USING INTER-PROCEDURAL SIDE-EFFECT INFORMATION IN JIT OPTIMIZATIONS

by

Anatole Le

School of Computer Science
McGill University, Montreal

February 2005

A THESIS SUBMITTED TO MCGILL UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF THE DEGREE OF
MASTER OF SCIENCE

Abstract

Side-effect analysis gives information about the set of locations that a statement may read or modify. This analysis can provide information useful in a compiler for performing aggressive optimizations. The impact of the use of side-effect analysis in compiler optimizations has been studied for programming languages such as Modula-3 and C, but no thorough investigation for Java has been done. We present a study of whether side-effect information improves performance in Java just-in-time (JIT) compilers, and if so, what level of analysis precision is needed. We also analyze the optimizations and benchmarks that benefit most from side-effect analysis.

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 various local and global analyses and optimizations such as 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 of up to 20% for some benchmarks. The main cause of the speedups is the use of side-effect information in load elimination. 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.

Résumé

Les analyses inter-procédures tel que l'*analyse d'effets secondaires* peuvent fournir de l'information utile pour effectuer des optimisations agressives. Nous présentons une étude qui a pour but de vérifier si l'utilisation de l'analyse d'effets secondaires peut améliorer les performances de compilateur juste-à-temps (JIT), et si tel est le cas, quel niveau de précision de l'analyse est requiert.

Nous avons utilisé SPARK, la composante d'analyse inter-procédure de SOOT, un cadre d'analyse et d'optimisation pour Java, pour faire l'analyse d'effets secondaires et l'encoder dans les fichiers *class* Java. Nous avons modifié Jikes RVM, un JIT de recherche, afin que l'analyse d'effets secondaires soit utilisée dans l'élimination de sous-expression commune, dans le *heap SSA*, dans l'élimination de charge redondante et dans le déplacement de code boucle-invariable. Sur les programmes standards de SpecJVM98, nous avons mesuré le nombre statique d'opérations de mémoire diminué, les comptes dynamiques d'instructions de lecture de mémoire éliminés, et le temps d'exécution.

Nos résultats démontrent que l'utilisation de l'analyse d'effets secondaires augmente jusqu'à 98% le nombre statique d'opportunité d'élimination d'opérations de charge, et réduit jusqu'à 27% le nombre dynamique d'instructions de lecture de champ. L'utilisation d'information sur les effets secondaires a produit une montée en vitesse de jusqu'à 20% pour certain programmes. La cause principale de ce résultat est l'utilisation de l'analyse d'effets secondaires dans l'optimisation de l'élimination de charge. Finalement, parmi les différents niveaux de précision de l'information sur les effets secondaires, une analyse relativement simple est habituellement suffisante pour obtenir la plupart des montées en vitesse.

Acknowledgments

I am very grateful to my advisor, Professor Laurie Hendren, for her help, guidance and encouragements throughout the completion of this thesis. I learned a lot from her high-level view, criticisms, many advices and suggestions, and her insights in compilers. I also very appreciated her financial support towards the end of my studies.

I would like to thank Professors Laurie Hendren, Clark Verbrugge, Prakash Panangaden and former Professor Karel Driesen for the courses that they taught in programming languages and that helped me accomplish this work. I am also very grateful to many professors and the Co-op program at the University of Waterloo, which provided me a solid foundation in Mathematics and Computer Science during my undergraduate studies.

I would also like to thank all members of the Sable Compiler and Concurrency Lab that made the environment stimulating, and also friendly. In particular, I cannot thank enough Ondřej Lhoták for all the help, explanations, comments and suggestions that he provided. I also thank Feng Qian and the people on the Jikes RVM mailing list for answering questions related to Jikes RVM. I thank Lin Wang, who was my partner in the compiler peephole optimization contest, and Marc Lanctot for his help on PowerPC machines.

I really enjoyed my graduate studies at McGill, and the time that I spent with the tennis team. I also met really good friends. Thanks to my friends and family, in particular, Marcia, my sister and my parents, who all supported me with encouragements during good as well as difficult times.

Finally, this research has been funded in part by FQRNT and McGill Graduate Studies fellowships.

Contents

List of Figures

List of Tables

Chapter 1
Introduction

1.1 Motivation

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.

Side-effect analysis provides an approximation of the set of memory locations that each instruction may read or write. This analysis can optimize code by eliminating redundant loads and stores in the presence of method calls. It can also improve precision of other intra-procedural analyses, which in turn may enable various other optimizations. Since all non-trivial data types in Java are objects which are always accessed through indirect references (pointers), one would expect optimizations using side-effect information to enable significant further improvements in the performance of Java programs.

The purpose of the study presented in this thesis is to answer three 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? Third, which optimizations benefit most from side-effect analysis and where in the code does it make a difference?

To study these questions, we used the SOOT [VRGH+00] bytecode analysis, optimization, and annotation framework, which implements a system consisting of various ahead-of-time inter-procedural side-effect analyses. SOOT supports three intermediate representations that can be used for transforming bytecode at different abstraction levels. The side-effect analyses computed in SOOT uses Jimple as its intermediate representation. The simplest, least precise side-effect analysis computed in Soot uses Class Hierarchy Analysis (CHA) [DGC95] for an approximation of the call graph and method summaries of fields read and written. More precise (though more expensive) side-effect analyses make use of call graph and points-to information computed by SPARK [Lho02, LH03]. SPARK is the points-to analysis framework component of SOOT that is used to estimate the set of locations in memory that a Java reference variable can point to.

The SOOT framework also supports the embedding of user-defined attributes in Java class files through its annotation framework [PQVR+01]. These attributes are used to encode optimization information that is determined during a static analysis of the program. JIT compilers have in the past used such information in optimizations such as array bounds check elimination and null pointer check elimination. In this thesis, we are interested in encoding side-effect information in class file attributes, and apply it in various optimizations of a JIT compiler. We chose the Jikes Research Virtual Machine (RVM) [AAB+00], an open source research software written in Java that can execute Java programs, as the JIT for our study.

1.2 Contributions

The contributions of this thesis are the following:

- We review in detail the different side-effect analyses implemented in SOOT, and the call graph and points-to analyses computed by SPARK that a side-effect analysis relies on.

- 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 present an analysis of the speedups obtained by pointing out the optimizations that benefit most from side-effect information and where in the code these optimizations achieve speedups.

In the following subsections, we describe in more detail each of these contributions.

1.2.1 Side-Effect Analysis in Soot

We first review the two inter-procedural analyses that a side-effect analysis depends on in SOOT :

- Call Graph Construction (Section 3.1)
- Points-to Analysis (Section 3.2)

We then explain how Soot computes side-effect analysis in Section 3.3. We present how side-effect information is encoded in class file attributes and the method it uses to reduce the attribute's size in Section 3.4. We describe the different side-effect variations and their relative precision, and show examples of optimizations that can be performed with the different analyses in Section 3.5.

1.2.2 Implementation

To take advantage of side-effect analysis, we made several modifications to Jikes RVM. We added code to read in the side-effect information produced in our analysis. We then modified the following analyses and optimizations to take advantage of side-effect information:

- Local Common Sub-Expression Elimination (Section 4.1)
- Heap Array SSA Construction (Section 4.2)

- Redundant Load Elimination (Section 4.2)
- Loop-Invariant Code Motion (Section 4.3)

We provide a description of each of these analyses and optimizations in Sections 4.1 to 4.3, and explain how these can benefit from side-effect analysis. We describe the modifications that we made and show examples of improvements enabled by the use of side-effect information in the optimizations adapted. We explain how we dealt with method inlining when using side-effect information in Section 4.4. Finally, to measure the effect of the availability of side-effect analysis in these optimizations, we inserted instrumentation code in Jikes RVM both to count the static opportunities for performing optimizations, and the dynamic effects on the improved optimizations.

1.2.3 Experiments

We performed experiments on the use of side-effect information in local and global optimizations on three different architectures (Intel, AMD and PowerPC). The different systems and benchmarks used in our experiments are described in detail in Chapter 5.

The results for local optimizations (only local CSE makes use of side-effect analysis) are presented in Section 6.1. Our experiments show that static opportunities for load elimination increased by up to 41%, but only resulted in a decrease of up to 0.87% of dynamic loads. This produced speedups in *mpegaudio* of 1.08x and 1.06x on our Intel and AMD systems, and 1.02x for *raytrace* on both of these systems. On PowerPC, the use of side-effect information did not enable speedups. Finally, the different side-effect variations produced identical static and dynamic counts, and as expected, similar execution times. A simple, inexpensive side-effect analysis thus appears sufficient in local optimizations.

For global optimizations (Section 6.2), all of the optimizations that we modified make use of side-effect information. The results of the experiments show that the use of side-effect analysis increased 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 of up to 20% for benchmarks *compress*, *raytrace/mtrt* and *mpegaudio*.

Our results also 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 discovered for other languages such as Modula-3 or C [GH98, DMM98].

1.2.4 Analysis of Speedups

For local optimizations where only local CSE makes use of side-effect analysis, a significant speedup was obtained only for benchmark *mpegaudio* on our Intel and AMD systems. We show in which method the additional loads eliminated using side-effect information improved execution times.

For global optimizations, we show that the availability of side-effect information was mostly beneficial in the redundant load elimination optimization for the speedups obtained for benchmarks *compress*, *raytrace/mtrt* and *mpegaudio*. For each of these benchmarks, we report the methods in which the use of side-effect analysis made a difference. We present static and dynamic counts, and execution times of runs when not using side-effect information in these methods, and compare them with our results in Chapter 6. For benchmarks *compress* and *mpegaudio*, more precise side-effect analyses improved speedups over our simple (CHA) side-effect analysis. We point out where in the code the redundant load elimination optimization took advantage of more precise side-effect information. We show that the additional loads eliminated is due to an improved precision of side-effect information on array elements. Finally, we note that adding a type-based analysis on array elements in our simple, inexpensive, side-effect analysis would find these load removal opportunities, and as a result, would produce speedups similar to our most precise side-effect analysis.

1.3 Thesis Organization

The remainder of this thesis is organized as follows. The next chapter discusses related work. Chapter 3 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 that we experimented with. In Chapter 4, we describe how we modified the optimizations in Jikes RVM to take advantage of side-effect information. We present in Chapter 5 the benchmarks that we used, properties about the benchmarks, and the environment in which we conducted our experiments. We report our empirical results, including static and dynamic effects of side-effect information usage in the optimizations that we modified, and execution times improvements for local and global optimizations in Chapter 6. We provide an analysis of the speedups obtained in Chapter 7, showing which optimizations benefited most from side-effect information and where in the code it made a difference. We conclude with Chapter 8 and discuss future work.

Chapter 2
Related Work

In this chapter, we present a survey of the previous work on side-effect analysis. The first section discusses the computation of summary information for method calls for languages without pointers, covering the early algorithms of the 1970's and Banning's solution to the side-effect problem which became the conventional framework on which other researchers worked on to improve in the 1980's. The second section presents recent work done in the 1990's and 2000's on side-effect analysis for languages with general-purpose pointer usage. This section also discusses the use of side-effect information as a metric in comparing points-to analyses, and the impact it has statically and dynamically in program optimizations.

2.1 Summary Information of Procedures

2.1.1 Early Algorithms

Early summary information algorithms for procedure calls dates back to the 1970's [Spi71, All74, Ros75, Bar78] and were mainly targeted for the FORTRAN programming language. This analysis was defined as an inter-procedural dataflow analysis used to summarize the semantic effects associated with subroutine calls and permitting global flow analysis to more effectively propagate information through programs. Each statement *s* in a program was annotated with *MOD* and *REF* sets defined as:

- **MOD**(s): set containing those variables whose values can be **changed** as a result of executing s.
- **REF**(s): set containing those variables whose values can be **used** as a result of executing s.

Summary information was used in various intra-procedural transformations, optimizations and parallelization. The early algorithms for computing this information differed on the language features they supported, the information that they computed, the precision of the analysis, their complexity, and whether they used one or multiple passes over the program. Some analyses ignored recursion or parameter aliasing [Spi71, All74], were inefficient [Ros75], or did not support the nesting of procedures [HS75, AU77]. The most powerful techniques worked with languages with recursion and sharing of variables through reference parameters, and was precise up to symbolic computation [Bar78]. The use of pointers was not supported. All of these techniques were based on some form of transitive closures of various relationship. A comparison of these algorithms for computing summary information can be found in Barth's PhD dissertation [Bar77].

2.1.2 Banning's Decomposition of the MOD Problem

Several researchers worked on the improvements of the early algorithms for computing summary information in both complexity and precision [Lom77, Ros79, Ban79]. Banning [Ban79] presented basic methods, using one pass, to find flow-insensitive side-effects and possible aliases of variables. He represented the MOD problem as a dataflow problem over the program's call multi-graph, which could be solved by efficient techniques developed for global dataflow analysis. His algorithm was more efficient than previous work, handled recursion and reference parameters, and was precise up to symbolic computation. The basic methods for computing MOD could also be extended to summarize flow-sensitive side-effects and cover other features and constructs present in programming languages. The extensions are covered in Banning's PhD thesis [Ban78].

To perform the MOD analysis, Banning decomposed the problem into two separate components:

- alias analysis, and
- side-effect computation.

To compute MOD, Banning first computed a set GMOD(p) for every procedure or function *p* representing the generalized MOD of *p*. Secondly, he calculated a set DMOD(s) for every statement *s* representing the direct MOD of *s*. Thus, GMOD sets applied to procedures, and DMOD sets to statements. MOD was then computed using the combination of these two sets. MOD(s) was simply derived from DMOD(s) by considering the potential aliases[1] due to reference parameters.

2.1.3 Improving Banning's Framework

Banning's work on MOD analysis became the conventional framework on which other researchers have worked on its improvements in the 1980's [Mye80, Mye81, Bur84, Bur90, CK84, BC86, CR87, CK88b]. Cooper and Kennedy [CK84] presented improvements in the complexity of computing flow-insensitive summary information by breaking the problem into two subproblems, a computation for global variables and one for call-by-reference formal parameters. Combining the solutions to these subproblems solved the original MOD problem. Using known efficient techniques to solve each of these subproblems, the MOD analysis could be computed in almost linear time. A few years later, Cooper and Kennedy [CK88b], again, presented new methods to solve each of the two subproblems by using a data structure, known as the *binding multi-graph*, to achieve a linear time complexity. Burke [Bur90] then showed that the two subproblems on globals and formals could be solved more effectively by a similar problem decomposition.

