1-1-2004

COMPILER OPTIMIZATION ORCHESTRATION FOR PEAK PERFORMANCE

Zhelong Pan

Rudolf Eigenmann

COMPILER OPTIMIZATION ORCHESTRATION FOR PEAK PERFORMANCE

ZHELONG PAN
RUDOLF EIGENMANN

TR-ECE 04-01
JANUARY 2004

SCHOOL OF ELECTRICAL
AND COMPUTER ENGINEERING
PURDUE UNIVERSITY
WEST LAFAYETTE, IN 47907-2035

COMPILER OPTIMIZATION ORCHESTRATION FOR PEAK PERFORMANCE

Zhelong Pan Rudolf Eigenmann [1]

School of Electrical and Computer Engineering
1285 Electrical Engineering Building
Purdue University
West Lafayette, IN 47907-1285

[1]This material is based upon work supported in part by the National Science Foundation under Grant No. 9703180, 9975275, 9986020, and 9974976.

TABLE OF CONTENTS

Page

LIST OF TABLES iii

LIST OF FIGURES iv

ABSTRACT v

1 Introduction and Motivation 1

2 Understanding the Performance Behavior of Compiler Optimizations 6

2.1 Experimental Environment 6

2.2 Performance of Optimization Levels O0 through O3 7

2.3 Performance of Individual Optimizations 9

2.4 Selected Harmful GCC Optimizations 13

2.4.1 Strict Aliasing 14

2.4.2 GCSE (Global Common Subexpression Elimination) 15

2.4.3 If-Conversion 16

3 Algorithms for Orchestrating Compiler Optimizations 19

4 Conclusions 23

LIST OF REFERENCES 24

LIST OF TABLES

Table Page

2.1 Average speedups of the optimization levels, relative to O0. In each entry, the first number is the arithmetic mean, and the second one is the geometric mean. The averages without ART are put in parentheses for the floating point benchmarks on the Pentium IV machine. 8

3.1 Algorithms for Orchestrating Compiler Optimizations 20

LIST OF FIGURES

Figure Page

2.1 Execution time of SPEC CPU 2000 benchmarks under different optimization levels compiled by GCC. (Four floating point benchmarks written in f90 are not included, since GCC does not compile them.) Each benchmark has four bars for O0 to O3. (a) and (c) show the integer benchmarks; (b) and (d) show the floating point benchmarks. (a) and (b) are the results on a Pentium IV machine; (c) and (d) are the results on a SPARC II machine. 7

2.2 Relative improvement percentage of all individual optimizations within O3 of GCC 10

2.3 Relative improvement percentage of all individual optimizations within O3 of GCC 11

2.4 Relative improvement percentage of all individual optimizations within O3 of GCC. SIXTRACK on a Pentium IV machine. 12

2.5 Relative improvement percentage of strict aliasing. 14

2.6 Relative improvement percentage of global common subexpression elimination. 16

2.7 Relative improvement percentage of if-conversion. 17

3.1 Speedup improvement of three algorithms relative to O1. (a) and (c) show the integer benchmarks; (b) and (d) show the floating point benchmarks. (a) and (b) are the results on a Pentium IV machine; (c) and (d) are the results on a SPARC II machine. 21

ABSTRACT

Although compile-time optimizations generally improve program performance, degradations caused by individual techniques are to be expected. Feedback-directed optimizations have recently begun to address this issue, by factoring runtime information into the decision process of which compiler optimization to apply where and when. While improvements for small sets of optimization techniques have been demonstrated, little empirical knowledge exists on the performance behavior of the large number of today's optimization techniques. This is especially true for the interaction of such techniques, which we have found to be of significant importance in navigating the search space of the best combination of techniques. The contribution of this paper is in (1) providing such empirical knowledge and (2) developing algorithms for efficiently navigating and pruning the search space.

To this end, we evaluate the optimization techniques of GCC on both a Pentium IV machine and a SPARC II machine, by measuring the performance of the SPEC CPU2000 benchmarks under different compiler flags. We analyze the performance losses that result from individual optimizations. We then present three heuristic algorithms that search for the best combination of compiler techniques using measured runtime as feedback.

1. INTRODUCTION AND MOTIVATION

Compiler optimizations for modern architectures have reached a high level of sophistication. Although they yield significant improvements in many programs, the potential for performance degradation in certain program patterns is known to compiler researchers and many users. Potential degradations are well understood for some techniques, while they are unexpected in other cases. For example, the difficulty of employing predicated execution or parallel recurrence substitutions is evident. On the other hand, performance degradation as a result of alias analysis is generally unexpected.

The state of the art is for compiler writers to let the user deal with this problem. Through command line flags, the user must decide which optimizations are to be applied in a given compilation run. Clearly, this is not a long-term solution. As compiler optimizations get increasingly numerous and complex, this problem must find an automated solution.

Compile-time performance prediction models are unable to deliver the necessary accuracy in deciding when and where best to apply what optimization technique - a process we refer to as *orchestration of compiler optimizations*. Among the main reasons are the unavailability of program input data, the unknown machine/environment parameters, and the difficulty of modeling interactions of optimization techniques. Runtime information is necessary for more accurate decisions. Many recent techniques have begun to exploit this potential. They range from state-of-the-art profiling techniques to feedback-directed optimizations to fully dynamic, adaptive compilation.

We are still only at the beginning of this development. Two important milestones towards advanced techniques that dynamically orchestrate compiler optimizations are (1) the quantitative understanding of the performance effects of today's compiler opti-

mizations and (2) the development of efficient methods that find the best combination of optimization techniques, given a specific program, machine, and compiler. The first issue is important, as we must understand and focus attention on those techniques that may cause large negative effects in (sections of) today's computer applications. The second issue is important, as the number of optimization techniques in any realistic compiler setting is so large that it is prohibitive to "try out all combinations" in hope to find the best. The contributions of the present paper are in providing answers and solutions to these two issues.

The specific contributions of this paper are as follows:

1. We evaluate the performance behavior of the GNU Compiler Collection (GCC) and its optimization options on both a Pentium IV machine and a SPARC II machine using the SPEC CPU2000 benchmarks. We analyze the situations where substantial performance losses occur. They are effects of the optimizations alias analysis, global common subexpression elimination, and if conversion.

2. We describe and evaluate three algorithms that maximize performance by orchestrating compiler optimizations in a better way. In our implementation, the algorithms are driver scripts that tune the optimization options of existing compilers. The best algorithm improves the performance up to 6.4% for integer benchmarks (3.0% on average) and up to 183.8% for floating point benchmarks (24.1% on average) on the Pentium IV machine, over O3, the highest GCC optimization level. It improves the performance up to 8.8% for integer benchmarks (3.3% on average) and up to 13.8% for floating point benchmarks (4.4% on average) on the SPARC II machine.

The present paper relates to several recent contributions: In an effort to consider the *interactions between optimizations*, Wolf, Maydan and Chen developed an algorithm that applies fission, fusion, tiling, permutation and outer loop unrolling to optimize loop nests [14]. They also use a *performance model* to estimate caches misses, software pipelining, register pressure and loop overhead. Similarly, Click and

Cooper showed that combining constant propagation, global value numbering, and dead code elimination leads to more optimization opportunities [4].

In an effort to avoid the inaccuracy of compile-time models, Whaley and Dongarra *select optimizations using actual execution time*. They developed ATLAS to generate numerous variants of matrix multiplication and to find the best one on a particular machine [13]. Similarly, Iterative Compilation [8] searches through the transformation space to find the best block sizes and unrolling factors.

Some *empirical methods* were introduced to tune the optimizations in the compiler. Meta optimization [10] uses machine-learning techniques to adjust the compiler heuristics automatically. Cooper uses a biased random search to discover the best order of optimizations [5]. To reduce the expensive search time, the Optimization-Space Exploration (OSE) compiler [11] defines sets of optimization configurations and an exploration space at compile-time. According to compile-time performance estimation, this compiler prunes the search space to find the best optimization configuration quickly.

To choose options intelligently, Granston and Holler presented an automatic system, which makes *application-specific recommendations* for use with PA-RISC compilers. They developed heuristics to deterministically select good optimizations, based on information from the user, the compiler and the profiler [6].

