SOFTWARE—PRACTICE AND EXPERIENCE
*Softw. Pract. Exper.* 2004; **01**:1–2 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

# Understanding the behavior of compiler optimizations

Han Lee[1,], Daniel von Dincklage,[1] Amer Diwan,[1,*] and J. Eliot B. Moss[2]

[1] *Department of Computer Science, University of Colorado, Boulder, CO 80309, U.S.A.*
[2] *Department of Computer Science, University of Massachusetts, Amherst, MA 01003, U.S.A.*

**SUMMARY**

**Compiler optimizations are difficult to implement and add complexity to a compiler. For this reason, compiler writers are selective about implementing them: they implement only the ones that they believe will be beneficial. To support compiler writers in this, we describe a method for measuring the cost and benefits of compiler optimizations, both individually and in synergy with other optimizations. We demonstrate our method by presenting results for the optimizations implemented in the Jikes RVM on the PowerPC and IA32 platforms.**

KEY WORDS: Java; compiler optimizations; performance evaluation of optimizations

## 1. INTRODUCTION

Optimizing compilers make many passes over representations of the program being optimized, each time applying one or more analyses or optimizations. Optimizations and analyses performed in a given pass may interact with optimizations and analyses in the same or other passes, either directly (e.g., annotations) or indirectly (e.g., code transformations). These interactions may be positive (e.g., one optimization may expose opportunities for another optimization) or negative (e.g., one optimization may eliminate opportunities for another optimization). To complicate matters further, optimizations may behave differently on different architectures.

Compiler writers are selective about implementing optimizations because optimizations are hard to implement and debug. Moreover, since modern run-time environments (e.g., Java virtual machines) interleave compilation and program execution, optimizations can increase overall execution time even if they decrease application execution time. To help compiler writers in deciding which optimizations to implement, we show how to evaluate the costs and benefits of optimizations, both in isolation and in combination with other optimizations. We demonstrate our methodology on the optimizations implemented in Jikes RVM running on the PowerPC and IA32 platforms.

*Correspondence to: Amer Diwan, 430 UCB, Boulder, CO 80309-0430, U.S.A., diwan@cs.colorado.edu

The remainder of this paper is organized as follows. Section 2 describes and discusses our experimental methodology. Section 3 presents the results. Section 4 explains the reasons for our results. Section 5 reviews prior work in the area. Finally, Section 6 concludes the paper.

## 2. METHODOLOGY

We now introduce our terminology, and describe our benchmarks, measurement infrastructure, and measurement methodology.

### Terminology

All Java systems make some distinction between parts of the system that are compiled in advance and parts (loosely, application code) that are loaded and compiled at run time. For example, in Jikes RVM, most of the system code is written in Java and pre-compiled into what is called the *boot image*. The boot image includes the class loaders, compilers, garbage collector, etc. We use the same boot image, optimized at level -O2 (i.e., highest level of optimization), for all of our experiments.

### Benchmarks

To evaluate the effectiveness of optimizations, we report results for the commonly used SPECjvm98 benchmarks [12] (using input 100), SPECjbb2000 (which prior work has found to be representative of real world applications), and *ipsixql*, a real-world application that implements an XML database. For the SPECjvm98 benchmarks we use the single-threaded version of *mtrt*, called *raytrace*, to get easily reproducible results. We use a modified version of SPECjbb2000 (called pseudojbb) that performs a fixed number of transactions [3].

### Infrastructure

We conduct our experiments using Jikes Research Virtual Machine (RVM) version 2.1.1 from IBM Research [4]. Jikes RVM is designed to support research into virtual machines and includes an aggressively optimizing just-in-time compiler and an adaptive compilation system [2].

Jikes RVM's baseline compiler (abbreviated as BASELINE) is a fast non-optimizing compiler that converts Java bytecodes, one at a time, to machine code. Jikes RVM's optimizing compiler performs one or more of the optimizations described in Table I. It performs the optimizations more or less in the order given in the table (Jikes RVM performs some optimizations, such as loc_cse, multiple times) [9]. The optimizations in Table I can be enabled or disabled individually using command line flags. Due to space considerations, we describe the optimizations only briefly; our technical report [10] gives more details. The "Home" column in Table I indicates the lowest optimization level (O0, O1, or O2) that enables the optimization.

