PEAK—A Fast and Effective Performance Tuning System via Compiler Optimization Orchestration

ZHELONG PAN and RUDOLF EIGENMANN
Purdue University

Compile-time optimizations generally improve program performance. Nevertheless, degradations caused by individual compiler optimization techniques are to be expected. Feedback-directed *optimization orchestration* systems generate optimized code versions under a series of optimization combinations, evaluate their performance, and search for the best version. One challenge to such systems is to tune program performance quickly in an exponential search space. Another challenge is to achieve high program performance, considering that optimizations interact. Aiming at these two goals, this article presents an automated performance tuning system, *PEAK*, which searches for the best compiler optimization combinations for the important code sections in a program. The major contributions made in this work are as follows: (1) An algorithm called *Combined Elimination (CE)* is developed to explore the optimization space quickly and effectively; (2) Three fast and accurate rating methods are designed to evaluate the performance of an optimized code section based on a partial execution of the program; (3) An algorithm is developed to identify important code sections as candidates for performance tuning by trading off tuning speed and tuned program performance; and (4) A set of compiler tools are implemented to automate optimization orchestration. Orchestrating optimization options in SUN Forte compilers at the whole-program level, our CE algorithm improves performance by 10.8% over the SPEC CPU2000 FP baseline setting, compared to 5.6% improved by manual tuning. Orchestrating GCC O3 optimizations, CE improves performance by 12% over O3, the highest optimization level. Applying the rating methods, PEAK reduces tuning time from 2.19 hours to 5.85 minutes on average, while achieving equal or better program performance.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Code generation, compilers, optimization, run-time environments*

General Terms: Performance

Additional Key Words and Phrases: Performance tuning, optimization orchestration, dynamic compilation

This work was supported, in part, by the National Science Foundation under Grants No. EIA-0103582, CCF-0429535, and CNS-0650016.
Authors' address: School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN, 47907-1285; email: pan@vmware.com.

ACM Reference Format:
Pan, Z. and Eigenmann, R. 2008. PEAK—A fast and effective performance tuning system via compiler optimization orchestration. ACM Trans. Program. Lang. Syst. 30, 3, Article 17 (May 2008), 43 pages. DOI = 10.1145/1353445.1353451 http://doi.acm.org/10.1145/ 1353445.1353451

1. INTRODUCTION

Compiler optimizations generally yield significant performance improvements in many programs on modern architectures. Nevertheless, the potential for performance degradation in certain program patterns is known to compiler writers and many programmers. The state of the art is to let programmers deal with this problem through compiler options. For example, a programmer can switch off an optimization after finding that it causes performance degradation. The presence of these options reflects the inability of today's compilers to make the best optimization decision at compile time. In this article, we refer to this process of finding the best optimization combination for a target program as *optimization orchestration*. The large number of compiler optimizations, the subtle interactions between optimizations, the sophistication of computer architectures, and the complexity of the program itself make this optimization orchestration problem difficult to solve.

Our PEAK system automates the optimization orchestration process in a fast and effective way. It consists of a few run-time and compile-time subsystems with the goal of finding effective optimization settings for important code segments of an application. PEAK employs a profiling system to create a call graph and to identify these important code segments, which we call *tuning sections*. For each tuning section, PEAK uses a fast and effective search technique to determine the best set of optimization flags with which to compile. Because the search technique needs to fairly compare the effectiveness of one optimization vector against another, this article presents accurate rating methods as well.

In more detail, PEAK generates a series of *experimental versions*, compiled under different optimization combinations, for each tuning section. The performance of each experimental version is *rated* based on a partial execution of the program, that is, a few invocations of the tuning section during a *training run* of the program. An orchestration algorithm chooses a sequence of experimental optimization combinations, based on these ratings, until performance returns diminish. Finally, the tuned versions of all segments are combined for the production run of the program.

To achieve the two goals of high program performance and fast tuning speed, this article develops fast and effective orchestration algorithms for empirically searching the optimization space [Pan and Eigenmann 2006]. It also presents fast and accurate rating methods to speed up performance evaluation [Pan and Eigenmann 2004b]. Compiler tools are implemented to analyze and to instrument the target program automatically.