Open Issues: Most approaches have focuses on a relatively small number of optimization techniques [4,10,13,14], where the need for considering their interactions has not been compelling. When considering interactions, the search space for finding the best combination of techniques increases dramatically. Interactions are very difficult to model, which adds even more complexity to the approach of orchestrating optimizations through compile-time modeling, pursued by several researchers [11, 13, 14]. General pruning techniques of the search space must be developed. They are necessary complements of methods that explore the full space [5, 10] or that deal with specialized environments [6, 11]. These techniques need to be developed based on

experience with realistic applications, in addition to the kernel benchmarks used in many initial contributions [4, 5, 14].

Addressing these issues, this paper sets the goals of: (1) quantitatively understanding the performance behavior of the large number of today's optimization techniques on realistic programs and, in particular, understanding their interactions; (2) developing efficient algorithms that can feed this information back into the compiler and orchestrate the best combination of techniques.

We have approached these two milestones as follows: In order to quantitatively understand the performance effects of a large number of compiler techniques, we have measured the performance of the SPEC CPU2000 benchmarks under different compiler configurations. We have obtained these results on two different computer architectures and two different compilers, giving answers to the questions (1) What optimizations may not always help performance? How large is the degradation? (2) How and to what degree do optimizations interact with each other? (3) Do the optimizations have different effects on integer benchmarks and on floating-point benchmarks? (4) Is the performance degradation specific to a particular architecture?

Although [15] shows model-driven optimizations may produce comparable performance as empirical optimizations, there is some difficulty in constructing a perfect performance model applicable to all optimizations. Moreover, it is not easy to estimate the model parameters. [11] displays that optimization evaluation by the real execution time is much better than by its performance model. Therefore, for simplicity, we use the actual execution time as the feedback in this paper. We evaluate the whole program by executing it under the SPEC ref dataset, which gives the potential peak performance by automatically orchestrating optimizations.

Using this information as feedback, we have developed several search algorithms that orchestrate optimization techniques in a compiler. They find a superior optimization combination for a particular application and machine, while maximizing the application performance. The optimization parameters tuned by our algorithms are those available to the user through compiler options. The algorithms make multiple

runs of an application, finding the best possible options. The "trivial" solution of trying all combinations of techniques would be infeasible, as it is of complexity $O(2^n)$ even for n on-off optimizations. Given the potential interactions, all *on* and *off* combinations would need to be tried, leading to days or months of test times per program for even modest numbers of optimization techniques and application run times. The interaction of optimization techniques is a key motivation for our goal of developing efficient search algorithms. We have observed many situations, where individual optimization techniques degraded performance, but switching all of them off led to further degradation. These observations are consistent with the work presented in [8], which shows that the execution time of matrix multiplication is not smooth across tile sizes and unroll factors. In this paper, we will evaluate three different algorithms that are of reasonable complexity and yield significant application performance improvements.

The remainder of this paper is organized as follows: In Section 2, we characterize the performance behavior of the compiler optimization techniques of the GCC compiler . We also analyze the situations where performance degrades substantially as a result of individual optimizations and discuss the need and opportunity for better orchestration of these techniques in the compiler. Section 3 presents and evaluates three heuristic algorithms that orchestrate the compiler optimizations, so as to achieve the best possible application performance. Section 4 presents conclusions.

2. UNDERSTANDING THE PERFORMANCE BEHAVIOR OF COMPILER OPTIMIZATIONS

2.1 Experimental Environment

We take our measurements using the SPEC CPU2000 benchmarks. To differentiate the effect of compiler optimizations on integer (INT) and floating-point (FP) programs, we display the results of these two benchmark categories separately.

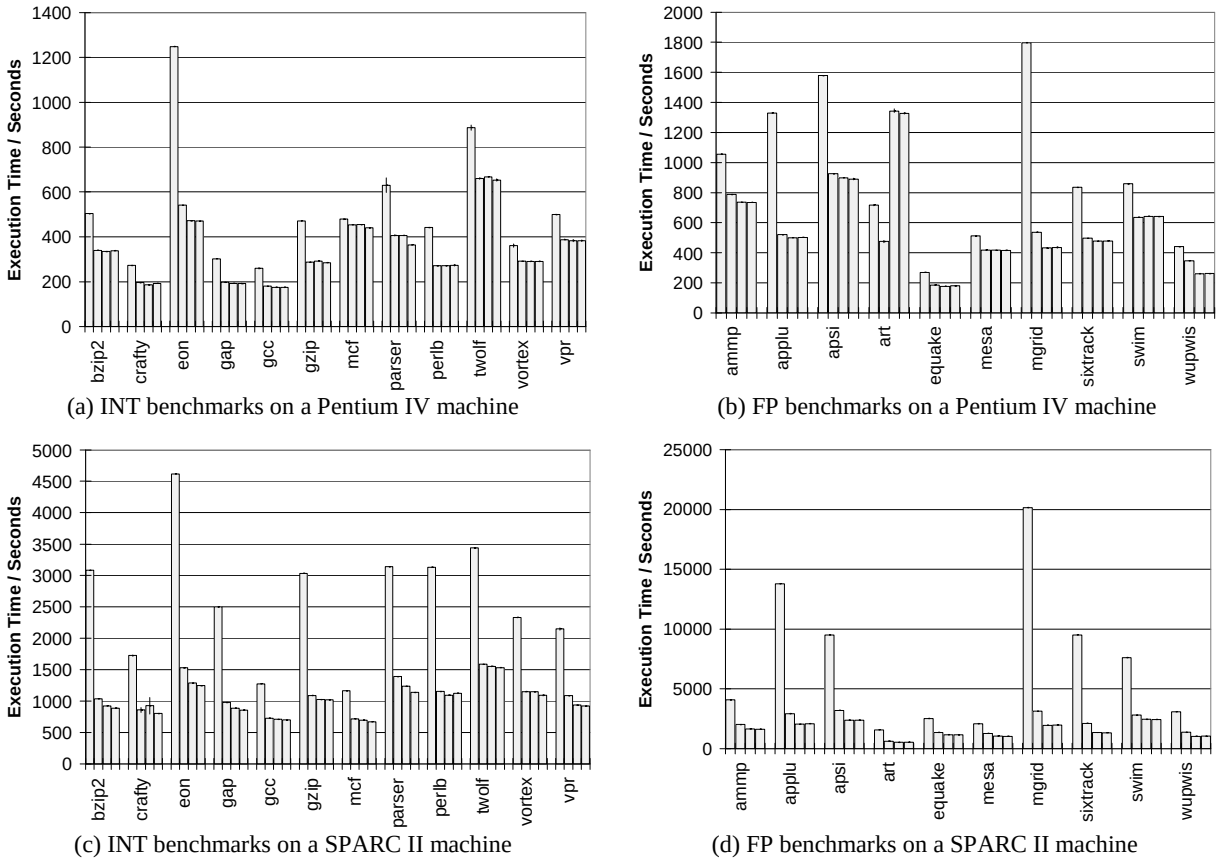
We measured GCC 3.3 [1] on two different computer architectures: a SPARC II machine and a Pentium IV machine. Among all the FP benchmarks, FACEREC, FMA3D, GALGEL, and LUCAS are written in f90. Because GCC cannot currently handle f90, we do not measure them in this paper. We chose GCC even though it may be outperformed by vendor-specific compilers. Our reasons are that GCC is widely used, has many easily controlled compiler optimizations, is portable across many different computer architectures, and its open-source nature helps us to understand the performance behavior. To verify that our results hold beyond the GCC compiler, we have conducted similar experiments with compilers from Sun Microsystems. We have found that these compilers generally outperform GCC, that many of their options cause significant performance degradation as well, and that our orchestration algorithms can yield substantial improvements. Hence, our conclusions are valid for these compiler as well.

To ensure reliable measurements, we ran our experiments multiple times. In several of the subsequent figures we indicate the degree of fluctuation through "error bars". This fluctuation is relevant where the performance gains and losses of an optimization technique are small.

[1] For compatibility, we use g++ 2.95.3 for EON, the only C++ benchmark in SPEC CPU2000.

