Side-Effect Analysis in Java Classfile Attributes
308-621 Project

Ondřej Lhoták

April 16, 2002

Abstract

Many compiler optimizations require side-effect information, especially when applied to object-oriented languages. Just-in-time compilers perform many optimizations, often making optimizations on bytecode redundant or even detrimental. However, precise side-effect information can be computed before run-time, and communicated to the just-in-time compiler in class file attributes. This is especially useful because precise side-effect analyses are too expensive to be included in just-in-time compilers.

This report presents and evaluates the implementation of a side-effect analysis for Java, and a representation of side-effect information as annotations in class file attributes. These are designed to be practical in terms of analysis speed, memory requirements, and size of the annotation, and the evaluation shows that these design goals are satisfied. The analysis is measured to be significantly more precise than the simple side-effect heuristic used by just-in-time compilers, in that it rules out up to 83.2% of the dependences found between instructions.

1 Introduction and Motivation

The just-in-time compilers found in commercial Java virtual machines being written today perform increasingly sophisticated optimizations. As a result, the native code that they produce is increasingly different from the bytecode that they take as input. This makes many of the low-level optimizations performed on bytecode redundant, because the just-in-time compiler performs them a second time, or perhaps undoes them. However, Soot remains useful for performing high-level optimizations, which are unlikely to be affected by the translation to native code, and for whole-program analyses that are too expensive for a just-in-time compiler to perform at run-time. A just-in-time compiler can make use of the pre-computed results of such an analysis encoded in class file attributes.

Side-effect analysis is an example of an expensive whole-program analysis that can aid the just-in-time compiler to produce more aggressively optimized native code. The purpose of this

Figure 1: Code example

```
foo() {
    this.x = 0;
    for( int i = 0; i < 1000000; i++ ) {
        this.bar();
        this.y[i] = this.x;
    }
}
```

analysis is to estimate the sets of run-time objects which each instruction and each method of the program may affect. Having such an estimate may allow a compiler to perform common-subexpression-elimination-like optimizations of loads and stores in the presence of method calls. As an example, consider the example code in figure 1. If we knew that `bar()` does not read or write `this.x`, then we could move the load of `this.x` out of the loop. Furthermore, if we knew that `bar()` performs no writes or native method calls, we could move the call out of the loop, or remove it entirely.

This project is an implementation of a side-effect analysis whose result is encoded in Java class file attributes, where it can be read by a just-in-time compiler. The implementation can analyze and annotate medium-sized benchmark programs in about half an hour using under 800 MB of memory, and the resulting annotation is compact enough to be included in the class files; in most cases, the annotation itself is much smaller than the bytecode, and it could easily be made smaller still. The interface between the points-to analysis and side-effect analysis is clearly defined, to enable experimentation with variations in the points-to analysis.

2 Background and Related Work

A large amount of research has been published on side-effect analysis, as well as on the alias or points-to analysis that it requires, and their many other uses; a long list of references can be found at `http://www.math.tau.ac.il/~guy/pa/pointer-analysis.html`. In this project report, I list only those publications specifically relevant to this project. The flow-insensitive and context-insensitive points-to analysis that I use to compute side-effect information is based on Andersen's points-to analysis for C [1], modified for Java. The resulting analysis is very similar to, but more precise than, Variable Type Analysis [9]. Liang, Pennings and Harrold [6] also discuss extending Andersen's analysis for Java, but they do not use it to compute side-effect information. Clausen describes a clear side-effect analysis for Java using very simple points-to information [3]. Clausen's side-effect analysis is similar to the one used in this project, but uses a less accurate points-to analysis. Rountev and Ryder [8] give a points-to and side-effect analysis which can analyze parts of a program, and merge the results together; this may be useful to split the analysis results between the various class files comprising the program.

Recently, several papers have been published about communicating results of static analyses to Java virtual machines. The Sable group describes a framework for embedding attributes in bytecode [7], based on Soot [10], and they give null check and array bounds check elimination as example analyses. Krintz and Calder [4] use annotations to communicate analysis results and profiling information to Intel's ORP JIT. Azevedo, Nicolau and Hummel [2] also use attributes to communicate pre-computed analysis results to a just-in-time compiler.

