Producer-Side Platform-Independent Optimizations and Their Effects on Mobile-Code Performance

Philipp Adler Wolfram Amme
Institut für Informatik
Friedrich-Schiller-Universität Jena, Germany
{phadler,amme}@informatik.uni-jena.de

Jeffery von Ronne
Department of Computer Science
University of Texas at San Antonio
vonronne@cs.utsa.edu

Michael Franz
Deptartment of Computer Science
University of California, Irvine
franz@uci.edu

***Abstract*— Portable mobile code is often executed by a host virtual machine using just-in-time compilation. In this context, the compilation time in the host virtual machine is critical. This compilation time would be reduced if optimizations can be performed ahead-of-time before distribution of the mobile code. Unfortunately, the portable nature of mobile code limits ahead-of-time optimizations to those that are platform-independent.**

This work examines the effect of platform-independent optimizations on the performance of mobile code applications. All experiments use the SafeTSA Format, a mobile code format that is based on Static Single Assignment Form (SSA Form). The experiments, which are performed on both the PowerPC and IA32 architectures, indicate that the effects of performing classical machine-independent optimizations are—in fact—quite platform-dependent. Nevertheless, the results indicate that an application of such optimizations can be profitable.

I. Introduction

In the era of the internet, we increasingly come across mobile code applications (i.e., programs that can be sent in a single form to a heterogeneous collection of processors and will then be executed on each of them with the same semantics [1]). Such mobile code is usually intended to be loaded across a network and executed by an interpreter or after dynamic compilation on the target machine.

More widespread use of mobile code puts increasing demands on it. Because the just-in-time (JIT) compilation occurs at run time and because the target systems of mobile code often have few resources and low performance, the program optimizations that are performed by a JIT compiler are often limited to those that are both simple and fast. Therefore, it is often beneficial to perform optimizations on the producer side before distributing the mobile code. Unfortunately, the target computer system is usually not known by the producer at the time of mobile code generation. Therefore, it would seem that only platform-independent optimizations should be performed on the producer side.

SafeTSA [2], [3] is a safe mobile code format designed as an alternative to the Java Virtual Machine's bytecode language (JVML). SafeTSA safely and compactly represents programs in Static Single Assignment Form (SSA Form [4]) using novel encoding techniques [2], [5]. The use of SSA Form simplifies verification and code generation and also allows for the natural and efficient application of producer-side platform-independent optimizations. (In contrast JVML is a stack-oriented format, which interferes with or complicates some of the optimizations discussed here. Nevertheless, it is possible to perform some ahead-of-time optimizations on JVML code; e.g. JAX [6] and IBM's SmartLinker [7] can perform extensive dead-code elimination and other optimizations.)

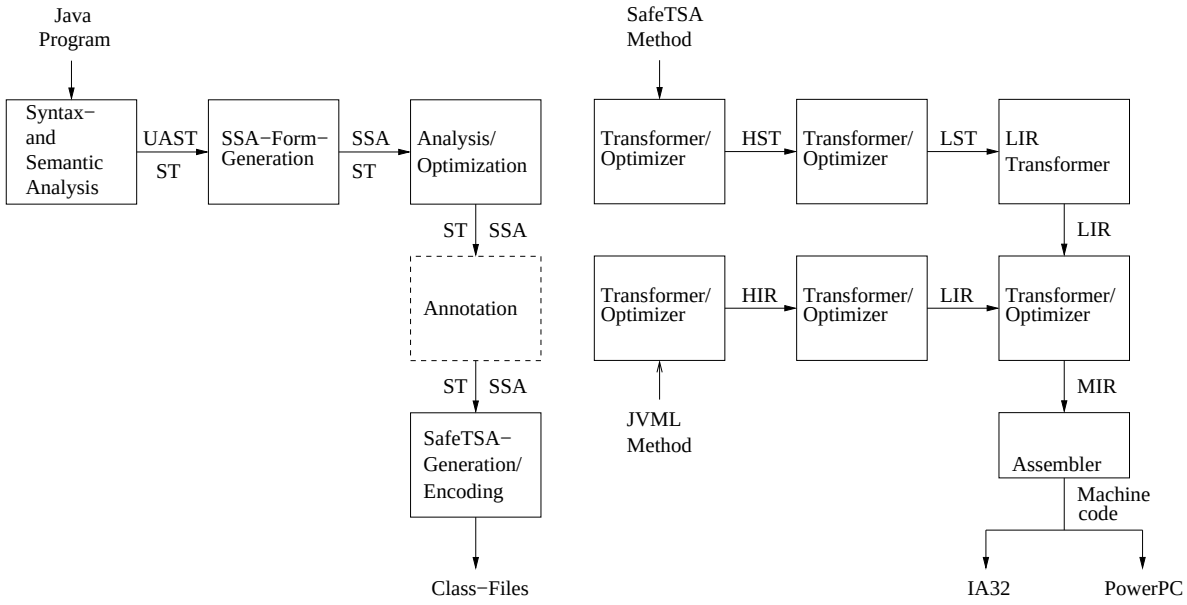
Producer-side platform-independent optimization would seem to be an unquestionable win, but the complexity of modern computer systems means that even these seemingly simple optimizations have complex and machine dependent effects [8]. In this paper, we describe our experiences selecting platform-independent optimizations for the intermediate representation SafeTSA.

