Fast and Effective Orchestration of Compiler Optimizations for Automatic Performance Tuning [*]

Zhelong Pan   Rudolf Eigenmann
Purdue University, School of ECE
West Lafayette, IN, 47907
{zpan, eigenman}@purdue.edu

Abstract

Although compile-time optimizations generally improve program performance, degradations caused by individual techniques are to be expected. One promising research direction to overcome this problem is the development of dynamic, feedback-directed optimization orchestration *algorithms, which automatically search for the combination of optimization techniques that achieves the best program performance. The challenge is to develop an orchestration algorithm that finds, in an exponential search space, a solution that is close to the best, in acceptable time. In this paper, we build such a fast and effective algorithm, called* Combined Elimination (CE)*. The key advance of CE over existing techniques is that it takes the least tuning time (57% of the closest alternative), while achieving the same program performance. We conduct the experiments on both a Pentium IV machine and a SPARC II machine, by measuring performance of SPEC CPU2000 benchmarks under a large set of 38 GCC compiler options. Furthermore, through orchestrating a small set of optimizations causing the most degradation, we show that the performance achieved by CE is close to the upper bound obtained by an exhaustive search algorithm. The gap is less than 0.2% on average.*

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 writers and many users. The state of the art is to let the users deal with this problem through compiler options. The presence of compiler options reflects the inability of today's optimizers to make optimal choices at compile time. The unavailability of program input data and insufficient knowledge of the target architecture can severely limit the accuracy of compile-time performance models. Thus, the determination of the best combination of compiler optimizations for a given program or program section remains an unattainable compile-time goal. Today's compilers have evolved to the point where they present to the user a large number of options. For example, GCC compilers include 38 options, roughly grouped into three optimization levels, $O1$ through $O3$. On the other hand, compiler optimizations interact in unpredictable manners, as many have observed [5, 11, 13, 15, 18]. A fast and effective *orchestration* algorithm to search for the best optimization combination for a program is desired.

Several automatic performance tuning systems have taken a dynamic, feedback-directed approach to orchestrate compiler optimizations. In this approach, many different binary code versions generated under different *experimental optimization combinations* are being evaluated. The performance of these versions is compared using either measured execution times or profile-based estimates. Iteratively, the orchestration algorithms use this information to decide the next experimental optimization combinations, until convergence criteria are reached.

The new algorithms presented in this paper follow the above model. We first develop two simple algorithms: (a) *Batch Elimination* (BE) identifies the harmful optimizations and removes them in a batch. (b) *Iterative Elimination* (IE) successively removes harmful optimizations, measured through a series of program executions. Based on the above two algorithms, we design our final algorithm, *Combined Elimination* (CE). We compare our algorithms with two algorithms proposed in the literature: (i) The "compiler construction-time pruning" algorithm in *Optimization-Space Exploration* (OSE) [18] iteratively constructs new optimization combinations using "unions" of the ones in the previous iteration. (ii) *Statistical Selection* (SS) in [15] uses orthogonal arrays [9] to compute the main effect of the optimizations based on a statistical analysis of profile infor-

[*]This work was supported, in part, by the National Science Foundation under Grants No. 9974976-EIA, 0103582-EIA, and 0429535-CCF.

mation, which in turn is used to find the best optimization combination.

In addition to the above algorithms that we compare our work with, several other approaches have been proposed. Typically, they need more than hundreds of compilations and experimental runs, when tuning a large number of optimizations (38 optimizations in our experiments). The goal of our work is to reduce this number to several tens, while achieving comparative or even better program performance. For the large number of benchmarks and optimizations experimented in this paper, we can only apply the algorithms in [18] and [15], which are closest to our new algorithm, CE, in terms of tuning time. To further verify that our CE algorithm achieves program performance comparable to other existing algorithms, we use a small set of optimizations and show that CE closely approaches the upper bound represented by exhaustive search. The other existing algorithms are as follows.

In [5], a fractional factorial design is developed based on aliasing or confounding [4]; it illustrates a half-fraction design with 2^{n-1} experiments. In [7], heuristics are designed to select PA-RISC compiler options based on information from the user, the compiler, and the profiler. While the use of *a priori* knowledge of the interaction between optimization techniques may reduce the complexity of the search for the best, it has been found by others [13] that the number of techniques that *potentially* interact is still large. ATLAS [20] starts with a parameterized, hand-coded set of matrix multiplication variants and evaluates them on the target machine to determine the optimum settings for that context. Similarly, Iterative Compilation [11] searches through the transformation space to find the best block sizes and unrolling factors. In more recent research [10], five different algorithms, genetic algorithm, simulated annealing, grid search, window search and random search are exploited to find the best blocking and unrolling parameters. Based on the random search in [10], some aim to find a general compiler optimization settings using GCC [8]. Meta optimization [17] uses machine-learning techniques to adjust the compiler heuristics automatically.

Different from our goal of finding the best optimization combination is finding the best *order* of optimization phases. In [6], a biased random search and genetic algorithm is used to discover the best order of optimizations. Others have added hill climbing and greedy constructive algorithms [3]. Furthermore, genetic algorithm has been improved to reduce search time [12].

In this paper, we make the following contributions:

- We present a new performance tuning algorithm, *Combined Elimination (CE)*, which aims at picking the best set of compiler optimizations for a program. We show that this algorithm takes the shortest tuning time, while achieving comparable or better performance than other algorithms. Using a small set of (6) important optimizations, we also verify that CE closely approaches the performance upper bound.
- We evaluate our and other algorithms on a large set of realistic programs. We use all 23 SPEC CPU2000 benchmarks that are amenable to the GCC compiler infrastructure (omitting 5 benchmarks, written in F90 and C++). By contrast, many previous papers have used small kernel benchmarks. Among the papers that used a large set of SPEC benchmarks are [13, 14, 18].
- Our experiments use all (38) GCC O3 options, where the speed of the tuning algorithm becomes of decisive importance. Except [8] and [18] that also use a large number of optimizations, previous papers have generally evaluated a small set of optimizations.

Using the full set of GCC O3 optimizations, the average *normalized tuning time*, which will be defined formally in Section 3.2, is 75.3 for our *CE* algorithm; 131.2 for the *OSE* algorithm described in Section 2.2.5; 313.9 for the *SS* algorithm described in Section 2.2.6. Hence, CE reduces tuning time to 57% of the closest alternative. CE improves performance by 6.01%, over O3, the highest optimization level; OSE by 5.68%; SS by 5.46%. (Compared to unoptimized programs, performance improvement achieved by CE would amount to 56.4%, on average.)

The remainder of this paper is organized as follows. In Section 2, we describe the orchestration algorithms that we use in our comparison. In Section 3, we compare tuning time and tuned program performance of these algorithms under 38 optimizations. In Section 4, we compare the performance of CE with the upper bound obtained using exhaustive search under a smaller set of optimizations.