The side-effect analysis described in this report is part of SPARK: the Soot Pointer Analysis Research Kit [5]. Please see the SPARK poster for a view of the big picture.

3 Specific Problem Statement

The purpose of this project is to design a representation of side-effect information in attributes, to implement a module within Soot to construct this representation from a variety of points-to analyses, and to link this module with some points-to analyses already implemented. The implementation should be evaluated in terms of analysis speed, the size of the resulting annotation, and the precision of the information computed.

4 Solution Strategy and Implementation

This project consists of the following tasks:

1. Design a representation of side-effect information in class file attributes. The specification of this representation is given in the appendix.
2. Implement a module within Soot which interfaces to a points-to analysis, produces side-effect information from it, and encodes the information in class file attributes.
3. Evaluate the efficiency of the implementation, and measure the approximate improvement in the precision of side-effect information. These results are presented later in this report.

I performed the above three steps iteratively until my implementation satisfied the following design goals:

1. The annotations themselves should be reasonable in size. Specifically, they should not be larger than the original class file.

2. The annotations should be computable in 800 MB of memory for small to medium-sized benchmarks, such as the SPECjvm benchmarks, or the benchmarks developed by students in this course.

3. The annotations should be computable in a reasonable time. I would like to be able to analyze small benchmarks in about five minutes, and medium-sized benchmarks in about half an hour.

4.1 Representation details

The information to be encoded takes the form of dependences between instructions. For example, a client might want to know whether a write to `p.f` in one instruction may overwrite the value written to `q.f` in another instruction.

It is difficult to encode an abstract location, such as `p.f`, because the local variable `p` may appear in the bytecode as an unlabeled stack location. Rather than try to encode variables in some way, I decided instead to encode the dependences between instructions. For example, a write to `p.f` overwriting the value written to `q.f` would be encoded as a Write-Write dependence between the two bytecode instructions writing `p.f` and `q.f`. A client reading the annotation can convert this dependence into whatever internal representation it has for `p.f` and `q.f`. For each pair of statements, the annotation specifies whether there is a Write-Write, Write-Read, Read-Write, or Read-Read dependence between them. Although the Read-Read dependences may not be useful to a just-in-time compiler, I included them for completeness; they could be removed.

The size of this representation grows quadratically as the number of inter-dependent instructions in the method being analyzed. Most methods are short, and even longer methods tend to have few instructions that are inter-dependent. However, some methods are like the constructor of `spec.io.TableOfExistingFiles`, contained in the harness of all the SPECjvm benchmarks. This method consists of 633 calls to the `put` method of `java.util.Hashtable`. Since all of these calls read and write the same locations, they should all have dependences between them encoded, leading to 400689 dependences of each type. Furthermore, the methods called from each of these call sites possibly call a large number of other methods, so the call sites take a long time and a large amount of memory to analyze.

To limit the growth of the annotation size and amount of computation required, I devised a way to reduce the size of the set of dependences as it is being computed. Each instruction is assigned a pair of numbers, representing the locations that the instruction can read and write. Dependences are then computed between these numbered locations, rather than the instructions themselves. The simplest such assignment of numbered locations would assign distinct locations to each instruction, and the resulting dependence graph would be as large as the dependence graph between instructions. However, some sets of instructions can easily be determined to read or write the same locations, and can therefore share the same numbered locations, reducing the effective number of instructions to be considered. Specifically, all method calls with equal sets of possible transitive targets share read and write locations. Also, all field reference expressions having the same base pointer and the same field share the same location. This reduces the 633 method calls in `spec.io.TableOfExistingFiles` to a single pair of numbered locations, drastically reducing the size of the annotation and the time and memory needed to compute it. However, this approach makes it slightly more difficult for the client to extract the information. In order to determine whether there is a dependence between two instructions, it must look up the numbered locations read and written by the instructions, and then look in the graph for dependences between these locations.

In addition to the relationships between the locations read and written by statements, the side-effect annotation encodes, for each call site, whether a native method may be called from the call site, or transitively from any methods that may be called from it. This information may be useful to clients of the side-effect analysis, and it is trivial to compute while computing the side-effect information.