II. The SafeTSA System

One objective of the SafeTSA project is to extend SafeTSA into a programming-language-independent and target-machine-independent transportation format for mobile code. As proof of concept prototype, we are developing an entire system for the transport of SafeTSA programs that can be used for programs written in several different programming languages and for different target architectures, and that applies safe and efficient techniques for program analysis, optimization, code generation, and encoding/decoding, at both the mobile-code producer and the mobile-code consumer. Current research activities of the SafeTSA project have focused on supporting programs written in the Java programming language, and we have built a producer-side implementation, that can be used to transform Java programs into SafeTSA. In addition, we have created implementations of the consumer side for Intel's IA32 architecture and for the PowerPC architecture.

Figure 1(a) shows an overview of our current producer-side compiler. The first phase uses syntactic and semantic analysis to transform a Java program into a unified abstract syntax tree (UAST) and its corresponding symbol table (ST). A UAST is a canonical syntax representation that can be used as the syntactical description of programs translated from several different programming languages. After the UAST is constructed, the program is transformed into a High-Level SSA Form, on which platform-independent optimizations then can be performed. After being optimized, the program can be translated into the SafeTSA Format and then stored into corresponding class files. Optionally, our system can be used to annotate programs with additional information that can be used by the code consumer; currently, escape information can be annotated [9].

Both implementations of the consumer side were created by adding an optimizing SafeTSA compiler to IBM's Jikes Research Virtual Machine (Jikes RVM) [10]: by adding



(a) SafeTSA's producer-side compiler.

(b) Structure of the consumer-side compiler.

Fig. 1. SafeTSA System Compilers

SafeTSA class loading and a SafeTSA compiler to the Jikes RVM system, we have built a virtual machine that can execute both SafeTSA- and JVML-compiled Java programs.

Figure 1(b) shows the internal structure of our SafeTSA compiler in comparison to Jikes RVM's JVML optimizing compiler. Initially, the SafeTSA compiler transforms the method into its high-level SafeTSA representation (HST). An HST representation of a SafeTSA method is an intermediate representation that is largely independent of the host runtime environment but differs from the original SafeTSA method in that there is some resolution of accessed fields and methods. Next, the SafeTSA method is transformed from HST into the low-level SafeTSA representation (LST). This process expands some HST instructions into host-JVM specific LST operations, specializing them for Jikes RVM's object layout and parameter passing mechanisms. After this transformation, the LST method is optimized and transformed into the same low-level intermediate representation that is used by Jikes RVM's JVML optimizing compiler.

III. Platform-Independent Optimizations

Optimizations can be divided in two groups: those that are considered to be platform independent and those that are platform dependent. In contrast to platform-dependent optimizations, which always must be performed on the consumer side, platform-independent optimizations have the advantage that they can be performed on the producer side of a mobile system, so that the time it takes to perform the optimization is not counted as part of the consumer-side execution time of a program. In the following, we briefly introduce the platform-independent optimizations that were considered for integration into our producer side.

A. Dead Code Elimination

Dead code elimination is used to remove unnecessary executed instructions. These are mostly operations whose value is never used in the program or whose elimination does not have any influence on the external program behavior. The following code fragment is a typical case:

```
public int addUp( int a, int b ) {
  int retVal = a + b;
  retVal += OwnUtils.myAdder( retVal );
  int fact = retVal * b;
  return retVal;
}
```

The line **int** fact = retVal * b; is not necessary, and the calculated value is never used. Therefore, this line can be removed resulting in a small program with less code and fewer instructions to be executed at runtime. In general, after the elimination of dead code, one should observe a shorter total execution time and less program code.

In addition, our producer side uses dead code elimination in order to remove those ϕ-instructions inserted into the program during the generation of SSA Form that do not have an impact on program operation. Therefore, dead code elimination should be considered an inherent part of the producer-side code generation, because it is required in order to produce programs in pruned SSA Form.

B. Constant Propagation and Folding

In practice, both constant propagation and constant folding are performed together. Constant propagation replaces references to variables that have a constant value with a direct reference to that constant value. Constant folding is used to

combine different constant values in a single expression [11]. A typical case follows:

```
final int CNT = 3;
int run = CNT * 100;
```

After performing constant propagation, this code fragment is reduced to one line:

```
int run = 3 * 100;
```

After constant folding is performed, the resulting code is

```
int run = 300;
```

As a result, the constants do not need to be placed in variables and the calculations using these do not need to be performed at runtime.

