) after

Using side-effect analysis, since `a.x` has no dependence with `b.x` (line 2 and 3) the available index set after line 3 is {`a`, `b`}. Thus, loads of `a.x` and `b.x` on line 7 and 9 can be eliminated. The available index set after line 7 is {`a`, `b`}, and after line 10, it is also {`a`, `b`}, since `nothing()` has no side-effect. The intersection at the merge point (line 12) results in the set {`a`, `b`}. The load of `a.x` can then be removed on line 14. The available index set after line 14 is {`a`, `b`}, allowing load elimination of `b.x` on line 15. The resulting code after performing load elimination is shown in Figure 3(b).

3.3 Loop-invariant code motion

The LICM algorithm in Jikes RVM is an implementation of the Global Code Motion algorithm introduced by Click [7] and is adapted to handle memory operations. As such, it requires the IR to be in heap SSA form. We provide the basic idea of the algorithm below. For more details, see [7].

The algorithm schedules each intruction early, i.e. finds the earliest legal basic block that an instruction could be moved to (all of the instruction's inputs must dominate this basic block). Similarly, it finds the latest legal basic block for each instruction (this block must dominate all uses of the instruction's result). Instructions such as `phi`, `branch` or `return` cannot be moved due to control dependences. Between the earliest and latest legal basic blocks, the heuristic is to place instructions in the basic block with the smallest loop depth. Global Code Motion differs from standard loop-invariant code motion techniques in that it moves instructions after, as well as before, loops.

In Figure 4(a), the compiler first transforms the code into heap SSA form and without side-effect information assumes that method `nothing()` can read and write any memory location. As a result, the compiler will be unable to move the loads of `a.x` and `a.y` outside of the loop. With side-effect information, knowing that method `nothing()` does not read or write to `a.x` or `a.y`, the loads of `a.x` and `a.y` will be moved before and after the loop respectively, resulting in the code in Figure 4(b).

(a)

```
  do {
    i = i + a.x;
    j = i + a.y;
    nothing();
  } while( i < n );
```

(b)

```
  t = a.x;
  do {
    i = i + t;
    nothing();
  } while( i < n );
  j = i + a.y;
```

Fig. 4. Loop-invariant code motion example (a) before (b) after

3.4 Using side-effect information for inlined bytecode

The side-effect attribute provides information about data dependences between instructions and refers to a bytecode by using its offset. Since the side-effect analysis is computed ahead-of-time, and thus is not aware of the JIT inlining decisions, the side-effect attribute does not have entries for inlined bytecodes. In Figure 5(a), let's assume that calls to `foo()` and `bar()` are inlined, resulting in the code in Figure 5(b). Since an inlined bytecode is associated with its original offset in the IR, it is, in general, incorrect to retrieve side-effect information for an inlined bytecode in the current method. For example, in the side-effect attribute of method `main()` in Figure 5(b), information about offset `0` is associated with bytecode `b0`, not `b1` or `b2`, which are from other methods.

To handle this case, we keep track of inlining sequences for each instruction. When comparing two bytecodes, we retrieve the least common method ancestor of the two bytecode inlining sequences, and use the side-effect information associated with that method. If a bytecode originally comes from that common method, we use its offset.

(a)

```
Offset  main() {
0 main    b0
1 main    invoke foo
        }
        foo() {
0 foo     b1
1 foo     invoke bar
        }
        bar() {
0 bar     b2
1 bar     b3
        }
```

(b)

```
Offset
        main() {
0 main    b0
0 foo     b1
0 bar     b2
1 bar     b3
        }
```

Fig. 5. Inlining example (a) before (b) after

Otherwise, we retrieve the *invoke* bytecode that it comes from in the common method, and use the offset associated with this *invoke* bytecode.

For example, in Figure 5(b), the least common method ancestor for bytecodes `b0` and `b1` is `main()`. Since `b0` originally comes from `main()`, we use its offset (i.e. 0). Since `b1` was not originally part of `main()`, we retrieve the *invoke* bytecode that it comes from in `main()`, i.e. *invoke* `foo`. We then use the offset associated with this *invoke* bytecode (i.e. `1`). Thus, when inquiring about data dependences between bytecodes `b0` and `b1`, we lookup information for offsets `0` and `1` in the side-effect attribute for method `main()`. Similarly, for bytecodes `b1` and `b2` we lookup offsets `0` and `1` in the side-effect attribute of method `foo()` (same result for `b1` and `b3`). For bytecodes `b2` and `b3`, we lookup offsets `0` and `1` in the side-effect attribute of `bar()`.