The optimizing compiler uses three intermediate representations: HIR (at about the bytecode level), LIR (low level), and MIR (machine level). While converting between these representations, the optimizing compiler performs a number of simple optimizations (such as limited forms of copy and constant propagation). We modified these simplifications to allow us to enable and disable them using command-line flags. We consider these simplifications to occur at a level below O0, which we call IR; in other words, O0, O1, and O2 all perform these simplifications. Finally, the optimizing compiler always performs linear-scan register allocation [11].

Table I. Optimizations provided by Jikes RVM

| Optimization | Home | SSA? | Description |
|---|---|---|---|
| inl_new | O1 | | Inline allocation of arrays and objects. |
| inl | O0 | | Inline statically resolvable calls. |
| preex_inl | O1 | | Pre-existence based inlining [5]. |
| guarded_inl | O1 | | Guarded inlining of virtual calls. |
| guarded_inl_interface | O1 | | Guarded inlining of interface calls. |
| scalar_replace_aggregates | O1 | | Treat fields/elements of objects as local variables. |
| monitor_removal | O1 | | Remove unnecessary synchronizations of non-escaping objects. Flow insensitive. |
| static_splitting | O1 | | Create hot traces using static heuristics. |
| unwhile | O2 | | Convert whiles into untils (i.e., loop inversion). |
| load_elim | O2 | Yes | Eliminate redundant loads [6]. |
| redundant_branch_elim | O2 | Yes | Eliminate redundant conditional branches using global value numbering and dominance relationships. |
| store_elim | O2 | Yes | Eliminate dead stores [6]. |
| expression_folding | O2 | Yes | Fold constants in addition and subtraction operations using flow sensitive analysis within a method. |
| licm | O2 | Yes | Loop invariant code motion. |
| gcse | O2 | Yes | Global common subexpression elimination. |
| loc_copy_prop | O1 | | Local copy propagation using flow sensitive analysis. |
| loc_constant_prop | O1 | | Local constant propagation (i.e., within basic blocks) using flow sensitive analysis. |
| loc_sr | O1 | | Local scalar replacement of loads of fields (i.e., within basic blocks) using flow sensitive analysis. |
| loc_cse | O1 | | Flow sensitive local common subexpression elimination. |
| loc_check | O1 | | Flow sensitive elimination of null, array bounds, and zero checks. |

**Measurement methodology**

We conducted our experiments on a 1.0 GHz Pentium III processor with 512 MB of memory running SUSE Linux 8.1 and on a 350 MHz PowerPC 750 processor with 576 MB of memory running Linux PowerPC 2000Q4.

To obtain end-to-end measurements, we access a cycle-accurate timer immediately before the program starts and immediately after the program ends. We ran each program twelve times and report the average of the last eleven runs. We drop the first run because it also includes compilation time. We perform a whole-heap garbage collection before every run in order to minimize memory management interference between the runs.

We performed the compilation time measurements using the built-in mechanisms of Jikes RVM. We took the measurements in separate runs to prevent them from interfering with the execution time measurements.

**Optimization combinations**

In order to explore the synergy between optimizations, we consider optimizations individually and in combination with other optimizations. We use the abbreviation pattern OPT$< \cdots >$ to describe the

optimizing configurations. The $\cdots$ enclosed in the <> gives the set of optimizations that are enabled. Recall that all optimizing configurations include register allocation. Here are the common patterns that we use to describe the optimizing configurations:

- OPT<>: Perform no optimizations (except, of course, for register allocation).
- OPT<IR>, OPT<O0>, OPT<O1>, and OPT<O2>: Use optimization levels IR, O0, O1, and O2 respectively. Note that OPT<O2> includes all optimizations.
- OPT<*s*+*opt*>: Perform all optimizations in the set *s* plus the optimization OPT<*opt*>. Per the usual definition of sets, if *opt* is already in *s*, then OPT<*s*+*opt*> is the same as OPT<*s*>.
- OPT<*s*−*opt*>: Perform all optimizations in the set *s* except for *opt*. If *opt* is not in *s*, then OPT<*s*−*opt*> is the same as OPT<*s*>.

For example, OPT<O1+load_elim> enables all optimizations in O1 plus load elimination. OPT<O2−load_elim> performs all optimizations in O2 with the exception of load elimination.