C. Static Final Field Resolution

Static final field resolution is a special case of constant propagation targeting class constants.[1] Without it, a field access would be needed every time one of these static final fields is used. Instead, this optimization replaces the field access with its constant value like normal constant propagation. Therefore, the field access including nullcheck is removed. This often results in second-order benefits, because such field accesses hamper other program transformations.

D. Common Subexpression Elimination

The goal of common subexpression elimination (CSE) is to find common (sub-)expressions, which redundantly produce the same result, and retain only one copy of the common subexpression. Running this optimization checks whether, for each instruction *inst* of a program, subsequently executed instructions will recalculate the same value. If such instructions exist, they will be deleted and their uses will be replaced by uses of *inst*. In this way, recalculation of values can be avoided at the cost of potentially extending the length of time for which the results of certain instructions must be retained, which may have undesirable consequences that manifest themselves during register allocation. These undesirable effects are lessened when there is a limit to such lengthening of live ranges. Therefore, CSE often is performed (locally) inside of basic blocks, instead of working (globally) on the whole program.

Since SafeTSA comes with explicit instructions for null- and indexchecks, CSE often results in the elimination of redundant null- and indexchecks. The following is an example of a typical program situation manifesting null- and indexchecks that can be eliminated:

[1]Constant propagation, constant folding, and static final field resolution are required by the Java Language Specification.

```
r1 = nullcheck (a)
i1 = indexcheck A, a, (i)
i2 = getelement A, r1, i1
r2 = nullcheck (a)
i3 = indexcheck A, a, (i)
i4 = getelement A, r2, i3
i5 = add int i2, i4
r3 = nullcheck (a)
i6 = indexcheck A, a, (i)
v0 = setelement A, r3, i6, i5
```

This is what would be generated by the SSA Form Generator of our producer side from the Java code fragment, which redefines the element *i* of a locally defined integer array *a*:

```
a[i] = a[i] + a[i];
```

In this program, only the first null- and indexchecks must be performed, whereas the execution of the other nullcheck and indexcheck instructions are unnecessary and therefore could be eliminated. For this reason the accomplishment of an CSE would result in the following unquestionably more efficient program:

```
r1 = nullcheck (a)
i1 = indexcheck A, a, (i)
i2 = getelement A, r1, i1
i3 = getelement A, r1, i1
i4 = add int i2, i3
v0 = setelement A, r1, i1, i4
```

E. Local Get Elimination

Every time an array is accessed null- and indexchecks are necessary before setting or retrieving the actual element. As we have already seen, for local or synchronized arrays this is often superfluous. With local get elimination former writing or reading accesses can be reused if the same element in an array is read again. Instead of performing null- and indexchecks, even get elements can be omitted, the value which was written or read before is used again.

The result of local get elimination can be shown with the same example program that was used for the explanation of CSE. In the optimized code fragment of this example, the second array access, `i3 = getelement A, r1, i1`, is also unnecessary, because that element had already been read by the first *getelement* instruction. Instead of performing this access, a more optimized version would just use the value *i2* everywhere the element *i3* is used.

F. Global Code Motion

The implementation of global code motion (GCM) is based on the algorithm of Click [12]. It is an optimization technique that reschedules instructions by moving them to an optimal spot. Two main motions can be identified:

1) Instructions are moved out of loops and into more control dependent (and often less frequently executed) basic blocks. On the other hand, moving instructions before loops usually causes an increase in register pressure.

2) If instructions cannot be moved out of loops or no loop is present, they are placed in the program such that live ranges are shortened and register pressure is reduced.

This optimization requires the derivation of a given program's control flow graph. After this, early scheduling is done, which determines for each instruction the first basic block at which it is still dominated by its inputs. The next phase is late scheduling; this phase finds each instruction's last permissible basic block that still dominates all of that instruction's uses. After finishing these two stages, the first and last possible basic blocks for instruction placement are given, and thus a safe-placement range is defined. Finally, it is necessary to find the optimal block with regard to loop nesting depth and control dependence, and place the instruction in that block.

To visualize the effects of global code motion, take a look at following code fragments. The listing on the left is the original fragment whereas the listing on the right depicts an optimized version.

```
double y = 2 * var1;
while ( i < 10 ) {
  int x = 2 * var2;
  i += x;
}
f( y );
```

```
int x = 2 * var2;
while ( i < 10 ) {
  i += x;
}
double y = 2 * var1;
f( y );
```

Two instructions were moved: The assignment "y = 2 * var1" in line one is moved to the last possible place where it dominates its use, that is directly before the function call. As a result the life span of variable y is reduced. The other assignment moved is the loop invariant "x = 2 * var2," which is placed before the loop. Thus, unnecessary calculations are avoided when the loop is run more than once.