Figure 1 shows the steps of our envisioned performance tuning scenario. Steps (1) to (4) are taken before tuning to construct a tuning driver for a given program. In these steps, our compiler analyzes and instruments the source code, selecting candidates for performance tuning and applying accurate rating

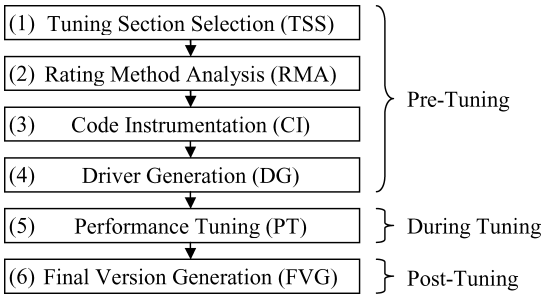


Fig. 1. Work flow of the PEAK system: Compile-time pre-tuning steps identify the important program sections to be tuned and suitable methods for comparing the measured performance. The actual tuning, through empirical search, happens during an offline execution, similar to a profile run. The individually optimized tuning sections are combined in a post-tuning step.

methods. During performance tuning at Step (5), the tuning driver continually runs the program under a training input until the best version is found for each tuning section. In this step, the system dynamically generates and loads optimized versions, and rates these versions. After tuning, in Step (6), each tuning section is compiled under its best optimization combination and linked to the main program to generate the final tuned version.

This article makes the following specific contributions:

(1) An optimization orchestration algorithm is developed to explore the optimization space fast and effectively. This algorithm achieves equal or better performance than related algorithms, but with less tuning time (57% of the closest alternative) [Pan and Eigenmann 2006].
(2) Three accurate and fast performance rating methods based on a partial execution of the program are designed to improve rating accuracy and to reduce tuning time. These rating methods reduce tuning time from several hours to several minutes, while achieving equal or higher program performance.
(3) An algorithm to select important code segments as candidates for performance tuning is presented. The selected tuning sections cover most of (typically more than 90% of) the total execution time. Each of them is invoked many times (typically several hundreds) in one run of the program. A high execution time coverage leads to high program performance; a large number of tuning-section invocations leads to fast tuning, because it means that many optimized versions can be experimented in one run of the program.
(4) An automatic performance tuning system via optimization orchestration is implemented. The PEAK compiler analyzes and instruments the source program before the tuning phase. The PEAK runtime system explores the optimization space and evaluates the performance of optimized versions automatically.
(5) Optimization orchestration performance is measured and analyzed comprehensively for GCC and the SUN Forte compiler, with a focus on GCC. The experiments are done for SPEC CPU2000 benchmarks on a Pentium 4 machine and a SPARC II machine.

The remainder of this article is organized as follows. Section 2 presents a fast and effective orchestration algorithm named *Combined Elimination (CE)*. Section 3 shows fast and accurate rating methods to evaluate the performance of optimized tuning-section versions based on a partial execution of the program. Section 4 develops an algorithm for selecting the important code sections in a program as tuning sections. Section 5 shows the design of our PEAK system, which tunes program performance at tuning-section granularity. Section 6 evaluates PEAK in terms of tuning time and tuned program performance. Section 7 discusses related work. Section 8 concludes this article.

2. A FAST AND EFFECTIVE OPTIMIZATION ORCHESTRATION ALGORITHM

Today's compilers have evolved to the point where they present to programmers a large number of optimization 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 [Chow and Wu 1999; Kisuki et al. 1999; Pan and Eigenmann 2004a; Pinkers et al. 2004; Triantafyllis et al. 2003]. How do we search for the best optimization combination for a given program in order to achieve the best performance? We call this process *optimization orchestration*, and this section aims at developing a fast and effective algorithm to do so. In Section 2.1, we develop a feedback-directed algorithm, Combined Elimination (CE), to orchestrate on-off options. All the GCC O3 optimization options are of this on-off type. In Section 2.2, we extend the algorithm to handle multivalued options. One example of a multivalued option is the "-unroll" option in SUN Forte compilers [Sun 2000], which takes an argument indicating the degree of loop unrolling.