2 Orchestration Algorithms

2.1 Problem Description

We define the goal of optimization orchestration as follows:

Given a set of compiler optimization options $\{F_1, F_2, ..., F_n\}$*, find the combination that minimizes the program execution time. Do this efficiently, without the use of a priori knowledge of the optimizations and their interactions. (Here,* n *is the number of optimizations.)*

In this section, we give an overview of several algorithms that pursue this goal. We first present the exhaustive search algorithm, ES. Then, we develop two of our algorithms, BE and IE, on which our final CE method builds, followed by CE itself. Next, we present two existing algorithms, OSE and SS, with which our algorithm compares. Each algorithm makes a number of full program runs, using the resulting run times as performance feedback for deciding on the next run. We keep the algorithms general and independent of specific compilers and optimization techniques. The algorithms tune the options available in the given compiler via command line flags. Here, we focus on *on-off* options, similar to several of the papers [5, 13, 14, 15]. In Section 2.3, we briefly discuss extensions to handle other types of options.

2.2 Orchestration Algorithms

2.2.1 Algorithm 1: Exhaustive Search (ES)

Due to the interaction of compiler optimizations, the exhaustive search approach, which is called the *factorial design* in [5, 15], would try every optimization combination to find the best. This approach provides an upper bound of an application's performance after optimization orchestration. However, its complexity is $O(2^n)$, which is prohibitive if a large number of optimizations are involved. For 38 optimizations in our experiments, it would take up to 2^{38} program runs – a million years for a program that runs in two minutes. We will not evaluate this algorithm under the full set of options. However, Section 4 will use a feasible set of (6) options to compare our algorithm with this upper bound. Using pseudo code, ES can be described as follows.

1. Get all 2^n combinations of n options, $\{F_1, F_2, ..., F_n\}$.
2. Measure application execution time of the optimized *version* compiled under every possible combination.
3. The best version is the one with the least execution time.

2.2.2 Algorithm 2: Batch Elimination (BE)

The idea of *Batch Elimination (BE)* is to identify the optimizations with negative effects and turn them off at once. BE achieves good program performance, when the optimizations do not interact with each other. It is the fastest among the feedback-directed algorithms.

The negative effect of one optimization, F_i, can be represented by its *Relative Improvement Percentage (RIP)*, $RIP(F_i)$, which is the relative difference of the execution times of the two versions with and without F_i, $T(F_i = 1)$ and $T(F_i = 0)$. $F_i = 1$ means F_i is on, 0 means off.

$$RIP(F_i) = \frac{T(F_i = 0) - T(F_i = 1)}{T(F_i = 1)} \times 100\% \quad (1)$$

The baseline of this approach switches on all optimizations. $T(F_i = 1)$ is the execution time of the baseline T_B as shown in Equation 2. The performance improvement by switching off F_i from the baseline B relative to the baseline performance can be computed with Equation 3.

$$T_B = T(F_i = 1) = T(F_1 = 1, F_2 = 1, ..., F_n = 1) \quad (2)$$

$$RIP_B(F_i = 0) = \frac{T(F_i = 0) - T_B}{T_B} \times 100\% \quad (3)$$

If $RIP_B(F_i = 0) < 0$, the optimization of F_i has a negative effect. The BE algorithm eliminates the optimizations with negative RIPs in a batch to generate the final, tuned version. This algorithm has a complexity of O(n).

1. Compile the application under the baseline $B = \{F_1 = 1, F_2 = 1, ..., F_n = 1\}$. Execute the generated code version to get the baseline execution time T_B.
2. For each optimization F_i, switch it off from B and compile the application. Execute the generated version to get $T(F_i = 0)$, and compute the $RIP_B(F_i = 0)$ according to Equation 3.
3. Disable all optimizations with negative RIPs to generate the final, tuned version.

2.2.3 Algorithm 3: *Iterative Elimination (IE)*

We design *Iterative Elimination (IE)* to take the interaction of optimizations into consideration. Unlike BE, which turns off all the optimizations with negative effects at once, IE iteratively turns off one optimization with the most negative effect at a time.

IE starts with the baseline that switches on all the optimizations. After computing the RIPs of the optimizations according to Equation 3, IE switches off the one optimization with the most negative effect from the baseline. This process repeats with all remaining optimizations, until none of them causes performance degradation. The complexity of IE is $O(n^2)$.

1. Let B be the option combination for measuring the baseline execution time, T_B. Let the set of S represent the optimization search space. Initialize $S = \{F_1, F_2, ..., F_n\}$ and $B = \{F_1 = 1, F_2 = 1, ..., F_n = 1\}$.
2. Compile and execute the application under the baseline setting to get the baseline execution time T_B.
3. For each optimization $F_i \in S$, switch F_i off from B and compile the application, execute the generated code version to get $T(F_i = 0)$, and compute the RIP of F_i relative to the baseline B, $RIP_B(F_i = 0)$, according to Equation 3.
4. Find the optimization F_x with the most negative RIP. Remove F_x from S, and set F_x to 0 in B.
5. Repeat Steps 2, 3 and 4 until all options in S have non-negative RIPs. B represents the final option combination.

2.2.4 Algorithm 4: Combined Elimination (CE)

CE, our final algorithm, combines the ideas of the two algorithms just described. It has a similar iterative structure as

IE. However, in each iteration, CE applies the idea of BE: after identifying the optimizations with negative effects, in this iteration, CE tries to eliminate these optimizations one by one in a greedy fashion.

We will see, in Section 3, that IE achieves better program performance than BE, since it considers the interaction of optimizations. However, when the interactions have only small effects, BE may perform close to IE in a faster way. CE takes the advantages of both BE and IE. When the optimizations interact weakly, CE eliminates the optimizations with negative effects in one iteration, just like BE. Otherwise, CE eliminates them iteratively, like IE. As a result, CE achieves both good program performance and fast tuning speed. CE has the complexity of $O(n^2)$.

1. Let B be the baseline option combination. Let the set of S represent the optimization search space. Initialize $S = \{F_1, F_2, ..., F_n\}$ and $B = \{F_1 = 1, F_2 = 1, ..., F_n = 1\}$.
2. Compile and execute the application under the baseline setting to get the baseline execution time T_B. Measure the $RIP_B(F_i = 0)$ of each optimization option F_i in S relative to the baseline B.
3. Let $X = \{X_1, X_2, ..., X_l\}$ be the set of optimization options with negative RIPs. X is sorted in an increasing order, that is, the first element, X_1, has the most negative RIP. Remove X_1 from S and set X_1 to 0 in B. (B is changed in this step.) For i from 2 to l,
   $*$ Measure the RIP of X_i relative to the baseline B.
   $*$ If the RIP of X_i is negative, remove X_i from S and set X_i to 0 in B.
4. Repeat Steps 2 and 3 until all options in S have non-negative RIPs. B represents the final solution.