G. Global Value Numbering

Like common subexpression elimination global value numbering (GVN) finds identical expressions, algebraic identities and performs constant folding. However, in contrast to CSE, this optimization identifies equivalent instructions that do not cause exceptions and which may reside in divergent execution paths, and replaces them with a single occurrence.

From this description, one might think that GVN subsumes CSE, but this would not be entirely accurate. The two optimizations overlap, but there are clear differences. For example, CSE can remove instructions that throw exceptions whereas GVN can not. Because global value numbering eliminates congruent instructions without regard to scheduling a run of global code motion is required afterwards. Only then can a correct schedule be guaranteed. Like global code motion this algorithm is based on the work of Click [12].

IV. Measurements

In order to empirically assess what kind of platform-independent optimizations can be expected to deliver performance benefits, we compiled a series of benchmarks from Java source into SafeTSA files and ran them through the consumer sides of our system. For each program and each platform-independent optimization, we generated an optimized and a non-optimized version of the benchmark and measured the total time required for the execution of each version.

A. Benchmark Programs and Environment

For our benchmarks, we used the programs contained in Sections 2 and 3 of the Java Grande Forum Sequential Benchmarks (JGF) [13], which were freely available in source code. All measurements were produced using the Size A version of the benchmark programs.

In order to ascertain the effects that platform-independent optimizations have on different architectures, benchmarking was conducted on both architectures of SafeTSA's consumer side—the IA32 and PowerPC architectures. Both of these architectures are common and widely used, and since IA32 is a CISC processor and PowerPC is a RISC architecture, they also represent much of the diversity present in practical instruction set architecture. One important difference between these two target architectures is the number of available registers. PowerPC has a total of 32 registers, whereas IA32 has only 8 general purpose registers and additional 8 floating point registers.

Measurements for the PowerPC architecture were obtained on a PowerMac with a 733 MHz PowerPC G4 (7450), 1.5GB of main memory, 64KB L1, 256KB L2, 1MB L3 caches running Mandrake Linux 7.1 (Linux 2.4.17 kernel). As a representative of the IA32 architecture, we have chosen a standard PC with an 1.333GHz AMD Athlon XP 1500+ processor, 2GB of main memory, 128KB L1 and 256KB L2 caches running SuSE Linux 9.0 (Linux 2.4.21 kernel).

B. Results

In this section, we present the results of our measurements. Although modest, the measurements of the effect of producer-side optimization on total execution time, especially for *CSE*, *GCM* and *GVN*, were quite insightful and somewhat surprising.

1) Dead Code Elimination: Most of the JGF benchmarks are well designed and contain only a few instructions that can be removed by dead code elimination. Therefore, the majority of instructions removed from the benchmark programs were ϕ-instructions, which had been unnecessarily inserted into the underlying intermediate code representation by our SSA generation algorithm. In fact, an examination of the benchmark results revealed that, for our benchmark programs, dead code elimination removed from 2.3% to 21.7% of the ϕ-instructions initially created during SSA conversion.

Although one might think that the removal of unused code or instructions should always result in same or better runtime performance, surprisingly for some benchmarks a performance degradation could be observed after dead code elimination had been applied. The measurements (see Figure 2) indicate that dead code elimination can negatively influence the execution time of programs on the IA32. For this architecture the measured changes in execution time ranges