2.1 The CE Algorithm to Tune On-Off Options

Focusing on on-off options, similar to several of the other papers [Chow and Wu 1999; Pan and Eigenmann 2004a, 2004b; Pinkers et al. 2004], we define the goal of optimization orchestration as follows:

Given a set of compiler optimization options $\{F_1, F_2, \ldots, F_n\}$*, find the combination that minimizes the program execution time. Each option* F_i *has two possible values: on and off. (Here, n is the number of optimizations.)*

We first develop two simple feedback-directed algorithms: (1) *Batch Elimination* (BE) identifies harmful optimizations and removes them in a batch. (2) *Iterative Elimination* (IE) successively removes harmful optimizations. Based on the above two algorithms, we design our final algorithm, *Combined Elimination* (CE). In our previous paper [Pan and Eigenmann 2006], we applied these algorithms to tune program performance at the whole-program level. In this article, we will apply CE at the tuning-section level, using the PEAK system.

2.1.1 *Batch Elimination (BE).* The idea of *Batch Elimination (BE)* is to identify the optimizations with negative effects and turn them off all at once. The negative effect of one optimization, F_i, can be represented by its *Relative*

(1) Compile the code under the baseline $B = \{F_1 = on, F_2 = on, \ldots, F_n = on\}$. Execute the generated version to get the baseline rating T_B.
(2) For each optimization F_i, switch it off from B and compile the code, execute the generated version to get $T(F_i = off)$, and compute the $RIP_B(F_i = off)$ according to Equation (2).
(3) Disable all optimizations that cause performance degradation, i.e., whose *RIP*s are negative, to generate the final, tuned version for the code.

Fig. 2. Pseudo-code of batch elimination.

Improvement Percentage (RIP), $RIP(F_i)$, which is the relative performance difference of the two versions with and without F_i, $T(F_i = on)$ and $T(F_i = off)$. $RIP(F_i)$ can be computed according to Eq. (1).

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

If the orchestration algorithm is applied at the whole-program level, $T(F_i = off)$ is the execution time of the whole program with F_i turned off; if the algorithm is applied at the tuning-section level, $T(F_i = off)$ is the *rating* generated based on the execution times of a few invocations to the tuning section compiled with F_i turned off. (We will discuss the rating methods in detail in Section 3.)

BE switches on all optimizations as the baseline. The performance improvement by switching off F_i from the baseline B relative to the baseline performance can be computed by Eq. (2).

$$RIP_B(F_i = off) = \frac{T(F_i = off) - T_B}{T_B} \times 100\%. \quad (2)$$

If $RIP_B(F_i = off) < 0$, the optimization F_i has a negative effect. BE eliminates the optimizations with negative *RIP*s in a batch to generate the final, tuned version. This algorithm has a complexity of $O(n)$, where n is the number of optimizations. Figure 2 lists the pseudo-code.

2.1.2 *Iterative Elimination (IE).* *IE* takes the interaction of optimizations into consideration, using a greedy method. 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 *RIP*s of the optimizations in accordance with Eq. (2), 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 $O(n^2)$. Figure 3 shows the pseudo-code.

2.1.3 *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 each iteration, CE tries to eliminate these optimizations one by one in a greedy fashion.

We have shown in Pan and Eigenmann [2006] that IE achieves better program performance than BE, since IE considers the interaction of optimizations

(1) Let B be the baseline optimization combination. Let S be the set of optimizations forming the search space. Initialize these two sets: $S = \{F_1, F_2, \ldots, F_n\}$ and $B = \{F_1 = on, F_2 = on, \ldots, F_n = on\}$.
(2) Compile and execute the code under the baseline setting to get the baseline rating T_B.
(3) For each optimization $F_i \in S$, switch F_i off from B and compile the code, execute the generated version to get $T(F_i = off)$, and compute the *RIP* of F_i relative to the baseline B, $RIP_B(F_i = off)$, according to Equation (2).
(4) Find the optimization F_x that causes the most performance degradation, i.e., whose *RIP* has the most negative value. Remove F_x from S, and set F_x to off in B.
(5) Repeat Steps (2), (3) and (4) until all options in S have non-negative *RIP*s. B represents the final optimization combination.

Fig. 3. Pseudo-code of iterative elimination.