Burke and Cytron [BC86], Triolet, Irgoin, and Feautrier [TIF86], and Callahan and Kennedy [CK88a] were interested in automatically restructuring sequential programs on parallel architectures to improve performance, mainly by scheduling loop iterations concurrently on multiple processors. When an array element was modified by a procedure call, current inter-procedural side-effect analyses conservatively assumed that the entire

[1]Two variables are potentially aliased to each other if the analysis considers that they could access the same memory location through a reference to either of them at a given point in the execution of a program.

array could be modified by the call. Within a loop, in this case, all elements of an array appeared to be referenced in each iteration. This much restricted the parallelization of loops containing calls. Studies were thus conducted to integrate subscript analysis with aliasing and to implement inter-procedural dataflow analysis to compute side-effects of method calls on subscripted references (individual array elements).

Ryder and Carroll [CR87] studied incremental algorithms for large, complex, and dynamically evolving systems. They presented an incremental interval algorithm for MOD analysis that could compute an updated side-effects solution in response to a change in the system rather than recalculating it in its entirety.

2.2 Side-Effects for Languages with Pointers

Existing techniques for the computation of side-effect information could handle call-by-reference induced aliasing but did not support the use of pointers. This was insufficient to perform aggressive transformations and optimizations in languages with general-purpose pointer usage. Choi, Burke and Carini [CBC93] were the first to show that conventional methods for side-effect analysis based on the decomposition in Banning's framework could not handle the presence of pointers correctly. They illustrated with examples that side-effect analysis could not be performed separately from alias analysis for languages with pointers. They mentioned an algorithm that could compute side-effects and that supported pointers, including passing of pointers as reference or value parameters. However, they did not provide a description of the algorithm or present implementation results. Landi, Ryder and Zhang [LRZ93] were the first to present a complete design and implementation of an inter-procedural modification side-effects algorithm for C programs that could handle the presence of general-purpose pointers.

Early side-effect analyses for languages with pointers by Choi, Burke and Carini [CBC93] and Landi, Ryder and Zhang [LRZ93] 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.

While prior work used the notion of alias-pairs analysis, Hendren *et al.* took a different approach and introduced the *points-to* abstraction in their McCAT C Compiler [HDE$^+$93, EGH94]. This method computes relationships between abstract memory locations. This analysis can provide points-to information based on the reads and writes abstract locations computed, and can be used directly in other transformations and optimizations. The following section discusses previous work on points-to analyses that used side-effect analysis to evaluate its precision and effectiveness.

2.2.1 Evaluating Points-to Analyses

To evaluate and compare the precision of various point-to analyses, researchers measured its effect on the precision of side-effect information, a client analysis, by reporting the size of the points-to sets at indirect memory access instructions (i.e. `*p=`) [EGH94, Ruf95, RR01, RMR01, MRR02]. Other points-to analysis work [LRZ93, Oli97, MSH97, SRLZ98, HP00, RLS$^+$01] 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 3.3.

Landi, Ryder and Zhang [LRZ93] measured the average and maximum number of side-effects found per assignment through pointer dereference (`*p=`), per procedure and per call site. They found that for their set of C programs, the number of locations assigned values per assignment statement through dereference pointer variable was on average 1.2. Thus, in most cases, there was only one alias for such variable at a given program point. However, their analysis excluded certain features of the C programming language such as union types, casting, pointers to functions, exception handling, *setjump* and *longjump*. In our experiments, we found that performing a context-insensitive points-to analysis to distinguish different *objects* in our set of Java benchmarks provided little benefit. This result also leads us to believe that on average, the number of possible aliases for a given variable is low.

Emami, Ghiya and Hendren [EGH94] presented the *points-to* abstraction implementation and results in the McCAT C Compiler framework [HGMS91]. They also showed how to compute the program's call graph and points-to analysis together. Their points-to analysis was context-sensitive, including recursive and mutually-recursive calling contexts, and handled general function pointers in C. They measured possible and definite points-to information at indirect references. For their set of benchmarks, their results showed that the overall average number of locations pointed to by a dereference pointer was 1.13, indicating that their points-to analysis was very precise.

Ruf [Ruf95] studied the empirical benefits of context-sensitive points-to analyses over context-insensitive ones. He calculated points-to pairs reaching the location of inputs of indirect memory reads or stores. His results showed that context-sensitivity offered no benefit or improved precision on his set of C benchmarks programs. However, Ruf warned that this result, somewhat surprising, might only apply to his set of benchmarks and that for larger programs, context-sensitive analyses may be beneficial.

Olivar [Oli97] implemented a side-effect analysis based on the type inference points-to algorithm by Steensgaard [Ste96] in the McCAT C compiler framework [HGMS91]. She compared the read and write sets computed using this algorithm with the context-sensitive points-to analysis [EGH94] implemented in McCAT. Her results showed that having a simple side-effect analysis over having none was very beneficial. The benefits of having a more precise side-effect analysis over a simple one were smaller. This result agrees with our runtime measurements, where we found that our simple side-effect analysis was sufficient to obtain most of the speedups.

Shapiro and Horwitz [MSH97] studied the effects of the relative accuracies of different points-to analysis algorithms on various subsequent analyses including MOD analysis. To determine whether and how much the choice of points-to analysis affects MOD analysis, they measured the sum of the sizes of the MOD sets for each function and the time to perform the MOD analysis. Their results showed that the size of MOD sets increased by about 70% when the size of points-to sets doubled on average. They also observed that a more precise points-to analysis leads to a faster MOD computation (since points-to sets are smaller). Still, the total time to perform both points-to analysis and side-effect computation was smaller using an imprecise but fast points-to algorithm.

Stocks, Ryder, Landi and Zhang [SRLZ98, RLS+01] reported comparative experiments on the effectiveness of a context-sensitive flow-sensitive points-to analysis versus a context-insensitive flow-insensitive one with respect to precision and scalability on the modification side-effects problem for C. On a large set of C programs, they gathered empirical measurements of the precision of the analysis at pointer dereference statements (`*p=`) and at function call statements. They noted that although the loss in accuracy of using a context-insensitive flow-insensitive analysis is a strong concern, the analysis provides a significant gain over worst-case assumptions and can be adequate for certain applications. Our run-time measurements confirm this result. They also concluded that a context-sensitive flow-sensitive analysis yields significant precision improvements at the expense of much greater complexity.

Hind and Pioli [HP00] compared the effectiveness of five pointer analysis algorithms on C programs. The analyses were context-insensitive and varied in their use of control flow and alias data structures. They measured the precision of the analyses and how the computed solution affects various client analyses of pointer information including side-effect analysis. Their empirical experiments reported the average MOD and REF size sets at each nodes in the control flow graph. Their results showed that the difference in precision of the side-effect information resulting from the various context-insensitive analyses was minimal. In our experiments, we also found that the various context-insensitive flow-insensitive points-to analyses used to compute side-effect information were about equivalent.

More recent work on the Java programming language also measured the precision of points-to analyses by reporting the size of the points-to sets at field read and write instructions [RMR01, MRR02]. Rountev and Ryder [RR01] evaluated their points-to analysis for precompiled libraries in this way. Rountev, Milanova and Ryder [RMR01] presented a points-to analysis for Java based on Andersen's points-to analysis for C [And94] using annotated inclusion constraints. Their results on a large set of Java programs showed that the points-to analysis solution has a significant impact on which objects may be read or written by program statements (object read-write information). Later, they performed similar measurements to evaluate object-sensitivity, a new form of context-sensitivity for flow-insensitive points-to analysis for Java [MRR02]. The precision improvements of object-sensitivity analysis over context-insensitive analysis significantly improved the precision of MOD information by reducing the number of modified objects per statement. The impact on execution time of using context-sensitive side-effect analysis in JIT optimizations is one of our areas for future work.

2.2.2 Impact of Side-Effect Analysis in Optimizations

When evaluating the effectiveness of points-to analyses, most researchers have used the common metric of average points-to set size at indirect read or write statements or at function calls. This metric only provides static results. This measure is thus not sufficient to understand the impact of using these analyses in optimizations has on achievable run-time performance improvements. Studies measuring the actual run-time impact of code optimized using side-effect information are surprisingly rare. We discuss them below.

Clausen's [Cla97] 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 applied his analysis results in an ahead-of-time early Java bytecode optimizer to a similar set of optimizations as we did: dead code removal, loop invariant removal, constant propagation, and common sub-expression elimination. For one benchmark, he obtained a speedup of 25%. However, his experiments were run using JDK 1.0.2, one of the earliest Java virtual machines which did not have a just-in-time compiler to perform aggressive optimizations like modern JVMs have today.

Ghiya and Hendren [GH98] measured the effectiveness of side-effect information on the run-time efficiency of code produced by an optimizing compiler for C. They used side-effect analysis in traditional analyses like common sub-expression elimination, loop-invariant removal, location-invariant removal (similar to the scalar replacement technique for array references), and array dependence testing. On a set of pointer intensive C benchmarks, they obtained up to 10% speedups. They observed that a reduction in memory references and instructions executed always translated into a speedup, but that there was no direct correlation between this reduction and the percentage of performance improvements. Our results also showed that the speedups obtained for our set of benchmarks is not in proportion with the percentage of dynamic loads eliminated.

Diwan, McKinley and Moss [DMM98] studied a version of redundant load elimination (RLE) which combines loop-invariant code motion and common sub-expression elimination of memory references in Modula-3. They used declared types to conservatively approximate aliasing relationships, and method read/write set summaries. They obtained up to 8% speedups when using their most precise type-based alias analysis. They noted that they expect RLE to be a profitable optimization since loads are expensive on modern machines and architects expect they will only get more expensive [HP95]. In our experiments, we found that using side-effect information in RLE had the largest impact on benchmarks with high load densities. 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. They show that for a type-safe languages like Modula-3 and Java, a fast and simple alias analysis may be sufficient for many applications.

Debray, Muth and Weippert [DMW98] presented an alias analysis and evaluated their algorithm by measuring the percentage reduction in dynamic loads when using this analysis in a redundant load elimination optimization. However, they did not provide execution time measurements. Their results showed that local alias analysis provided none to small improvements, but for global alias analysis, it produced a reduction of up to 7% of dynamic loads. Similarly, we concluded that side-effect analysis has little impact on local optimizations, but improves significantly global optimizations.

Razafimahefa [Raz99] performed loop invariant code motion using side-effect information on Java in an ahead-of-time bytecode optimizer, and reported run-time speedups comparable with ours on an early-generation Java VM (up to 20%). He observed that many invariant expressions were not moved due to the context-insensitive nature of the analysis, and that a context-sensitive side-effect analysis would be beneficial.

Cheng and Hwu [CH00] performed a study of the impact of memory disambiguation on optimizations such as redundant load and store elimination, loop-invariant memory access migration, and load and store scheduling. They performed experiments on numerous C benchmarks using fully resolved pointers and function side-effects. Their empirical results using the SPECcint92 and SPECcint95 benchmarks suite produced a reduction of up to 40% of dynamic loads, and speedups of 42% on average (in large part due to the load and store scheduling optimization). The performance gains that they obtained were over programs compiled without pointer analysis, side-effect analysis and memory access optimizations. Although we obtained smaller speedups on our set of Java programs (up to 20%), we note that the base that they used for comparison is much more conservative than ours. Our base includes some form of memory disambiguation using types and global value numbering.

Ghiya, Lavery and Sehr [GLS01] evaluated the benefits of a complete memory disambiguation framework in transformations and optimizations, and its impact on program performance. Their framework includes numerous techniques including pointer analysis, and MOD and REF analyses for function calls. They compared performance improvements achievable using several memory disambiguation techniques and obtained up to 26% speedups on the SPECcint2000 C benchmarks. They also concluded that there was no direct correlation between the static improvements and the performance gains.

Pechtchanski and Sarkar [PS02] presented 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 the redundant load elimination and loop-invariant code motion 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.

Chowdhury, Djeu, Cahoon, Burrill and McKinley [CDC$^+$04] studied the effect of alias analysis precision on the number of optimization opportunities for a range of scalar optimizations. However, they only measured 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 found, unsurprisingly, that alias analysis precision had little effect. Other work studying the effect of alias analysis on scalar optimizations also suggests that a simple alias analysis may be sufficient [DLFR01, DMM01].

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.

Chapter 3
Background

In this chapter, we review the implementation of side-effect analysis in SOOT [VRGH+00], a framework for analyzing, optimizing, and annotating Java bytecode. The side-effect analysis depends on two other inter-procedural analyses, call graph construction and points-to analysis. We describe the construction of the call graph in SOOT in Section 3.1. An important difference from most other work on call graph construction is that to obtain a conservative side-effect analysis, the call graph must include all methods invoked, including those invoked implicitly by the Java VM. In Section 3.2, we briefly explain the output of the SPARK points-to analysis framework [Lho02, LH03]. Section 3.3 explains how the information from these two analyses is put together to produce side-effect information. In Section 3.4, 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 3.5, we describe how variations in the precision of the call graph and points-to analyses affect the side-effect information.

3.1 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 in SPARK of computing call graphs. First, we used call graphs computed using Class Hierarchy Analysis (CHA) [DGC95], an inexpensive method which considers only the static type of each receiver object, and does not require any inter-procedural analysis. Second, we used SPARK 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 to determine 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 [LY99, Section 2.17.4]. The SPARK analysis assumes 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, SPARK uses 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`. SPARK 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 SPARK's 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. SPARK conservatively assumes 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 by SPARK, 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.

3.2 Points-to Analysis

We use the SPARK [Lho02, LH03] points-to analysis framework to compute points-to information. For each *pointer* p in the program, SPARK 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; SPARK models all run-time objects created at a given allocation site as a single entity. In addition, special *objects* must be included 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*, the exact run-time type may not be known. Therefore, SPARK conservatively assumes 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 3.5.

3.3 Side-Effect Analysis

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

For each statement s, SPARK computes 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, only the direct read and write sets for each statement in the program are computed, 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, the read and write sets of each method are continually aggregated and propagated 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, an interference relation between all the read and write sets in the method is computed: $int(m) = \{(set_1, set_2) \mid set_1 \cap set_2 \neq \emptyset\}$. The interference relation is mapped on read and write sets to 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 $(read(s_1), write(s_2)) \in int(m)$. It is the dependences between statements that are encoded in class files for the JIT to use in performing optimizations.