4 Experiments

4.1 Environment and benchmarks

We modified Jikes RVM version 2.3.0.1 to use side-effect information in the optimizations described in the previous section. We used the production configuration (namely FastAdaptiveCopyMS) in Jikes RVM with the JIT-only option (every method is compiled on first invocation and no recompilation occurs thereafter). We ran the SpecJVM98 [1] benchmarks (size 100) with Jikes RVM at optimization level 1 and 2 using the six side-effect variations described in section 2. A description of the benchmarks is given in Table 1. For each benchmark and at each optimization level, we show the number of memory reads per second performed (load density). This shows how important memory operations are in each benchmark. We expect the benchmarks with high load densities, compress, raytrace, mtrt and mpegaudio, to benefit most from side-effect analysis. We computed side-effect information using the development version of SOOT, revision 1621.

		Load density 1000's	
Benchmark	Description	Level 1	Level 2
compress	Lempel-Ziv compressor/uncompressor	207383	138570
jess	A Java expert shell system based on NASA's CLIPS system	56371	68353
raytrace	Ray tracer application	106271	127806
db	Performs several database functions on a memory-resident database	7140	11776
javac	JDK 1.0.2 Java compiler	21645	19208
mpegaudio	MPEG-3 audio file compression application	82137	179070
mtrt	Dual-threaded version of raytrace	92599	122821
jack	A Java parser generator with lexical analyzers (now Java CC)	14632	15240

Table 1. Benchmark description and load density property

We ran our benchmarks on two different architectures to see whether we would get similar trends in our results. The first system that we used runs Linux Debian on an Intel Pentium 4 1.80GHz CPU with 512Mb of RAM. The second one also runs Linux Debian on an dual processor AMD Athlon MP 2000+ 1.66GHz CPU with 2Gb of RAM. For our experiment, Jikes RVM was configured to run on a single processor machine.

4.2 Results

Our primary goal for this study was to see whether side-effect information could improve performance in JITs, and if so, our secondary objective was to determine the level of precision of side-effect information required. To obtain accurate answers to these questions, we measured for each run the static number of loads removed in local CSE and in the redundant load elimination optimization, and the static number of instructions moved in the loop-invariant code motion phase. These numbers provide us details on how much improvement each optimization achieves statically using side-effect information. We also measured dynamic counts of memory load operations eliminated and execution times (best of four runs, not including compilation time). The architecture-independent dynamic counts help us see whether a direct correlation exists between a reduction in memory operations performed and speedups.

It should be noted that although we used the JIT-only option in Jikes RVM where no method recompilation is expected, some optimizations such as inlining can cause invalidation and recompilation. In this case, for our static numbers, we only counted the number of static loads eliminated (in local CSE or load elimination) or instructions moved (in LICM) in the last method compilation before execution.

To examine the effect of side-effect analysis in both local and global optimizations, we ran our benchmarks using Jikes RVM at optimization level 1 and 2. For level 1, only local CSE uses side-effect information. For level 2, local CSE, redundant load elimination and loop-invariant code motion use side-effect analysis. We present in the next two sections our results for level 1 and level 2 optimizations.

Optimization level 1 Level 1 optimizations in Jikes RVM include standard optimizations such as local copy propagation, local constant propagation, local common subexpression elimination, null check elimination, type propagation, constant folding, dead code elimination, inlining, etc. Among these, only local CSE uses our side-effect analysis for eliminating *getfield* and *getstatic* instructions.