(1) Let B be the baseline optimization combination. Let S be the set of optimizations forming the search space. Initialize these two sets: $S = \{F_1, F_2, \ldots, F_n\}$ and $B = \{F_1 = on, F_2 = on, \ldots, F_n = on\}$.
(2) Compile and execute the code under the baseline setting to get the baseline rating T_B.
(3) Measure the $RIP_B(F_i = off)$ of each optimization option F_i in S relative to the baseline B.
(4) Let $X = \{X_1, X_2, \ldots, X_l\}$ be the list of optimization options with negative *RIP*s. These optimizations cause performance degradation. X is sorted in an increasing order, i.e., the first element, X_1, causes the most degradation. Remove X_1 from S and set X_1 to off in B. (B is changed in this step.) For i from 2 to l,
   (a) Measure the *RIP* of X_i relative to the baseline B.
   (b) If the *RIP* of X_i is negative, i.e, X_i causes performance degradation, remove X_i from S and set X_i to off in B.
(5) Repeat Steps (2), (3), and (4) until all options in S have non-negative *RIP*s. B represents the final optimization combination.

Fig. 4. Pseudo-code of combined elimination.

using a greedy method. But, when the interactions have only small effects, BE may perform close to IE and more quickly provide the solution. 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 $O(n^2)$. The pseudo-code is listed in Figure 4.

2.2 The GCE Algorithm to Tune Multi-Valued Options

Generally, to tune multivalued options, the optimization orchestration problem can be described as follows:

Given a set of optimization options $\{F_1, F_2, \ldots, F_n\}$, where each F_i has K_i possible values $\{V_{i,j}, j = 1..K_i\}$ $(i = 1..n)$, find the combination that minimizes the program execution time. (Moreover, one F_i can actually contain multiple optimizations that have a high possibility of interaction; the possible values of this F_i are all possible combinations of the values of the two optimizations.)

We name our algorithm to handle multivalued options *General Combined Elimination (GCE)*. GCE has an iterative structure similar to CE with a few extensions. The initial baseline of GCE is the default optimization setting used in the compiler. In each iteration of GCE, all nondefault values of the remaining options in the search space S are evaluated; for each of these options, GCE

(1) Let B be the baseline option setting. Let S be the set of optimizations forming the search space. Initialize $B = \{ F_1 = f_1, F_2 = f_2, \ldots, F_n = f_n \mid f_i$ is the default value of option $F_i \}$ and $S = \{F_1, F_2, \ldots, F_n\}$.
(2) Measure the *RIP*s of all the non-default values of the options in S relative to the baseline B, that is, $RIP_B(F_i = V_{i,j})$ s.t. $F_i \epsilon S$ and $V_{i,j} \neq f_i$. The definition of *RIP* is the same as in Eq. (2).
(3) Let $X = \{X_1 = x_1, X_2 = x_2, \ldots, X_l = x_l\}$ be the list of options with negative *RIP*s and x_i have the most negative *RIP* among the possible values of option X_i. This means that $X_1 = x_1$, $X_2 = x_2$, ..., and $X_l = x_l$ improve the baseline performance, and value x_i has the best performance among all the possible values of Option X_i $(i = 1..l)$. X is sorted in an increasing order, that is, the first element in X, X_1, has the best performance among $X_i (i = 1..l)$. Remove X_1 from S and set X_1 in B to be x_1. (B is changed in this step.) For i from 2 to l,
    (a) Measure $RIP_B(X_i = x_i)$.
    (b) If it is negative, i.e., setting $X_i = x_i$ achieves better performance than the baseline, remove X_i from S and set X_i in B to be x_i.
(4) Repeat Step (2) and Step (3) until all options in S have non-negative *RIP*s. B represents the final option setting.

Fig. 5. Pseudo-code of general combined elimination.

records the value causing the most negative *RIP*, which is computed according to Eq. (2); GCE tries to apply these recorded values of the options with these negative *RIP*s one by one in a greedy fashion just like CE. When none of the remaining options in S has a value improving the performance, GCE gives the final optimization setting for the program. GCE has a complexity of $O(\Sigma_{i=1}^{n} K_i \times n)$. In most cases, $O(\Sigma_{i=1}^{n} K_i) = O(n)$, so roughly the complexity is still $O(n^2)$. (The techniques developed in Kisuki et al. [2000] could also be included in GCE to handle options with a large number of possible values, for example, blocking factors.) The pseudo-code of GCE is shown in Figure 5.