4.2 Implementation details

A detailed discussion of the points-to analysis used is beyond the scope of this report. For all experimental results presented, the points-to analysis has the following properties:

- Flow-insensitive
- Context-insensitive
- Subset-based (Andersen)
- Object instances distinguished when considering field and array references
- Declared types and casts respected
- Initial call graph created using class hierarchy analysis
- Variables split into UD-DU webs

The points-to analysis produces, for each local variable of pointer type, an abstract set of the possible locations to which it could point. From this information, the side-effect analysis computes abstract sets of locations read and written by each instruction. These locations include instance fields, static fields, and array elements. The abstract sets for each instruction can be combined into larger abstract sets for each method. These sets contain all locations accessed within the method, but not those accessed in other methods that it may call. Finally, the sets for each method can be combined into even larger sets that encode, for each call site, the set of locations accessed in all the methods possibly called from the call site, and other methods transitively called from them. This yields sets for all instructions, including invoke instructions, that can be used to determine whether dependences exist between them.

A naive implementation of this recursive definition of read and write sets would be hopelessly slow, because many call sites have large numbers of transitive targets, and the sets for each target would have to be recomputed at each call site. On the other hand, I found that an implementation that memoizes all read and write sets requires more than 800 MB of memory to analyze even medium-sized programs. I chose a compromise between memory requirements and running time. For the experiments that I am presenting in this report, I cached the abstract set accessed by each statement and method, but not the abstract set accessed by each call site.

5 Experimental Framework

I used the implementation to produce side-effect information in attributes for a set of twenty benchmarks, in order to measure how well the design satisfies the design goals.

Without a just-in-time compiler to use the side-effect information in attributes, this project has little experimental value on its own. However, I evaluated the relative precision of the side-effect information produced by counting the number of dependences found. This number can be compared to the number of dependences found by a simpler analysis, such as the one in the `NaiveSideEffectTester` in Soot. This conservative analysis considers accesses of the same field to always be potentially aliased, regardless of the base pointer, and it considers method calls to possibly read and write to all locations. A comparison of the number of dependences alone has limited value, because it tells us nothing about the importance of the individual dependences found.

Eventually, I will modify a just-in-time compiler to use the side-effect information to perform optimizations. I will then be able to measure actual speed improvements caused by optimizations enabled by the side-effect information. Also, I will be able to study the effects of details of the points-to analysis and side-effect analysis on the usefulness of the side-effect information.

5.1 Benchmark programs

To evaluate my implementation, I used the benchmarks from the SPECjvm suite and the benchmarks created by students in this course. Most of the benchmarks are either numerical, using many arrays, or object-oriented. The benchmarks range from small (about 1000 lines of code) to medium-sized (about 15000 lines of code). The largest is flBench, Florian Loitsch's compiler from 308-520, with over 100000 lines. The SPECjvm benchmarks are also quite large, because they share the large harness that must be analyzed. Table 1 gives the size of each benchmark, in lines of source code, and in bytes of bytecode. I did not have the complete source of the SPECjvm benchmarks, so I cannot report the exact number of lines. Note that

Table 1: Sizes of benchmarks

Benchmark	Bytes (bytecode)	Lines (source)
BSCT20	16544	2378
DFT	25671	1519
Froggy	8709	386
GivensQR	2397	230
ICF	4720	315
Integrate	4386	362
LUAckermann	4893	417
MersennePrime	4687	364
PointsToGraph	8921	357
Puzzle	9535	529
coefficients	13346	969
compress	69139	?
db	66891	?
flBench	827088	112584
jack	166004	?
javac	540286	?
jess	299272	?
mpegaudio	167410	?
mtrt	104570	?
ramconfig	3003	162
raytrace	104543	?

these sizes do not include the large Java library, which must be analyzed, but for which side-effect information is not produced.