2.2.5 Algorithm 5: Optimization Space Exploration(OSE)

In [18], the following method is used to orchestrate optimizations. First, a "compiler construction-time pruning" algorithm selects a small set of optimization combinations that perform well on a given set of code segments. Then, these combinations are used to construct a search tree, which is traversed to find good combinations for code segments in a target program. To fairly compare this method with other orchestration algorithms, we slightly modify the "compiler construction-time pruning" algorithm, which is then referred to as the OSE algorithm. (In [18], the pruning algorithm aims at finding a set of good optimization combinations; while the modified OSE algorithm in this paper finds the best of this set. The modified algorithm is applied to the whole application instead of code segments.)

The basic idea of the pruning algorithm is to iteratively find better optimization combinations by merging the beneficial ones. In each iteration, a new test set Ω is constructed by merging the optimization combinations in the old test set using "union" operations. Next, after evaluating the optimization combinations in Ω, the size of Ω is reduced to m by dropping the slowest combinations. The process repeats until the performance increase in the Ω set of two consecutive iterations becomes negligible. The complexity of OSE is $O(m^2 * n)$. We use the same $m = 12$ as in [18]. Roughly, m can be viewed as $O(n)$, hence, the complexity of OSE is approximately $O(n^3)$. The specific steps are as follows:

1. Construct a set, Ω, which consists of the default optimization combination, and n combinations, each of which assigns a non-default value to a single optimization. (In our experiments, the default optimization combination, O3, turns on all optimizations. The non-default value for each optimization is off.)
2. Measure the application execution time for each optimization combination in Ω. Keep the m fastest combinations in Ω, and drop the rest.
3. Construct a new Ω set, each element in which is a union of two optimization combinations in the old Ω set. (The "union" operation takes non-default values of the options in both combinations.)
4. Repeat Steps 2 and 3, until no new combinations can be generated or the increase of the fastest version in Ω becomes negligible. We use the fastest version in the final Ω as the final version .

2.2.6 Algorithm 6: Statistical Selection (SS)

SS was developed in [15]. It uses a statistical method to identify the performance effect of the optimization options. The options with positive effects are turned on, while the ones with negative effects are turned off in the final version, in an iterative fashion. This statistical method takes the interaction of optimizations into consideration.

The statistical method is based on *orthogonal arrays (OA)*, which have been proposed as an efficient design of experiments [4, 9]. Formally, an OA is an $m \times k$ matrix of zeros and ones. Each column of the array corresponds to one compiler option. Each row of the array corresponds to one optimization combination. SS uses the OA with strength 2, that is, two arbitrary columns of the OA contain the patterns $00, 01, 10, 11$ equally often. Our experiments use the OA with 38 options and 40 rows, which is constructed based on a Hadamard matrix taken from [16].

By a series of program runs, this SS approach identifies the options that have the largest effect on code performance. Then, it switches on/off those options with a large positive/negative effect. After iteratively applying the above solution to the options that have not been set, SS finds an optimal combination of the options. *SS* has a complexity of $O(n^2)$. The pseudo code is as follows.

1. Compile the application with each row from orthogonal array A as the compiler optimization combination and execute the optimized version.
2. Compute the *relative effect*, $RE(F_i)$, of each option using Equations 4 and 5, where $E(F_i)$ is the *main effect* of F_i, s is one row of A, $T(s)$ is the execution time of the version under s.

$$E(F_i) = \frac{(\sum_{s \epsilon A: s_i=1} T(s) - \sum_{s \epsilon A: s_i=0} T(s))^2}{m} \quad (4)$$

$$RE(F_i) = \frac{E(F_i)}{\sum_{j=1}^{k} E(F_j)} \times 100\% \quad (5)$$

3. If the relative effect of an option is larger than a threshold of 10%,
   $*$ if the option has a positive *improvement*, $I(F_i) > 0$, according to Equation 6, switch the option on.
   $*$ else if it has a negative improvement, switch the option off.

$$I(F_i) = \frac{\sum_{s \epsilon A: s_i=0} T(s) - \sum_{s \epsilon A: s_i=1} T(s)}{\sum_{s \epsilon A: s_i=0} T(s)} \quad (6)$$

4. Construct a new orthogonal array A by dropping the columns corresponding to the options selected in the previous step.
5. Repeat all above steps until all of the options are set.

2.2.7 Summary of the Orchestration Algorithms

The goal of optimization orchestration is to find the optimal point in a high-dimension space $S = F_1 \times F_2 \times ... \times F_n$. BE probes each dimension to find and adopt the ones that benefit performance. SS works in a similar way, but via a statistical and iterative approach. OSE probes multiple directions, each of which may involve multiple dimensions, and searches along the direction combinations that may benefit performance. IE probes each dimension and fixes the dimension that achieves the most performance at a time. CE probes each dimension and greedily fixes the dimensions that benefit performance at each iteration.

Table 1 summarizes the complexities of all six algorithms compared in this paper.

Table 1. Orchestration algorithm complexity (n is the number of optimization options.)

ES	BE	IE	OSE	SS	CE
$O(2^n)$	$O(n)$	$O(n^2)$	$O(n^3)$	$O(n^2)$	$O(n^2)$

2.3 Non-on-off Options

We will extend our CE algorithm to handle "non-on-off" options in the future, although our experiments have been done with "on-off" options, for simplicity. (All the GCC O3 optimization options are of this "on-off" type.) An example of a non-on-off option is the "-unroll" option in Forte compilers [1], which takes an argument indicating the degree of loop unrolling. The extended CE method will tune these options by trying several values instead of just *on* and *off*. The techniques developed in [10] will also be included in CE to handle options with a large number of possible values, for example, blocking factors.

3 Experimental Results

3.1 Experimental Environment

We evaluate our algorithm using the optimization options of the GCC 3.3.3 compiler on two different computer architectures: Pentium IV and SPARC II. Our reasons for choosing GCC is that this compiler is widely used, has many easily accessible compiler optimizations, and is portable across many different computer architectures.

In this section, we use all 38 optimization options implied by "O3", the highest optimization level. These options are listed in Table 2 and are described in the GCC manual [2].