## 3. RESULTS

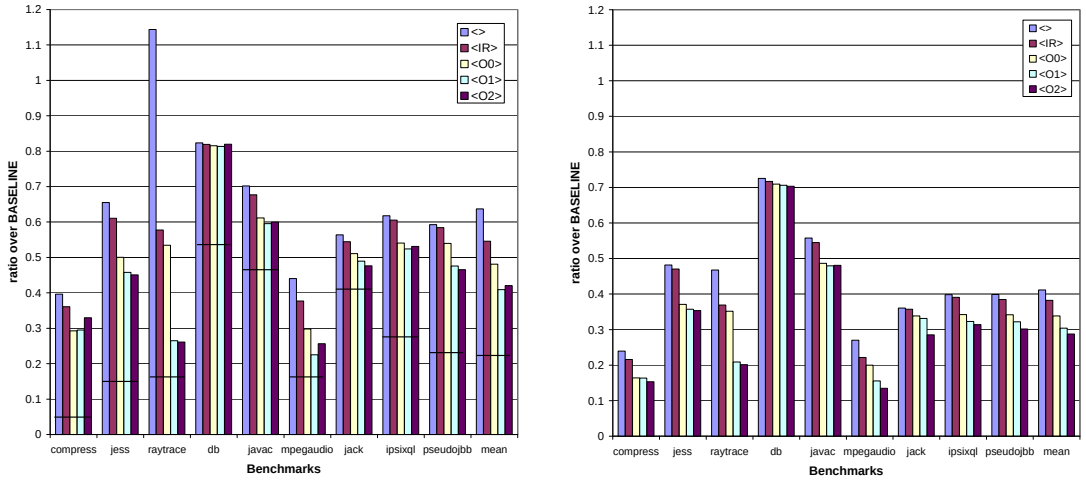
We now present and discuss detailed results evaluating the optimizations implemented in Jikes RVM. Section 3.1 gives the overall speedup due to the optimizations. Section 3.2 explores the individual and synergistic benefit of optimizations. Section 3.3 presents similar results for compilation time.

### 3.1. Overall speedups

Figure 1 gives the execution time using various optimizing configurations as a fraction of the execution time using the BASELINE configuration. The "mean" column gives the geometric mean of all benchmarks. From Figure 1 we see that OPT<> (register allocation) gives the greatest benefits; subsequent optimizations are also beneficial but not as dramatic. Also, we see that the speedups on the PowerPC are often larger than those on the IA32. For example, register allocation reduces execution time of *db* to 82% of BASELINE execution time on the IA32 and to 73% on the PowerPC. Also, while the general shapes of the curves on the IA32 and PowerPC are similar, we see that on the PowerPC, adding more optimizations almost always improves performance, whereas on the IA32 it can also degrade performance (e.g., OPT<O2> is slower than OPT<O1> for *mpegaudio* on the IA32). The reason for this is the small register set on the IA32 architecture (see Section 4.1).

### 3.2. Benefits of individual optimizations

In order to describe the benefits of individual optimizations, we need to introduce two additional notations. We use T(OPT<*s*>) to refer to the running time of the benchmark optimized with optimization set OPT<*s*>, and S(*x*, *s*) to refer to the speedup due to the optimization OPT<*x*> over optimization set OPT<*s*> using the following equation:




(a) IA32 (b) PowerPC

Figure 1. Ratio of execution time of various optimizing configurations to execution time of BASELINE for IA32 and PowerPC. The horizontal lines across each set of bars in (a) indicate the contribution of the boot image to the bar.

$$S(x,s) = \begin{cases} \frac{T(OPT<s-x>)-T(OPT<s>)}{T(OPT<IR>)} & \text{if} \quad x \in s \\ \frac{T(OPT<s>)-T(OPT<s+x>)}{T(OPT<IR>)} & \text{if} \quad x \notin s \end{cases} \tag{1}$$

Figure 2 gives the performance benefit of optimizations on the IA32 and PPC respectively for the *ipsixql* benchmark. The graphs for other benchmarks are mostly similar; our technical report [10] contains all the graphs. The graphs have four points for each optimization. The "S($x$, IR)" points (labeled "|") for each optimization present the benefit of turning on only optimization $x$ along with the IR optimizations. The "S($x$, O0)" points (labeled "0") give the benefit of turning on a single optimization, $x$, when all other optimizations in O0 are already enabled. "S($x$, O1)" and "S($x$, O2)" (labeled "1" and "2") are defined similarly. Unlike Figure 1, which is normalized to execution time of BASELINE, Figure 2 normalizes all speedups to execution time with OPT<IR> to make it easy to understand the effect of the optimizations. OPT<IR> includes register allocation, simple transformations performed during conversion between intermediate representations, and simple optimizations on HIR. Positive values indicate speedups; negatives are slow-downs.