3.4 Encoding Side-Effects in Class File Attributes

All of the SPARK 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, SPARK must convert them to refer to bytecode instructions during the translation of Jimple to bytecode. SOOT includes a universal tagging framework [PQVR+01] 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, SPARK identifies 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, many of the read and write sets in the method are identical. Therefore, a level of indirection is added. Instead of expressing the dependence relations in terms of statements, SPARK enumerates all distinct read and write sets, and expresses the dependence relations between those sets. For each statement, SPARK indicates 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 his M.Sc. thesis [Lho02, Sections 6.2.2 and 6.2.6], Lhoták observed that this encoding limits the annotation size to acceptable levels.

Figure 3.1 shows the side-effect attribute format in class files. Each method is associated with two attributes. The first one, `SideEffectAttribute`, maps each bytecode that has side-effects to a read and write set. The extra byte contains a bit that indicates whether a bytecode explicitly or implicitly invokes a native method, and other bits for future use. For our purpose, we did not use this extra byte. The second attribute, `DependenceGraph`, denotes which sets interfere.

SideEffectAttribute:

```
BytecodeOffset   ReadSet   WriteSet   ExtraByte
(2 bytes)        (2 bytes) (2 bytes)  (1 byte)
```

DependenceGraph:

```
Set         Set
(2 bytes)   (2 bytes)
```

Figure 3.1: Side-Effect Attribute Format

In Figure 3.2, we show sample code and the resulting encoding of side-effect information. Method `foo` contains instructions that, once compiled, would be represented by a *putfield* and two *invokevirtual* bytecodes at offset 2, 6 and 10. Since only the *putfield* and *invokevirtual* bytecodes at offset 2 and 6 have side-effects (`a.nothing()` has none), only two entries appear in the `SideEffectAttribute` of method `foo`. For both of these, the read set value is -1 (they do not read anything), and their write set values are 0 and 1 respectively. Since these two write sets interfere (both contain field `f`), the `DependenceGraph` attribute denotes a write-write dependence between sets 0 and 1.

3.5 Analysis Variations

In our empirical study presented in Chapter 6, we compare the effectiveness of six variations of side-effect analyses in Soot. In this section, we explain the differences between these variations. In Figure 3.3, we present examples of code that distinguishes the variations: it may be optimized only if the information provided by specific variations is available. In line 28, the code writes a constant to the field `b.f`. In line 30, the constant is read out again. Our goal is to optimize away the constant field read. If we substitute each of the code snippets **(a)** through **(e)** on the right of Figure 3.3 for line 29, the resulting code will never change the value (4) loaded in line 30. However, analyses of different precision are required to prove that the code snippets do not have side-effects affecting the value of

```
class A {
  int f;
  void setF( int n ) { this.f = n; }
  void nothing() {}

  void foo( A a ) {
      a.f = 4;     // Offset  2: putfield
      a.setF( 3 ); // Offset  6: invokevirtual
      a.nothing(); // Offset 10: invokevirtual
  }
}

SideEffectAttribute (method foo):
  Offset  ReadSet  WriteSet
       2       -1         0
       6       -1         1

DependenceGraph (method foo):
  Set  Set
    0    1
```

Figure 3.2: Example of Side-Effect Attribute

```
class Box {
    A a;
}
abstract class A {
  int f;
  abstract void nothing();
  abstract void maybe();
  abstract void setF();
  abstract A id();
}
class B extends A {
  void nothing() {}
  void maybe() { this.f = 1; }
  void setF() { this.f = 2; }
  A id() { return this; }
}
class C extends A {
  void nothing() {}
  void maybe() {}
  void setF() { this.f = 3; }
  A id() { return this; }
}
class Main {
  public static void main(String[] args) {
     new Main().run(new B(), new C());
  }
  void run(A b, A c) {
    b.f = 4;
    // insert possible side-effect here
    int n = b.f; // eliminate this load
  }
}
```

(a)

```

```

(b)

```
c.nothing();
```

(c)

```
c.maybe();
```

(d)

```
Box b1 = new Box();
Box b2 = new Box();

b1.a = c;
b2.a = b;

c = b1.a;
c.setF();
```

(e)

```
c = c.id();
b = b.id();
c.maybe();
```

Figure 3.3: Code Examples

`b.f`. Figure 3.4 gives an overview of the relative precision of the variations, with precision increasing from bottom to top. After each variation, we list the subset of the code snippets that can be optimized using the information provided by the variation.

$$\begin{tikzcd}
 & \text{otf-fs } \{abcde\} & \\
\text{otf-fb } \{abcd\} \arrow[ur] & & \text{aot-fs } \{abce\} \arrow[ul] \\
 & \text{aot-fb } \{abc\} \arrow[ul] \arrow[ur] & \\
 & \text{CHA } \{ab\} \arrow[u] & \\
 & \text{none } \{a\} \arrow[u] &
\end{tikzcd}$$

Figure 3.4: 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. In this case, Jikes RVM is able to remove the read in line 30 only when the empty snippet (a) is inserted at line 29. The JIT determines that the field being loaded is the same as the field to which the constant was written, and since no statements have been executed since the write, the value could not have been affected. However, as soon as we insert any method call between the write and read (in each of the code snippets (b) through (e)), the JIT cannot optimize the read, because it knows nothing about the side-effects of the method called.

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 3.1. In this case, we can optimize code snippet (b), because the analysis determines that the call `c.nothing()` calls the method `nothing()` in either class `B` or `C`, and neither of these methods write to field `f`. However, for the call to `maybe()` in snippet (c), CHA cannot tell which of the two `maybe()` methods will be invoked. Since `B.maybe()` writes to field `f`, the analysis conservatively assumes that `b.f` may be overwritten, and prevents the optimization.

The remaining variations all take advantage of points-to analysis information to compute side-effects. The differences between them are whether the points-to analysis is field-based (fb) or field-sensitive (fs), and whether it uses a call graph computed ahead-of-time (aot), or whether it computes its own call graph on-the-fly (otf). All of the points-to analysis variations determine that `c` can only be of run-time type `B`. Therefore, the call to `c.maybe()` does not write to field `f`, so the read in line 30 can be optimized when code snippet (c) is inserted into line 29.

The distinction between a field-based and field-sensitive analysis defines how the points-to analysis treats pointer flow through fields of heap objects. In a field-based analysis, each field is treated as a *pointer* with a single points-to set, i.e. the object to which a field belongs to is not considered. Thus, it is assumed that any *object* stored into a field `f` (regardless of the object it is part of) may be retrieved from field `f` of any object. On the other hand, a field-sensitive analysis 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`). This is illustrated by code snippet (d). In the code, `c` is stored and later on read out of `b1.a`, and `b` is stored into `b2.a`. A field-based points-to analysis cannot distinguish between the field `a` of the two different boxes `b1` and `b2`, and therefore assumes that `c` and `b` could point to the same object, so `b.f` could be written to at the end of the code snippet. A field-sensitive analysis, on the other hand, proves that when `c` read out of field `a` of box `b1`, it is distinct from `b`, and so the call to `c.setF()` does not affect the value of `b.f`.

In order to propagate points-to sets inter-procedurally, a points-to analysis requires an approximation of the call graph. However, the points-to analysis can be used to build the call graph. One solution to this circular dependency is to build an imprecise call graph ahead-of-time using CHA, only for the use of the points-to analysis. After the points-to analysis completes, the points-to information is used to construct a more precise call graph to be used in the side-effect analysis. The other alternative is to build the call graph on-the-fly as the points-to analysis proceeds: as points-to sets grow, edges are added to the call graph. Results from prior work [LH03] show the latter approach to be more costly, but to produce more precise results. The difference in precision is illustrated by code snippet (e). In the code, `c` and `b` are passed through identity methods that return themselves. An ahead-of-time CHA-based call graph says that each `id()` method calls may call either of the two `id()` methods, so both objects end up in the points-to sets of both `c` and `b`. Therefore, the analysis cannot determine that the call to `c.maybe()` will not change `b.f`. However, if the analysis builds the call graph on-the-fly, the call graph only contains the single correct target method for each of the `id()` method calls, and the object pointed to by `b` does not flow into the points-to set of `c`. The analysis therefore determines that the call to `c.maybe()` does not write to `b.f`, and the load may be eliminated.

Chapter 4
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) [AAB$^+$00]. Jikes RVM is an open source research platform for executing Java bytecode. It includes three levels of JIT optimizations: level 0 (dataflow basics), level 1 (flow-insensitive, inlining, commoning) and level 2 (advanced). We adapted three optimizations in Jikes RVM to make use of side-effect information. The first one is local common sub-expression elimination (CSE), a level 1 optimization, and the other two are redundant load elimination (RLE) and loop-invariant code motion (LICM), both level 2 optimizations. Sections 4.1 to 4.3 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 4.4 describes how we dealt with this case.

4.1 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 4.1. 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* instructions 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 *putfield* or *putstatic* instruction of a field, say X, will remove any expression in the cache associated with field X (the algorithm conservatively assumes that any object references may be aliased). A call or synchronization instruction kills all expressions in the cache.

In this algorithm, we used side-effect information to reduce the set of expressions killed (lines 20 and 22 in Figure 4.1). When the current instruction is a *putfield*, *putstatic* 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.

An example is shown in Figure 4.2. 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 location. 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 7 by an available expression (line 4).

4.2 Redundant Load Elimination

The redundant load elimination algorithm relies on extended Array SSA (also known as Heap Array SSA or Heap SSA) [FKS00] and Global Value Numbering [AWZ88]. We explain the general idea of the algorithm below. For a detailed description, please refer to [FKS00].

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, in the code of Figure 4.3, there are two heap arrays, X and Y. On line 4, ”heap Array X `[a]` = *exp1*” means 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

```
for each basic block bb do
    cache = createNewEmptyCache();

    for each instruction s in bb do
        if isVolatileFieldLoadOrStore( s ) then
            continue

        // Part 1: try to replace s by an available expression, and update cache
        if isGetField( s ) or isGetStatic( s ) then
            if cache.availableExpression( s ) then
                T = findOrCreateTemporary( expression( s ) )
                replace s by copyTemporaryInstruction( T )
            else
                add expression( s ) to cache
        else if isPutField( s ) or isPutStatic( s ) then
            add expression( s ) to cache

        // Part 2: remove cache entries that s kills
        if isPutField( s ) or isPutStatic( s ) of some field X then
            remove all expressions with field X from cache (excluding expression( s ))
        else if s is a call or synchronization then
            remove all expressions from cache

```

Figure 4.1: Original Local Common Sub-Expression Algorithm in Jikes RVM

```
A obj1 = new A();
A obj2 = new A();

i = obj1.x;
obj2.x = 10;
nothing();
j = obj1.x;
```

Figure 4.2: Local Common Sub-Expression Example

```
a = new A();
b = new A();

a.x = exp1     -> heap Array X [a] = exp1
a.y = exp2     -> heap Array Y [a] = exp2
b.x = exp3     -> heap Array X [b] = exp3
n = a.x         -> n = heap Array X [a]
```

Figure 4.3: Before Scalar Replacement

computed. The global value numbering algorithm 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). In Figure 4.3, since `a` and `b` are the results of different allocation sites (lines 1 and 2), *DD*(`a`, `b`) = true and *DS*(`a`, `b`) = false.

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.

In Figure 4.3, after `a.x` is assigned on line 4, the set of available indices for heap Array X is {`a`}. Similarly, {`a`} is available for heap Array Y after the assignment to `a.y` on line 5. For the store of `b.x` on line 6, since global value numbering tells us that *DD*(`a`, `b`) = true, we have {`a`, `b`} available for heap Array X after line 6. If *DD*(`a`, `b`) had returned false, we would have conservatively assumed that a store to heap Array X [`b`] could have overwritten heap Array X [`a`], and thus, only {`b`} would have been available after line 6. On line 7, heap Array X is used at index `a`. Since `a` is available, a new temporary is introduced and scalar replacement is performed. Figure 4.4 shows the resulting code.

```
a = new A();
b = new A();

T =  exp1
a.x = T        -> heap Array X [a] = T
a.y =  exp2    -> heap Array Y [a] =  exp2
b.x =  exp3    -> heap Array X [b] =  exp3
n = T
```

Figure 4.4: After Scalar Replacement

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 4.5, without side-effect information, since `a` and `b` are both 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 13 represents a merge point of the available index sets after lines 7 and 10. The intersection of these two sets is the empty set. After the load of `a.x` on line 16, {`a`} is available. Since *DS*(`a`, `b`) = false, the load of `b.x` on line 17 cannot be eliminated. Thus, without side-effect analysis, the algorithm only finds one opportunity for load elimination in this example.

Using side-effect analysis, since `a.x` has no dependence with `b.x` (lines 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 13) results in the set {`a`, `b`}. The load of `a.x` can then be removed on line 16. The available index set after line 16 is {`a`, `b`}, allowing load elimination of `b.x` on line 17. Thus, having side-effect information allowed three additional loads to be eliminated. The resulting code after performing load elimination is shown in Figure 4.6.

```
int foo( A a, A b, int n ) {
  a.x = 2;
  b.x = 3;

  int i;
  if( n > 0 ) {
    i = a.x;
  } else {
    i = b.x;
    nothing();
  }

  // Merging point: a phi is
  // placed here in heap SSA

  int j = a.x;
  int k = b.x;

  return i + j + k;
}

public static void main( String[] args ) {
  foo( new A(), new A(), 1 );
}
```

Figure 4.5: Before Redundant Load Elimination

```
int foo( A a, A b, int n ) {
  t1 = 2;
  a.x = t1;

  t2 = 3;
  b.x = t2;

  int i;
  if( n > 0 ) {
    i = t1;
  } else {
    i = t2;
    nothing();
  }

  // Merging point: a phi is
  // placed here in heap SSA

  int j = t1;
  int k = t2;

  return i + j + k;
}

public static void main( String[] args ) {
  foo( new A(), new A(), 1 );
}
```

Figure 4.6: After Redundant Load Elimination

4.3 Loop-Invariant Code Motion

The LICM algorithm in Jikes RVM is an implementation of the Global Code Motion algorithm introduced by Click [Cli95] 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 [Cli95].

The algorithm schedules each instruction 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 to choose which basic block to place instructions is to pick the one 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.

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

Figure 4.7: Before Loop-Invariant Code Motion

In Figure 4.7, 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 4.8.