		static counts		dynamic counts		Intel		AMD	
benchmark	side-effect	getfield	getstatic	getfield	getstatic	time(s)	speedup	time(s)	speedup
compress	none	108	1	1 871 398 009	33 418 641	9.215		9.185	
	any	112 (3.7 %)	2	1 871 397 929 (0.0 %)	33 418 641	9.395	0.98x	9.184	1.00x
jess	none	229	0	209 404 162	2 326 905	4.583		3.756	
	any	245 (7.0 %)	1	209 402 840 (0.0 %)	2 326 905	4.615	0.99x	3.77	1.00x
raytrace	none	166	0	287 993 152	1 359	4.276		2.71	
	any	188 (13.3 %)	1	287 979 508 (0.0 %)	1 359	4.198	1.02x	2.662	1.02x
db	none	130	0	160 088 294	96 012	22.023		22.434	
	any	133 (2.3 %)	3	160 087 709 (0.0 %)	96 012	22.054	1.00x	22.453	1.00x
javac	none	415	0	149 595 624	4 028 976	11.047		7.097	
	any	431 (3.9 %)	1	149 407 295 (0.1 %)	4 028 946	11.215	0.99x	7.177	0.99x
mpegaudio	none	340	174	456 136 442	52 215 347	8.874		6.189	
	any	347 (2.1 %)	176	455 026 631 (0.2 %)	52 215 346	8.219	1.08x	5.85	1.06x
mtrt	none	166	0	291 501 667	2 063	4.744		3.148	
	any	188 (13.3 %)	1	291 474 379 (0.0 %)	2 063	4.727	1.00x	3.087	1.02x
jack	none	470	1	50 029 731	1 534 965	6.095		3.524	
	any	663 (41.1 %)	2	49 579 043 (0.9 %)	1 534 977	6.108	1.00x	3.509	1.00x

Table 2. Level 1 results

When running our benchmarks with Jikes RVM at optimization level 1 (which also includes all level 0 optimizations), the use of the five side-effect variations (CHA, aot-fb, aot-fs, otf-fb and otf-fs) produced identical static and dynamic counts, and similar runtimes. To avoid repeating identical results, we grouped these five side-effect variations under the name any in the side-effect column of Table 2, and the time reported is the average execution times of runs using these five side-effect variations. The values in brackets denote the percentage increase in static opportunities or the percentage decrease in dynamic counts when compared with the none side-effect variation.

Table 2 shows that using side-effect information in local CSE increased the number of static opportunities for *getfield* elimination by 2% to 41%, but only resulted in a decrease of up to 0.9% dynamically (*getstatic* instructions are almost unaffected). As a result, most benchmarks have similar execution times with or without side-effect analysis. However, the use of side-effect information produced speedups of 1.08x and 1.06x for mpegaudio on our Intel and AMD systems, and 1.02x for raytrace on both systems. Although the dynamic counts show a reduction in load instructions, we note small slowdowns for compress and jess on our Intel system, and javac on both Intel and AMD machines. These slowdowns were reproducible, and are possibly due to secondary effects such as register pressure or cache behaviour.

These results show that the simplest side-effect analysis, CHA, is sufficient for level 1 optimizations in Jikes RVM. Only local CSE uses side-effect analysis, and since it is only performed on basic blocks (typically small in Java programs), the effect is minimal.

Optimization level 2 The more advanced and expensive analyses and optimizations in Jikes RVM are level 2 optimizations. They include redundant branch elimination, heap SSA, redundant load elimination, coalescing after heap SSA, expression folding, loop-invariant code motion, global CSE, and transforming while into until loops. As described in section 3, we made use of side-effect information in the heap SSA construction, redundant load elimination, and loop-invariant code motion.

Our benchmarks were run at optimization level 2 in Jikes RVM (all level 0 and 1 optimizations are also performed), and produced identical counts and similar runtimes for the side-effect variations aot-fb, aot-fs, otf-fb and otf-fs (except for one case in compress where the static number of loads eliminated is 388 for aot-fb and aot-fs, and 389 for otf-fb and otf-fs). Thus, we grouped these four variations of side-effect analysis that are based on points-to analysis under the name PTA in Tables 3 and 4. In Table 4, the time under PTA is the average runtime of these four variations.