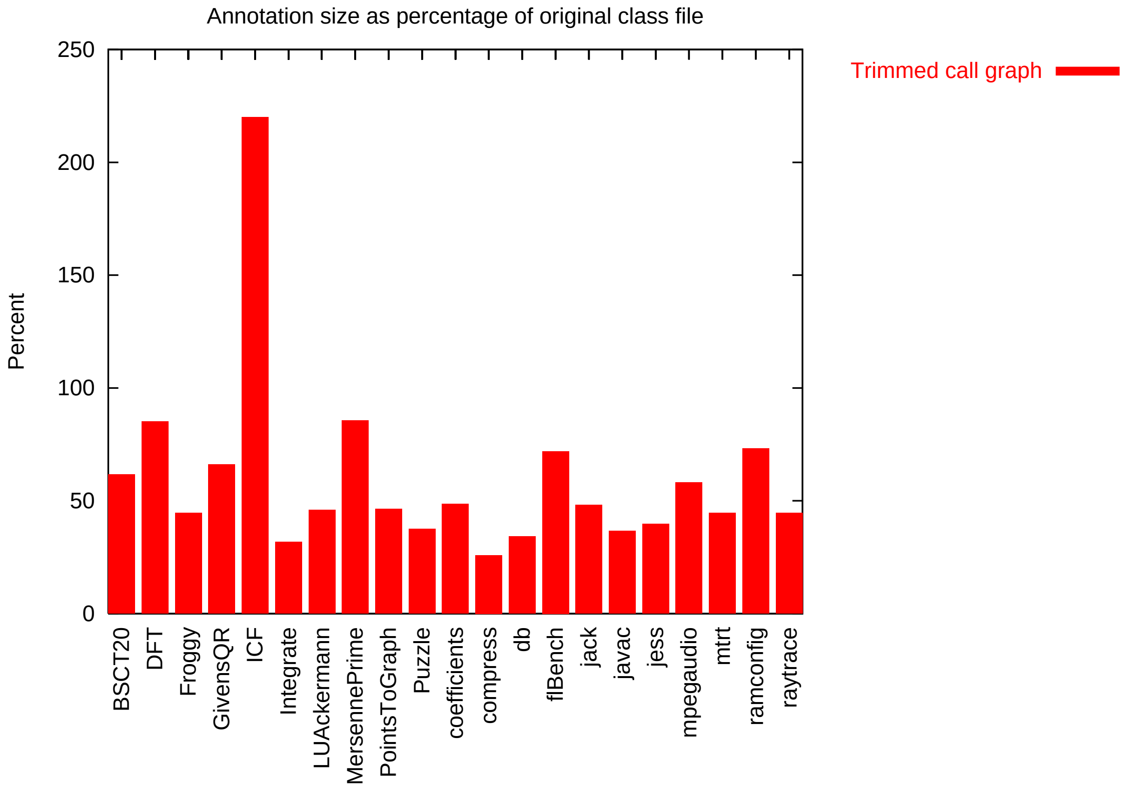
6 Results

On the benchmarks used, the implemented analysis satisfies the stated design goals. It is able to produce a reasonably-sized side-effect annotation in five minutes for small and half an hour for medium-sized benchmarks. All benchmarks can be successfully analyzed in the available 800 MB of memory. The analysis result rules out up to 83.2% of the dependences that would be found by a fully conservative side-effect analysis commonly used in just-in-time compilers.

6.1 Detailed Results

I studied extensively the annotations produced by the side-effect analysis. In the process, I discovered and corrected several bugs in the handling of multi-dimensional arrays in the

Figure 2: Annotation size



underlying points-to analysis. Because the results of compiler analyses are large and difficult to verify, I cannot guarantee that my implementation contains no other bugs that may affect the results that I am presenting.

Figure 2 shows the size of the annotation encoded in attributes as a percentage of the size of the original class files. The size of the original class files is measured after they have been processed by Soot with no optimization or annotation, which typically makes them slightly smaller. The side-effect information encoded for this measurement is the most precise information produced by this analysis: it is produced by using the result of a precise points-to analysis to first trim the call graph, and then as the basis of a precise side-effect analysis.