3. FAST AND ACCURATE RATING METHODS

In this article, we tune performance for important code sections of a program, called *tuning sections (TS)*. A tuning section is essentially a procedure including all its callees. We define one *invocation* of a TS as the total execution of this TS during one call to that procedure. To tune a program at this tuning-section level, we apply the CE algorithm to search for an effective optimization combination for each TS independently. Noticing that each TS is invoked many (usually hundreds or thousands of) times, we develop *rating* methods to evaluate the performance of one optimized version based on a small number of invocations of the TS, called a *window*. In this way, a partial execution of the program can be used to rate the performance of an optimized version. It leads to a large speedup of the tuning process compared to the whole-program tuning, which uses the total execution time of the whole program to evaluate the performance.

Rating based on a number of invocations of the tuning sections leads to fast tuning speed, but, the workload may change from one invocation to the next. Directly averaging the execution times of a number of invocations is not always accurate. This section presents the rating methods that compare in a fair way the execution times of the optimized versions under different invocations. The key ideas of our rating methods are as follows: *Context-Based Rating* (CBR) identifies and compares invocations of a tuning section that have the same workload, in the course of the program run. *Model-Based Rating* (MBR)

formulates relationship between different workloads, which it factors into the comparison. *Re-execution-Based Rating* (RBR) directly re-executes a tuning section under the same input for fair comparison. (In our previous paper [Pan and Eigenmann 2004b], we have presented automated compiler techniques to analyze the source code for choosing the most appropriate rating method for each tuning section.)

The rating methods are applied in an offline performance tuning scenario as follows. Before tuning, the program is partitioned by our PEAK compiler into a number of tuning sections. The tuning system runs the program one or several times under a training input, while dynamically generating and swapping in/out new optimized versions for each TS. The performance of these versions is compared using the proposed rating methods. The winning version will be used in the final tuned program.

3.1 Context Based Rating (CBR)

Context-based rating identifies the invocations to a tuning section that have the same workload in the course of program execution. The PEAK compiler finds the set of *context variables*, which are the program variables that influence the execution time of the tuning section. For example, the context variables include the variables that determine the conditions of control regions, such as if or loop constructs. (These variables can be function parameters, global variables, or static variables.) Thus, the context variables determine the workload of a tuning section. We define the *context* of one TS invocation as the set of values of all context variables. Therefore, each context represents one unique workload.

CBR rates one optimized version under a certain context by using the average execution time of several invocations. (Typically, this number is tens of times.) The best versions for different contexts may be different, in which case CBR could report the context-specific winners. PEAK makes use of only the best version under the most important context, which covers most (e.g., more than 80%) of execution time spent in the TS. This major context is determined by one profile run of the program. (If a tuning section has no major context or the number of invocations of the major context is too small, the MBR method described next is preferred.)

In summary, the rating of a version v, $R(v)$, is computed according to Eq. (3), where x is the most time-consuming context of version v, $T(i, x)$ is the execution time of the ith invocation under context x, and w is the window size, the number of invocations used to rate one version. $Var(v)$ records the variance of the measurement.

$$R(v) = \sum_{i=1..w} T(i, x)/w \tag{3}$$

$$Var(v) = \sum_{i=1..w} (T(i, x) - R(v))^2/w. \tag{4}$$

We presented in Pan and Eigenmann [2004b] the compiler analysis to find the context variables so as to determine the applicability of CBR. The algorithm traverses each control statement and recursively finds the related variables;

that is, it finds all of the input variables that may influence the values used in control statements. All these variables are considered to be context variables. If there exist one or more nonscalar context variables, there is usually no major context. So, in this case, CBR is not applicable. Similarly, if the context variable is floating-point, CBR is not applicable. To reduce the context match overhead during tuning, we eliminate the *runtime constants* from the context variable list. The runtime constant variables are found in the same profile run that determines the major context.