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

Figure 4.8: After Loop-Invariant Code Motion

4.4 Using Side-Effect Information for Inlined Bytecode

The side-effect attribute of each method provides information about data dependences between instructions. The attribute refers to a bytecode instruction by using its offset in the method it is part of. When a method is inlined, bytecodes are added in the current compiled method. 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 this section, we show an example and explain how we dealt with this special case.

In Figure 4.9, let's assume that calls to `foo()` and `bar()` are inlined, resulting in the code in Figure 4.10. 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 4.10, information about offset `0` is associated with bytecode `b0`, not `b1` or `b2`.

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. 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 4.10, the least common method ancestor for bytecodes `b0` and

```
Offset    main() {
0 main      b0
1 main      invoke foo
          }

          foo() {
0 foo       b1
1 foo       invoke bar
          }

          bar() {
0 bar       b2
1 bar       b3
          }
```

Figure 4.9: Before Inlining

`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 of method `main()`. Similarly, for bytecodes `b1` and `b2`, we lookup offsets `0` and `1` in the side-effect attribute of method `foo()`, the least common method ancestor of `b1` and `b2`. The same result holds for `b1` and `b3`. For bytecodes `b2` and `b3`, since they both come from method `bar()`, we lookup their original offsets, `0` and `1` respectively, in the side-effect attribute of method `bar()`.

```
Offset
          main() {
0 main      b0    // inlining sequence: main
0 foo       b1    // inlining sequence: main->foo
0 bar       b2    // inlining sequence: main->foo->bar
1 bar       b3    // inlining sequence: main->foo->bar
          }
```

Figure 4.10: After Inlining

Chapter 5
Experimental Framework

This chapter gives a description of the environment, the systems, and the various tools that were used in our experiments, and the measurements that we computed. The next section describes the different systems used for our experiments. Section 5.2 describes the Jikes RVM configuration and related tools that it relies on. In Section 5.3, we specify the benchmarks that we used, and provide some properties for each benchmark. Finally, Section 5.4 discusses our static and dynamic measurements.

5.1 Systems

We used three systems with different architectures in our experiments to see whether we would get similar trends in our results. All three systems run Linux Debian Stable (kernel 2.4.20). The three systems are listed below:

- Intel system
  - Pentium 4 1.80GHz CPU
  - 512Mb of RAM.
- AMD system
  - Athlon MP 2000+ 1.66GHz CPU (dual-processor)

  - 2Gb of RAM

- PowerPC system
  - 533MHz CPU
  - 1152Mb of RAM

5.2 Jikes RVM and Related Tools

We used the development version of SOOT (revision 1621) to perform the side-effect analysis and annotate class files. We modified Jikes RVM version 2.3.0.1 to read in the side-effect attributes and use it in the optimizations described in the previous chapter. 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). For our experiments, Jikes RVM was configured to run on a single processor machine.

To build Jikes RVM, various third-party tools are required. Below is a list of the versions that we used:

- classpath 0.06
- Sun JDK 1.4.2-b28 (for Intel and AMD systems)
- JRE Blackdown-1.3.1-02b-FCS (for PowerPC system)
- jikes 1.15
- gcc 2.95.4
- g++ 2.95.4

Benchmark	Description
compress	Lempel-Ziv compressor/uncompressor
jess	A Java expert shell system based on NASA's CLIPS system
raytrace	Ray tracer application
db	Performs several database functions on a memory-resident database
javac	JDK 1.0.2 Java compiler
mpegaudio	MPEG-3 audio file compression application
mtrt	Dual-threaded version of raytrace
jack	A Java parser generator with lexical analyzers (now Java CC)

Table 5.1: Benchmark Description

5.3 Benchmarks

For our experiments, we used the SpecJVM98 [spe] benchmarks. A description of the benchmarks is given in Table 5.1.

We ran each benchmark using size 100 with Jikes RVM at optimization level 1 and 2 using the six side-effect variations described in Section 3.5. Tables 5.2 and 5.3 show, for each benchmark at optimization level 1 and 2 respectively, the load density measure (number of memory reads performed per second). This metric shows how important memory operations are for each benchmark. We expect the benchmarks with high load densities, *compress*, *raytrace*, *mtrt* and *mpegaudio*, to benefit most from side-effect analysis. For these benchmarks, we also show profiling information gathered using Jikes RVM profiling option on our Intel system in Tables 5.4 to 5.8. We see in Table 5.4 that for *compress*, the first two methods account for over 70% of the execution time for both level 1 and 2. For *raytrace* and *mtrt* (Tables 5.5 and 5.6), the four methods shown account for about half of the runtime. Profiling information for *mpegaudio* at level 1 and 2 is split into two tables since the methods are different (Tables 5.7 and 5.8).

	Load density in 1000's		
	Level 1		
Benchmark	AMD	Intel	PowerPC
compress	207383	206708	95041
jess	56371	46199	21226
raytrace	106271	67351	41054
db	7140	7273	5394
javac	21645	13906	8792
mpegaudio	82137	57285	30721
mtrt	92599	61446	36338
jack	14632	8460	5506

Table 5.2: Benchmarks Load Density Property at Level 1

	Load density in 1000's		
	Level 2		
Benchmark	AMD	Intel	PowerPC
compress	138570	126339	86146
jess	68353	55210	26617
raytrace	127806	79806	49914
db	11776	12081	9161
javac	19208	12532	7738
mpegaudio	179070	114851	79647
mtrt	122821	75566	47422
jack	15240	8761	5761

Table 5.3: Benchmarks Load Density Property at Level 2

	% of Execution	
Method	Level 1	Level 2
void Compressor.compress()	53.8 %	50.5 %
void Decompressor.decompress()	20.7 %	20.8 %
void Compressor.output(int)	7.4 %	7.4 %
int Decompressor.getcode()	6.0 %	6.0 %

Table 5.4: Profiling Information for Benchmark Compress on Intel System

	% of Execution	
Method	Level 1	Level 2
OctNode OctNode.Intersect(Ray, Point, float)	21.5 %	18.4 %
boolean PolyTypeObj.Intersect(Ray, IntersectPt)	20.8 %	17.5 %
OctNode OctNode.FindTreeNode(Point)	15.4 %	11.1 %
boolean IntersectPt.FindNearestIsect(OctNode, Ray, int, int, OctNode)	3.2 %	2.9 %

Table 5.5: Profiling Information for Benchmark Raytrace on Intel System

	% of Execution	
Method	Level 1	Level 2
OctNode OctNode.Intersect(Ray, Point, float)	19.9 %	17.2 %
boolean PolyTypeObj.Intersect(Ray, IntersectPt)	19.8 %	17.2 %
OctNode OctNode.FindTreeNode(Point)	13.9 %	11.1 %
boolean IntersectPt.FindNearestIsect(OctNode, Ray, int, int, OctNode)	2.5 %	2.3 %

Table 5.6: Profiling Information for Benchmark Mtrt on Intel System

Method	% of Execution Level 1
void q.m(float[], float[])	29.6 %
boolean ub.(g)	4.7 %
void p.e(int[], g, short[][], int)	3.2 %
boolean cb.(g)	2.8 %
void tb. T(float[], float[], float[])	2.4 %
void d.I(int[], int, int, float[], int)	2.2 %
int lb.read(byte[], int, int)	1.7 %

Table 5.7: Profiling Information for Mpegaudio on Intel System at Level 1

Method	% of Execution Level 2
int q.l(short[], int)	27.8 %
void tb. T(float[], float[], float[])	13.3 %
void q.m(float[], float[])	12.3 %
void p.e(int[], g, short[][], int)	3.2 %
boolean cb.(g)	3.0 %
boolean ub.(g)	2.7 %
void tb. S(float[], float[])	2.7 %
void tb. W(float[], float[])	2.3 %
int lb.read(byte[], int, int)	2.1 %
void d.I(int[], int, int, float[], int)	1.8 %
void p.g(int[], g, int[], cb[])	1.4 %
int q.o(short[], int, float[][], float[][])	1.2 %

Table 5.8: Profiling Information for Mpegaudio on Intel System at Level 2

5.4 Measurements

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. Our third objective was to find out in the code where side-effect analysis makes a difference. We thus looked at the benchmarks that benefited from side-effect analysis, and in the methods that account for a high percentage of the execution time (given by the profiling information in the previous section), we disabled the use of side-effect information in those methods only and computed running times. We analyzed the difference in speedups, as well as static and dynamic counts, and looked at the methods and optimizations that made a difference. Chapter 7 provides a detailed analysis.

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. In the next two chapters, we present our results for local and global optimizations.

Chapter 6
Impact on Optimizations

In this Chapter, we show our static and dynamic measurements of the use of side-effect information in JIT optimizations. Sections 6.1 and 6.2 discuss our results for local and global optimizations.

6.1 Local Optimizations

Level 1 optimizations in Jikes RVM include standard optimizations such as local copy propagation, local constant propagation, local common sub-expression 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 redundant *get-field* and *getstatic* instructions.

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 run-times. To avoid repeating identical results, we grouped these five side-effect variations under the name any in the side-effect column of Tables 6.1 to 6.3. As expected, the execution times of runs using these five side-effect variations were almost identical. We thus also grouped them under any in the second column of Tables 6.4 and 6.5, and reported the average execution times of runs using these five side-effect variations. The values in brackets in these tables denote the percentage increase in static opportunities (Table 6.1) or the percentage decrease in dynamic counts (Tables 6.2 and 6.3) when compared with the none side-effect variation.

The last column of Table 6.1 shows that using side-effect information in local CSE increased the total number of static opportunities for load elimination by 2% to 41%. We note that most of these eliminated loads are *getfields*. Except for *mpegaudio*, there is only 0 or 1 *getstatic* instructions eliminated for each benchmark using the original local CSE algorithm, and 1 to 3 additional ones eliminated using side-effect information. Local CSE thus affects mostly *getfield* instructions. Since it has little impact on *getstatic* instructions, not surprisingly, the use of side-effect analysis had little effect on these instructions as well.

In Table 6.2, we see that the additional loads eliminated using side-effect analysis in local CSE resulted in a decrease of up to 0.90% of dynamic *getfields*, 0.0% of *getstatic* instructions, and 0.87% in total (Table 6.3). 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 of these systems (Tables 6.4 and 6.5). 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. On our PowerPC system, the use of side-effect information had no effect on runtime (Table 6.6). We note from Table 5.2 that the load density property on our PowerPC system is significantly smaller than on our AMD and Intel systems, and thus we conclude that the removal of loads is less beneficial on this slower machine.

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.

Benchmark	Side-effect	Local CSE Performed		
		getfield	*getstatic*	Total
compress	none	108	1	109
	any	112 (3.70 %)	2 (100.00 %)	114 (4.59 %)
jess	none	229	0	229
	any	245 (6.99 %)	1	246 (7.42 %)
raytrace	none	166	0	166
	any	188 (13.25 %)	1	189 (13.86 %)
db	none	130	0	130
	any	133 (2.31 %)	3	136 (4.62 %)
javac	none	415	0	415
	any	431 (3.86 %)	1	432 (4.10 %)
mpegaudio	none	340	174	514
	any	347 (2.06 %)	176 (1.15 %)	523 (1.75 %)
mtrt	none	166	0	166
	any	188 (13.25 %)	1	189 (13.86 %)
jack	none	470	1	471
	any	663 (41.06 %)	2 (100.00 %)	665 (41.19 %)

Table 6.1: Level 1 Static Counts for Local CSE with % Increase Using Side-Effects

Benchmark	Side-effect	*getfield*	*getstatic*
compress	none	1871398009	33418641
	any	1871397929 (0.00 %)	33418641
jess	none	209404162	2326905
	any	209402840 (0.00 %)	2326905
raytrace	none	287993152	1359
	any	287979508 (0.00 %)	1359
db	none	160088294	96012
	any	160087709 (0.00 %)	96012
javac	none	149595624	4028976
	any	149407295 (0.13 %)	4028946 (0.00 %)
mpegaudio	none	456136442	52215347
	any	455026631 (0.24 %)	52215346 (0.00 %)
mtrt	none	291501667	2063
	any	291474379 (0.01 %)	2063
jack	none	50029731	1534965
	any	49579043 (0.90 %)	1534977 (0.00 %)

Table 6.2: Level 1 Dynamic Load Counts with % Reduction Using Side-Effects

Benchmark	Side-effect	Total
compress	none	1904816650
	any	1904816570 (0.00 %)
jess	none	211731067
	any	211729745 (0.00 %)
raytrace	none	287994511
	any	287980867 (0.00 %)
db	none	160184306
	any	160183721 (0.00 %)
javac	none	153624600
	any	153436241 (0.12 %)
mpegaudio	none	508351789
	any	507241977 (0.22 %)
mtrt	none	291503730
	any	291476442 (0.01 %)
jack	none	51564696
	any	51114020 (0.87 %)

Table 6.3: Level 1 Dynamic Total Counts with % Reduction Using Side-Effects

Benchmark	Side-effect	Time (s)	Speedup
compress	none	9.215	
	any	9.395	0.98x
jess	none	4.583	
	any	4.615	0.99x
raytrace	none	4.276	
	any	4.198	1.02x
db	none	22.023	
	any	22.054	1.00x
javac	none	11.047	
	any	11.215	0.99x
mpegaudio	none	8.874	
	any	8.219	1.08x
mtrt	none	4.744	
	any	4.727	1.00x
jack	none	6.095	
	any	6.108	1.00x

Table 6.4: Level 1 Running Time on Intel

Benchmark	Side-effect	Time (s)	Speedup
compress	none	9.185	
	any	9.184	1.00x
jess	none	3.756	
	any	3.77	1.00x
raytrace	none	2.71	
	any	2.662	1.02x
db	none	22.434	
	any	22.453	1.00x
javac	none	7.097	
	any	7.177	0.99x
mpegaudio	none	6.189	
	any	5.85	1.06x
mtrt	none	3.148	
	any	3.087	1.02x
jack	none	3.524	
	any	3.509	1.00x

Table 6.5: Level 1 Running Time on AMD

Benchmark	Side-effect	Time (s)	Speedup
compress	none	20.069	
	any	20.089	1.00x
jess	none	9.975	
	any	9.974	1.00x
raytrace	none	6.985	
	any	6.991	1.00x
db	none	29.851	
	any	29.762	1.00x
javac	none	17.537	
	any	17.467	1.00x
mpegaudio	none	16.552	
	any	16.557	1.00x
mtrt	none	7.454	
	any	7.446	1.00x
jack	none	9.397	
	any	9.387	1.00x

Table 6.6: Level 1 Running Time on PowerPC

6.2 Global Optimizations

The more advanced and expensive analyses and optimizations in Jikes RVM are level 2 optimizations. They include redundant branch elimination, heap SSA construction, redundant load elimination, coalescing after heap SSA, expression folding, loop-invariant code motion, global CSE, transforming *while* into *until* loops, and loop unrolling. As described in Chapter 4, we used side-effect information in the heap SSA construction, RLE and LICM.