		redundant load elimination (RLE)			loop-invariant code motion (LICM)		
benchmark	side-effect	getfield	getstatic	aload	getfield	total HIR	total LIR
compress	none	359	4	0	87	118	29
	CHA	386 (7.5 %)	5 (25.0 %)	0	90 (3.5 %)	122 (3.4 %)	29
	PTA	388 (8.1 %)	5 (25.0 %)	0	90 (3.5 %)	122 (3.4 %)	29
jess	none	722	1	129	139	280	250
	CHA	1050 (45.4 %)	2 (100.0 %)	149 (15.5 %)	144 (3.6 %)	287 (2.5 %)	251 (0.4 %)
	PTA	1106 (53.2 %)	3 (200.0 %)	196 (51.9 %)	161 (15.8 %)	309 (10.4 %)	255 (2.0 %)
raytrace	none	342	1	32	87	184	54
	CHA	613 (79.2 %)	2 (100.0 %)	84 (162.5 %)	96 (10.3 %)	210 (14.1 %)	56 (3.7 %)
	PTA	613 (79.2 %)	2 (100.0 %)	127 (296.9 %)	96 (10.3 %)	210 (14.1 %)	56 (3.7 %)
db	none	243	1	2	61	88	31
	CHA	274 (12.8 %)	4 (300.0 %)	2	64 (4.9 %)	92 (4.6 %)	32 (3.2 %)
	PTA	274 (12.8 %)	4 (300.0 %)	3 (50.0 %)	64 (4.9 %)	92 (4.6 %)	32 (3.2 %)
javac	none	1519	26	90	44	116	479
	CHA	1842 (21.3 %)	30 (15.4 %)	101 (12.2 %)	48 (9.1 %)	121 (4.3 %)	479
	PTA	1847 (21.6 %)	30 (15.4 %)	108 (20.0 %)	48 (9.1 %)	121 (4.3 %)	479
mpegaudio	none	706	212	367	128	299	98
	CHA	804 (13.9 %)	216 (1.9 %)	370 (0.8 %)	152 (18.8 %)	327 (9.4 %)	102 (4.1 %)
	PTA	804 (13.9 %)	216 (1.9 %)	426 (16.1 %)	152 (18.8 %)	327 (9.4 %)	102 (4.1 %)
mtrt	none	342	1	32	87	184	55
	CHA	613 (79.2 %)	2 (100.0 %)	84 (162.5 %)	96 (10.3 %)	210 (14.1 %)	57 (3.6 %)
	PTA	613 (79.2 %)	2 (100.0 %)	127 (296.9 %)	96 (10.3 %)	210 (14.1 %)	57 (3.6 %)
jack	none	678	2	69	23	39	58
	CHA	999 (47.4 %)	16 (700.0 %)	69	23	39	58
	PTA	999 (47.4 %)	16 (700.0 %)	69	23	39	58

Table 3. Level 2 static results

The first part of Table 3 shows that using side-effect information in RLE increased static opportunities for *getfield* removal by 8% to 79%. There were very few improvements for removing *getstatic* instructions, but the increase was large for *aload* (array load) instructions for some benchmarks (jess, raytrace, mpegaudio and mtrt). For raytrace and mtrt, the total load increase when combining these three bytecode instructions is 98%. Interestingly, PTA improved over CHA for all benchmarks except jack.

The second part of Table 3 shows static counts of instructions moved during LICM. The last two columns are the total instructions moved when LICM is performed on high-level (HIR) and low-level (LIR) intermediate representation in Jikes RVM. Note that memory operations are not moved during LICM on LIR; interestingly, the use of side-effect information in HIR optimizations enabled some other transformations that allowed some instructions to be moved during LICM on LIR. We see that side-effect analysis enabled an increase in the number of moved *getfield* by up to 19%, and the total instructions during HIR by up to 14%. For only one benchmark (jess), using PTA side-effect information allowed more instructions to be moved than CHA. There were no *putstatic*, *aload* or *astore* instructions moved, and only one additional *putfield* moved for javac (not shown). Note that since RLE is performed before LICM, improved side-effect information can cause loads that would have been moved in LICM to be removed in RLE. Therefore, to measure the impact of side-effect information on LICM, we disabled RLE when collecting the static LICM counts. We do not show static counts for local CSE, which are minimal because redundant load elimination is performed before local CSE.