Table 2. Optimization options in GCC 3.3.3

F_1	rename-registers	F_2	inline-functions
F_3	align-labels	F_4	align-loops
F_5	align-jumps	F_6	align-functions
F_7	strict-aliasing	F_8	reorder-functions
F_9	reorder-blocks	F_{10}	peephole2
F_{11}	caller-saves	F_{12}	sched-spec
F_{13}	sched-interblock	F_{14}	schedule-insns2
F_{15}	schedule-insns	F_{16}	regmove
F_{17}	expensive-optimizations	F_{18}	delete-null-pointer-checks
F_{19}	gcse-sm	F_{20}	gcse-lm
F_{21}	gcse	F_{22}	rerun-loop-opt
F_{23}	rerun-cse-after-loop	F_{24}	cse-skip-blocks
F_{25}	cse-follow-jumps	F_{26}	strength-reduce
F_{27}	optimize-sibling-calls	F_{28}	force-mem
F_{29}	cprop-registers	F_{30}	guess-branch-probability
F_{31}	delayed-branch	F_{32}	if-conversion2
F_{33}	if-conversion	F_{34}	crossjumping
F_{35}	loop-optimize	F_{36}	thread-jumps
F_{37}	merge-constants	F_{38}	defer-pop

We take our measurements using all SPEC CPU2000 benchmarks written in F77 and C, which are amenable to GCC. 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. Our overall tuning process is similar to profile-based optimizations. A *train* dataset is used to tune the program. A different input, the SPEC *ref* dataset, is usually used to measure performance. To separate the performance effects attributed to the tuning algorithms from those caused by the input sets, we measure program performance under both the *train* and *ref* datasets. For our detailed comparison of the tuning algorithms, we will start with the *train* set. In Section 3.3.2, we will show that, overall, the tuned benchmark suite achieves similar performance improvement under the *train* and *ref* datasets.

To ensure accurate measurements and eliminate perturbation by the operating system, we re-execute each code version multiple times under a single-user environment, until the three least execution times are within a range of $[-1\%,1\%]$. In most of our experiments, each version is executed exactly three times. Hence, the impact on tuning time is negligible.

In our experiments, the same code version may be generated under different optimization combinations. This observation allows us to reduce tuning time. We keep a repository of code versions generated under different optimization combinations. The repository allows us to memorize and reuse their performance results. Different orchestration algorithms use their own repositories and get affected in similar ways, so that our comparison remains fair.

3.2 Metrics

Two important metrics characterize the behavior of orchestration algorithms.

1. The *program performance* of the best optimized version found by the orchestration algorithm. We define it as the performance improvement percentage of the best version relative to the base version under the highest optimization level O3.

2. The total *tuning time* spent in the orchestration process. Because the execution times of different benchmarks are not the same, we normalize the tuning time (TT) by the time of evaluating the base version, i.e., one compilation time (CT_B) plus three execution times (ET_B) for the base version.

$$NTT = TT/(CT_B + 3 \times ET_B) \qquad (7)$$

This *normalized tuning time* (NTT) roughly represents the number of experimented versions. (The number may be larger or smaller than the actual number of tested optimization combinations due to three effects: a) Some optimizations may not have any effect on the program, allowing the version repository to reduce the number of experiments. b) Perturbation filtering mechanism in Section 3.1 may increase the number of runs of some versions. c) The experimental versions may be faster or slower than the base version.)

A good optimization orchestration method is meant to achieve both high *program performance* and short *normalized tuning time*. We will show that our CE algorithm has the shortest tuning time, while achieving comparable or better performance than other algorithms.

3.3 Results

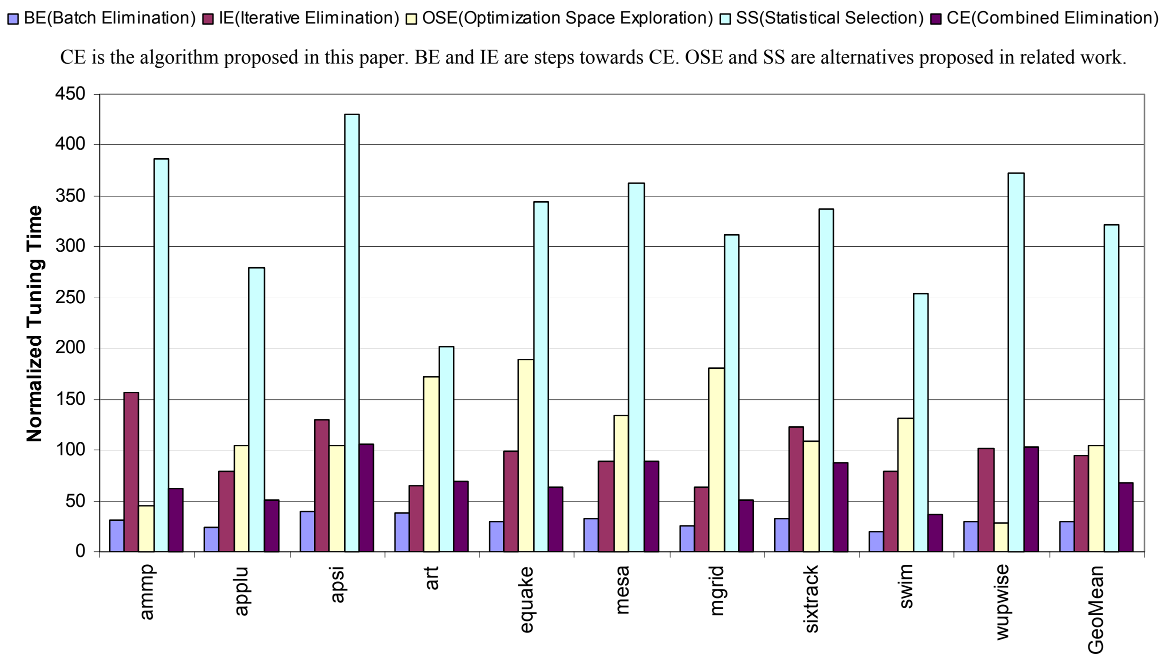
In this section, we compare our final optimization orchestration algorithm CE with the four algorithms BE, IE, OSE and SS. Recall, that BE and IE are steps towards CE; OSE and SS are algorithms proposed in related work. Figure 1 and Figure 2 show the results of these five orchestration algorithms on the Pentium IV machine for the SPEC CPU2000 FP and INT benchmarks in terms of the two metrics. They provide evidence for our claim that CE has the fastest tuning speed while achieving program performance comparable to the best alternatives. We will discuss the basic BE method first, then the other four algorithms. Tuning time will be analyzed first, then program performance.

3.3.1 Tuning Time

For the applications used in our experiments, the slowest of the measured algorithms takes up to several days to orchestrate the large number of optimizations. Figure 1(a) and Figure 1(b) show that our new algorithm, CE, is the fastest among the four orchestration algorithms that consider interactions. The absolute tuning time, for CE, is 2.19 hours, on average, for FP benchmarks and 3.66 hours for INT benchmarks on the 2.8 GHZ Pentium IV machine. On the 400 MHZ SPARC II machine, 9.92 hours for FP benchmarks and 12.31 hours for INT benchmarks. we compare the algorithms by normalized tuning time, shown in Figure 1(a) and Figure 1(b).