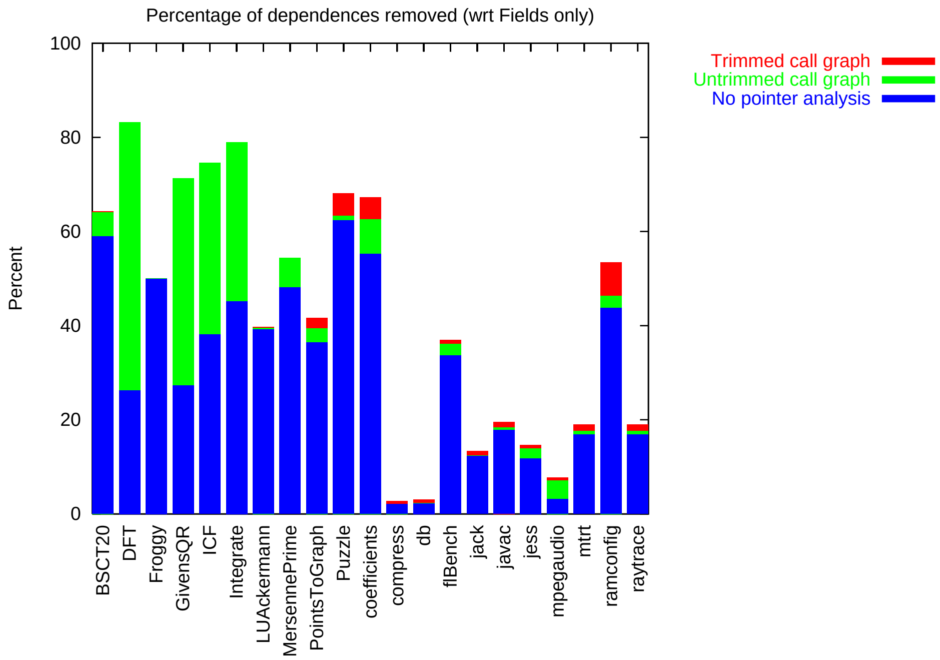
The annotation size is about half the size of the original bytecode, and larger than the bytecode only on the ICF benchmark. The relative annotation size is slightly larger on the small benchmarks from this course, because they tend to contain more of their code in a small number of large methods than the large SPECjvm benchmarks. Numerical benchmarks, such as DFT, ICF, and MersennePrime have larger annotation sizes, because array operations use many instructions with possible side-effects. The annotation for the ICF benchmark is 220% of the size of the bytecode because this benchmark does many operations on two arrays in one very large main method. Because the method operates on only two arrays, almost all the instructions have dependences between each other.

Table 2: Analysis time

Benchmark	Total analysis time (minutes)				
	Soot only	Fields only	No pointer analysis	Untrimmed call graph	Trimmed call graph
BSCT20	0.4	3.0	3.1	4.1	4.2
DFT	0.4	3.2	3.3	4.7	4.7
Froggy	0.2	2.2	2.3	3.0	3.1
GivensQR	0.1	2.0	2.0	2.7	2.8
ICF	0.2	2.3	2.4	3.4	3.5
Integrate	0.2	2.4	2.3	3.2	3.2
LUAckermann	0.2	2.2	2.2	3.0	3.1
MersennePrime	0.1	2.0	2.0	2.7	2.8
PointsToGraph	0.2	2.5	2.5	3.5	3.5
Puzzle	0.2	2.4	2.4	3.2	3.2
coefficients	0.2	2.3	2.3	3.1	3.2
compress	1.0	9.8	10.0	19.5	19.4
db	0.8	8.8	10.1	18.2	17.8
flBench	7.0	33.0	115.4	130.0	114.8
jack	4.3	17.0	21.4	39.8	36.2
javac	6.3	36.0	40.0	84.0	76.5
jess	4.9	27.2	33.9	72.3	65.1
mpegaudio	2.8	15.9	16.0	33.8	33.1
mtrt	1.4	11.5	12.0	24.6	24.0
ramconfig	0.1	2.0	2.0	2.8	2.8
raytrace	1.4	11.5	12.0	24.3	24.3