From Figure 2 we see that most optimizations offer little benefit either by themselves or in synergy with other optimizations. Across all the benchmarks, inlining is the most consistent optimization, significantly benefiting all except one (db) benchmark on both architectures. Other optimizations,

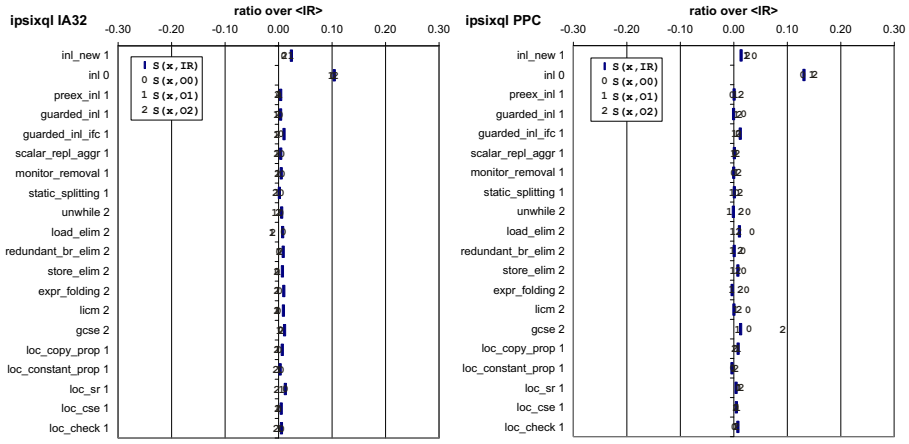



Figure 2. Individual and combined improvement due to optimizations for *ipsixql*

such as gcse benefit individual benchmarks. Optimizations (particularly the O2 optimizations) also sometimes degraded performance on the IA32 (but not the PPC). Thus, the O2 optimizations are a double-edged sword on the IA32: sometimes they offer significant benefits (e.g., gcse for *ipsixql*) but other times they degrade performance.

From Figure 2 we also see relatively little synergy between optimizations: in other words the S($x$, IR), S($x$, O0), S($x$, O1), and S($x$, O2) points are usually close to each other. This goes against common wisdom that indicates that inlining interacts positively and significantly with other optimizations by exposing opportunities for them.

### 3.3. Compilation times

We applied our method for measuring the individual and synergistic benefit of optimizations to measure the individual and synergistic costs of compilation. As expected, the O0 and O1 optimizations are fast, taking an insignificant time compared to T(OPT $<$ IR $>$). The O2 optimizations can easily take 50% or more of T(OPT $<$ IR $>$) and interact poorly with inlining. In some cases, performing an O2 optimization with inlining enabled could take three times T(OPT $<$ IR $>$). This occurs because O2 optimizations are mostly quadratic time optimizations and increasing the code size can rapidly increase their cost.

## 4. EXPLANATION FOR OUR RESULTS

This section explores the reasons for the behavior describe in Section 3 and tries to interpret our results more broadly than our current set of benchmarks.

### 4.1. Why do optimizations behave differently on different architectures?

From the data in Section 3 and particularly Figure 1 we see two main differences between our data for the PowerPC and IA32 architectures: (i) we get slightly better speedup on the PowerPC than on the IA32 (i.e., the bars in Figure 1 are often lower for PowerPC than for IA32); and (ii) in *raytrace* we get a severe performance degradation from register allocation on the IA32 but not on the PowerPC.

We hypothesized that the reason for the difference between the two architectures was due to the small number of registers on the IA32 compared to the PPC. To test this we modified the linear scan register allocator in Jikes RVM so that it would use approximately the same number of registers on the PowerPC as available on the IA32. We then repeated our experiments and plotted graphs similar to the ones in previous sections. Our results confirmed that our hypothesis was indeed correct.

### 4.2. A closer look at SSA-based optimizations

From Figure 2 we see that gcse, an SSA-based optimization, benefits *ipsixql*; overall, we found that all the SSA based optimizations benefited at least one program. We wanted to determine if the benefit of these optimizations comes from the optimizations themselves or from the SSA form. Converting into and out of SSA transforms the code (e.g., it splits live ranges much like a register allocator might) and thus may affect performance.

To investigate this possibility, we measured the performance of our benchmarks with the IR optimizations plus SSA form without performing any SSA-based optimizations on the IA32. We compared this performance to a configuration that performed the IR optimizations plus one SSA-based optimization.

We found that frequently the SSA-based optimizations did not improve performance once we subtracted out the benefit of just going through SSA form without performing any optimizations. In other words, the conversions into and out of SSA form were helping the (relatively naive) register allocator to make better decisions.