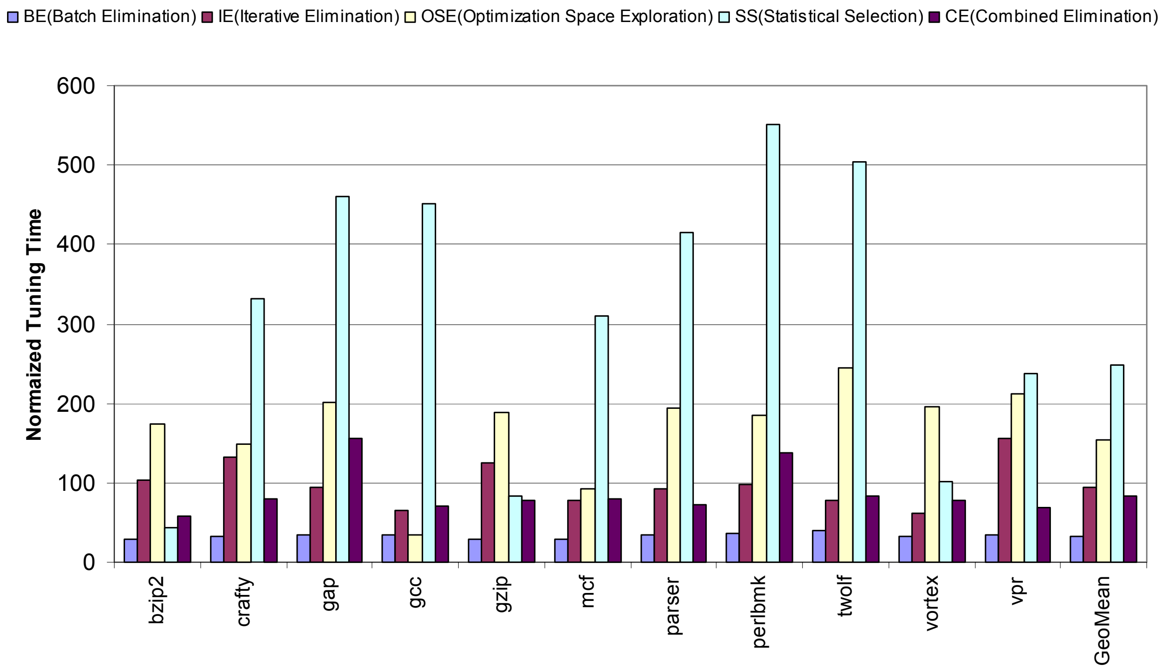
Although BE achieves the least program performance, its tuning speed is the fastest, which is consistent with its complexity of $O(n)$. BE can be viewed as a lower bound on the tuning time for a feed-back directed orchestration algorithm that does not have *a priori* knowledge of the optimizations. For such an algorithm, each optimization must be tried at least once to find its performance effect.

OSE is of higher complexity and thus slower than IE and CE. However, SS turns out to be the slowest method, even though its complexity is $O(n^2)$, less than OSE's $O(n^3)$. The reason for the long tuning time of SS is the higher number of iterations it takes to converge.



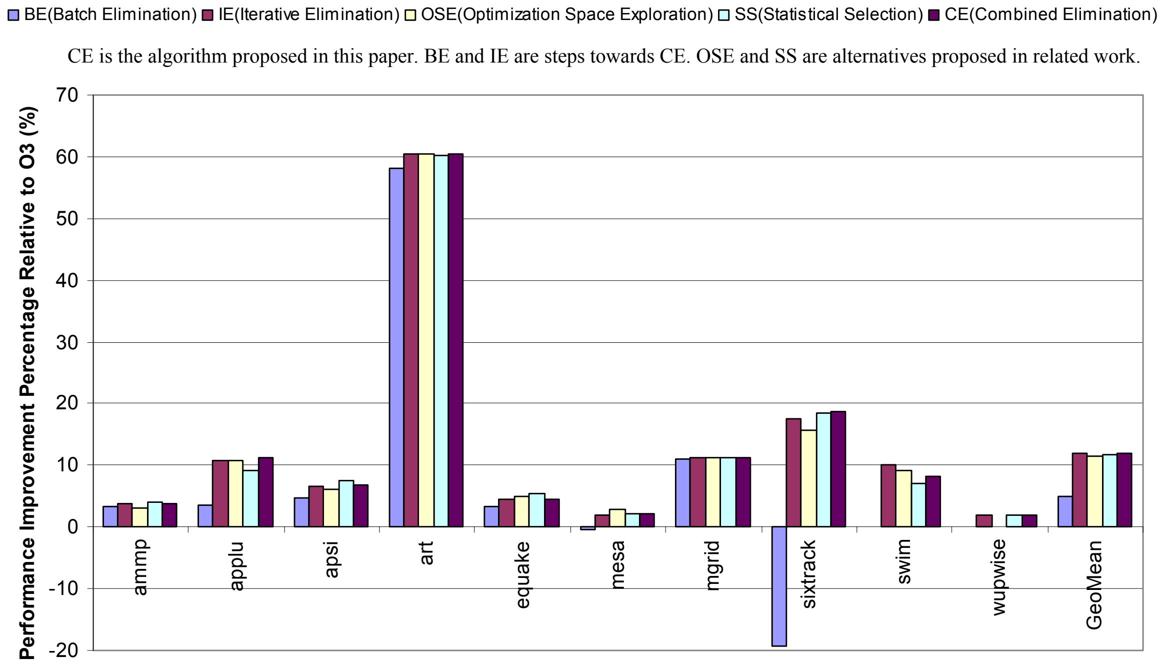
CE is the algorithm proposed in this paper. BE and IE are steps towards CE. OSE and SS are alternatives proposed in related work.

(a) Normalized tuning time of the orchestration algorithms for SPEC CPU2000 FP benchmarks on Pentium IV