3.2 Model-Based Rating (MBR)

Model-based rating formulates mathematical relationships between different contexts of a tuning section and adjusts the measured execution time accordingly. In this way, different contexts become comparable.

The execution time of a tuning section consists of the execution time spent in all of its basic blocks:

$$T_{TS} = \sum_{b=1..m} (T_b \times C_b). \tag{5}$$

T_{TS} is the execution time in one invocation to the whole tuning section; T_b is the execution time in one entry to the basic block b; and C_b is the number of entries to the basic block b in the TS invocation; m is the number of basic blocks.

If the numbers of entries to two basic blocks, C_{b1} and C_{b2}, are linearly dependent on each other in every TS invocation through the whole run of the program, (that is, $C_{b1} = \alpha * C_{b2} + \beta$, where α and β are constants), our compiler merges the items corresponding to these basic blocks into one *component*. Hence, MBR uses the following execution time estimation model:

$$T_{TS} = \sum_{i=1..n} (T_i \times C_i). \tag{6}$$

T_{TS} consists of several components, each of which has a *component count* C_i and a *component time* T_i.

During tuning, PEAK collects execution times of a number of invocations to an optimized version. It gathers a TS-execution-time vector, Y, and a component-count matrix, C, in which $Y(j)$ is the T_{TS} in the jth invocation and $C(i, j)$ is the ith component count C_i in the jth invocation. Solving the following linear regression problem yields the component-time vector T

$$Y = T \times C \tag{7}$$

Here, $T = (T_1, T_2, \ldots, T_n)$ represents the component-time vector of one particular version. The version with smaller T_i's performs better. Hence, MBR may compare different versions using the rating, $R(v)$, computed based on their T vectors according to the following equation.

$$R(v) = \sum_{i=1..n} (T_i \times C_{avgi}) \tag{8}$$

$$Var(v) = \sum_{j=1..w} \left(Y_j - \sum_{i=1..n} (T_i \times C_{i,j}) \right)^2 / w. \tag{9}$$

RBR(TuningSection TS) :
1. Swap Version 1 and Version 2
2. Save *Modified_Input*(TS)
3. Run the preconditional version
4. Restore *Modified_Input*(TS)
5. Time Version 1
6. Restore *Modified_Input*(TS)
7. Time Version 2
8. Return the two execution times

Fig. 6. Re-execution-based rating method.

C_{avgi} is the average count of component i during one whole run of the program. These data are obtained from the profile run. And, w is the number of invocations to compute the rating. The variance of the rating, $Var(v)$ is the residual error of this linear regression.

Adding a counter into the code may have side effects on the compiler optimizations. To avoid this, we can require that the value of the counter be determined before the entry to the tuning section.

If there are many (for example, ten) components in the execution time model, a large number of invocations need to be experimented in order to perform an accurate linear regression. MBR would lead to a long tuning time in this case and so is not applied. Instead, RBR will be applied, which is described next.

3.3 Re-execution Based Rating (RBR)

Figure 6 shows the basic idea of RBR: the input data is saved before each invocation to the TS (Step (2)), Version 1 is timed (Step (5)), the input is restored (Step (6)), and Version 2 is executed (Step (7)). These two execution times can be compared directly to decide which version is better, since both versions are executed with the same input. Suppose that the execution times of these two versions are T_{v1} and T_{v2}. Then, the performance of Version 2 relative to Version 1 is $R_{v2/v1}$:

$$R_{v2/v1} = T_{v1}/T_{v2}. \tag{10}$$

If $R_{v2/v1}$ is larger than 1, Version 2 performs better than Version 1. Otherwise, Version 2 performs worse. For multiple versions, we compare their performance relative to the same *base version*. For example, if $R_{v2/v1}$ is less than $R_{v3/v1}$, Version 3 performs better than Version 2. In our tuning system, we use the average of $R_{vx/vb}$'s across a number of TS invocations as the rating of Version vx relative to the base version vb. The rating of v is computed based on Eq. (11), where w is the number of invocations. Similar to CBR and MBR, we compute the rating variance $Var(v)$.