In the first part of Table 4, we see that side-effect analysis enabled a reduction in dynamic *getfield* operations by up to 27%, but only reduced *getstatic* and *aload* instructions by up to 3%. For compress and jess, using PTA side-effect information allowed a larger reduction of *getfield* than CHA. For mpegaudio, it improved the removal of *aload* instructions. The second part of the table shows speedups achieved for compress, ray-

		dynamic counts			Intel		AMD	
benchmark	side-effect	getfield	getstatic	aload	time(s)	speedup	time(s)	speedup
compress	none	836 681 238	29 585 886	450 569 851	10.423		9.503	
	CHA	713 879 612 (14.7%)	29 585 886	450 569 851	9.635	1.08x	9.316	1.02x
	PTA	694 156 483 (17.0%)	29 585 886	450 569 851	9.386	1.11x	9.03	1.05x
jess	none	193 400 124	2 326 905	74 199 530	4.889		3.949	
	CHA	177 280 681 (8.3%)	2 326 905	74 197 591 (0.0%)	4.945	0.99x	3.962	1.00x
	PTA	141 340 271 (26.9%)	2 326 572 (0.0%)	74 188 965 (0.0%)	4.872	1.00x	4.002	0.99x
raytrace	none	278 990 954	1 359	70 558 731	4.38		2.735	
	CHA	217 369 769 (22.1%)	1 359	70 189 162 (0.5%)	3.93	1.11x	2.607	1.05x
	PTA	217 369 769 (22.1%)	1 359	70 125 938 (0.6%)	3.905	1.12x	2.615	1.05x
db	none	160 085 986	96 012	113 165 950	22.625		23.212	
	CHA	154 814 883 (3.3%)	96 012	113 165 950	22.605	1.00x	23.222	1.00x
	PTA	154 814 883 (3.3%)	96 012	113 165 950	22.471	1.01x	23.141	1.00x
javac	none	129 704 466	3 728 755	3 947 221	10.962		7.154	
	CHA	123 962 720 (4.4%)	3 726 381 (0.1%)	3 947 158 (0.0%)	11.138	0.98x	7.21	0.99x
	PTA	123 962 933 (4.4%)	3 726 306 (0.1%)	3 947 133 (0.0%)	11.142	0.98x	7.231	0.99x
mpegaudio	none	258 084 245	16 092 989	796 126 083	9.319		5.977	
	CHA	254 421 559 (1.4%)	16 075 411 (0.1%)	794 492 856 (0.2%)	8.41	1.11x	5.175	1.15x
	PTA	254 421 559 (1.4%)	16 075 411 (0.1%)	773 557 981 (2.8%)	7.932	1.17x	4.987	1.20x
mtrt	none	282 145 314	2 063	71 578 275	4.681		2.88	
	CHA	220 136 202 (22.0%)	2 063	71 124 467 (0.6%)	4.201	1.11x	2.788	1.03x
	PTA	220 136 202 (22.0%)	2 063	70 998 019 (0.8%)	4.208	1.11x	2.796	1.03x
jack	none	46 154 208	1 534 965	5 727 775	6.097		3.505	
	CHA	42 805 654 (7.3%)	1 530 924 (0.3%)	5 727 775	6.122	1.00x	3.47	1.01x
	PTA	42 805 654 (7.3%)	1 530 924 (0.3%)	5 727 775	6.101	1.00x	3.51	1.00x

Table 4. Level 2 dynamic results

trace, mtrt and mpegaudio. The speedups vary from 1.08x to 1.17x on our Intel system, and from 1.02x to 1.20x on AMD. On both systems, mpegaudio has the largest speedup. These benchmarks are the ones with the highest load densities (Table 1), and the ones that we expected would benefit the most from side-effect information.

A higher level of precision of side-effect information made a difference in performance for compress and mpegaudio. Using PTA side-effect analysis vs CHA increased the speedup of compress from 1.08x to 1.11x on Intel, and 1.02x to 1.05x on AMD. For mpegaudio, it went from 1.11x to 1.17x on Intel and from 1.15x to 1.20x on AMD.