Table 2 shows the time taken to analyze each benchmark. These times were measured on a 750 MHz Sun UltraSPARC-III with 1 GB of memory, using the Sun HotSpot Client VM, version 1.3.1-b24. Each time given is the complete running time of Soot, including (as applicable) reading in the class files, converting to Jimple, building a call graph using Class Hierarchy Analysis, performing a points-to analysis, performing a side-effect analysis, encoding side-effect information in attributes, emitting jasmin files, and assembling these into class files. The first column lists the name of each benchmark. The second column lists the time taken to read the benchmark into Soot, and immediately write it back to class files with no optimization or annotation. Unlike the remaining columns, this time does not including reading the standard Java libraries and constructing a call graph. The third column lists the time taken to annotate the benchmark with fully conservative side-effect information (assuming all pointers to be potentially aliased, and all methods to potentially write to all locations). This information takes almost no time to compute, so the time is almost completely spent reading and writing the class files, and encoding the side-effect information in attributes. The fourth column lists the time taken to annotate the benchmark using a precise side-effect analysis, but assuming that

Figure 3: Number of dependences removed



all pointers may be aliased (and therefore requiring no points-to analysis). The fifth column lists the time taken to annotate the benchmark using a precise side-effect analysis based on a precise points-to analysis. Finally, the sixth column lists the time taken to annotate the benchmark using the same analyses as in the fifth column, but trimming the call graph based on the points-to information before starting the side-effect analysis.

The small benchmarks are fully analyzed and annotated in under five minutes each, as required. The SPECjvm benchmarks take about half an hour each, except for the two largest ones, javac and jess, which take over an hour each. The side-effect analysis for flBench takes close to two hours. While these times are acceptable for experimentation with side-effect analysis, some improvement would be welcome. For most of the benchmarks, the time in the Fields only column is more than half the time in the Trimmed call graph column, suggesting that a significant part of the time is the overhead of Soot, especially the reading and writing of the class files. However, especially for the large benchmarks, the time for the side-effect analysis is also significant, showing that the analysis itself needs to be tuned for performance. In the non-trivial benchmarks, the time used to trim the call graph is more than recovered in the resulting speedup of the side-effect analysis. The points-to analysis appears expensive for some of the benchmarks, but this is misleading: the longest points-to analysis (for javac) takes five minutes and two seconds. Most of the difference between the No pointer analysis and Untrimmed call graph columns is due to the side-effect analysis taking longer when precise points-to information is available.

Figure 3 shows the number of dependences ruled out by the analysis as a fraction of the total number of dependences that would be found using a fully conservative analysis (which considers all pointers to be potentially aliased, and all method calls to potentially write to all locations). Write-Write, Write-Read, Read-Write and Read-Read dependences are all grouped together in this graph, but I found that there were always approximately equal numbers of each type. The dark, bottom-most part of each bar represents the fraction of dependences ruled out by a side-effect analysis that takes into account fields accessed by each method, but that considers all pointers to be possibly aliased (and therefore needs no points-to analysis). The lighter part of each bar represents the additional dependences ruled out when the side-effect analysis uses precise points-to information. The slightly darker part at the top of each bar represents additional dependences ruled out if (prior to the side-effect analysis), the call graph is trimmed based on the result of the points-to analysis.