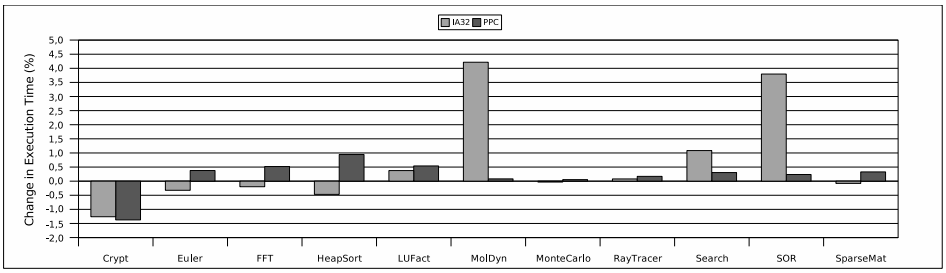


Fig. 2. Dead code elimination

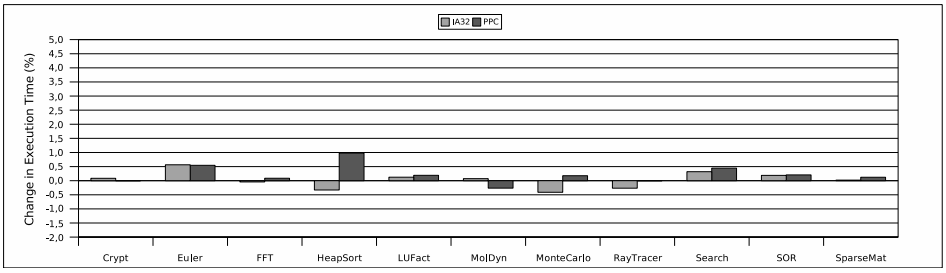


Fig. 3. Constant propagation and folding

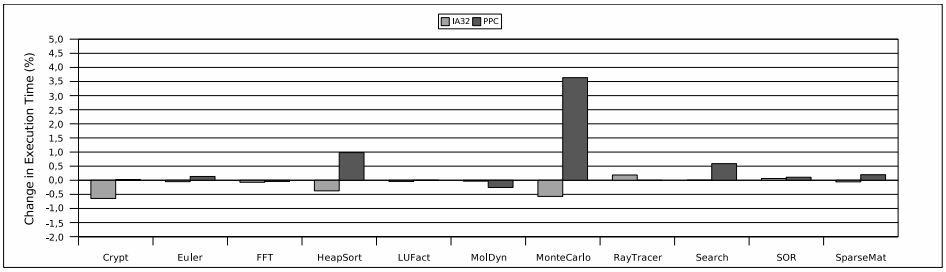


Fig. 4. Static final field resolution

from -1.3% to 4.2% of the time measured for non-optimized SafeTSA programs.

The main reason for this degradation in performance is the different cache behavior of optimized and non-optimized program versions. Although, the influence that a dead code elimination has on program's execution time appears to be random, this optimization is still useful especially for mobile code applications, because it reduces the size of transmitted programs and also the time required to load the program in the code producer.

2) Constant Propagation and Folding: Figure 3 identifies the effect that constant propagation and folding has on the execution time of SafeTSA programs. As can be seen, this optimization is beneficial for most of the benchmark programs, showing slight improvements in execution time (up to 1.0% for HeapSort), both on the IA32 and PowerPC. The performance degradations observed for some programs were negligible.

As consequence, constant propagation and folding can be seen as an optimization that needs little effort, reduces code size, but shows no real influence on the execution time of a program. As a rule, if it is necessary for the accomplishment of succeeding optimization phases a constant propagation and folding should be conducted on the producer side; otherwise this optimization can be omitted.

3) Static Final Field Resolution: The objectives of static final field resolution are similar to those of constant propagation and folding, except that this optimization works exclusively on constant fields. Results obtained for this optimization (see Figure 4) reveal that, for most of the benchmarks, static final field resolution yields a slight reduction in total execution time. As an example, when static field resolution was activated, the *MonteCarlo* benchmark's execution time was reduced by 3.6% compared to the non-optimized version.

4) Local Get Elimination: Only the benchmark programs *Euler*, *LUFact* and *Search* contain instructions that are affected by a local get elimination. *Euler* gets the most out of this optimization, with improvements of 1.8% on IA32 and 3.8% on PowerPC. This optimization's effect on *LUFact* and *Search* is negligible. *LUFact* has a degradation in performance of around 0.5% on IA32, whereas on PowerPC an improvement of 1.7% resulted. The measured values for *Search* show no significant improvement on IA32 and minor performance decrease on PowerPC with 0.4%.

Although the results obtained for local get elimination are

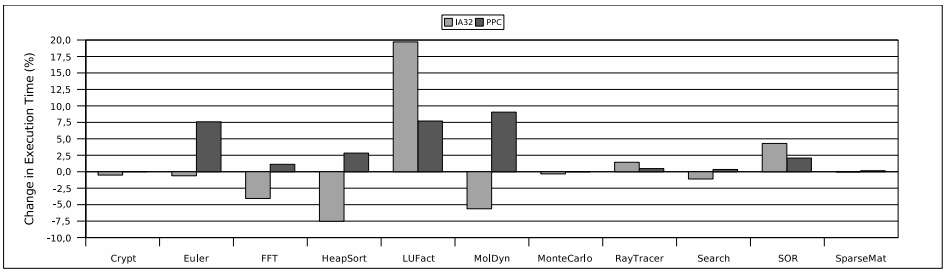


Fig. 5. Global common subexpression elimination

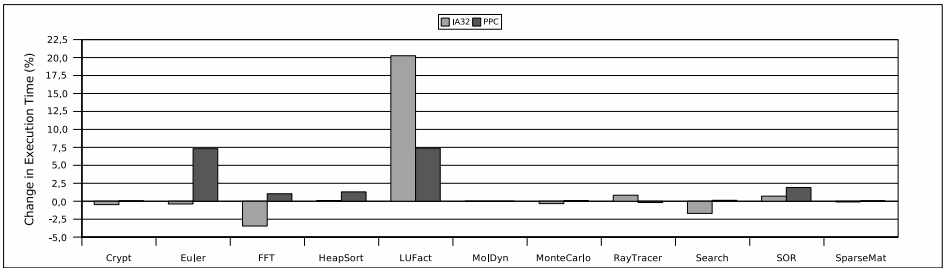


Fig. 6. Local common subexpression elimination

of limited applicability, the potential for large performance increases suggests that this optimization should be performed at the producer side.