In this paper, we have measured the impact of compiler optimizations on the overall program performance. For our ultimate goal of tuning the options on the basis of individual code sections, a more fine-grained analysis will be of interest. We expect that these results would be similar to our overall measurements, although potentially different in magnitude.

2.2 Performance of Optimization Levels O0 through O3



(a) INT benchmarks on a Pentium IV machine

(b) FP benchmarks on a Pentium IV machine

(c) INT benchmarks on a SPARC II machine

(d) FP benchmarks on a SPARC II machine

Fig. 2.1. Execution time of SPEC CPU 2000 benchmarks under different optimization levels compiled by GCC. (Four floating point benchmarks written in f90 are not included, since GCC does not compile them.) Each benchmark has four bars for O0 to O3. (a) and (c) show the integer benchmarks; (b) and (d) show the floating point benchmarks. (a) and (b) are the results on a Pentium IV machine; (c) and (d) are the results on a SPARC II machine.

GCC provides four optimization levels, O0 through O3 [1], each applying a larger number of optimization techniques. O0 does not apply any substantial code optimizations. From Figure 2.1, we make the following observations.

1. There is consistent, significant performance improvement from O0 to O1. However, O2 and O3 do not always lead to additional gains; in some cases, performance even degrades. (In Section 2.4 we will analyze the significant degradation of ART). For different applications, any one of the three levels O1 through O3 may be the best.

Table 2.1
Average speedups of the optimization levels, relative to O0. In each entry, the first number is the arithmetic mean, and the second one is the geometric mean. The averages without ART are put in parentheses for the floating point benchmarks on the Pentium IV machine.

	INT Pentium IV	FP Pentium IV	INT SPARC II	FP SPARC II
O1	1.49/1.47	1.74(1.77)/1.65(1.67)	2.32/2.28	3.17/2.88
O2	1.53/1.50	1.81(1.95)/1.60(1.81)	2.50/2.43	4.40/3.78
O3	1.55/1.51	1.80(1.94)/1.60(1.80)	2.58/2.52	4.38/3.79

2. As expected, Table 2.1 shows that O2 is better on average than O1[2] and, for the integer benchmarks, O3 is better than O2. However, for the floating point benchmarks O2 is better than or close to O3. Most of the performance is gained from the optimizations in level O1. The performance increase from O1 to O2 is bigger than that from O2 to O3.

3. Floating point benchmarks benefit more from compiler optimizations than integer benchmarks. Possible reasons are that floating point benchmarks tend to have fewer control statements than integer benchmarks and are written in a more regular way. Six of them are written in Fortran 77.

[2]except the anomalous ART, to be discussed in Section 2.4

4. Optimizations achieve higher performance on the SPARC II machine than on the Pentium IV machine. Possible reasons are the regularity of RISC versus CISC instruction sets and the fact that SPARC II has more registers than Pentium IV. The latter gives the compiler more freedom to allocate registers, resulting in less register spilling on the SPARC II machine.

5. EON, the only C++ benchmark, benefits more from optimization than all other integer benchmarks. On the Pentium IV machine, the highest speedup of EON is 2.65, while the highest one among other integer benchmarks is 1.73. On the SPARC II machine, the highest speedup of EON is 3.70, while the highest one among other integer benchmarks is 3.47.

2.3 Performance of Individual Optimizations

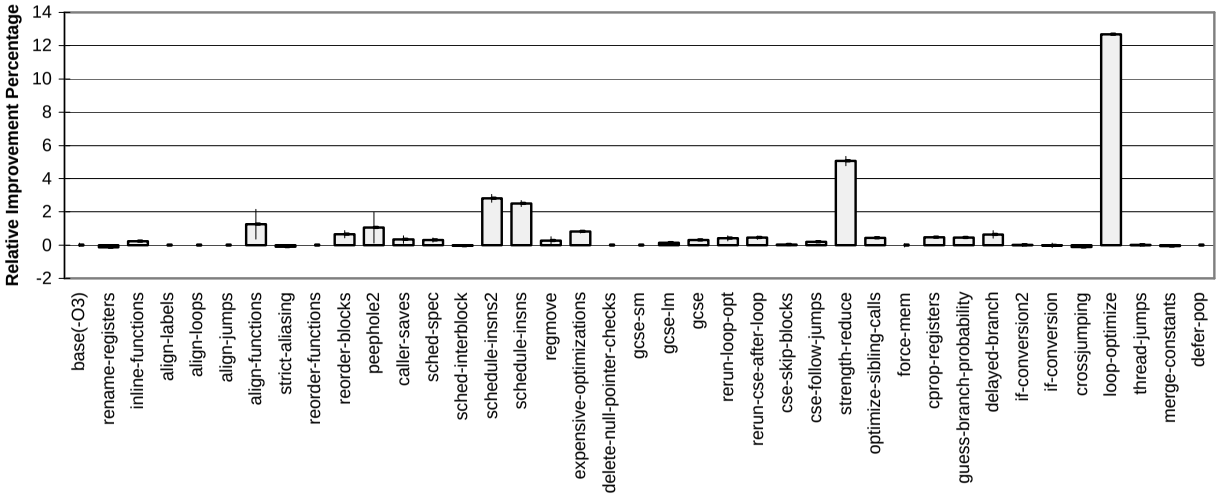
In this section, we discuss the performance of all individual optimization techniques included in the levels discussed previously. GCC includes additional optimization techniques, not enabled by any O-level. We have measured these options as well and found that their effect is generally small. They are not discussed further.

We chose O3 – which generally has the highest performance – as the baseline. We turned off each individual optimization *xxx*, by using the corresponding GCC compiler flag "*-fno-xxx*". We display the results using the metric *Relative Improvement Percentage*, defined as follows:

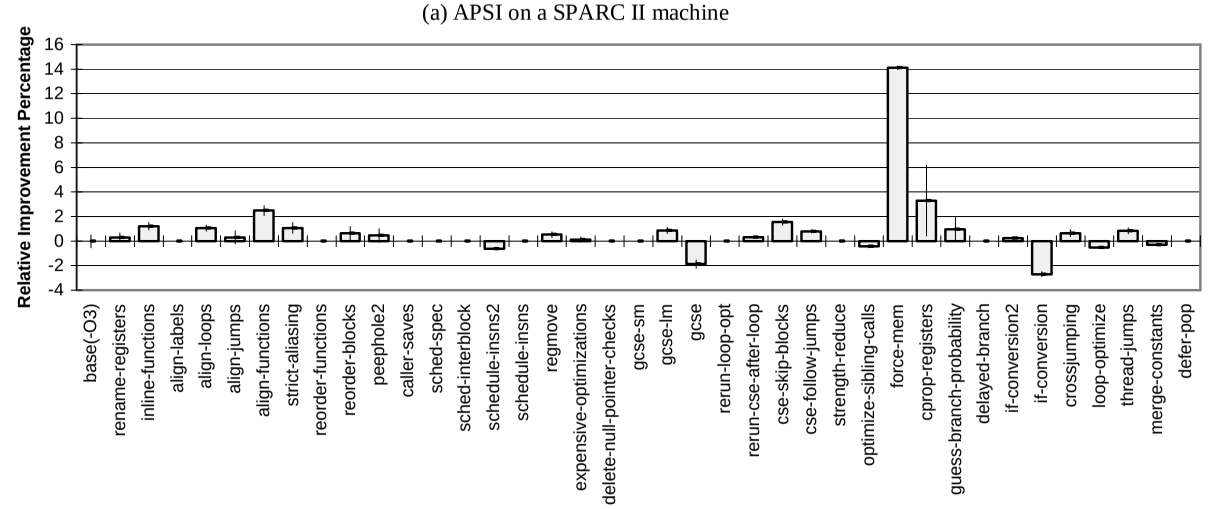
$$RIP = (\frac{execution\_time\_without\_xxx}{execution\_time\_of\_baseline} - 1) * 100$$

RIP represents the percent increase of the program execution time when disabling a given optimization technique. A larger RIP value, indicates a bigger positive impact of the technique.