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 6.7 to 6.16 of this chapter. In Tables 6.7 to 6.10, the value in brackets represents the percentage increase in static opportunities (the base is the value for the none side-effect variation). For Tables 6.11 to 6.13, it is the percentage reduction in dynamic loads. In Tables 6.14 to 6.16, the reported time for PTA is the average runtime of the four variations above.

The following three sections present our static and dynamic measurements. Sections 6.2.1 and 6.2.2 discuss the static counts for the RLE and LICM optimizations. In Section 6.2.3, we present our dynamic results which include the speedups obtained.

6.2.1 Redundant Load Elimination (RLE)

Table 6.7 shows that the use of side-effect information improved the removal of *getfield* instructions by up to 79% statically. It also significantly increased the static number of opportunities for eliminating *aload* (array load) bytecodes for benchmarks *jess*, *raytrace*, *javac*, *mpegaudio* and *mtrt*. However, as was the case for local optimizations, RLE does not affect many *getstatic* instructions, and thus there were very few improvements for removing these operations using side-effect analysis. Table 6.8 shows that using side-effect information in RLE increased the total number of load eliminations performed by 7% to 98%. Interestingly, PTA improved over CHA for all benchmarks except *jack*.

Benchmark	Side-effect	Load elimination performed		
		getfield	*getstatic*	*aload*
compress	none	359	4	0
	CHA	386 (7.52 %)	5 (25.00 %)	0
	PTA	388 (8.08 %)	5 (25.00 %)	0
jess	none	722	1	129
	CHA	1050 (45.43 %)	2 (100.00 %)	149 (15.50 %)
	PTA	1106 (53.19 %)	3 (200.00 %)	196 (51.94 %)
raytrace	none	342	1	32
	CHA	613 (79.24 %)	2 (100.00 %)	84 (162.50 %)
	PTA	613 (79.24 %)	2 (100.00 %)	127 (296.88 %)
db	none	243	1	2
	CHA	274 (12.76 %)	4 (300.00 %)	2
	PTA	274 (12.76 %)	4 (300.00 %)	3 (50.00 %)
javac	none	1519	26	90
	CHA	1842 (21.26 %)	30 (15.38 %)	101 (12.22 %)
	PTA	1847 (21.59 %)	30 (15.38 %)	108 (20.00 %)
mpegaudio	none	706	212	367
	CHA	804 (13.88 %)	216 (1.89 %)	370 (0.82 %)
	PTA	804 (13.88 %)	216 (1.89 %)	426 (16.08 %)
mtrt	none	342	1	32
	CHA	613 (79.24 %)	2 (100.00 %)	84 (162.50 %)
	PTA	613 (79.24 %)	2 (100.00 %)	127 (296.88 %)
jack	none	678	2	69
	CHA	999 (47.35 %)	16 (700.00 %)	69
	PTA	999 (47.35 %)	16 (700.00 %)	69

Table 6.7: Level 2 Static Counts for RLE with % Increase Using Side-Effects

Benchmark	Side-effect	Load elimination performed
		Total
compress	none	363
	CHA	391 (7.71 %)
	PTA	393 (8.26 %)
jess	none	852
	CHA	1201 (40.96 %)
	PTA	1305 (53.17 %)
raytrace	none	375
	CHA	699 (86.40 %)
	PTA	742 (97.87 %)
db	none	246
	CHA	280 (13.82 %)
	PTA	281 (14.23 %)
javac	none	1635
	CHA	1973 (20.67 %)
	PTA	1985 (21.41 %)
mpegaudio	none	1285
	CHA	1390 (8.17 %)
	PTA	1446 (12.53 %)
mtrt	none	375
	CHA	699 (86.40 %)
	PTA	742 (97.87 %)
jack	none	749
	CHA	1084 (44.73 %)
	PTA	1084 (44.73 %)

Table 6.8: Level 2 Static Total Count for RLE with % Increase Using Side-Effects

6.2.2 Loop-Invariant Code Motion (LICM)

In Tables 6.9 and 6.10, we show static counts of instructions moved during LICM. In Table 6.9, we have counts for *getfield*, *getstatic* and *putfield* instructions. The table does not contain information for *putstatic*, *aload* or *astore* bytecodes since none of these were moved during LICM. We see that the use of side-effect analysis enabled an increase in the number of moved *getfields* by up to 19%, and in one case of a *putfield*. Table 6.10 shows the total number of instructions moved when LICM is performed on high-level (HIR) and low-level (LIR) intermediate representation in Jikes RVM. The table illustrates that using side-effect analysis increased the total number of HIR instructions moved by up to 14%. For one benchmark (*jess*), using PTA side-effect analysis allowed more instructions to be moved than CHA. Since memory instructions are not moved during LICM on LIR, and that in some cases we see an increased in LIR instructions moved, this suggests that, 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 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.

Benchmark	Side-effect	*getfield*	*getstatic*	*putfield*
compress	none	87	0	1
	any	90 (3.45 %)	0	1
jess	none	139	0	0
	CHA	144 (3.60 %)	0	0
	PTA	161 (15.83 %)	0	0
raytrace	none	87	0	47
	any	96 (10.34 %)	0	47
db	none	61	0	0
	any	64 (4.92 %)	0	0
javac	none	44	0	5
	any	48 (9.09 %)	0	6 (20.00 %)
mpegaudio	none	128	27	1
	any	152 (18.75 %)	27	1
mtrt	none	87	0	47
	any	96 (10.34 %)	0	47
jack	none	23	0	2
	any	23	0	2

Table 6.9: Level 2 Static Counts for LICM with % Increase Using Side-Effects

Benchmark	Side-effect	Total HIR	Total LIR
compress	none	118	29
	any	122 (3.39 %)	29
jess	none	280	250
	CHA	287 (2.50 %)	251 (0.40 %)
	PTA	309 (10.36 %)	255 (2.00 %)
raytrace	none	184	54
	any	210 (14.13 %)	56 (3.70 %)
db	none	88	31
	any	92 (4.55 %)	32 (3.23 %)
javac	none	116	479
	any	121 (4.31 %)	479
mpegaudio	none	299	98
	any	327 (9.36 %)	102 (4.08 %)
mtrt	none	184	55
	any	210 (14.13 %)	57 (3.64 %)
jack	none	39	58
	any	39	58

Table 6.10: Level 2 Static Total Count for LICM with % Increase Using Side-Effects

6.2.3 Dynamic Measurements

Tables 6.11 and 6.12 show 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%. Level 2 optimizations using side-effect information reduced total dynamic load operations in the range of 1% to 19% (Table 6.13). For most benchmarks, using PTA side-effect information allowed a larger reduction of dynamic loads than CHA.

Tables 6.14 and 6.15 show speedups achieved for *compress*, *raytrace*, *mtrt* and *mpegaudio*. For these benchmarks, the speedups vary from 1.08x to 1.17x on our Intel system, and from 1.02x to 1.20x on our AMD machine. On both systems, *mpegaudio* has the largest speedup. These benchmarks are also the ones with the highest load densities (Table 5.3), and the ones that we expected would benefit the most from side-effect information. For our PowerPC system, we did not obtain any speedup (Table 6.16). However, we note that the load density value of each benchmark (Table 5.3) is much smaller for our PowerPC machine than for our AMD and Intel systems, and thus the removal of loads has less impact for this slower machine.

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 our Intel system, and 1.02x to 1.05x on our AMD one. For *mpegaudio,* it went from 1.11x to 1.17x on our Intel machine and from 1.15x to 1.20x on our AMD machine.

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 and produced in some cases improvements in runtime performance compared to only using CHA, our simple side-effect analysis variation that does not make use of points-to information. Finally, the differences between points-to analysis variations are negligible.

Benchmark	Side-effect	*getfield*	*getstatic*
compress	none	836681238	29585886
	CHA	713879612 (14.68 %)	29585886
	PTA	694156483 (17.03 %)	29585886
jess	none	193400124	2326905
	CHA	177280681 (8.33 %)	2326905
	PTA	141340271 (26.92 %)	2326572 (0.01 %)
raytrace	none	278990954	1359
	CHA	217369769 (22.09 %)	1359
	PTA	217369769 (22.09 %)	1359
db	none	160085986	96012
	CHA	154814883 (3.29 %)	96012
	PTA	154814883 (3.29 %)	96012
javac	none	129704466	3728755
	CHA	123962720 (4.43 %)	3726381 (0.06 %)
	PTA	123962933 (4.43 %)	3726306 (0.07 %)
mpegaudio	none	258084245	16092989
	CHA	254421559 (1.42 %)	16075411 (0.11 %)
	PTA	254421559 (1.42 %)	16075411 (0.11 %)
mtrt	none	282145314	2063
	CHA	220136202 (21.98 %)	2063
	PTA	220136202 (21.98 %)	2063
jack	none	46154208	1534965
	CHA	42805654 (7.26 %)	1530924 (0.26 %)
	PTA	42805654 (7.26 %)	1530924 (0.26 %)

Table 6.11: Level 2 Dynamic Counts for *getfield* and *getstatic* Instructions with % Reduction Using Side-Effects

Benchmark	Side-effect	*aload*
compress	none	450569851
	CHA	450569851
	PTA	450569851
jess	none	74199530
	CHA	74197591 (0.00 %)
	PTA	74188965 (0.01 %)
raytrace	none	70558731
	CHA	70189162 (0.52 %)
	PTA	70125938 (0.61 %)
db	none	113165950
	CHA	113165950
	PTA	113165950
javac	none	3947221
	CHA	3947158 (0.00 %)
	PTA	3947133 (0.00 %)
mpegaudio	none	796126083
	CHA	794492856 (0.21 %)
	PTA	773557981 (2.83 %)
mtrt	none	71578275
	CHA	71124467 (0.63 %)
	PTA	70998019 (0.81 %)
jack	none	5727775
	CHA	5727775
	PTA	5727775

Table 6.12: Level 2 Dynamic Count for *aload* Instructions with % Reduction Using Side-Effects

Benchmark	Side-effect	Total
compress	none	1316836975
	CHA	1194035349 (9.33 %)
	PTA	1174312220 (10.82 %)
jess	none	269926559
	CHA	253805177 (5.97 %)
	PTA	217855808 (19.29 %)
raytrace	none	349551044
	CHA	287560290 (17.73 %)
	PTA	287497066 (17.75 %)
db	none	273347948
	CHA	268076845 (1.93 %)
	PTA	268076845 (1.93 %)
javac	none	137380442
	CHA	131636259 (4.18 %)
	PTA	131636372 (4.18 %)
mpegaudio	none	1070303317
	CHA	1064989826 (0.50 %)
	PTA	1044054951 (2.45 %)
mtrt	none	353725652
	CHA	291262732 (17.66 %)
	PTA	291136284 (17.69 %)
jack	none	53416948
	CHA	50064353 (6.28 %)
	PTA	50064353 (6.28 %)

Table 6.13: Level 2 Dynamic Loads Total Count with % Reduction Using Side-Effects

Benchmark	Side-effect	Time (s)	Speedup
compress	none	10.423	
	CHA	9.635	1.08x
	PTA	9.386	1.11x
jess	none	4.889	
	CHA	4.945	0.99x
	PTA	4.872	1.00x
raytrace	none	4.38	
	CHA	3.93	1.11x
	PTA	3.905	1.12x
db	none	22.625	
	CHA	22.605	1.00x
	PTA	22.471	1.01x
javac	none	10.962	
	CHA	11.138	0.98x
	PTA	11.142	0.98x
mpegaudio	none	9.319	
	CHA	8.41	1.11x
	PTA	7.932	1.17x
mtrt	none	4.681	
	CHA	4.201	1.11x
	PTA	4.208	1.11x
jack	none	6.097	
	CHA	6.122	1.00x
	PTA	6.101	1.00x

Table 6.14: Level 2 Running Time on Intel

Benchmark	Side-effect	Time (s)	Speedup
compress	none	9.503	
	CHA	9.316	1.02x
	PTA	9.03	1.05x
jess	none	3.949	
	CHA	3.962	1.00x
	PTA	4.002	0.99x
raytrace	none	2.735	
	CHA	2.607	1.05x
	PTA	2.615	1.05x
db	none	23.212	
	CHA	23.222	1.00x
	PTA	23.141	1.00x
javac	none	7.154	
	CHA	7.21	0.99x
	PTA	7.231	0.99x
mpegaudio	none	5.977	
	CHA	5.175	1.15x
	PTA	4.987	1.20x
mtrt	none	2.88	
	CHA	2.788	1.03x
	PTA	2.796	1.03x
jack	none	3.505	
	CHA	3.47	1.01x
	PTA	3.51	1.00x

Table 6.15: Level 2 Running Time on AMD

Benchmark	Side-effect	Time (s)	Speedup
compress	none	15.446	
	CHA	15.53	0.99x
	PTA	15.375	1.00x
jess	none	9.829	
	CHA	9.817	1.00x
	PTA	9.841	1.00x
raytrace	none	6.878	
	CHA	6.916	0.99x
	PTA	6.914	0.99x
db	none	29.695	
	CHA	29.649	1.00x
	PTA	29.668	1.00x
javac	none	17.69	
	CHA	17.887	0.99x
	PTA	17.729	1.00x
mpegaudio	none	13.503	
	CHA	13.485	1.00x
	PTA	13.464	1.00x
mtrt	none	7.325	
	CHA	7.333	1.00x
	PTA	7.362	0.99x
jack	none	9.795	
	CHA	9.811	1.00x
	PTA	9.788	1.00x

Table 6.16: Level 2 Running Time on PowerPC

Chapter 7
Analysis of Speedups

In this chapter, we analyze the benchmarks where significant speedups were obtained in local and global optimizations (Chapter 6). We look at the methods causing these speedups and where in the optimizations the use of side-effect information benefited. The following three sections discuss speedups obtained for the benchmarks *compress*, *mpegaudio* and *raytrace/mtrt*.