5) Common Subexpression Elimination: Generally, common subexpression elimination yields an improvement in runtime performance (see Figure 5 and Figure 6), but the different behavior on IA32 and PowerPC is conspicuous. Although, both local and global CSE are nearly always beneficial on PowerPC, it can also be seen that, in general, global CSE yields better speedups than local CSE. On IA32, however, the results are the other way around: local CSE yields faster programs than global CSE.

Except for one benchmark program, global common subexpression elimination on PowerPC consistently resulted in performance gains—often considerable—and three benchmark programs ran over 7.5% faster with the code-producer optimized SafeTSA files. In contrast, an examination of the performance of the same optimized SafeTSA programs reveals that, on IA32, only one benchmark (LUFact) was sped up by global common subexpression elimination. In most cases, however, common subexpression had only a negligible effect on total execution time, and for three benchmark programs, there was a significant degradation. Although at first sight these results are surprising, they can be explained by the limited number of registers which are available on the IA32 architecture (see also [14]) and with Jikes RVM's register allocation strategy.

Thus, Jikes RVM's linear scan register allocation [15] tends to assign registers to variables with short live ranges and should result in acceptable performance for simple programs that use short-lived variables exclusively. In contrast, in programs that make extensive use of long-lived global variables, the register strategy used by Jikes RVM will result in suboptimal performance, especially since the frequency of variable use is not considered during register allocation.

Global common subexpression elimination usually increases the live range of a temporary variable so that it can be reused without being recomputed. With the spill heuristic of Jikes RVM's register allocator, this can be especially damaging to loop invariant subexpression eliminations. In such a case, a variable used heavily in an inner-loop can be spilled and allocated to a memory location just because it is defined outside of the loop and has a long live range.

Measurements of the execution time of SafeTSA programs optimized with local common subexpression elimination on the IA32 architecture indicate that in most cases with local common subexpression elimination the significant performance degradation that would sometimes result from global common subexpression elimination can be avoided. In fact, with the exception of the benchmark programs *FFT* and *Search*, whose runtime degradations of 2.7% and 1.7% seems tolerable, a local common subexpression led to improved—or, at least, the same—performance.

Further investigations were performed to determine what kind of subexpression elimination was responsible for the performance gains. These investigations revealed that 7% of all instructions, 17% of all null-checks, and 8% of all indexchecks could be eliminated from the programs. A further review of the program sources showed that the elimination of superfluous indexchecks is the primary cause of the speedup for the optimized SafeTSA files. In contrast, the elimination of nullchecks had only an insignificant influence on the runtime behavior because in the underlying Jikes RVM system in most of all cases nullchecks will be replaced by gratuitous hardware checks [16].

6) Global Code Motion: The effect of performing global code motion on SafeTSA programs can be seen in Figure

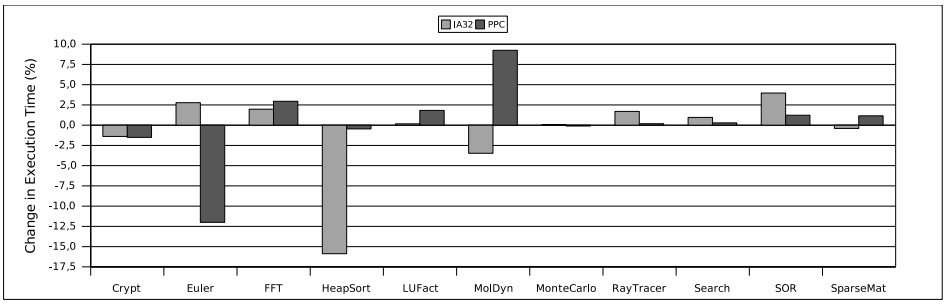


Fig. 7. Global code motion

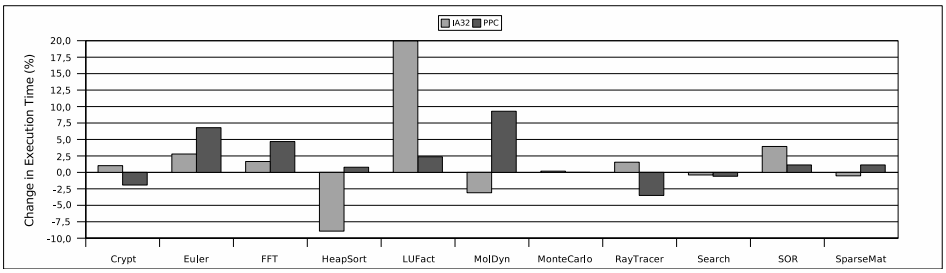


Fig. 8. Global value numbering

7. For the IA32 architecture this optimization moderately decreases the execution time of most of the benchmark programs. The drastic degradation in execution time for some programs (by more than 15% for *HeapSort*) arose primarily from the movement of loop invariant instructions. In general, these movements increase performance by preventing instructions from being executed unnecessarily inside loops but with the disadvantage that the live ranges of variables are extended, which as a consequence will increase the register pressure. For some benchmark programs, especially *HeapSort* and *Moldyn*, the additional register pressure results in an overall slowdown instead of a speedup.