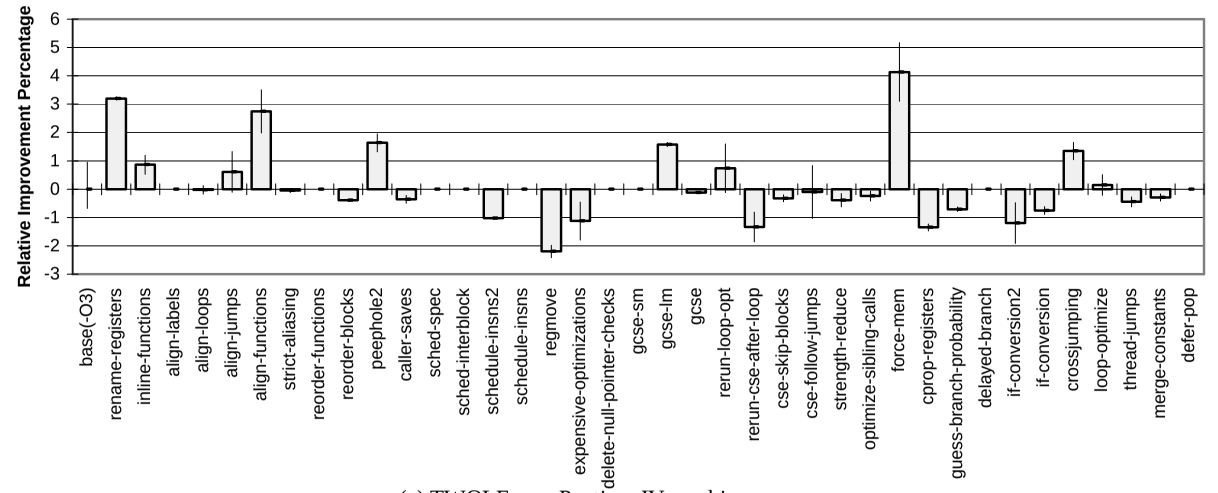
Figures 2.2, 2.3, and 2.4 show the results. We make a number of observations and discuss opportunities and needs for better orchestration of the techniques.



(a) APSI on a SPARC II machine

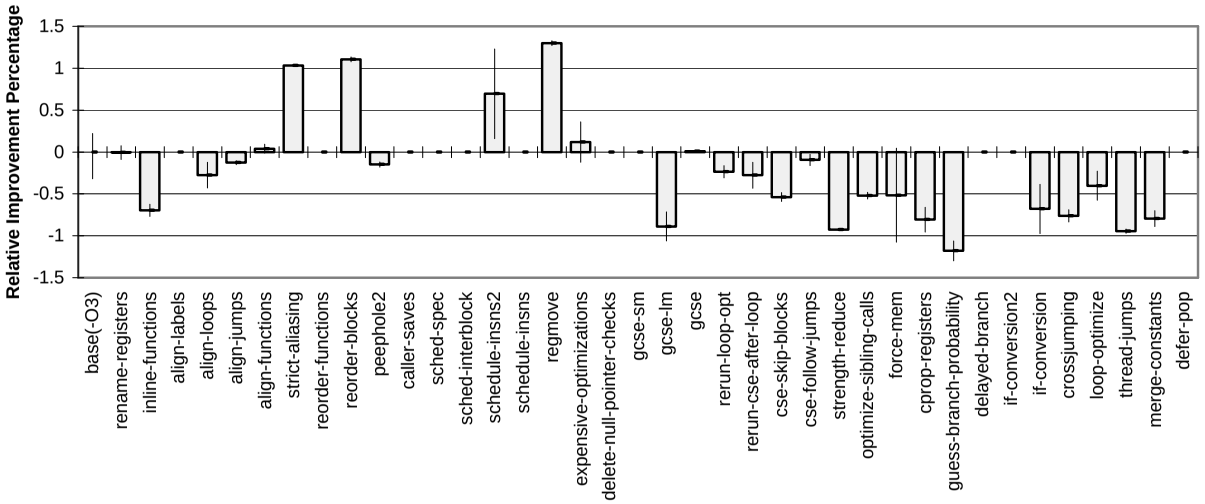


(b) VORTEX on a Pentium IV machine

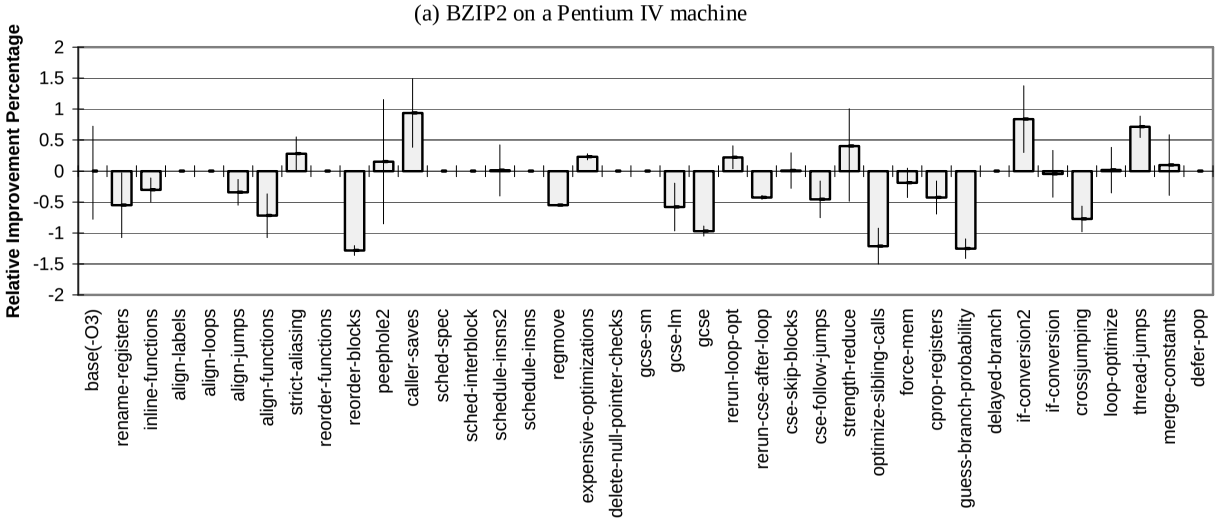


(c) TWOLF on a Pentium IV machine

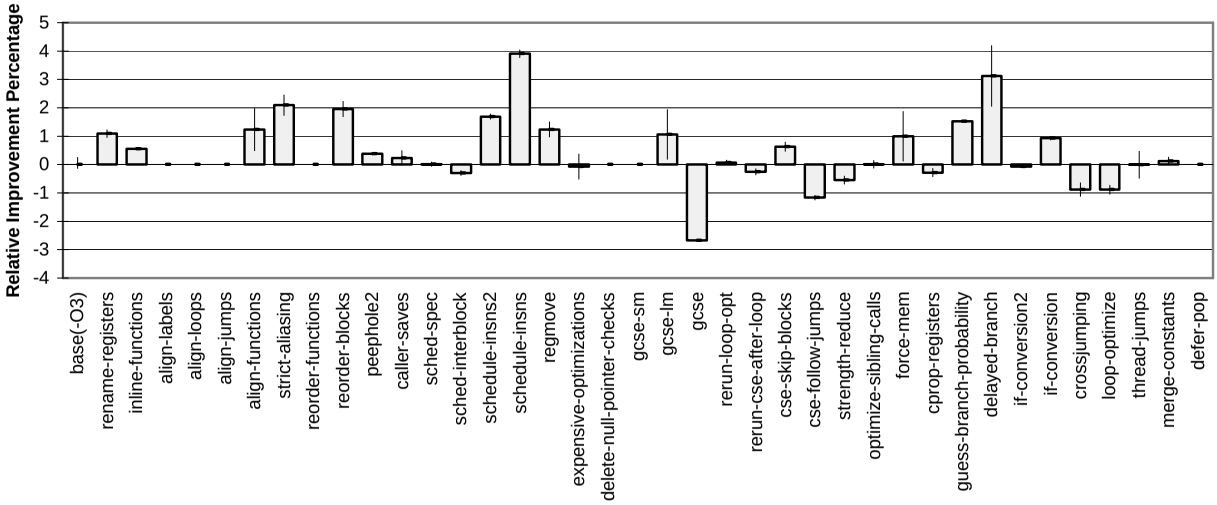
Fig. 2.2. Relative improvement percentage of all individual optimizations within O3 of GCC



(a) BZIP2 on a Pentium IV machine



(b) VPR on a Pentium IV machine



(c) TWOLF on a SPARC II machine

Fig. 2.3. Relative improvement percentage of all individual optimizations within O3 of GCC

1. Ideally, one expects that most optimization techniques yield performance improvements with no degradation. This is the case for APSI on the SPARC II machine, shown in Figure 2.2 (a). This situation indicates little or no need and opportunity for better orchestration.