7.1 Compress

In Section 6.2, we saw that the use of side-effect information resulted in speedups of up to 1.11x for benchmark *compress*. The following section provides an analysis of the methods and optimizations producing these speedups. Section 7.1.2 shows the changes in the original code that caused these runtime improvements.

7.1.1 Methods and Optimizations Causing Speedups

In Table 5.4 of Chapter 5, profiling information is shown for the benchmark *compress*. From this table, we see that for level 2 optimizations, methods `Compressor.compress()` and `Decompressor.decompress()` account for more than 70% of the execution time. To find out where in the code the use of side-effect information produced speedups for *compress* in the range of 1.08x to 1.11x on Intel and 1.02x to 1.05x on AMD (Tables 6.14

and 6.15), we disabled the use of side-effect analysis in these two methods and computed runtime.

Table 7.1 shows that when the use of side-effect analysis is disabled in both methods `Compressor.compress()` and `Decompressor.decompress()`, the speedups on Intel go down from 1.08x to 1.01x and 1.11x to 1.01x for the CHA and PTA side-effect variations respectively (first row compared with second row). On AMD, the speedups decrease from 1.02x to 1.00x and 1.05x to 1.00x (Table 7.2). Thus, as expected by the profiling information, using side-effect information in these two methods is responsible for most of the speedups. When side-effect analysis is disabled only in method `Compressor.compress()`, we get speedups of 1.01x and 1.02x on Intel, and 0.99x and 1.02x on AMD (third row in Tables 7.1 and 7.2). When it is disabled only in method `Decompressor.decompress()`, the speedups are 1.08x on Intel and 1.04x on AMD (fourth row). These results show that having side-effects in method `Compressor.compress()` is the main cause of the speedups. Since the speedups are the same for the CHA and PTA side-effect variations when they are disabled in method `Decompressor.decompress()`, this method is responsible for the difference in speedups between these two side-effect variations (1.08x versus 1.11x on Intel, 1.02x versus 1.05x on AMD).

Local CSE, redundant load elimination and LICM are the three optimizations that were modified to take advantage of side-effect information. To find out which ones are responsible for the speedups, we disabled the use of side-effect analysis in these optimizations separately for methods `Compressor.compress()` and `Decompressor.decompress()`. For LICM, our results showed that the speedups stayed about the same [1]. Thus, having side-effect information in LICM does not affect the speedups obtained for *compress*. This is also confirmed by the static and dynamic counts that were unchanged. Tables 7.3 and 7.4 show the speedups when not using side-effect analysis in both local CSE and RLE. In this case, we see from these two tables that the loads eliminated using side-effect analysis in local CSE and RLE affect significantly the speedups. Comparing the first row with the third row in these two tables shows that having side-effect information in local CSE and RLE for method `Compressor.compress()` caused most of the speedups. The fourth row

[1]full results are in Appendix A, Tables A.1 and A.2

Methods without side-effects	Side-effect used in other methods	Time(s)	Speedup
none	none	9.751	
	CHA	9.049	1.08x
	PTA	8.769	1.11x
void Compressor.compress() void Decompressor.decompress()	none	9.747	
	CHA	9.678	1.01x
	PTA	9.654	1.01x
void Compressor.compress()	none	9.742	
	CHA	9.657	1.01x
	PTA	9.544	1.02x
void Decompressor.decompress()	none	9.757	
	CHA	9.01	1.08x
	PTA	9.05	1.08x

Table 7.1: Level 2 Runtime without Side-Effects in Selected Methods of Compress on Intel

Methods without side-effects	Side-effect used in other methods	Time(s)	Speedup
none	none	9.514	
	CHA	9.312	1.02x
	PTA	9.026	1.05x
void Compressor.compress() void Decompressor.decompress()	none	9.516	
	CHA	9.505	1.00x
	PTA	9.491	1.00x
void Compressor.compress()	none	9.532	
	CHA	9.64	0.99x
	PTA	9.356	1.02x
void Decompressor.decompress()	none	9.504	
	CHA	9.144	1.04x
	PTA	9.137	1.04x

Table 7.2: Level 2 Runtime without Side-Effects in Selected Methods of Compress on AMD

of these tables shows that the difference in speedups between the CHA and PTA side-effect variations is due to loads eliminated in method `Decompressor.decompress()` (since when the use of side-effects is disabled in local CSE and RLE for this method, speedups for these two variations are the same).

Methods without side-effects in LCSE & RLE	Side-effect used in other methods	Time(s)	Speedup
none	none	9.751	
	CHA	9.049	1.08x
	PTA	8.769	1.11x
void Compressor.compress() void Decompressor.decompress()	none	9.797	
	CHA	9.776	1.00x
	PTA	9.759	1.00x
void Compressor.compress()	none	9.811	
	CHA	9.722	1.01x
	PTA	9.536	1.03x
void Decompressor.decompress()	none	9.805	
	CHA	9.05	1.08x
	PTA	9.042	1.08x

Table 7.3: Level 2 Runtime without Side-Effects in LCSE and RLE for Compress on Intel

In Tables 7.5 and 7.6, we show the effect of disabling side-effect analysis on the static counts of loads eliminated in the redundant load elimination optimization and on the dynamic counts of *getfields* performed. Counts for *getstatic* and *aload* instructions are not shown since they are not affected. The third row compared with the first row in Table 7.5 shows that when side-effect information is disabled in method `Compressor.compress()`, there is a reduction of five *getfields* eliminated statically (381 versus 386 for CHA and 383 versus 388 for PTA). This results in a decrease of dynamic *getfields* eliminated from 14.68% to 6.80% and 17.03% to 9.16% for the CHA and PTA side-effect variations (Table 7.6, row 1 and 3). Since we saw that the effect on speedups is a decrease from 1.08x to 1.01x and from 1.11x to 1.03x for CHA and PTA on Intel (Table 7.3, row 1 and 3) , and

Methods without side-effects in LCSE & RLE	Side-effect used in other methods	Time(s)	Speedup
none	none	9.514	
	CHA	9.312	1.02x
	PTA	9.026	1.05x
void Compressor.compress() void Decompressor.decompress()	none	9.459	
	CHA	9.544	0.99x
	PTA	9.532	0.99x
void Compressor.compress()	none	9.461	
	CHA	9.726	0.97x
	PTA	9.427	1.00x
void Decompressor.decompress()	none	9.467	
	CHA	9.113	1.04x
	PTA	9.105	1.04x

Table 7.4: Level 2 Runtime without Side-Effects in LCSE and RLE for Compress on AMD

1.02x to 0.97x and 1.05x to 1.00x on AMD (Table 7.4, row 1 and 3), the removal of only few additional *getfields* (five statically) is responsible for almost all of the speedups.

In Table 7.5, we note that, when comparing row 1 and 4, using side-effect information in method `Decompressor.decompress()` allowed the elimination of 1 and 3 more loads for the CHA and PTA variations respectively. The two additional loads eliminated using the more precise side-effect variation (PTA) resulted in a larger reduction of dynamic *getfield* instructions from 14.68% to 17.03% (Table 7.6, row 1), and produced an increase in speedups from 1.08x to 1.11x on Intel (Table 7.3, row 1), and from 1.02x to 1.05x on AMD (Table 7.4, row 1).

Methods without side-effects in LCSE & RLE	Side-effect in other methods	*getfield*	*getstatic*	*aload*
none	none	359	4	0
	CHA	386	5	0
	PTA	388	5	0
void Compressor.compress() void Decompressor.decompress()	none	359	4	0
	CHA	380	5	0
	PTA	380	5	0
void Compressor.compress()	none	359	4	0
	CHA	381	5	0
	PTA	383	5	0
void Decompressor.decompress()	none	359	4	0
	CHA	385	5	0
	PTA	385	5	0

Table 7.5: Level 2 Static Counts without Side-Effects in LCSE and RLE for Compress

Methods without side-effects in LCSE & RLE	Side-effect in other methods	*getfield*
none	none	836681238
	CHA	713879612 (14.68 %)
	PTA	694156483 (17.03 %)
void Compressor.compress() void Decompressor.decompress()	none	836681238
	CHA	789621577 (5.62 %)
	PTA	789621577 (5.62 %)
void Compressor.compress()	none	836681238
	CHA	779760012 (6.80 %)
	PTA	760036882 (9.16 %)
void Decompressor.decompress()	none	836681238
	CHA	723741182 (13.50 %)
	PTA	723741182 (13.50 %)

Table 7.6: Level 2 Dynamic Counts without Side-Effects in LCSE and RLE for Compress

7.1.2 Original Code

In Figure 7.1, we show part of the original code of method `Compressor.compress()`. In the previous section, we saw that there were five additional loads eliminated using side-effect information in this method that was the main cause of the speedups. We list them below:

- *getfield* to `Input` at line 12 is eliminated by a copy of the `Input` *getfield* at line 3
- *getfield* to `htab` at line 16 is eliminated by a copy of the `htab` *getfield* at line 9
- *getfield* to `htab` at line 20 is eliminated by a copy of the `htab` *getfield* at line 16
- methods `htab.of(i)` and `htab.set(i, fcode)` (lines 16 and 20) are both inlined and contain a *getfield* to a `tab` field; the second load (in `htab.set(i, fcode)`) is eliminated by a copy of the first one (in `htab.of(i)`)
- *getfield* to `in_count` at line 22 is eliminated by a copy of the `in_count` *getfield* at line 13

Figure 7.2 shows part of method `Decompressor.decompress()`. We saw that having side-effects in this method allowed one more load to be eliminated using CHA and three more with PTA:

- *getfield* to `Output` at line 24 is eliminated by a copy of the `Output` *getfield* at line 3 (any side-effect variation finds this)
- both calls to method `Output.putbyte(..)` at lines 3 and 24 are inlined and contain *getfields* to `OutBuff` and `OutCnt` fields, both of which are eliminated in the second occurrence of the call (only PTA finds this)

In our side-effect analysis, the elements of an array are considered a (special) field. Without points-to analysis, it is not possible to distinguish different methods writing to different arrays. Since methods `Output.putbyte(..)` (line 3) and `de_stack.push(..)` (line 14) both write to arrays, the CHA side-effect analysis thus conservatively assumes that there is a write-write dependence between these two calls. Thus, the loads to `OutBuff` and `OutCnt` fields can only be eliminated using the PTA variation. However, we note that these two calls write to arrays of different and unrelated types. The write-write dependence could thus be removed if a type analysis on arrays would be added to the CHA side-effect computation. In this case, it would make CHA as good as PTA in finding load removal opportunities.

```
public void compress() {
  ...
  ent = Input.getbyte () ;
  hshift = 0;
  for ( fcode = htab.hsize();  fcode < 65536; fcode *= 2 )
      hshift++;
  hshift = 8 - hshift;
  hsize_reg = htab.hsize();
  htab.clear();

  next_byte:
  while ( (c = Input.getbyte()) != -1) {
    in_count++;
    fcode = (((int) c << maxbits) + ent);
    i = ((c << hshift) ˆ ent);
    int temphtab = htab.of (i);
    ...
    if ( free_ent < maxmaxcode ) {
      codetab.set(i, free_ent++);
      htab.set(i, fcode);
    }
    else if ( (in_count >= checkpoint) && (block_compress != 0) )
      cl_block ();
  }
  ...
}
```

Figure 7.1: Part of Method `Compressor.compress()`

```
public void decompress() {
  ...
  Output.putbyte( (byte)finchar );
  while ( (code = getcode()) > -1 ) {
    if ( (code == Compress.CLEAR) && (block_compress != 0) ) {
      tab_prefix.clear(256);
      clear_flg = 1;
      free_ent = Compress.FIRST - 1;
      if ( (code = getcode ()) == -1 )
        break;
    }
    incode = code;
    if ( code >= free_ent ) {
      de_stack.push((byte)finchar);
      code = oldcode;
    }
    while ( code >= 256 ) {
      de_stack.push(tab_suffix.of(code));
      code = tab_prefix.of(code);
    }
    de_stack.push((byte)(finchar = tab_suffix.of(code)));

    do
      Output.putbyte ( de_stack.pop());
    while ( !de_stack.is_empty());
    ...
  }
}
```

Figure 7.2: Part of Method `Decompressor.decompress()`

7.2 Mpegaudio

In Section 6.1, we saw that we obtained speedups of up to 1.08x for *mpegaudio* in local optimizations. For global optimizations, the use of side-effect information in local CSE, RLE and LICM enabled speedups of up to 1.20x. The following two sections discuss these speedups. Section 7.2.1 provides an analysis of the methods causing speedups in local CSE. Section 7.2.2 discusses which methods and which optimizations benefited from side-effect analysis, and where in the code the use of the most precise side-effect analysis (PTA) produced better runtime improvement over the basic side-effect analysis (CHA).

7.2.1 Local Optimizations

In Section 6.1, we saw that having side-effect information in local optimizations resulted in speedups of 1.08x and 1.06x on our Intel and AMD systems (Tables 6.4 and 6.5). To find out where the use of side-effect analysis in local CSE produced these results, we disabled side-effects in the most frequently executed methods. Table 5.7 shows profiling information for the seven methods that account for the highest percentage of the execution time. Surprisingly, disabling side-effect analysis in these methods did not affect speedups. We thus disabled side-effects in more methods incrementally and found that method `q.o(short[], int, float[][], float[][])`, which account for less than 1% of the execution time, is responsible for all of the speedups. We see in Tables 7.7 and 7.8 (second row) that the speedups on Intel and AMD become null without side-effects in local CSE for method `q.o(..)`. The static counts in Table 7.9 show that this behaviour is due to a single *getfield* that is not eliminated (346 without side-effects versus 347 with). This causes a reduction of dynamic *getfield* instructions by 0.18% compared to 0.24% originally (Table 7.10). Thus, the changes in static and dynamic counts are very minimal, but the impact on runtime is quite large. This is likely due to secondary effects such as cache behaviour or register pressure. Finally, for legal reasons, we are unable to show the original code.