Overall, an application of global code motion on the PowerPC architecture leads to better runtime performance than it does on the IA32 architecture. The *Euler* benchmark, however, slowed down by more than 12% after this optimization had been performed, which is particularly surprising, since the PowerPC architecture usually has a sufficient number of free registers to prevent register pressure from having a significant impact on execution time. For this benchmark, however, an excessive number of invariants (95 instructions) were hoisted out of the loop. As a consequence, register pressure had a noticeable effect on the PowerPC architecture, but on the IA32 architecture, global code motion did not cause such a reduction in performance of the *Euler* benchmark. At first glance, this is confusing, but the reason for this is that on the IA32 architecture, register pressure was already high, and many of the temporary values were already spilled, and as a result, the additional register pressure and the resulting spill code did not increase execution time enough to outweigh the benefits of moving invariant computations out of the loop.

7) Global Value Numbering: Figure 8 shows the execution times we measured after applying global value numbering and global code motion. (Global code motion is required to reschedule instructions, because global value numbering can leave instructions in an illegal ordering.) As expected, global value numbering had an effect on the benchmark programs' execution time similar to that global common subexpression elimination had. With global value numbering, however, the performance degradation that had been observed for global common subexpression elimination on the IA32 architecture was reduced but not entirely eliminated.

On the IA32 architecture, the degradation in performance when executing the *HeapSort* and *Moldyn* programs was primarily a consequence of global code motion. In order to further explore this, we desired to restrict such movements to basic blocks, so we reran the benchmarks using SafeTSA files which had been optimized using local common subexpression elimination in combination with our modified version of global code motion that does not hoist loop-invariant instruction out of loops. The results of this benchmark run resulted in execution times that—except for *HeapSort*—were nearly identical to those that result from local common subexpression elimination. Unfortunately, this combined optimization still produces a performance degradation of 14.8% when applied to the *HeapSort* benchmark.

V. Selection of Producer-Side Optimizations

The experiments described in the last section demonstrate that platform-independent optimizations—in the truest sense—do not exist. In fact, none of the candidate optimizations could be characterized as absolutely platform-independent, since for each of them, the execution time of at least one benchmark program on at least one of the platforms was negatively impacted by the optimization.

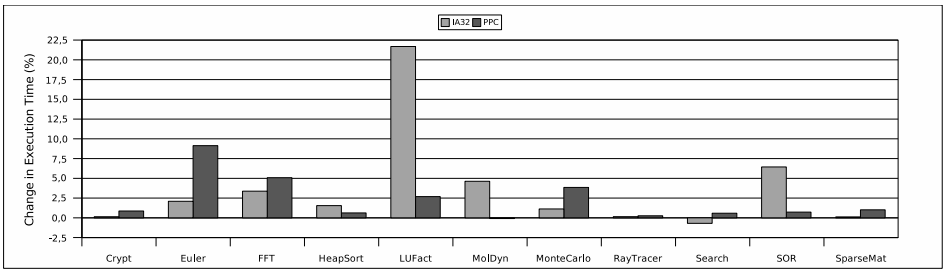


Fig. 9. Combination of selected optimizations

Nevertheless, except for a few of the global optimizations the occasional performance degradation fell within a tolerable range and should not be taken to mean that these optimizations do not result in overall performance increases. Therefore, we integrated into the producer side of the SafeTSA system all those optimizations which do not use global program restructuring, i.e., dead code elimination, constant propagation, static field resolution, local getfield elimination, and local common subexpression elimination. Global code motion and global value numbering have not been integrated into our producer side, since these optimizations did not result in a better runtime performance than that what could be achieved with a local common subexpression elimination.

Figure 9 shows the execution times for SafeTSA programs generated with a combination of these selected optimizations relative to the execution times of the unoptimized versions. Except for one benchmark program, the combination of optimizations resulted in consistent performance gains. On the PowerPC architecture, two benchmark programs ran over 5% faster with the code-producer optimized SafeTSA files. On the IA32 architecture, four optimized SafeTSA programs outperform corresponding unoptimized versions by over 3%, and one of them ran 20% faster.[2]

It is noteworthy that the speedups due to the selected combination of optimizations are not the sum of the speedups achieved during the execution of separately optimized programs. In fact, the measurements show that the optimizations interact with each other, i.e., a degradation introduced by one optimization is reversed by another optimization and the other way around.[3]

VI. Conclusion

In this paper, we have described our experience selecting platform-independent optimizations for SafeTSA's producer side. Experiments, performed on both PowerPC and IA32 architecture, indicate that none of the platform-independent optimizations reduced the execution time of all benchmark programs on both IA32 and PowerPC.