### 4.3. Kernel benchmarks

Section 3 reports that most optimizations offer little or only occasional benefits. This could be because (i) Our benchmarks suite is poor; (ii) The optimizations are implemented incorrectly; or (iii) The optimizations are actually useless. To investigate this we created up to 8 kernel benchmarks for each optimization. The kernel benchmarks for a given optimization heavily exercise that optimization and thus represent an optimistic situation for the optimization. For example, kernels for inlining contained calls to an inlinable method in a tight loop executed millions of times. Since these kernels are small programs, we were not only able to measure them with optimizations but were also able to inspect the machine code produced as a result of compiling and optimizing them.

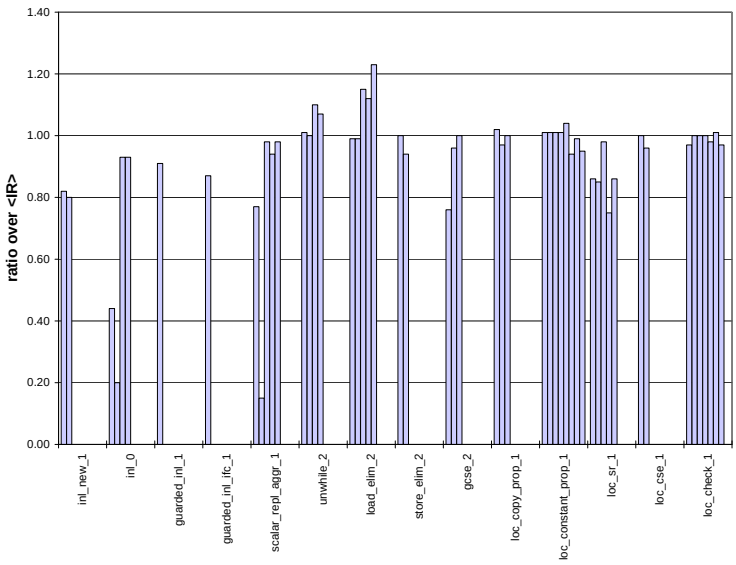



Figure 3. Results of running kernel benchmarks

Our experience in writing and inspecting the optimized versions of the kernel benchmarks revealed that Jikes RVM's optimizations are actually implemented correctly. In other words, the lack of benefits due to many optimizations is not because Jikes RVM implements them poorly.

Figure 3 presents results of kernel benchmarks on IA32. Each group of bars in Figure 3 presents the speedups for a single optimization on all of its kernels (smaller is better). Except otherwise noted, Figure 3 gives data for S($x$, IR). For most optimizations we wrote multiple kernels in order to try to present the most optimistic case to the optimization. In some cases we were able to derive kernels based on published examples. For example, we based one of the kernels for load_elimination on the example in the paper that introduced load_elimination [6]. We also wrote kernel tests that were not expected to exhibit any benefit to verify the validity of optimizations (not shown in Figure 3). Based on the results of our kernel tests, we classify each optimization into three categories as follows:

- *Beneficial: Optimizations that easily give significant benefit*. These are optimizations for which we could easily construct kernel tests that benefited greatly due to the corresponding optimization. Optimizations in this group are inl_new, inl, guarded_inl, guarded_inl_ifc, gcse, and loc_sr. Of these optimizations only inl gave consistent and significant benefit for our benchmarks. This emphasizes the need to use kernel as well as real benchmarks when determining the full potential of an optimization.

- *Limited: Optimizations with limited applicability*. These are optimizations for which we were able to construct kernels that contained opportunities for the optimization, but constructing the kernels was hard. There were two reasons that contributed to the difficulty of coming up with kernels for optimizations in this group: (i) the optimization was often subsumed by simpler optimizations in OPT<IR> and thus we had to come up with kernels that would not be optimized by the simpler optimizations but would be optimized by the target optimization; and (ii) the optimization required other optimizations, usually inlining, also to run in order to be effective. Optimizations in this group are scalar_repl_aggr, unwhile, loc_copy_prop, loc_constant_prop, loc_cse, and loc_check.

  scalar_repl_aggr works well for arrays, but it needs to have inl enabled to be effective for objects because object allocation is always followed by a call to a constructor. The escape analysis used with scalar_repl_aggr assumes the worst case about calls (Table I). Thus, to get the improvement due to scalar_repl_aggr in Figure 3 we enabled inlining.