$$R(v) = \sum_{i=1..w} R_{v/vb}(i)/w \tag{11}$$

$$Var(v) = \sum_{i=1..w} (R_{v/vb}(i) - R(v))^2/w. \tag{12}$$

To improve the rating accuracy, RBR (1) swaps Version 1 and Version 2 at each invocation, so that their order does not bias the result (Step (1)), and

Step (2) inserts a *preconditional version* before Version 1 to bring the used data into cache (Step (3)). In addition, to reduce the roll-back overhead, RBR saves and restores only the input variables that are modified in the invocation, the set of $Modified\_Input(TS)$, which is the intersection of the input set and def set of TS

$$Modified\_Input(TS) = Input(TS) \cap Def(TS). \tag{13}$$

Our compiler finds the input variables using a combination of compile-time and run-time techniques. If it fails to do so during static analysis, for example, due to complicated memory aliases, our compiler instruments the source code to get the input during tuning. So, generally, RBR is applicable to all tuning sections, except the ones that call library functions with side effects, such as `malloc`, `free`, and I/O operations. Right now, we do not tune these tuning sections. As future work, these library functions with side effects could be rewritten, so that their execution can be rolled back. In our experiments, RBR is applicable to all SPEC CPU2000 FP benchmarks and half of the INT benchmarks.

3.4 The Use of Rating Methods in PEAK

We have presented three rating methods: CBR, MBR and RBR. Context-based rating (CBR) has the least overhead but is not applicable to code without a major context. Model-based rating (MBR) works for code without a major context, but is not applicable to irregular programs. Re-execution-based rating (RBR) can be applied to almost all programs, however, the overhead is the highest among the three. Generally, the applicability of these three rating approaches increases in the order of CBR, MBR and RBR; so does the overhead.

Before tuning, our PEAK compiler divides the target program into several tuning sections. The original source program is analyzed according to the techniques presented in the previous sections. The details about the compiler analysis can be found in Pan and Eigenmann [2004b]. After one profile run, the PEAK compiler finds the major context, if it exists, for CBR, and the execution time model for MBR. From the static analysis and profile information, the PEAK compiler decides which applicable rating method should be used for each tuning section, in the priority order of CBR, MBR and RBR. Again, CBR is applicable if there is a major context whose execution time is more than 80% of total execution time spent in the tuning section. MBR is applicable if there are only a few (less than 10) components in its execution time model. Then, the PEAK compiler inserts three kinds of instrumentation code into the source program to construct a tuning driver: (1) code to activate performance tuning; (2) code to measure the execution times and to trigger the rating methods; (3) code to facilitate the rating methods, for example, context match code for CBR and the preconditional version for RBR.

During tuning, the tuning driver generates the *rating*, $R(v)$, and the *rating variance*, $Var(v)$, across a number of TS invocations, which is called a *window*. The tuning driver compares $R(v)$ of different versions to know which version is the best one. In summary, $R(v)$ and $Var(v)$ are computed as follows.

—CBR: Suppose that $T(i, x)$ is the execution time of the ith invocation under context x. $R(v)$ and $Var(v)$ under context x are the mean and the variance

of $T(i, x)$, $i = 1 \cdots w$, where w is the window size. They are computed in accordance with Eqs. (3) and (4).

—MBR: $R(v)$ is execution time estimated from the execution time model, and $Var(v)$ is the residual error of the linear regression. They are computed in accordance with Eqs. (8) and (9).

—RBR: Suppose that $R_{v/vb}(i)$ is the relative performance of version v over base version vb at the ith invocation. $R(v)$ and $Var(v)$ are the mean and the variance of $R_{v/vb}(i)$, $i = 1 \ldots w$, in accordance with Eqs. (11) and (12).

To improve rating accuracy, the tuning system applies two optimizations:

(1) The tuning system identifies and eliminates measurement outliers, which are far away from the average. Such data may result from system perturbations, such as interrupts.

(2) The tuning system uses a dynamic approach to determining the window size. It continually executes and rates a version until the rating variance $Var(v)$ falls below a threshold. This optimization is applied based on the observation that $Var(v)$ decreases with increasing size of the window.

4. TUNING SECTION SELECTION

This section deals with the problem of how to select the important code sections in a program as the tuning sections, in order to achieve fast tuning speed and high tuned program performance. To do so, tuning sections need to meet the following requirements.