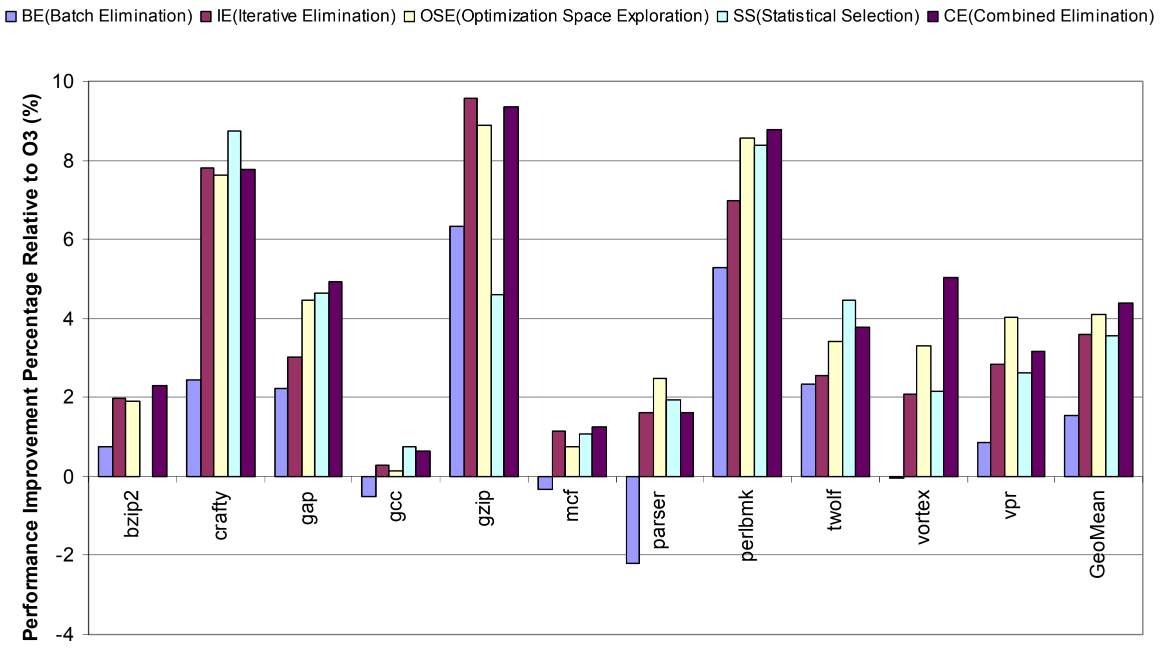


(b) Normalized tuning time of the orchestration algorithms for SPEC CPU2000 INT benchmarks on Pentium IV

Figure 1. Normalized tuning time of five optimization orchestration algorithms for SPEC CPU2000 benchmarks on the Pentium IV machine. Lower is better. CE has the shortest tuning time in all except a few cases. In all those cases, the extended tuning time leads to significantly higher performance.



(a) Program performance achieved by the orchestration algorithms relative to the baseline under the highest optimization level "O3" for the SPEC CPU2000 FP benchmarks on Pentium IV



(b) Program performance achieved by the orchestration algorithms relative to the baseline under the highest optimization level "O3" for the SPEC CPU2000 INT benchmarks on Pentium IV

Figure 2. Program performance of five optimization orchestration algorithms for SPEC CPU2000 benchmarks on Pentium IV. Higher is better. In all cases, CE performs the best or within 1% of the best.

Among the four algorithms (excluding BE), CE has the fastest average tuning speed. For *ammp, wupwise, bzip2, gap, gcc, perlbmk* and *vortex*, CE is not the fastest. However, the faster algorithms achieve their speed at significant expense of program performance.

3.3.2 Program Performance

In both Figure 2(a) and Figure 2(b), BE almost always achieves the least program performance among the five algorithms. As described in Section 2, BE ignores the interaction of the optimizations. Therefore, it does not achieve good performance when the interaction has a significant negative performance effect. In the cases of *sixtrack* and *parser*, BE even significantly degrades the performance.

In Figure 2(a), for *art*, all the algorithms improve performance by about 60% on Pentium. This is mainly due to eliminating the option of "strict-aliasing", which does alias analysis, removes false data dependences, and increases register pressure. This option results in lots of spill code for *art*, causing substantial performance degradation. However, on the SPARC machine, the orchestration algorithms do not have the above behavior for *art*. "Strict-aliasing" does not cause performance degradation, as the SPARC machine has more registers than the Pentium machine. In [13], we have analyzed reasons for negative performance effects by several optimizations, in detail.

The average performance improvement of all other orchestration algorithms, which consider the interactions of optimizations, are about twice as high as BE's. Moreover, Figure 2(a) shows that these four algorithms perform essentially the same for the FP benchmarks. On one hand, the regularity of FP programs contributes to this result. On the other hand, the optimizations in GCC limit performance tuning on FP benchmarks, because GCC options do not include advanced dependence-based transformations, such as loop tiling. We expect that such transformations would be amenable to our tuning method and yield tangible improvement. In Figure 2(b), performance similarity still holds in most of the INT benchmarks, with a few exceptions. For *gap*, *twolf* and *vortex*, IE does not achieve as good a performance as CE, though the performance *gap* is small. SS does not produce consistent performance; for *bzip2*, SS does not achieve any performance; for *bzip2*, *gzip* and *vortex*, SS's performance is significantly inferior to CE. CE and OSE always achieve good program performance improvement.

The fact that none of the algorithms constantly outperforms the others, reflects the exponential complexity of the optimization orchestration problem. All five algorithms use heuristics, which lead to sub-optimal results. Among these algorithms, CE achieves consistent performance. Although, for *crafty*, *parser*, *twolf*, and *vpr*, CE does not achieve the best performance, the gap is less than 1%.

The small performance differences between the measured algorithms indicate that all methods properly deal with the primary interactions between optimization techniques. However, there are differences in the ways the algorithms deal with secondary interactions. These properties are consistent with those of a general optimization problem, in which the main effects tend to be larger than two-factor interactions, which in turn tend to be larger than three-factor interactions, and so on [4].

In Figure 2, we measured program performance under the *train* dataset. It is important to evaluate how the algorithm performs under different input. To this end, we measured execution times of each benchmark using the *ref* dataset as input, for both the O3 version and the optimal version found by CE. (Still, the *train* dataset is the input for the tuning process.) On average, CE improves FP benchmarks by 11.7% (compared to 11.9% under *train*) relative to O3; INT benchmarks by 3.9% (4.4% under *train*). This shows that CE works well when the input is different from the tuning input. On the other hand, we do find a few benchmarks that do not achieve the same performance under the *ref* dataset as under *train*. The highest differences are, for *gzip* and *vortex*, 1.95% and 2.18%. If the training input of the orchestration algorithm differs significantly from actual workloads, our *offline* (profile-based) tuning approach may not reach the full tuning potential. In that case, an *online* approach [19] could tune the program using the actual input. This is a complementary approach that we are pursuing in ongoing work.

3.3.3 Overall Comparison of Algorithms

CE achieves both fast tuning speed and high program performance. It does so by combining the advantages of IE and BE: Like IE, it considers the interaction of optimizations, leading to high program performance; like BE, it keeps tuning time short when the interaction does not have a significant performance effect.

Similar observations hold on the SPARC II machine. Limited by space, Table 3 only lists the mean performance of each algorithm across the integer and floating point benchmarks, respectively.

Figure 3 provides an overall comparison of the algorithms. The X-axis is average program performance achieved by the algorithm; the Y-axis is average normalized tuning time. The averages are taken across all benchmarks and machines. (The figure under each benchmark and machine setting would be similar.) A good algorithm achieves high program performance and short tuning time, represented by the bottom-right corner of Figure 3. The figure shows that CE is the best algorithm. The runner-up is IE, which we developed as a step towards CE.

Table 3. Mean performance on SPARC II CE achieves both fast tuning speed and high program performance on SPARC II.