2. In some benchmarks, only a few optimizations make a significant performance difference, while others have very small effects. VORTEX on Pentium IV Figure 2.2 (b) is such an example. Here also, little opportunity exists for performance gain through better orchestration of the techniques.

3. It is possible that many optimizations cause performance degradation, such as in TWOLF on Pentium IV (Figure 2.2 (c)), or individual degradations are large, as in SIXTRACK on Pentium IV (Figure 2.4). In these cases, better orchestration may help significantly. Section 2.4 will show that the search for the optimum is non-trivial and that performance surprises can happen.

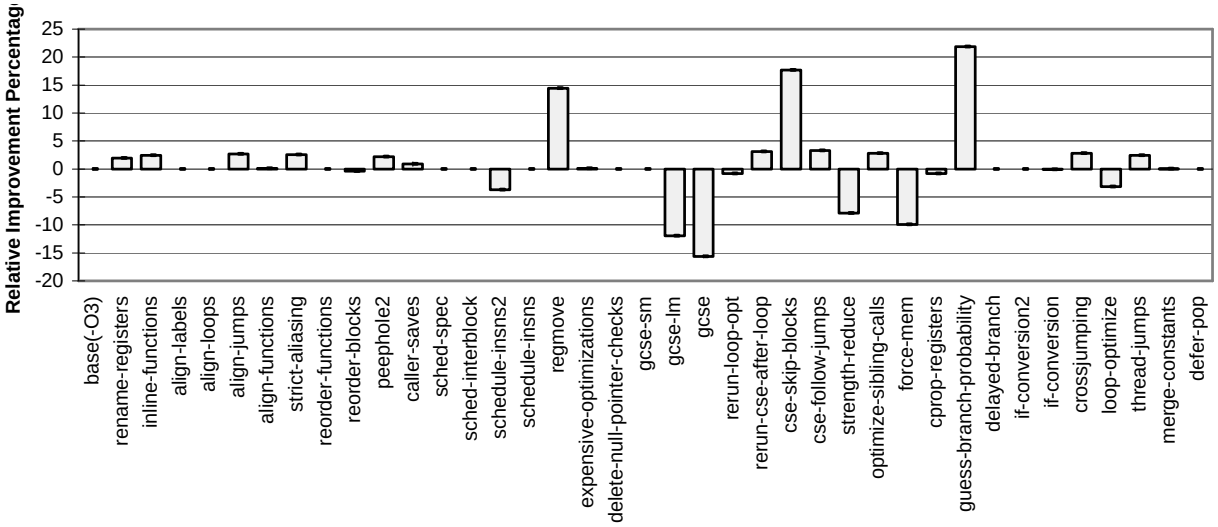


Fig. 2.4. Relative improvement percentage of all individual optimizations within O3 of GCC. SIXTRACK on a Pentium IV machine.

4. In some programs, the RIPs of individual optimizations are between -1.5 and 1.5. For example, in Figure 2.3 (a) and Figure 2.3 (b) the improvements are

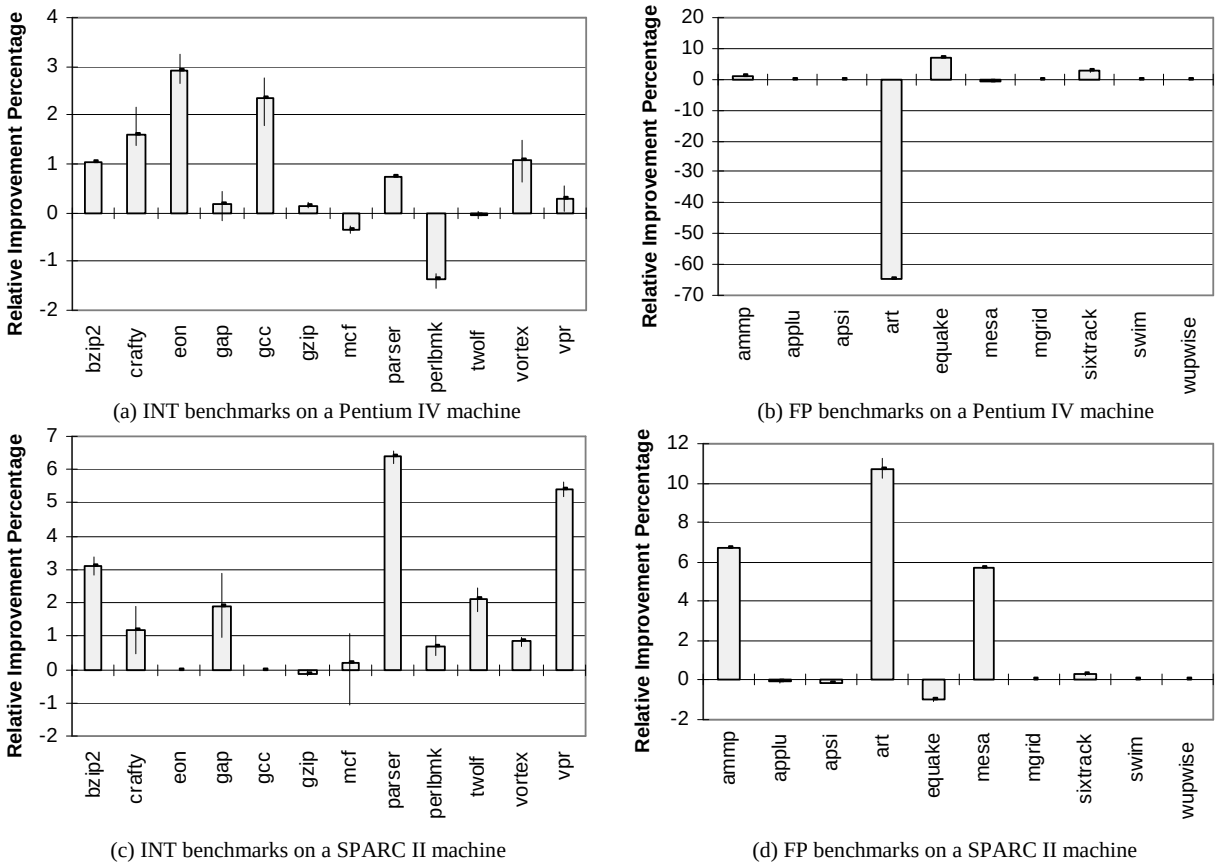
in the same order of magnitude as their variance. While better orchestration may combine small individual gains to a substantial improvement, the need for accurately measuring the performance feedback becomes evident. Effects such as OS activities and interference of other applications must be considered carefully.

5. The performance improvement or degradation may depend on the computer architectures. We compare TWOLF on the SPARC II machine, in Figure 2.3 (c), with TWOLF compiled on the Pentium IV machine, in Figure 2.2 (c). The optimizations causing degradations are completely different on these two platforms. It clearly shows that advanced orchestration must consider the application as well as the execution environment.

We have also found that, in some experiments, few or none of the tested optimizations cause significant speedups (e.g. Figure 2.3 (a)); however, there is significant improvement from O0 to O1, as shown in Figure 2.1. This is because some basic optimizations are not controllable by compiler options. From reading the GCC source code we determined that these optimizations include the expansion of built-in functions, and basic local and global register allocation.

2.4 Selected Harmful GCC Optimizations

In this section, we discuss the major reasons for performance degradations seen in Section 2.3. Some of the major degradations are caused by the techniques strict aliasing, global common subexpression elimination, and if-conversion.