  unwhile was not effective because most modern Java source to bytecode compilers, including IBM's Jikes compiler (which we used), perform rudimentary loop inversion. However, when we used version 1.4.2 of Sun's javac compiler, which does not perform loop inversion on the IA32/Linux platform, we found unwhile improved the performance of our test programs by up to 8%.

  loc_copy_prop, loc_constant_prop, and loc_check were often subsumed by optimizations in OPT<IR>. loc_cse was not subsumed by simpler optimizations but still failed to yield any benefit in one of our test cases even though we could confirm by looking at machine code that it applied.
- *Cannot Explain: Optimizations that we cannot yet explain*. These optimizations had unpredictable behavior, probably due to severe interactions with the underlying hardware. Seemingly insignificant changes to the kernels would lead to significantly different behavior. The different behavior was accompanied by wide swings in the number of L1-cache misses (which we measured using performance counters). The optimizations in this group are load_elim and store_elim. We are continuing our efforts to understand the behavior of these optimizations. These optimizations further underline the benefit of using our strategy of using kernel tests: effectiveness of an optimization may be indirect and it is worthwhile to confirm where the benefits are coming from using kernel tests.

## 5. RELATED WORK

While there has been much work on looking at the costs and benefits of optimizations (e.g., [1, 7, 8] we are not aware of any prior work that thoroughly explores the synergies between optimizations and uses kernel benchmarks along with real benchmarks to explore the full potential for the optimizations.

## 6. CONCLUSIONS

Compiler optimizations interact with each other and the environment in many ways. An optimization may increase or decrease the effectiveness of another optimization. An optimization that yields

significant benefit on one architecture may yield much less benefit on another architecture. An optimization that decreases application execution time may increase overall execution time if the optimization itself is expensive. To help compiler writers balance these considerations and thus build compilers that are both effective and efficient, we presented a detailed study of the cost and benefit of optimizations. We present results for twenty optimizations (all implemented in Jikes RVM), on two architectures (IA32 and PowerPC), and using nine benchmarks (including SPECjvm98, SPECjbb2000, and an XML database).

Besides exploring the cost and benefit of individual optimizations, we also report on the effects of interactions between optimizations and between optimizations and architectures. To explain our results, we report and explain results for a number of kernel programs we wrote to explore the behavior of the optimizations. We expect our results will provide valuable guidance to compiler writers.

## 7. Acknowledgments

This work is supported by NSF ITR grant CCR-0085792 and Career award CCR-0133457. Any opinions, findings and conclusions or recommendations expressed in this material are the authors' and do not necessarily reflect those of the sponsors.

### REFERENCES

1. Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.
2. Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter Sweeney. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 47–65, 2000.
3. Stephen M. Blackburn, Richard Jones, Kathryn S. McKinley, and J. Eliot B. Moss. Beltway: getting around garbage collection gridlock. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Languages Design and Implementation (PLDI)*, pages 153–164, Berlin, Germany, June 2002.
4. Michael Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio Serrano, V. C. Sreedhar, and Harini Srinivasan. The Jalapeño dynamic optimizing compiler for Java. In *ACM Java Grande Conference*, pages 129–141, San Francisco, CA, June 1999.
5. David Detlefs and Ole Agesen. Inlining of virtual methods. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 258–278. Springer-Verlag, 1999.
6. 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.
7. Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Mikio Takeuchi, Takeshi Ogasawara, Toshio Suganuma, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. Design, implementation, and evaluation of optimizations in a just-in-time compiler. In *ACM Java Grande Conference*, pages 119–128, San Francisco, CA, June 1999.
8. Kazuaki Ishizaki, Mikio Takeuchi, Kiyokuni Kawachiya, Toshio Suganuma, Osamu Gohda, Tatsushi Inagaki, Akira Koseki, Kazunori Ogata, Motohiro Kawahito, Toshiaki Yasue, Takeshi Ogasawara, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. Effectiveness of cross-platform optimizations for a Java just-in-time compiler. In *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programing Systems, Languages, and Applications*, pages 187–204. ACM Press, 2003.
9. Jikes Research Virtual Machine (RVM). Jikes Research Virtual Machine User's Guide. http://www.ibm.com/developerworks/oss/ jikesrvm/userguide/HTML/userguide.html.
10. Han Lee, Daniel von Dincklage, Amer Diwan, and J. Eliot B. Moss. Understanding the behavior of compiler optimizations. Technical Report CU-CS-972-04, University of Colorado, Boulder, CO, 2004.
11. Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(5):895–913, 1999.
12. Standard Performance Evaluation Corporation (SPEC). SPECjvm98 benchmarks. http://www.specbench.org/osg/jvm98.