Benchmark	Algorithm	improvement over "O3"	Normalized Tuning Time
FP	BE	−4.1 %	30.8
FP	IE	4.1 %	105.4
FP	OSE	4.0 %	142.0
FP	SS	3.7 %	384.9
FP	CE	4.1 %	63.4
INT	BE	−0.8 %	36.2
INT	IE	3.6 %	98.7
INT	OSE	3.4 %	130.0
INT	SS	3.1 %	317.0
INT	CE	3.9 %	88.4

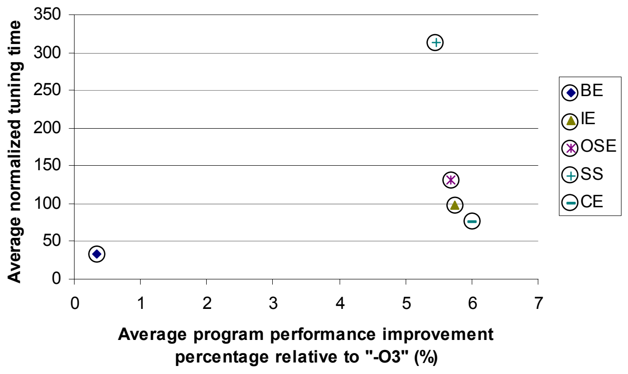


Figure 3. Overall comparison of the orchestration algorithms. *CE achieves both fast tuning speed and high program performance.*

4 Upper Bound Analysis

We have shown that CE achieves good performance improvement. This section attempts to answer the question of how much better than CE an algorithm could perform. To this end, we look for a performance upper bound, which we find by an exhaustive search (ES) through all optimization combinations. As it would be impossible to do exhaustive search with 38 optimizations, we pick a small set of six optimizations. This section will show that the performance improvement by CE is close to this upper bound.

The six optimizations that have the largest performance effects are picked to conduct upper bound analysis. (The performance effect of an optimization is the total negative relative performance improvement of this optimization on all the benchmarks.) On the SPARC II and Pentium IV machines, these six optimizations are picked separately. They are strict-aliasing, schedule-insns2, regmove, gcse, rerun-loop-opt and force-mem for Pentium IV, and rename-registers, reorder-blocks, sched-interblock, schedule-insns, gcse and if-conversion for SPARC II.

4.1 Results

In Figure 4, ES represents the performance upper bound. Comparing the first two columns in Figure 4(a) and Figure 4(b), we find that, under the 6 optimizations, CE performs close to ES. In about half of the benchmarks, they both find the same best version. Another important fact shown in Figure 4(c) and Figure 4(d) is that CE is more than 4 times as fast as ES, even for this small set of optimizations. For comparison, the figures also show the tuning speed of CE for 38 options. ES for 38 options would be millions of years!

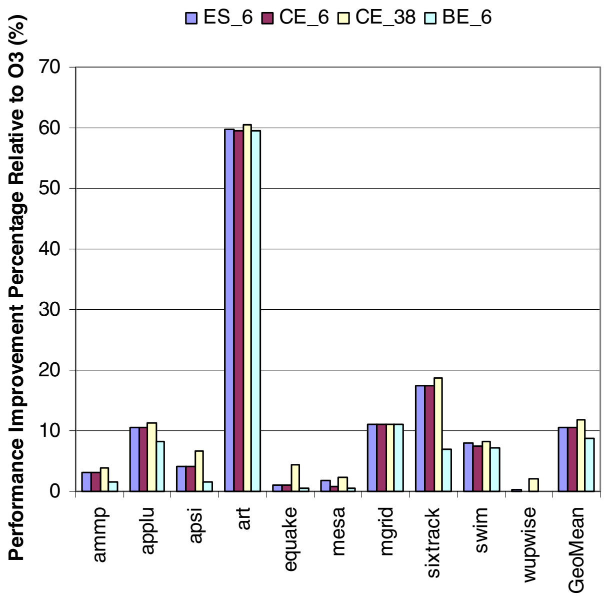
Figure 4 provides evidence that the heuristic-based search algorithms can achieve performance close to the upper bound. This confirms our analysis in Section 3.3.2. The heuristics find the primary and secondary performance effects, which are the individual performance of an optimization and the main interaction with other optimizations, respectively. These arguments hold for the SPARC II machine. Table 4 shows average program performance achieved under different machine and benchmark settings.

Table 4. Upper bound analysis under four different machine and benchmark settings

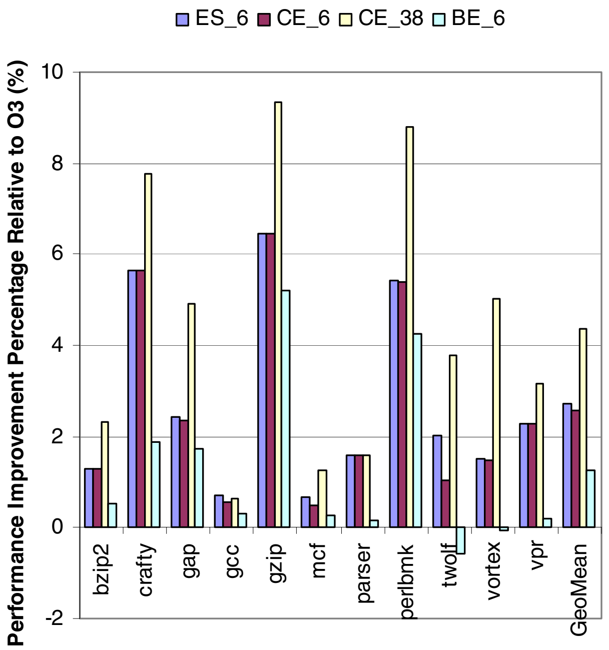
Machine	Benchmark	RIP by ES over "O3"	RIP by CE over "O3"
Pentium IV	FP	10.6 %	10.4 %
Pentium IV	INT	2.7 %	2.6 %
SPARC II	FP	2.9 %	2.7 %
SPARC II	INT	3.2 %	3.0 %

Comparing CE_6 with CE_38, the performance gap for FP benchmarks is negligible, but not for INT benchmarks. This result is consistent with the finding in [13] that INT programs are sensitive to a larger number of interactions between optimization techniques than FP programs. These results suggest that *a priori* knowledge of a small set of potentially interacting optimizations may help tuning numerical programs. Exhaustive search within this small set can be feasible. However, this is not the case for non-numerical applications.