These results show that using side-effect analysis in global optimizations improved opportunities for load elimination and moving instructions, reduced dynamic load operations, and improved performance in runtimes. Benchmarks with higher load densities benefited most from side-effect information. The results also show that points-to analysis improves side-effect information compared to only using CHA, but that the differences between points-to analysis variations are negligible.

5 Related Work

Early side-effect analyses for languages with pointers by Choi *et al.* [4] and Landi *et al.* [14] made use of may-alias analysis to distinguish reads and writes to locations known to be different. These analyses were mainly targeted at analysis of C, so the call graph was assumed to be mostly static. Therefore, in comparison with our work, in that setting, the information about pointers was most important, while the call graph was much easier to compute.

In contrast, Clausen's [6] side-effect analysis for Java was based on a call graph constructed with a CHA-like analysis, but it did not use any pointer information. This analysis computed read and write information for each field, ignoring which specific object contained the field read or written. In comparison with our work, Clausen's analysis is most similar to our CHA-based side-effect analysis. Clausen applies his analysis results in an ahead-of-time early Java bytecode optimizer to a similar set of optimizations as we do: dead code removal, loop invariant removal, constant propagation, and common subexpression elimination.

When evaluating the precision of points-to analyses, it is common to report the size of the points-to sets at field read and write instructions, as in [18, 25]. Rountev and Ryder [26] evaluate their points-to analysis for precompiled libraries in this way. Other points-to analysis work [13, 19, 27, 28] takes this evaluation one step further, by also computing read and write sets summarizing the effects of entire methods, rather than just individual statements, and propagating this information along the call graph. This is similar to the read and write set computation we mention in Section 2.3. In general, these studies conclude that differences in precision of the underlying analyses do have a significant effect on the static precision of side-effect information.

Chowdhury *et al.* [5] study the effect of alias analysis precision on the number of optimization opportunities for a range of scalar optimizations. However, they only measure the static number of optimizations performed (rather than their run-time effect), and their benchmarks are mostly pointer-free C programs, some translated directly from FORTRAN, so they find, unsurprisingly, that alias analysis precision has little effect.

Studies measuring the actual run-time impact of code optimized using side-effect information are surprisingly rare. Ghiya *et al.* [11, 12] measure the effectiveness of side-effect information on the run-time efficiency of code produced by an optimizing compiler for C. Diwan *et al.* [9] study redundant load elimination in Modula-3, using declared types to conservatively approximate aliasing relationships, and method read/write set summaries. The results of Diwan *et al.* on Modula-3 and Ghiya *et al.* on C are comparable to ours on Java. In particular, all three studies show that significant run-time improvements are possible, and that even simple, imprecise alias information enables many of the improvements. Razafimahefa [24] performs loop invariant code motion using side-effect information on Java in an ahead-of-time bytecode optimizer, and reports run-time speedups comparable with ours on an early-generation Java VM.

Pechtchanski and Sarkar [20] present a preliminary study of a framework which allows programmers to provide annotations indicating absence of side-effects. Like our side-effect information, these annotations are communicated to Jikes RVM in class file attributes and used to improve optimizations. Only limited, preliminary, empirical results of the effect of these annotations are provided, and verification of the correctness of the programmer-provided annotations has yet to be done.

In summary, existing work on other languages largely agrees with our findings on Java. Some side-effect information is useful for real run-time improvements from compiler optimizations. Although precision of the underlying analyses tends to have large effects on static counts of optimization opportunities, the effects on dynamic behaviour are much smaller; even simple analyses provide most of the improvement. Distinctions of our work from previous work are that we provide a study of run-time effects of side-effect information on Java, and that we show how to communicate analysis results from an off-line analyzer to a JIT.

6 Conclusion

In this study, we showed that side-effect analysis does improve performance in just-in-time (JIT) compilers, and that relatively simple analyses are sufficient for significant improvements. On level 1 optimizations, side-effect analyses had little impact on performance, except for one benchmark. On level 2 optimizations, however, our results showed an increase of up to 98% of static opportunities for load removal, a reduction of up to 27% of the dynamic fields reads, and execution time speedups ranging from 1.08x to 1.20x. As we expected, using side-effect analysis had the largest impact on the benchmarks with high load densities.