The measurements from our experiments indicate, however, that optimizations which avoid global code restructuring are generally useful optimizations and therefore should be utilized on the producer side of all mobile-code systems. In contrast, global optimizations, i.e., global common subexpression elimination, global code motion and global value numbering, often result in additional register pressure which on the IA32 architecture can often slow down program execution time dramatically.

In particular, the measurements obtained on the IA32 architecture reveal that, due to the scarcity of registers, the register allocation strategy and the effect of optimizations on register pressure are among the most important factors influencing program performance. In this context, it turned out that Jikes RVM linear scan register allocation algorithm was inadequate for use as a general purpose register allocator since slight modifications to the program code would often result in unexpected performance losses. Therefore, it is not clear to what extent our results would generalize to mobile-code systems utilizing other register allocation strategies.

[2] At first glance, these seems rather disappointing results. However, it should be mentioned that a lack of data encapsulation in the JGF benchmark programs prevents extensive optimization.

[3] A comparable behavior of combined optimization can be found for example in [17].

References

[1] A.-R. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe, "Efficient and language-independent mobile programs," in *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'1996)*, ser. ACM SIGPLAN Notices, vol. 31. New York: ACM Press, May 1996, pp. 127–136.

[2] W. Amme, N. Dalton, M. Franz, and J. von Ronne, "SafeTSA: A type safe and referentially secure mobile-code representation based on static single assignment form," in *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'2001)*, ser. ACM SIGPLAN Notices, vol. 36. Snowbird, Utah, USA: ACM Press, June 2001, pp. 137–147.

[3] W. Amme, J. von Ronne, and M. Franz, "Quantifying the benefits of ssa-based mobile code," in *Proceedings of the 4th International Workshop on Compiler Optimization Meets Compiler Verification (COCV'2005)*, Edinburgh, Scotland, Apr. 2005.

[4] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, pp. 451–490, Oct. 1991.

[5] J. von Ronne, "A safe and efficient machine-independent code transportation format based on static single assignment form and applied to just-in-time compilation," Ph.D. dissertation, University of California, Irvine, 2005.

[6] F. Tip, P. F. Sweeney, C. Laffra, A. Eisma, and D. Streeter, "Practical extraction techniques for java," *ACM Trans. Program. Lang. Syst.*, vol. 24, no. 6, pp. 625–666, 2002.

[7] F. Tip, P. F. Sweeney, and C. Laffra, "Extracting library-based java applications," *Commun. ACM*, vol. 46, no. 8, pp. 35–40, 2003.

[8] M. E. Benitez and J. W. Davidson, "The advantages of machine-dependent global optimization," in *Programming Languages and System Architectures*, ser. Lecture Notes in Computer Science, J. Gutknecht, Ed., vol. 782. Springer Verlag, Mar. 1994, pp. 105–124.

[9] A. Hartmann, W. Amme, J. von Ronne, and M. Franz, "Code annotation for safe and efficient dynamic object resolution," in *Proceedings of the 2nd International Workshop on Compiler Optimization Meets Compiler Verification (COCV'2003)*, Warsaw, Poland, Apr. 2003.

[10] *Jikes RVM User's Manual*, v2.4.1 ed., IBM Research, Sept. 2005. [Online]. Available: http://jikesrvm.sourceforge.net/userguide/HTML/userguide.html

[11] S. S. Muchnick, *Advanced compiler design and implementation*. 2929 Campus Drive, Suite 260, San Mateo, CA 94403, USA: Morgan Kaufmann Publishers, 1997.

[12] C. N. Click, "Combining Analyses, Combining Optimizations," *PhD Dissertation, Rice University, Houston, Texas*, Feb. 1995.

[13] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey, "A benchmark suite for high performance Java," *Concurrency: Practice and Experience*, vol. 12, no. 6, pp. 375–388, May 2000.

[14] R. Gupta and R. Bodik, "Register pressure sensitive redundancy elimination," *Lecture Notes in Computer Science*, vol. 1575, pp. 107–121, 1999.

[15] M. Poletto and V. Sarkar, "Linear scan register allocation," *ACM Transactions on Programming Languages and Systems*, vol. 21, no. 5, pp. 895–913, Sept. 1999.

[16] D. Grove, H. Srinivasan, J. Whaley, J. deok Choi, M. J. Serrano, M. G. Burke, M. Hind, S. Fink, V. C. Sreedhar, and V. Sarkar, "The Jalapeño dynamic optimizing compiler for Java," in *Proceedings of the ACM 1999 Conference on Java Grande*. ACM Press, May 1999, pp. 129–141.

[17] H. Lee, D. von Dincklage, A. Diwan, and J. E. B. Moss, "Understanding the behavior of compiler optimizations," Department of Computer Science, University of Colorado, Boulder, Tech. Rep. CU-CS-972-04, 2004.