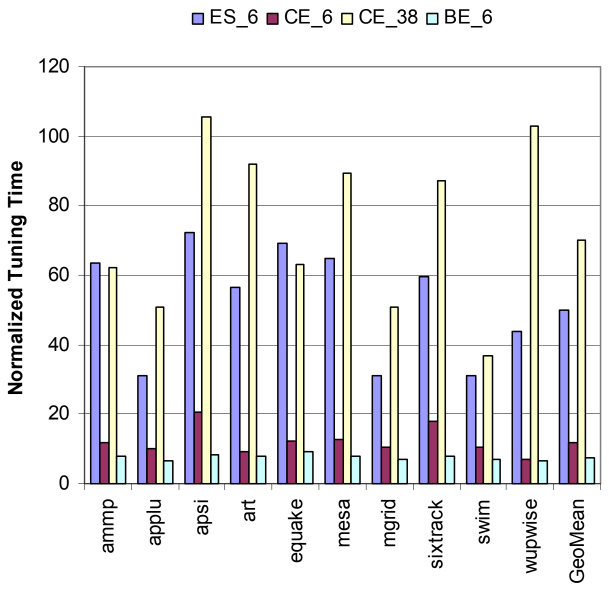
In order to verify that the interaction between these six optimizations has a significant performance effect, we apply BE as well. The result is shown as the last column, BE_6, in Figure 4. From this figure, the performance of BE_6 is much worse than CE_6, for example, in *ammp*, *apsi*, *sixtrack*, *crafty*, *parser*, and *vpr*.



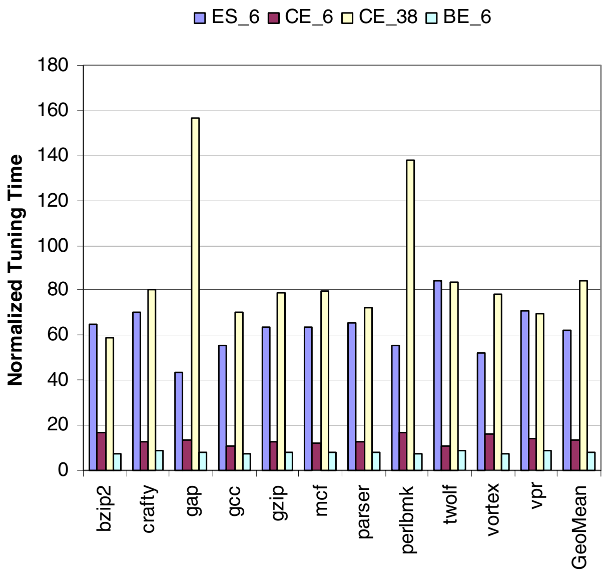
(a) Program performance of the SPEC CPU2000 FP benchmarks achieved by the orchestration algorithms on Pentium IV



(b) Program performance of the SPEC CPU2000 INT benchmarks achieved by the orchestration algorithms on Pentium IV



(c) Normalized tuning time of the orchestration algorithms for the SPEC CPU2000 FP benchmarks on Pentium IV



(d) Normalized tuning time of the orchestration algorithms for the SPEC CPU2000 INT benchmarks on Pentium IV

Figure 4. Upper bound analysis on Pentium IV. ES_6: exhaustive search with 6 optimizations; CE_6: combined elimination with 6 optimizations; CE_38: combined elimination with 38 optimizations; BE_6: batch elimination with 6 optimizations. CE_6 achieves nearly the same performance as ES_6, in all cases. CE_38 performs significantly better – exhaustive search with 38 optimizations would be infeasible. BE_6 is much worse than CE_6. CE_6 is about 4 times faster than ES_6.

5 Conclusion

This paper has presented a new compiler optimization orchestration algorithm, Combined Elimination (CE). CE allows programs to be optimized automatically using full, standard compilers, via an offline dynamic tuning process. By detecting both the main effects and the interactions of the involved compiler optimizations, CE is able to tune programs fast and achieves significant performance improvements. It does so for both floating point and integer benchmarks on Pentium and SPARC machines.

Compared to Optimization Space Exploration (OSE) [18] and Statistical Selection (SS) [15], CE takes the least tuning time while performing as well as the other approaches. On our Pentium IV machine, CE takes 2.96 hours on average, while OSE takes 4.51 hours and SS takes 11.96 hours. By orchestrating a small set of (6) optimizations, we have shown that the gap between performance improvement by CE and the upper bound is less than 0.2% on average. Thus, CE provides a fast and effective automatic performance tuning method.

Some work [18] has shown that different parts of a program have different optimal set of optimization options. In ongoing work, we are applying our CE algorithm to performance tuning at a finer granularity than the whole program. We are exploring the possibility of developing algorithms for online adaptive tuning [19] as well.

References

[1] *Forte C 6 /Sun WorkShop 6 Compilers C User's Guide*. http://docs.sun.com/app/docs/doc/806-3567.

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

[3] L. Almagor, K. D. Cooper, A. Grosul, T. J. Harvey, S. W. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Finding effective compilation sequences. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 231–239, New York, NY, USA, 2004. ACM Press.

[4] G. E. P. Box, W. G. Hunter, and J. S. Hunter. *Statistics for experimenters : an introduction to design, data analysis, and model building*. John Wiley and Sons, 1978.

[5] K. Chow and Y. Wu. Feedback-directed selection and characterization of compiler optimizations. In *Second Workshop on Feedback Directed Optimizations*, Israel, November 1999.

[6] K. D. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *The Journal of Supercomputing*, 23(1):7–22, 2002.

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

[8] M. Haneda, P. Knijnenburg, and H. Wijshoff. Generating new general compiler optimization settings. In *Proceedings of the 19th ACM International Conference on Supercomputing*, pages 161–168, June 2005.

[9] A. Hedayat, N. Sloane, and J. Stufken. *Orthogonal Arrays: Theory and Applications*. Springer, 1999.

[10] T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *IEEE PACT*, pages 237–248, 2000.

[11] T. Kisuki, P. M. W. Knijnenburg, M. F. P. O'Boyle, F. Bodin, and H. A. G. Wijshoff. A feasibility study in iterative compilation. In *International Symposium on High Performance Computing (ISHPC'99)*, pages 121–132, 1999.

[12] P. Kulkarni, S. Hines, J. Hiser, D. Whalley, J. Davidson, and D. Jones. Fast searches for effective optimization phase sequences. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 171–182, New York, NY, USA, 2004. ACM Press.

[13] Z. Pan and R. Eigenmann. Compiler optimization orchestration for peak performance. Technical Report TR-ECE-04-01, School of Electrical and Computer Engineering, Purdue University, 2004.

[14] Z. Pan and R. Eigenmann. Rating compiler optimizations for automatic performance tuning. In *SC2004: High Performance Computing, Networking and Storage Conference*, page (10 pages), November 2004.

[15] R. P. J. Pinkers, P. M. W. Knijnenburg, M. Haneda, and H. A. G. Wijshoff. Statistical selection of compiler options. In *The IEEE Computer Societys 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'04)*, pages 494–501, Volendam, The Netherlands, October 2004.

[16] N. J. A. Sloane. *A Library of Orthogonal Arrays*. http://www.research.att.com/ njas/oadir/.

[17] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. 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.

[18] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August. Compiler optimization-space exploration. In *Proceedings of the international symposium on Code generation and optimization*, pages 204–215, 2003.

[19] M. J. Voss and R. Eigenmann. 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.

[20] R. C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *SuperComputing 1998: High Performance Networking and Computing*, 1998.