(a) INT benchmarks on a Pentium IV machine

(b) FP benchmarks on a Pentium IV machine

(c) INT benchmarks on a SPARC II machine

(d) FP benchmarks on a SPARC II machine

Fig. 2.5. Relative improvement percentage of strict aliasing.

2.4.1 Strict Aliasing

Strict aliasing is controlled by the flag *strict-aliasing*. When turned on, objects of different types are always assumed to reside at different addresses.[3] If strict aliasing is turned off, GCC assumes the existence of aliases very conservatively [1].

Generally, one expects a throughout positive effect of strict aliasing, as it avoids conservative assumptions. Figure 2.5 confirms this view for most cases. However, the technique also leads to significant degradation in ART, shown in Figure 2.5 (b). The RIP is -64.5.

[3] *Strict-aliasing* in combination with type casting may lead to incorrect programs. We have not observed any such problems in the SPEC CPU2000 benchmarks.

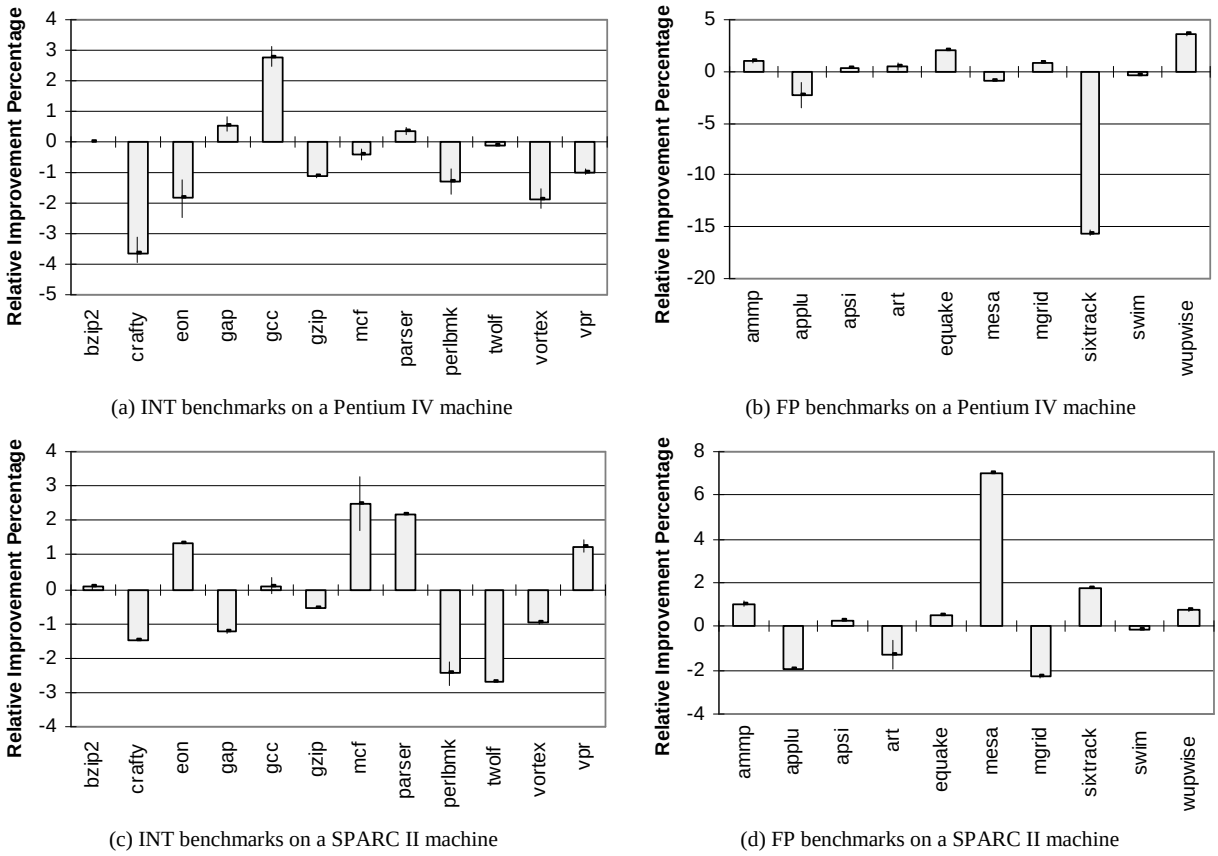
From inspecting the assembly code, we found that the degradation is an effect of the register allocation algorithm. GCC implements a graph coloring register allocator [2, 3]. With strict aliasing, the live ranges of the variables become longer, leading to high register pressure and ‘ spilling. With more conservative aliasing, the same variables incur memory transfers at the end of their (shorter) live ranges as well. However, in the given compiler implementation, the spill code includes substantially more memory accesses than these transfers, and thus causes the degradation in ART.

We also observed from Figure 2.5 (d) that, on the SPARC II machine, strict aliasing does not degrade ART, but improves the performance by 10.7%. We attribute this improvement to less spilling due to the larger number of general purpose registers in the SPARC II than in the Pentium IV processor,

2.4.2 GCSE (Global Common Subexpression Elimination)

GCSE is controlled by the flag *gcse* in GCC. GCSE employs PRE (partial redundancy elimination), global constant propagation, and copy propagation [1]. GCSE removes redundant computation and, therefore, generally improves performance. In rare cases it increases register pressure by keeping the expression values longer. PRE may also create additional move instructions, as it attempts to place the results of the same expression computed in different basic blocks into the same register.

We have also found that GCSE can degrade the performance, as it interacts with other optimizations. In APPLU (Figure 2.6 (b)), we observed a significant performance degradation in the function JACLD. Detailed analysis showed that this problem happens when GCSE is used together with the flag *force-mem*. This flag forces memory operands to be copied into registers before arithmetic operations, which generally improves code by making all memory references potential common subexpressions. However, in APPLU, this pass evidently interferes with the GCSE algorithm. Comparing the assembly code with and without *force-mem*, we found the former recognized fewer common subexpressions.



(a) INT benchmarks on a Pentium IV machine

(b) FP benchmarks on a Pentium IV machine

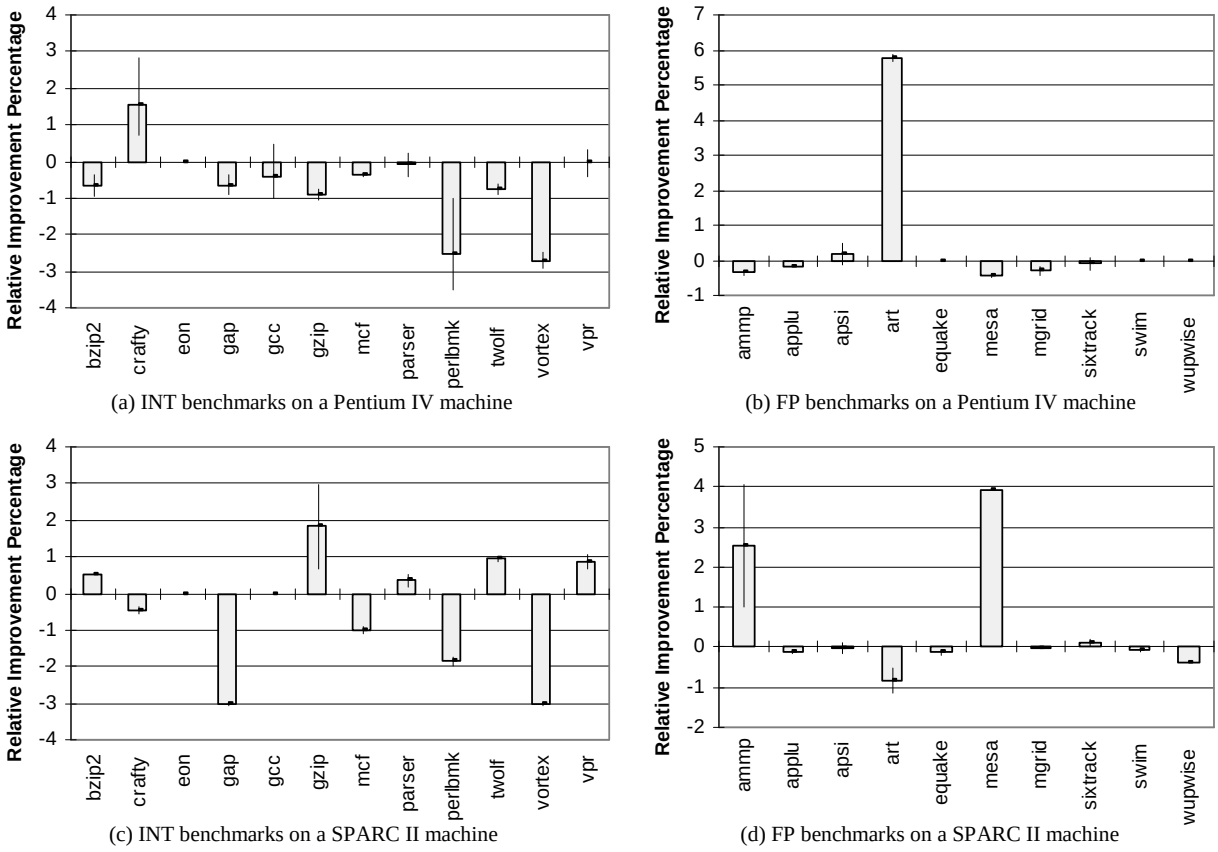
(c) INT benchmarks on a SPARC II machine