Methods without side-effects in LCSE	Side-effect used in other methods	Time(s)	Speedup
none	none	8.874	
	any	8.219	1.08x
int q.o(short[], int, float[][], float[][])	none	8.878	
	any	8.839	1.00x

Table 7.7: Level 1 Runtime without Side-Effects in Local CSE for Mpegaudio on Intel

Methods without side-effects in LCSE	Side-effect used in other methods	Time(s)	Speedup
none	none	6.189	
	any	5.85	1.06x
int q.o(short[], int, float[][], float[][])	none	6.208	
	any	6.187	1.00x

Table 7.8: Level 1 Runtime without Side-Effects in Local CSE for Mpegaudio on AMD

Methods without side-effects in LCSE	Side-effect in other methods	*getfield*	*getstatic*
none	none	340	174
	any	347 (2.06 %)	176 (1.15 %)
int q.o(short[], int, float[][], float[][])	none	340	174
	any	346 (1.76 %)	176 (1.15 %)

Table 7.9: Level 1 Static Counts without Side-Effects in Local CSE for Mpegaudio

Methods without side-effects in LCSE	Side-effect in other methods	*getfield*
none	none	456136442
	any	455026631 (0.24 %)
int q.o(short[], int, float[][], float[][])	none	456136442
	any	455307827 (0.18 %)

Table 7.10: Level 1 Dynamic Counts Using Side-Effects in Local CSE for Mpegaudio

7.2.2 Global Optimizations

In Section 6.2, we saw that we obtained speedups for *mpegaudio* in the range of 1.11x to 1.17x on Intel and 1.15x to 1.20x on AMD using side-effect analysis. In a similar manner to the previous section, we incrementally disabled side-effects in the most frequently executed methods, and found that the two methods that caused the speedups are `q.o(short[], int, float[][], float[][])` and `q.m(float[], float[])`. The profiling information in Table 5.8 shows that `q.m(float[], float[])` account for 12.3% of the execution time, but `q.o(short[], int, float[][], float[][])`, account for less than 1% (not shown in the table).

Analysis of Method `q.o(short[], int, float[][], float[][])`

We see in the second row of Table 7.11 that on Intel, disabling side-effects in `q.o(..)` results in a slowdown of 0.70x and 0.66x for the CHA and PTA side-effect variations respectively. On our AMD system, the speedups decrease from 1.15x to 1.02x and from 1.20x to 1.01x (Table 7.12, row 1 and 2). The impact of not using side-effect information in `q.o(..)` is thus much larger on the Intel architecture. To see whether this behaviour was caused by LICM or load elimination, we disabled side-effects in each of these optimizations separately. Doing so in LICM made no difference since speedups stayed about the same [2]. However, Tables 7.13 and 7.14 (second row) show that disabling side-effects in

[2] full results are in Appendix A, Tables A.3 and A.4

local CSE and RLE resulted in the same large slowdowns that was obtained in Tables 7.11 and 7.12. Thus, it is the load elimination optimization that is responsible for the slowdowns. Comparing the first and second rows in the static results of Table 7.15 show that only 778 *getfields* are eliminated without using side-effect information in `q.o(..)` versus 804 with side-effects. Note that *getstatic* and *aload* instructions are unaffected. Note that there is no column for *getstatic* since the counts did not change for any rows. The effect on dynamic counts is a reduction by 0.79% of *getfields* versus 1.42% originally (Table 7.16, row 1 and 2). Thus, the removal of additional loads using side-effect analysis in `q.o(..)` has a large impact on runtime. Since the changes statically and dynamically are small, this is likely due to secondary effects such as register pressure or cache behaviour.

Analysis of Method `q.m(float[], float[])`

Tables 7.11 and 7.12 (third row) show that for method `q.m(float[], float[])`, the speedups for CHA and PTA on Intel are 1.11x and 1.10x respectively, and on AMD they are 1.15x and 1.16x. Since the speedups are about the same for CHA and PTA when the use of side-effects is disabled in `q.m(float[], float[])`, having more precise side-effect information in this method is responsible for the better runtime improvement by PTA (1.17x on Intel, 1.20x on AMD) versus CHA (1.11x on Intel, 1.15x on AMD). To see whether this is due to LICM or load elimination, we disabled side-effects in these optimizations separately and computed runtime. Our results show that having or not having side-effects in LICM did not affect the speedups [3]. However, when disabling side-effects in local CSE and RLE, the speedups are 1.10x on Intel and 1.16x on AMD (Tables 7.13 and 7.14, third row). It is thus the load elimination optimization in `q.m(float[], float[])` that caused the difference in the original speedups between CHA and PTA. In Table 7.15 (row 3), the static results show that the *getfield* instruction counts are unaffected by not using side-effects in `q.m(float[], float[])` (same counts as original ones). Though, for *aload* instructions, there are 407 eliminated versus 426 originally with PTA (row 1 and 3). For CHA, having or not having side-effects in `q.m(float[], float[])` did not affect the *aload* counts (370 in both cases). Thus, not using the most precise side-effect

[3]full results are in Appendix A, Tables A.3 and A.4

analysis PTA in `q.m(float[], float[])` reduced by 19 statically the number of *aloads* eliminated. As a result, the reduction dynamically is 1.49% versus 2.83% originally (Table 7.16, row 1 and 3 for PTA). Since the speedups are about the same for CHA and PTA when the use of side-effects is disabled in `q.m(float[], float[])`, and originally it was 1.11x versus 1.17x on Intel and 1.15x versus 1.20x on AMD, the use of most precise side-effect analysis (PTA) in this method is responsible for this difference.

Methods without side-effects	Side-effect used in other methods	Time(s)	Speedup
none	none	9.319	
	CHA	8.41	1.11x
	PTA	7.932	1.17x
int q.o(short[], int, float[][], float[][])	none	9.22	
	CHA	13.177	0.70x
	PTA	13.917	0.66x
void q.m(float[], float[])	none	9.224	
	CHA	8.303	1.11x
	PTA	8.411	1.10x

Table 7.11: Level 2 Runtime without Side-Effects in Selected Methods of Mpegaudio on Intel

Methods without side-effects	Side-effect used in other methods	Time(s)	Speedup
none	none	5.977	
	CHA	5.175	1.15x
	PTA	4.987	1.20x
int q.o(short[], int, float[][], float[][])	none	5.976	
	CHA	5.882	1.02x
	PTA	5.895	1.01x
void q.m(float[], float[])	none	5.977	
	CHA	5.201	1.15x
	PTA	5.131	1.16x

Table 7.12: Level 2 Runtime without Side-Effects in Selected Methods of Mpegaudio on AMD

Methods without side-effects in LCSE & RLE	Side-effect used in other methods	Time(s)	Speedup
none	none	9.319	
	CHA	8.41	1.11x
	PTA	7.932	1.17x
int q.o(short[], int, float[][], float[][])	none	9.223	
	CHA	13.182	0.70x
	PTA	13.914	0.66x
void q.m(float[], float[])	none	9.222	
	CHA	8.412	1.10x
	PTA	8.402	1.10x

Table 7.13: Level 2 Runtime without Side-Effects in LCSE and RLE for Mpegaudio on Intel

Methods without side-effects in LCSE & RLE	Side-effect used in other methods	Time(s)	Speedup
none	none	5.977	
	CHA	5.175	1.15x
	PTA	4.987	1.20x
int q.o(short[], int, float[][], float[][])	none	5.977	
	CHA	5.88	1.02x
	PTA	5.886	1.02x
void q.m(float[], float[])	none	5.974	
	CHA	5.153	1.16x
	PTA	5.139	1.16x

Table 7.14: Level 2 Runtime without Side-Effects in LCSE and RLE for Mpegaudio on AMD

Methods without side-effects in LCSE & RLE	Side-effect in other methods	*getfield*	*aload*
none	none	706	367
	CHA	804 (13.88 %)	370 (0.82 %)
	PTA	804 (13.88 %)	426 (16.08%)
int q.o(short[], int, float[][], float[][])	none	706	367
	CHA	778 (10.20 %)	370 (0.82 %)
	PTA	778 (10.20 %)	426 (16.08%)
void q.m(float[], float[])	none	706	367
	CHA	804 (13.88 %)	370 (0.82 %)
	PTA	804 (13.88 %)	407 (10.90 %)

Table 7.15: Level 2 Static Counts without Side-Effects in LCSE and RLE for Mpegaudio

Methods without side-effects in LCSE & RLE	Side-effect in other methods	*getfield*	*aload*
none	none	258084245	796126083
	CHA	254421559 (1.42 %)	794492856 (0.21 %)
	PTA	254421559 (1.42 %)	773557981 (2.83 %)
int q.o(short[], int, float[][], float[][])	none	258084245	796126083
	CHA	256046247 (0.79 %)	794492856 (0.21 %)
	PTA	256046247 (0.79 %)	773557981 (2.83 %)
void q.m(float[], float[])	none	258084245	796126083
	CHA	254421559 (1.42 %)	794492856 (0.21 %)
	PTA	254421559 (1.42 %)	784243429 (1.49 %)

Table 7.16: Level 2 Dynamic Counts without Side-Effects in LCSE and RLE for Mpegaudio

7.3 Raytrace/Mtrt

In this section, we analyze where side-effect information produced speedups for the benchmarks *raytrace* and *mtrt*. Since *mtrt* is a multi-threaded version of *raytrace*, and that we found that the cause of the speedups was the same for both benchmarks, we will only discuss *raytrace* here. The same analysis applies for *mtrt*.

In Section 6.2, we saw that using side-effect information improved runtime for *raytrace* in the range of 1.11x to 1.12x on Intel and 1.05x on AMD (Tables 6.14 and 6.15). To find out where the use of side-effect information resulted in these speedups, we disabled it in the hot methods given by the profiling information of Table 5.5. To our surprise, the speedups stayed the same. We thus incrementally added methods with side-effects disabled and narrowed our search to method `run()`, which is part of the `Runner` class in the file RayTracer.java. We note that `Runner` is a thread, and that for benchmark *raytrace*, only one `Runner` thread is created to render the scene, whereas two threads are used for *mtrt*.

The code of the `Runner` class is shown in Figure 7.3. When we disabled side-effect information in method `run()`, we found that the main cause of the speedups was the removal of the *getfield* to the `parent` field on line 17. (The `parent.threadCount--` statement on line 17, when transformed to bytecode, performs a *getfield* to a `RayTracer` object, `parent`, in order to decrement the `threadCount` counter.) With side-effect analysis, this load to `parent` can be eliminated by a copy of this field (line 13). When leaving this *getfield* instruction on line 17 (i.e. not replacing it with a cached copy) and applying side-effect analysis everywhere else as originally, the speedups went down from 1.11x to 1.02x on Intel and 1.05x to 1.00x on AMD. We also obtained similar results when removing the entire statement `parent.threadCount--` (line 17). Although we cannot explain this behaviour, we note that this *getfield* instruction is performed to retrieve an object and decrement the `threadCount` counter, which can be manipulated by both the `Runner` and the main threads. This behaviour may thus be due to the way Jikes RVM handles shared objects.

7.4 Summary

In this chapter, we analyzed the methods of the benchmarks where significant speedups were obtained. We found that for *compress*, only five additional loads eliminated using side-effect information was the main cause of the speedups. For *mpegaudio*, we noted that the removal of few additional loads likely caused secondary effects such as register pressure and/or cache behaviour to produce the performance improvements. Finally, for *raytrace/mtrt*, we found that the cause of the speedups was mainly due to an additional load eliminated to a shared object.

```
class Runner extends Thread {
    RayTracer parent;
    int section;
    int nsections;

    public Runner(RayTracer parent, int section, int nsections) {
        this.parent    = parent;
        this.section   = section;
        this.nsections = nsections;
    }

    public void run() {
        new Scene(parent.name).RenderScene(parent.canvas,
                                           parent.width,
                                           section,
                                           nsections);
        parent.threadCount--;
    }
}
```

Figure 7.3: Code of Class `Runner` in Benchmark Raytrace

Chapter 8
Conclusions and Future Work

8.1 Conclusions

This research presented a study of whether the use of inter-procedural side-effect analysis in Java just-in-time (JIT) compilers improves performance. Our experiments showed that relatively simple analyses are sufficient for significant improvements. Our results also showed that the benchmarks with high load densities benefited the most from side-effect information. Among the optimizations adapted to use side-effects, load elimination was the one causing the speedups.

In this thesis, we first reviewed how side-effect analysis is computed ahead-of-time in SOOT, based on different call graph constructions and various points-to analyses. We explained the difference in precision of the various side-effect analyses that we experimented with, and how they can be communicated to JIT compilers through Java class files attributes.

The three optimizations in Jikes RVM that were modified to take advantage of side-effect information are local common-sub-expression, redundant load elimination and loop-invariant code motion. The last two optimizations use the Heap SSA construction [FKS00], which we also adapted to use side-effect analysis. For each of these optimizations, we explained the algorithms, the changes that were made, and showed examples of improvements that are possible with the knowledge of side-effects. We also discussed how JIT inlining decisions affect the use of ahead-of-time side-effect information.

In our experiments, we ran the SpecJVM98 benchmarks on three different architectures (Intel, AMD and PowerPC). In local and global optimizations, we gathered various measurements including static counts of instructions moved during LICM, and the static number of loads eliminated in local CSE and RLE. We also measured the dynamic effects of using side-effect information by computing the reduction in memory reads operations and execution times.

For local optimizations, side-effect analyses had little impact on the static and dynamic counts. Except for one benchmark, the effect on performance was negligible. Since local optimizations are only performed within basic blocks, typically small in Java programs, this behaviour was expected.

In global optimizations, our results showed an increase of up to 98% of static opportunities for load removal and up to 18% of memory reads moved, a reduction of up to 27% of the dynamic fields reads, and execution time speedups of up to 17% on our Intel system and up to 20% on our AMD machine. On PowerPC, no speedups were obtained. However, we noted that our PowerPC machine is significantly slower than our Intel and AMD systems. The load density property of the benchmarks on PowerPC is thus considerably smaller than on Intel and AMD, making the use of side-effect analysis less effective.

Finally, we analyzed the methods and optimizations that were the cause of the speedups obtained for *compress*, *mpegaudio* and *raytrace/mtrt*. We found that for all of these benchmarks, the optimization that was responsible for the speedups was load elimination (LICM had little effect on runtime). We noted that only few additional important loads eliminated (statically) using side-effect analysis was the main cause of the runtime improvements. We also found that the difference in speedups between the CHA and PTA side-effect variations was due to the analysis precision on array reads and writes. For our set of benchmarks, adding a type analysis on arrays would make CHA as good as PTA in finding load removal opportunities.

8.2 Future Work

8.2.1 Experimenting with a Fast PowerPC Machine

The first step in continuing this work would be to perform our experiments and gather measurements on a fast PowerPC machine to see whether we would get speedups comparable to the ones obtained on our Intel and AMD systems. Obtaining significant runtime improvements would strengthen the belief that the use of side-effect information is more effective on fast machines, and thus for benchmarks with high load densities.