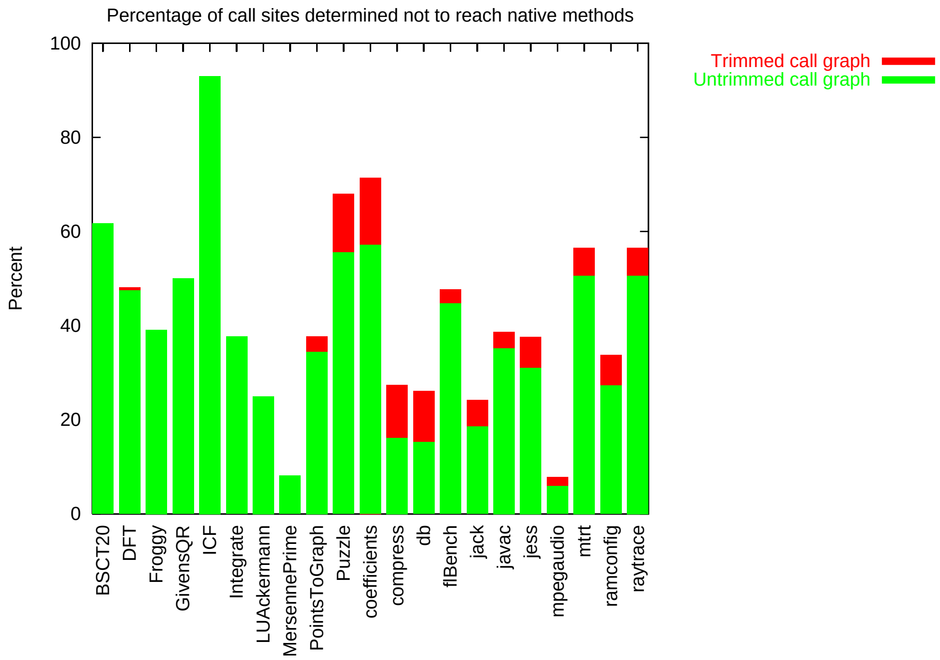
Note that the number of dependences ruled out tells us very little about how important the dependences were that were ruled out. Having said that, we can see that the analysis does rule out many dependences (up to 83.2% in DFT), especially in the smaller benchmarks. Fewer dependences are ruled out in the large benchmarks. This is probably because they contain large amounts of code in methods that are far from being leaf methods, in that they call many other methods, and this code is likely to have actual dependences due to some of the methods that may be called. On the other hand, in the smaller benchmarks, almost all methods are leaf methods, and therefore not likely to have many side effects. The precise computation of interprocedural side-effects rules out many more dependences in the large benchmarks than the precise points-to analysis. This is to be expected, since the large benchmarks contain many methods. In spite of this, I believe that the precision of the points-to information rules out important dependences, and that it will be found to have a significant effect on the usefulness of the side-effect information.

Figure 4 shows the fraction of call sites determined by the side-effect analysis not to reach native methods. That is, neither the target of the call, nor any methods called transitively from the target may be native. For each measurement, precise points-to and side-effect analyses were performed. The lighter part of each bar represents the fraction of call sites determined not to reach native methods when the call graph was not trimmed using the result of the points-to analysis. The darker part at the top of each bar represents the additional fraction of call sites determined not to reach native methods when the call graph was trimmed before starting the side-effect analysis.

7 Future Work

Since the purpose of this project was to build infrastructure for experiments, the obvious next step is to perform those experiments. In particular, I would like to modify a just-in-time compiler to make use of the side-effect information that this analysis produces. I could then observe how much it helps the just-in-time compiler to produce efficient code. I also plan to experiment with variations in the points-to analyses used by the side-effect analysis.

Figure 4: Call sites not reaching native methods



The results of the side-effect analysis should be made available to other Soot analyses, not just encoded in attributes. This requires implementing the `SideEffectTester` interface. This should be a small job, but it is one that I have been putting off while finishing this report.

The pointer analysis framework, including the side-effect analysis, will always benefit from additional testing. In order to facilitate this, it would be nice to design a way to browse the pointer assignment graph, points-to analysis result, and side-effect information. These results are sometimes so large that they are difficult to browse if dumped to a single large file. They could perhaps be encoded in a group of HTML files with links between them.

Finally, it may become necessary to improve either the running time or annotation size of the side-effect analysis further. The annotation result could be made smaller in several ways. First, its encoding in the class file could be made more efficient. The current encoding uses the same seven byte record format for every instruction that could possibly have a side effect. However, some instructions obviously read but don't write, or vice versa, and only invoke instructions have any possibility of causing native methods to be executed. A different, shorter record could be used for these instructions. Also, each record includes two bytes for the bytecode offset. If the records were sorted by their bytecode offset, this field would not need to be encoded, because the client could reconstruct it using the bytecode. These simple optimizations would reduce the annotation size by about 50% at the cost of more work on the client side. The annotation size could also be reduced by further condensing the dependence graph that it contains. The current implementation shares numbered locations between statements that can be easily seen to have the same read and write sets, but it does not find all such possibilities for sharing. In fact, it is possible to use the same numbered location for instructions whose read and write sets are not the same, but similar enough that using the same numbered location for them would not change the side-effect relationships encoded. Specifically, the following are necessary and sufficient conditions allowing two numbered locations x and y to be merged without reducing the precision of the side-effect information:

- x and y have a non-empty intersection, and
- the set of other numbered locations having a non-empty intersection with x is the same as the set of other numbered locations having a non-empty intersection with y

It would be interesting to see how much the annotation size could be reduced in this way.