(d) FP benchmarks on a SPARC II machine

Fig. 2.6. Relative improvement percentage of global common subexpression elimination.

2.4.3 If-Conversion

If-conversion attempts to transform conditional jumps into branch-less equivalents. It makes use of conditional moves, min, max, set flags and abs instructions, and applies laws of standard arithmetic [1]. If the computer architecture supports predication, if-conversion may be used to enable predicated instructions [9]. By removing conditional jumps, if-conversion not only reduces the number of branches, but also enlarges basic blocks, thus helps scheduling. The potential overhead of such transformations and the opportunities for dynamic optimization are well-known [7]. In our measurements, we found many cases where if-conversion degrades the performance.



(a) INT benchmarks on a Pentium IV machine

(b) FP benchmarks on a Pentium IV machine

(c) INT benchmarks on a SPARC II machine

(d) FP benchmarks on a SPARC II machine

Fig. 2.7. Relative improvement percentage of if-conversion.

In VORTEX, there is a frequently called function named ChkGetChunk, which is called billions of times in the course of the program. After if-conversion, the number of basic blocks is reduced, but the number of instructions in the converted if-then-else construct is still the same. However, the number of registers used in this function is increased, because the coalesced basic block uses more physical registers to avoid spilling. Thus, this function has to save the status of more registers. The additional register saving causes substantial overhead, as the function is called many times

From the above analysis, we make the following observations:

- Optimizations may exhibit unexpected performance behavior. Even generally-beneficial techniques may degrade performance. Degradations are often com-

plex side-effects of the interaction with other optimizations. They are near-impossible to predict analytically.

- Better orchestration techniques of compiler optimizations using knowledge of the observed runtime behavior are called for. The subtle interactions of compiler optimizations requires such techniques to consider combinations rather than decoupled, individual optimizations.

- A larger number of optimization techniques cause performance degradations in integer benchmarks. Integer benchmarks often contain irregular code with many control statements, which tends to reduce the effectiveness of optimizations. On the other hand, larger degradations (of fewer techniques) occur in floating point benchmarks. This is consistent with the generally larger effect of optimization techniques in these programs.

- On different architectures, the optimizations may behave differently. This means that compilers orchestrated for one architecture will not necessarily perform best for others.

3. ALGORITHMS FOR ORCHESTRATING COMPILER OPTIMIZATIONS

The previous sections have indicated significant potential for performance improvement by better orchestrating the compiler optimizations. In this section, we develop three algorithms to efficiently determine the best combination of techniques, while maximizing the application's performance. Each algorithm makes a number of full application runs, using the resulting run times as performance feedback for deciding on the next run. We keep the algorithm general and independent of specific compilers and optimization techniques. It tunes the options available in the given compiler via command line flags.

We can describe our goal as follows:

Given a set of on-off optimization flags $\{F_1...F_n\}$*, find the combination that minimizes the application execution time.*

Because there are interactions between optimizations and the interactions are unpredictable, a simple approach would have to try each combination to find the best. Its complexity would be $O(2^n)$, which is prohibitive. We design three heuristic algorithms, shown in Table 3.1, to obtain the result in polynomial time. We have evaluated the resulting application performance, relative to the default optimization level, -O1. We took the measurements on a Pentium IV machine.

From Section 2.2, we know that any of the three optimization levels, O1 through O3, may be the best among them. This observation leads to the first, basic algorithm, which has complexity $O(1)$. Figure 3.1 shows that this simple algorithm improves the speedup by up to 14.9% (3.9% on average) for integer benchmarks, and up to 32.7% (8.1% on average) for floating point benchmarks on Pentium; and up to 22.6% (11.0%

Table 3.1
Algorithms for Orchestrating Compiler Optimizations

Algorithms	**Description**
Algorithm 1: *Basic Algorithm*	1. Measure all three optimization levels. 2. Choose the best one as the best optimization combination.
Algorithm 2: *Batch Elimination*	1. Measure the RIPs of all individual optimizations. 2. Disable all optimizations with negative RIPs.
Algorithm 3: *Iterative Elimination*	1. Let B be the flag set for measuring the baseline performance. Let the set S represent the optimization search space. Initialize $B = S = \{F_1...F_n\}$ 2. Measure the RIPs of all optimization flags in S relative to the baseline B. 3. Find the optimization flag f with the most negative RIP. Remove f from both B and S. 4. Repeat Step 2 and Step 3 until all flags in S have non-negative RIPs. B represents the final flag combination.

on average) for integer benchmarks, and up to 60.4%(33.3% on average) for floating point benchmarks on SPARC.

The second heuristic is to find the techniques degrading the baseline performance, and then to disable all of them in a batch. Since most optimizations improve performance most of the time, we include all of them in the baseline. The complexity of this algorithm is $O(n)$. Figure 3.1 shows that Algorithm 2 achieves better performance than the first in half of the benchmarks, and that on average it is worse than Algorithm 1. In a few cases Algorithm 2 is even worse than the default optimization combination. The main disadvantage of Algorithm 2 is that it does not consider the interactions of different optimizations. These observations hold for both architectures.

Algorithm 3 eliminates the degrading optimizations one by one in a greedy way. This approach takes the interaction among optimizations into consideration. We include all optimizations into the initial baseline in order to make the algorithm converge faster. This baseline is close to the optimal decision, because the optimizations