8.2.2 Using Context-Sensitive Analyses

In this thesis, the side-effect analyses used were computed using a flow-insensitive, context-insensitive, subset-based points-to analysis. Context-sensitive points-to analyses can produce much more precise information than context-insensitive ones. In an object-oriented language that encourages encapsulation, such as Java, the information lost due to context-insensitivity is especially significant [Lho02]. Context-sensitive points-to analysis is planned to be included in the SOOT framework in the near future [Lho05]. An area for future research would be to perform a similar set of measurements using context-sensitive points-to analyses to compute side-effect information, which would be more precise than our PTA analysis. In the analysis of speedups, we saw that only few additional loads eliminated was responsible for the runtime improvements. Thus, a more precise side-effect analysis may enable the removal of further key loads, leading to even bigger performance gains.

8.2.3 Computing Side-Effects at Runtime

The feasibility of performing side-effect analysis inside the JIT is also a topic for future research. The dynamic call graph construction presented in [QH04, QH05] is a first step in this work. A simple side-effect analysis, similar to our CHA analysis, could be computed using this dynamic call graph to build method summaries of fields read and written. A simple type analysis could be implemented to distinguish reads and writes to unrelated arrays.

8.2.4 Investigating Secondary Effects

In Chapter 7, we found that for benchmark *mpegaudio*, secondary effects such as register pressure and/or cache behaviour likely was the main cause of the performance gains using side-effect information. Studying whether and how the impact of load elimination on caches and register allocation contributed to performance variations is a topic for further investigation.

Appendix A
Miscellaneous Tables

In the following tables, full results are shown for benchmarks that do not make use of side-effects in LICM. Since the speedups stay about the same, side-effect information in LICM has little effect.

Methods without side-effects in LICM	Side-effect used in other methods	Time(s)	Speedup
none	none	9.751	
	CHA	9.049	1.08x
	PTA	8.769	1.11x
void Compressor.compress() void Decompressor.decompress()	none	9.815	
	CHA	9.196	1.07x
	PTA	8.954	1.10x
void Compressor.compress()	none	9.822	
	CHA	9.109	1.08x
	PTA	8.967	1.10x
void Decompressor.decompress()	none	9.807	
	CHA	9.131	1.07x
	PTA	8.93	1.10x

Table A.1: Level 2 Runtime without Side-Effects in LICM for Compress on Intel

Methods without side-effects in LICM	Side-effect used in other methods	Time(s)	Speedup
none	none	9.514	
	CHA	9.312	1.02x
	PTA	9.026	1.05x
void Compressor.compress() void Decompressor.decompress()	none	9.471	
	CHA	9.301	1.02x
	PTA	9.004	1.05x
void Compressor.compress()	none	9.485	
	CHA	9.309	1.02x
	PTA	9.03	1.05x
void Decompressor.decompress()	none	9.487	
	CHA	9.298	1.02x
	PTA	9.023	1.05x

Table A.2: Level 2 Runtime without Side-Effects in LICM for Compress on AMD

Methods without side-effects in LICM	Side-effect used in other methods	Time(s)	Speedup
none	none	9.319	
	CHA	8.41	1.11x
	PTA	7.932	1.17x
int q.o(short[], int, float[][], float[][])	none	9.222	
	CHA	8.329	1.11x
	PTA	7.984	1.16x
void q.m(float[], float[])	none	9.221	
	CHA	8.32	1.11x
	PTA	7.897	1.17x

Table A.3: Level 2 Runtime without Side-Effects in LICM for Mpegaudio on Intel

Methods without side-effects in LICM	Side-effect used in other methods	Time(s)	Speedup
none	none	5.977	
	CHA	5.175	1.15x
	PTA	4.987	1.20x
int q.o(short[], int, float[][], float[][])	none	5.978	
	CHA	5.16	1.16x
	PTA	4.963	1.20x
void q.m(float[], float[])	none	5.976	
	CHA	5.155	1.16x
	PTA	4.924	1.21x

Table A.4: Level 2 Runtime without Side-Effects in LICM for Mpegaudio on AMD

Bibliography

[AAB$^+$00] 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.

[AC76] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Commun. ACM*, 19(3):137, 1976.

[All74] F.E. Allen. Interprocedural data flow analysis. In *Proceedings of the 1974 IFIPS Congress*, pages 398–402. North Holland Publishing Company, Amsterdam, 1974.

[And94] L. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[AU77] A.V. Aho and J.D. Ullman. *Principles of Compiler Design*. Addison-Wesley, 1977.

[AWZ88] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–11. ACM Press, 1988.

[Ban78] John P. Banning. *A Method for Determining the Side Effects of Procedure Calls*. PhD thesis, Stanford University, 1978.

[Ban79] John P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 29–41. ACM Press, 1979.

[Bar77] J.M. Barth. *A practical interprocedural data flow analysis algorithm and its applications*. PhD thesis, University of California, Berkeley, May 1977.

[Bar78] Jeffrey M. Barth. A practical interprocedural data flow analysis algorithm. *Commun. ACM*, 21(9):724–736, 1978.

[BC86] Michael Burke and Ron Cytron. Interprocedural dependence analysis and parallelization. In *SIGPLAN '86: Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, pages 162–175. ACM Press, 1986.

[Bur84] M. Burke. An interval analysis approach toward interprocedural data flow. Technical report, IBM T.J. Watson Research Center RC 10640, July 1984.

[Bur90] Michael Burke. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. *ACM Trans. Program. Lang. Syst.*, 12(3):341–395, 1990.

[CBC93] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 232–245. ACM Press, 1993.

[CDC$^+$04] Rezaul Alam Chowdhury, Peter Djeu, Brendon Cahoon, James H. Burrill, and Kathryn S. McKinley. The limits of alias analysis for scalar optimizations. In Evelyn Duesterwald, editor, *Compiler Construction, 13th International Conference, CC 2004*, volume 2985 of *Lecture Notes in Computer Science*, pages 24–38. Springer, 2004.

[CH00] Ben-Chung Cheng and Wen-Mei W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 57–69. ACM Press, 2000.

[CK84] Keith D. Cooper and Ken Kennedy. Efficient computation of flow insensitive interprocedural summary information. *SIGPLAN Not.*, 19(6):247–258, 1984.

[CK88a] David Callahan and Ken Kennedy. Analysis of interprocedural side effects in a parallel programming environment. *J. Parallel Distrib. Comput.*, 5(5):517–550, 1988.

[CK88b] K. D. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language Design and Implementation*, pages 57–66. ACM Press, 1988.

[Cla97] Lars R. Clausen. A Java bytecode optimizer using side-effect analysis. *Concurrency: Practice and Experience*, 9(11):1031–1045, November 1997.

[Cli95] Cliff Click. Global code motion/global value numbering. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 246–257. ACM Press, 1995.

[CR87] Martin Carroll and Barbara G Ryder. An incremental algorithm for software analysis. In *SDE 2: Proceedings of the second ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 171–179. ACM Press, 1987.

[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP '95, object-oriented programming: 9th European Conference*, volume 952 of *Lecture Notes in Computer Science*, pages 77–101, 1995.

[DLFR01] Manuvir Das, Ben Liblit, Manuel F&#228;hndrich, and Jakob Rehof. Estimating the impact of scalable pointer analysis on optimization. In *SAS '01: Proceedings of the 8th International Symposium on Static Analysis*, pages 260–278, London, UK, 2001. Springer-Verlag.

[DMM98] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 106–117. ACM Press, 1998.

[DMM01] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Using types to analyze and optimize object-oriented programs. *ACM Trans. Program. Lang. Syst.*, 23(1):30–72, 2001.

[DMW98] Saumya Debray, Robert Muth, and Matthew Weippert. Alias analysis of executable code. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–24. ACM Press, 1998.

[EGH94] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 242–256. ACM Press, 1994.

[FKS00] Stephen J. Fink, Kathleen Knobe, and Vivek Sarkar. Unified analysis of array and object references in strongly typed languages. In *Static Analysis Symposium*, pages 155–174, 2000.

[GH98] Rakesh Ghiya and Laurie J. Hendren. Putting pointer analysis to work. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 121–133. ACM Press, 1998.

[GLS01] Rakesh Ghiya, Daniel Lavery, and David Sehr. On the importance of points-to analysis and other memory disambiguation methods for C programs. In

Proceedings of the ACM SIGPLAN'01 Conference on Programming Language Design and Implementation, pages 47–58. ACM Press, 2001.

[HDE$^+$93] Laurie J. Hendren, C. Donawa, Maryam Emami, Guang R. Gao, Justiani, and B. Sridharan. Designing the mccat compiler based on a family of structured intermediate representations. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, pages 406–420. Springer-Verlag, 1993.

[HGMS91] Laurie J. Hendren, Guang R. Gao, Chandrika Mukerji, and Bhama Sridharan, Introducing McCAT - The McGill Compiler Compiler-Architecture Testbed. ACAPS Technical Memo 27, School of Computer Science, McGill University, Montreal, Quebec, September 1991.

[HP95] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach*. Morgan-Kaufmann, 1995.

[HP98] Michael Hind and Anthony Pioli. Assessing the effects of flow-sensitivity on pointer alias analyses. In *SAS '98: Proceedings of the 5th International Symposium on Static Analysis*, pages 57–81. Springer-Verlag, 1998.

[HP00] Michael Hind and Anthony Pioli. Which pointer analysis should I use? In *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 113–123. ACM Press, 2000.

[HS75] M.S. Hecht and J.B. Shaffer. Ideas on the design of a 'quad improver' for simpl-t, part i: Overview and intersegment analysis. Technical report, TR-405, University of Maryland, College Park, Maryland, August 1975.

[LH03] Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using Spark. In G. Hedin, editor, *Compiler Construction, 12th International Conference*, volume 2622 of *LNCS*, pages 153–169, Warsaw, Poland, April 2003. Springer.

[Lho02] Ondřej Lhoták. Spark: A flexible points-to analysis framework for Java. Master's thesis, McGill University, December 2002.

[Lho05] Ondřej Lhoták. *Program Analysis using Binary Decision Diagrams*. PhD thesis, School of Computer Science, McGill University. In preparation, September 2005.

[Lom77] D.B. Lomet, Data Flow Analysis in the Presence of Procedure Calls. IBM Journal of Research and Development, 21(6), page 559-571, November, 1977.

[LRZ93] William Landi, Barbara G. Ryder, and Sean Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pages 56–67. ACM Press, 1993.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.

[MRR02] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 1–11. ACM Press, 2002.

[MSH97] II Marc Shapiro and Susan Horwitz. The effects of the precision of pointer analysis. In *SAS '97: Proceedings of the 4th International Symposium on Static Analysis*, pages 16–34. Springer-Verlag, 1997.

[Mye80] E. Myers. A precise and efficient algorithm for determining existential summary data flow information. Technical report, CU-CS-175-80, University of Colorado at Boulder, Department of Computer Science, 1980.

[Mye81] Eugene M. Myers. A precise inter-procedural data flow algorithm. In *POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 219–230. ACM Press, 1981.

[Oli97] Guirlyn 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`, April 1997.

[PQVR+01] Patrice Pominville, Feng Qian, Raja Vallée-Rai, Laurie Hendren, and Clark Verbrugge. A framework for optimizing Java using attributes. In *Compiler construction: 10th International Conference, CC 2001*, volume 2027 of *Lecture Notes in Computer Science*, pages 334–354, 2001.

[PS02] Igor Pechtchanski and Vivek Sarkar. Immutability specification and its applications. In *Proceedings of the 2002 Joint ACM-ISCOPE Conference on Java Grande*, pages 202–211. ACM Press, 2002.

[QH04] Feng Qian and Laurie J. Hendren. Towards dynamic interprocedural analysis in jvms. In *Virtual Machine Research and Technology Symposium*, pages 139–150, 2004.

[QH05] Feng Qian and Laurie Hendren. A study of type analysis for speculative method inlining in a JIT environment. In *Compiler Construction, 14th International Conference, CC 2005*, Lecture Notes in Computer Science, pages 255–270, Edinburgh, Scotland, April 2005.

[Raz99] Chrislain Razafimahefa. A study of side-effect analyses for Java. Master's thesis, McGill University, December 1999.

[RLS+01] Barbara G. Ryder, William A. Landi, Philip A. Stocks, Sean Zhang, and Rita Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM Transactions on Programming Languages and Systems*, 23(2):105–186, March 2001.

[RMR01] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Points-to analysis for Java using annotated constraints. In *Proceedings of the OOPSLA '01 Conference on Object-Oriented Programming Systems Languages and Applications*, pages 43–55. ACM Press, 2001.

[Ros75] B.K. Rosen. Data flow analysis for recursive pl/i programs. Technical report, RC5211, IBM T.J. Watson Research Center, Yorktown Heights, N.Y., January 1975.

[Ros79] Barry K. Rosen. Data flow analysis for procedural languages. *J. ACM*, 26(2):322–344, 1979.

[RR01] Atanas Rountev and Barbara G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. In *Compiler construction: 10th International Conference, CC 2001*, volume 2027 of *Lecture Notes in Computer Science*, pages 20–36, 2001.

[Ruf95] Erik Ruf. Context-insensitive alias analysis reconsidered. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 13–22. ACM Press, 1995.

[spe] SPEC JVM98 benchmarks. `http://www.spec.org/osg/jvm98/`.

[Spi71] T.C. Spillman. Exposing side-effects in a pl/i optimizing compiler. In *Proceedings of the 1971 IFIPS Congress*, pages 376–381. North Holland Publishing Company, 1971.

[SRLZ98] Philip A. Stocks, Barbara G. Ryder, William A. Landi, and Sean Zhang. Comparing flow and context sensitivity on the modification-side-effects problem. In *Proceedings of ACM SIGSOFT international symposium on Software testing and analysis*, pages 21–31. ACM Press, 1998.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41. ACM Press, 1996.

[TIF86] R. Triolet, F. Irigoin, and P. Feautrier. Direct parallelization of call statements. In *SIGPLAN '86: Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, pages 176–185. ACM Press, 1986.

[VRGH$^+$00] Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing Java bytecode using the Soot framework: is it feasible? In *Compiler Construction, 9th International Conference (CC 2000)*, volume 1781 of *Lecture Notes in Computer Science*, pages 18–34, 2000.