8 Conclusions

This report described an implementation of a side-effect analysis for Java whose result is encoded in class file attributes. The experiments showed that the analysis rules out a significantly higher number of dependences than the simple approximation often used, in which method calls are assumed to possibly access any memory location, and pointers are all assumed to be potentially aliased. However, more work is necessary to determine whether the dependences ruled out are truly important ones. The annotation is small enough, and the analysis fast enough to be used for typical benchmarks. This infrastructure will enable measurements of the usefulness of side-effect information in just-in-time compilers, and of the effects of various points-to analyses on the precision of this information.

References

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[2] A. Azevedo, J. Hummel, and A. Nicolau. Java annotation-aware just-in-time (AJIT) compilation system. In *Proceedings of the ACM 1999 Java Grande Conference*, pages 142–151, Palo Alto, CA, 1999.

[3] Lars R. Clausen. A Java bytecode optimizer using side-effect analysis. *Concurrency: Practice and Experience*, 9(11):1031–1045, 1997.

[4] Chandra Krintz and Brad Calder. Using annotations to reduce dynamic optimization time. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation*, pages 156–167, Snowbird, Utah, June 20–22, 2001.

[5] Ondřej Lhoták, Feng Qian, John Jorgensen, and Laurie Hendren. Spark: Soot pointer analysis research kit. Submitted to *Student Research Forum, '02 Conference on Programming Language Design and Implementation (PLDI-02)*, Berlin, Germany, 2002.

[6] Donglin Liang, Maikel Pennings, and Mary Jean Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for Java. In *ACM SIGPLAN–SIGSOFT workshop on Program analysis for software tools and engineering: June 18–19, 2001, Snowbird, Utah, USA: PASTE'01*, pages 73–79, New York, NY, USA, 2001. ACM Press.

[7] Patrice Pominville, Feng Qian, Raja Vallee-Rai, Laurie Hendren, and Clark Verbrugge. A framework for optimizing Java using attributes. In *Proceedings of the International Conference on Compiler Construction*, pages 334–554, 2001.

[8] Atanas Rountev and Barbara G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. *Proceedings of the International Conference on Compiler Construction 2001, Lecture Notes in Computer Science*, 2027:20–36, 2001.

[9] Vijay Sundaresan, Laurie J. Hendren, Chrislain Razafimahefa, Raja Vallee-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*, pages 264–280, 2000.

[10] Raja Vallee-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the soot framework: Is it feasible? In *Proceedings of the International Conference on Compiler Construction*, pages 18–34, 2000.

A Annotation format

The side-effect information for each method is encoded in two attributes: a code attribute with the name SideEffectAttribute, and a method attribute with the name DependenceGraph.

A.1 SideEffectAttribute

This attribute maps statements to abstract locations read and written, and also indicates which invoke statements may transitively call native methods.

0	1	0	1	2	3	4	5	6	
record count		bytecode offset		read set		write set		calls native	•••

The first two bytes of the attribute are a big-endian integer specifying the number of records to follow.

Each record that follows consists of seven bytes:

- The first two bytes are a big-endian integer specifying the bytecode offset of the instruction that this record describes.
- The third and fourth bytes are the number of the numbered location *read* by the instruction that this record describes.
- The fifth and sixth bytes are the number of the numbered location *written* by the instruction that this record describes.
- The least significant bit of the seventh byte is one if the instruction that this record describes invokes a method that may be a native method, and zero otherwise. The remaining bits are reserved.

The special numbered location `0xffff` indicates a non-existent location, and is used to indicate that an instruction does not read or write anything. For example, the record for a getfield instruction will specify the location that the instruction reads, and `0xffff` for the location that it writes.

A.2 DependenceGraph

This attribute specifies dependences between numbered locations.

<table>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td rowspan="2">• • •</td></tr>
<tr><td colspan="2">set</td><td colspan="2">set</td></tr>
</table>

It consists of a number of records, each four bytes in length. The first two bytes and the last two bytes of each record each specify a numbered location. If a numbered location may overlap another numbered location, then the two locations will appear as a record in this attribute. Note that each unordered pair of locations is encoded in the attribute only once, with the lower-numbered location listed first, but the relation is symmetric.