The feasibility of performing side-effect analysis inside the JIT is a topic for future research. The dynamic call graph construction presented in [22, 23] is a first step in this work.

References

1. SPEC JVM98 benchmarks. `http://www.spec.org/osg/jvm98/`.
2. B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Syst. J.*, 39(1):211–238, 2000.
3. B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of POPL 1988*, pages 1–11, 1988.
4. J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of POPL 1993*, pages 232–245, 1993.

5. R. A. Chowdhury, P. Djeu, B. Cahoon, J. H. Burrill, and K. S. McKinley. The limits of alias analysis for scalar optimizations. In *CC 2004*, volume 2985 of *LNCS*, pages 24–38, 2004.
6. L. R. Clausen. A Java bytecode optimizer using side-effect analysis. *Concurrency: Practice and Experience*, 9(11):1031–1045, Nov. 1997.
7. C. Click. Global code motion/global value numbering. In *Proceedings of PLDI 1995*, pages 246–257, 1995.
8. J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP 95*, volume 952 of *LNCS*, pages 77–101, 1995.
9. A. Diwan, K. S. McKinley, and J. E. B. Moss. Type-based alias analysis. In *Proceedings of PLDI 1998*, pages 106–117, 1998.
10. S. J. Fink, K. Knobe, and V. Sarkar. Unified analysis of array and object references in strongly typed languages. In *Static Analysis Symposium*, pages 155–174, 2000.
11. R. Ghiya and L. J. Hendren. Putting pointer analysis to work. In *Proceedings of POPL 1998*, pages 121–133, 1998.
12. R. Ghiya, D. Lavery, and D. Sehr. On the importance of points-to analysis and other memory disambiguation methods for C programs. In *Proceedings of PLDI 2001*, pages 47–58, 2001.
13. M. Hind and A. Pioli. Which pointer analysis should I use? In *Proceedings of ISSTA 2000*, pages 113–123, 2000.
14. W. Landi, B. G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of PLDI 1993*, pages 56–67, 1993.
15. O. Lhoták. Spark: A flexible points-to analysis framework for Java. Master's thesis, McGill University, Dec. 2002.
16. O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *CC 2003*, volume 2622 of *LNCS*, pages 153–169, Warsaw, Poland, 2003.
17. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.
18. A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of ISSTA 2002*, pages 1–11, 2002.
19. G. Olivar. Fast points-to and side-effect analysis for the McCAT C compiler. M.Sc. project, McGill University, `http://citeseer.ist.psu.edu/350797.html`, Apr. 1997.
20. I. Pechtchanski and V. Sarkar. Immutability specification and its applications. In *Proceedings of the 2002 Joint ACM-ISCOPE Conference on Java Grande*, pages 202–211, 2002.
21. P. Pominville, F. Qian, R. Vallée-Rai, L. Hendren, and C. Verbrugge. A framework for optimizing Java using attributes. In *CC 2001*, volume 2027 of *LNCS*, pages 334–354, 2001.
22. F. Qian and L. Hendren. A study of type analysis for speculative method inlining in a JIT environment. In *CC 2005*, LNCS, Edinburgh, Scotland, April 2005. Springer.
23. F. Qian and L. J. Hendren. Towards dynamic interprocedural analysis in jvms. In *Virtual Machine Research and Technology Symposium*, pages 139–150, 2004.
24. C. Razafimahefa. A study of side-effect analyses for Java. Master's thesis, McGill University, Dec. 1999.
25. A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for Java using annotated constraints. In *Proceedings of OOPSLA '01*, pages 43–55, 2001.
26. A. Rountev and B. G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. In *CC 2001*, volume 2027 of *LNCS*, pages 20–36, 2001.
27. B. G. Ryder, W. A. Landi, P. A. Stocks, S. Zhang, and R. Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM Transactions on Programming Languages and Systems*, 23(2):105–186, Mar. 2001.
28. P. A. Stocks, B. G. Ryder, W. A. Landi, and S. Zhang. Comparing flow and context sensitivity on the modification-side-effects problem. In *Proceedings of ISSTA 1998*, pages 21–31, 1998.
29. R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: is it feasible? In *CC 2000*, volume 1781 of *LNCS*, pages 18–34, 2000.