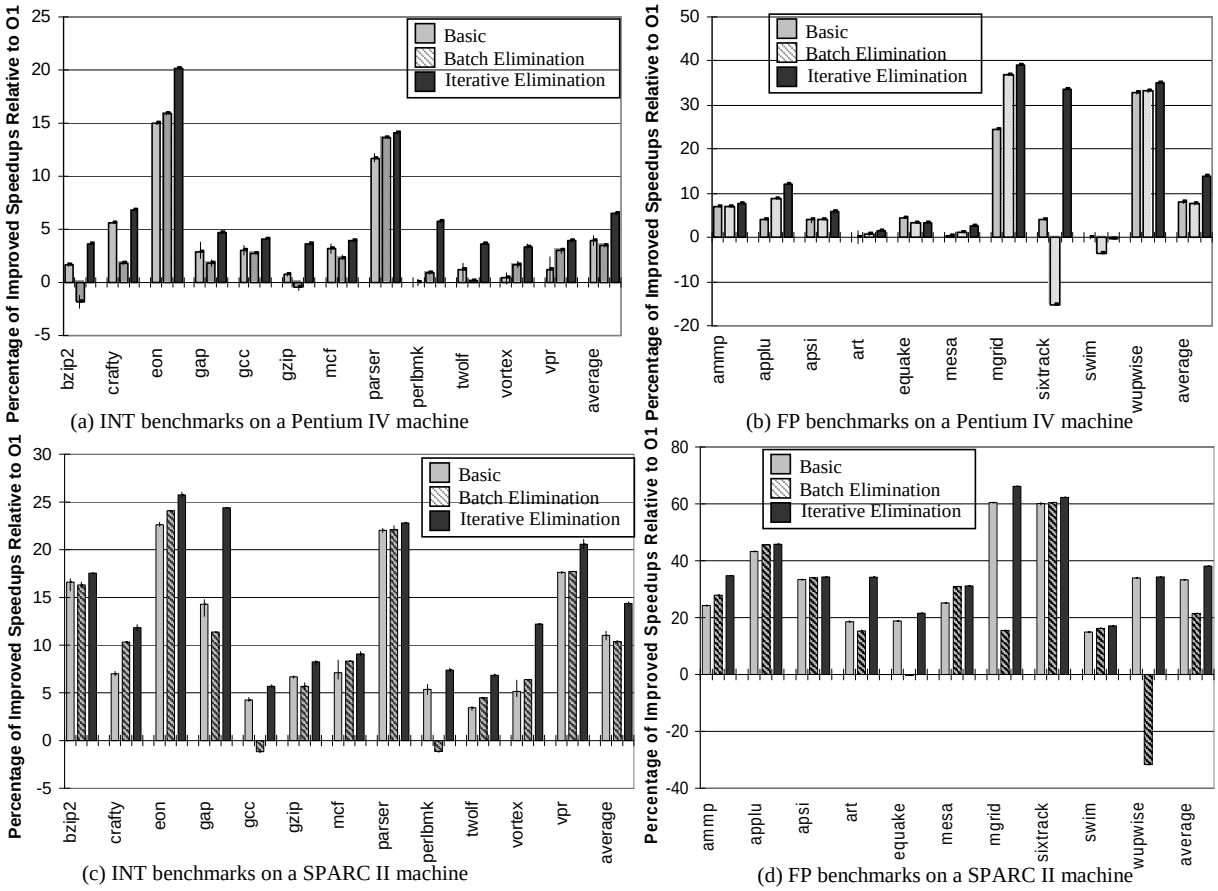


Fig. 3.1. Speedup improvement of three algorithms relative to O1. (a) and (c) show the integer benchmarks; (b) and (d) show the floating point benchmarks. (a) and (b) are the results on a Pentium IV machine; (c) and (d) are the results on a SPARC II machine.

are beneficial in most cases. Therefore, we only need a few iterations, in practice. This algorithm has a worst-case complexity of $O(n^2)$.

Figure 3.1 shows that Algorithm 3 achieves the highest performance in all but one case, EQUAKE on the Pentium IV, and it consistently out-performs Algorithm 2. In EQUAKE, the iterative algorithm is slightly worse than the basic one. This can be due to the greedy nature, which may converge on a local optimum in the complex search space. Figure 3.1 shows many cases where the iterative algorithm is significantly better than Algorithm 1, showing the benefit of orchestrating optimizations.

On the Pentium IV machine, Algorithm 3 improves the speedup over O1 by up to 20.1% (6.5% on average) for integer benchmarks, and up to 38.9% (14.0% on average) for floating point benchmarks. The improvement over O3 is up to 6.4% for integer benchmarks (3.0% on average) and up to 183.8% for floating point benchmarks (24.1% on average). Floating point benchmarks improve more from orchestrating optimizations than the integer benchmarks, which is consistent with the generally higher effect of optimizations on this class of programs. We also observed that integer benchmarks take more time to converge on the Pentium machine. On average, the integer benchmarks need 4 iterations and the floating point benchmarks need 3.3 iterations.

On the SPARC II machine, Algorithm 3 significantly improves the performance as well. The speedup over O1 is improved by up to 25.7% (14.4% on average) for integer benchmarks, and up to 66.2% (38.1% on average) for floating point benchmarks. The improvement over O3 is up to 8.8% (3.3% on average) for integer benchmarks, and up to 13.8% (4.4% on average) for floating point benchmarks. Similar to Pentium IV, floating point benchmarks improve more from orchestrating optimizations than the integer benchmarks. However, integer benchmarks take less time to converge on SPARC II. On average, the integer benchmarks need 2.75 iterations and the floating point benchmarks need 3.9 iterations. Comparing the results from the SPARC II to those from the Pentium IV, we know that different orchestrations achieve the optimal performance on the two machines (SIXTRACK's results are substantially different).

We also considered an improved algorithm to reduce the search time by making the searching set S in Step 3 contain only the flags with negative RIPs, obtained in Step 2. Since the interactions between optimizations can be very strong in some benchmarks, this approach did not result in better performance.

4. CONCLUSIONS

In this paper, we used the metric *Relative Improvement Percentage* to evaluate a large number of compiler optimizations. We found that, although the optimizations are beneficial in many programs, significant degradations may occur. Optimizations perform differently on different applications, different computer platforms, and different compilers. Of particular importance and concern are the interactions between the optimizations, which are subtle and near-impossible to predict. Based on these observations, we have developed three heuristic algorithms to search for the best optimization combination. Among the three algorithms, the iterative method outperforms the other two in all but one insignificant case. This method uses the RIP metrics to identify harmful optimizations, and iteratively removes the harmful optimization, so as to get the best possible flag combination. Our work demonstrates promising performance improvement by orchestrating compiler techniques using measured runtime as feedback.

The presented work is an important step towards our ultimate goal of orchestrating compiler optimizations on the basis of individual functions. Our implementation environment is the ADAPT system [12] which supports the monitoring of application through timers and hardware counters and provides for the application of compiler optimizations in a dynamic, adaptive manner.

The knowledge about performance-degrading optimization techniques and their interactions are not only important for enabling an adaptive compilation process, it also helps compiler engineers to continuously develop better optimization techniques and compiler driver algorithms. Clearly, both methods for advancing compiler capabilities are complementary and necessary to achieve high performance levels for future machine generations.

LIST OF REFERENCES

[1] *GCC online documentation*. http://gcc.gnu.org/onlinedocs/.

[2] Peter Bergner, Peter Dahl, David Engebretsen, and Matthew T. O'Keefe. Spill code minimization via interference region spilling. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 287–295, 1997.

[3] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, May 1994.

[4] Cliff Click and Keith D. Cooper. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(2):181–196, 1995.

[5] Keith D. Cooper, Devika Subramanian, and Linda Torczon. Adaptive optimizing compilers for the 21st century.

[6] Elana D. Granston and Anne Holler. Automatic recommendation of compiler options. In *4th Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*. December 2001.

[7] Kim M. Hazelwood and Thomas M. Conte. A lightweight algorithm for dynamic if-conversion during dynamic optimization. In *2000 International Conference on Parallel Architectures and Compilation Techniques*, pages 71–80, 2000.

[8] Toru Kisuki, Peter M. W. Knijnenburg, Michael F. P. O'Boyle, Francois Bodin, and Harry A. G. Wijshoff. A feasibility study in iterative compilation. In *ISHPC*, pages 121–132, 1999.

[9] J.C.H. Park and M. Schlansker. On predicated execution. Technical Report HPL-91-58, Hewlett-Packard Software Systems Laboratory, May 1991.

[10] Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. Meta optimization: improving compiler heuristics with machine learning. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 77–90. ACM Press, 2003.

[11] Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I. August. Compiler optimization-space exploration. In *Proceedings of the international symposium on Code generation and optimization*, pages 204–215, 2003.

[12] Michael J. Voss and Rudolf Eigemann. High-level adaptive program optimization with adapt. In *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 93–102. ACM Press, 2001.

[13] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. Technical Report UT-CS-97-366, 1997.

[14] Michael E. Wolf, Dror E. Maydan, and Ding-Kai Chen. Combining loop transformations considering caches and scheduling. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 274–286, 1996.

[15] Kamen Yotov, Xiaoming Li, Gang Ren, Michael Cibulskis, Gerald DeJong, Maria Garzaran, David Padua, Keshav Pingali, Paul Stodghill, and Peng Wu. A comparison of empirical and model-driven optimization. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 63–76. ACM Press, 2003.