(1) A tuning section should be invoked a large number of times, for example, more than 100 times, in one run of the program. A large number of invocations usually leads to fast tuning. For tuning section TS_i, let the average number of invocations used to rate one optimized version be $N_1(TS_i)$, the total number of invocations to the tuning section be $N_t(TS_i)$, the number of optimized versions rated in one run of the program be $N_v(TS_i)$

$$N_v(TS_i) = N_t(TS_i)/N_1(TS_i). \tag{14}$$

So, when the number of invocations $N_t(TS_i)$ is large, the PEAK system may rate a large number of, $N_v(TS_i)$, versions in each run of the program, which means a fast tuning.[1] If there are multiple tuning sections, the tuning time is bound by the slowest one. Denote the smallest number of invocations to the tuning sections as N_{min}.

$$N_{min} = min_i(N_t(TS_i)). \tag{15}$$

So, the first requirement for good section selections is to have a large N_{min}.

(2) Tuning sections should *cover* the largest possible part of the program. The *coverage* of the tuning sections is computed based on the execution times. Denote the time spent in tuning section TS_i as $T_t(TS_i)$, which includes the time

[1]Although $N_1(TS_i)$ may not be the same for different tuning sections because it is determined dynamically as described in the previous section, generally, a large $N_t(TS_i)$ still means a large $N_v(TS_i)$.

spent in all the functions/subroutines invoked within this tuning section, and the total execution time of the program as T_{total}

$$Coverage = \frac{\sum T_t(TS_i)}{T_{total}} \times 100\%. \tag{16}$$

A large *coverage* means that a large percentage of the program is tuned via optimization orchestration. So, the system can achieve good program performance.

(3) A tuning section should be big enough, so that the average execution time spent in one invocation is big enough, for example, greater than 100 μ sec. The average execution time $T_{avg}(TS_i)$ is computed based on the number of invocations, $N_t(TS_i)$, and the execution time spent in TS_i, $T_t(TS_i)$

$$T_{avg}(TS_i) = T_t(TS_i)/N_t(TS_i). \tag{17}$$

We do not choose tiny tuning sections, because they usually result in a low timing accuracy, even though we use a high-resolution timer. The low timing accuracy results in low rating accuracy and a large number of invocations per version, $N_1(TS_i)$.

This section presents a tuning section selection algorithm, which meets all these three requirements. Section 4.1 shows the call graph annotated with execution time profiles used for tuning section selection. Based on the call graph, the problem of tuning section selection is formally defined in Section 4.2. Section 4.3 presents our tuning section selection algorithm. In this algorithm, nodes for recursive functions are merged to construct a simplified call graph, which is a directed acyclic graph. An algorithm is designed to select the tuning sections by maximizing the program coverage under a given constraint for N_{min}, working on the simplified call graph. The final algorithm iteratively calls the previous algorithm to trade off the coverage and N_{min}.

4.1 Profile Data for Tuning Section Selection

From the previous section, a tuning section is selected based on its number of invocations and its execution time. These data are collected from a profile pass. In our implementation, we use the call graph profile generated by *gprof*. This call graph profile shows how much time is spent in each function and its children, how many times each function is called and how many times the function calls its children, during the profile run.

Our tuning section selection algorithm reads the output of *gprof* [Graham et al. 1982] and generates a call graph $G = (V, E)$.[2] This call graph is a directed graph. It has one source (root) node, δ, whose in-degree is 0, and a set, Γ, of sink (leaf) nodes whose out-degrees are 0. Each node $v \in V$ identifies a function.

[2]Here, call graph G contains the dynamic calls during the profile run, not the static calls in the program code. So, if a function call is not executed during the profile run, G does not include this call. However, a *static call graph* generated by a compiler should include this call. For the purpose of tuning section selection, the dynamic call graph is good enough. After tuning sections are selected, our compiler tools analyze and transform the program. During this process, a static call graph is used. Our tuning selection algorithm can be applied to the static call graph as well, if profile information is assigned to its nodes and edges.

δ identifies the function *main*. The nodes in Γ identify the functions that do not call any other function. Each edge $e \in E$ identifies a function call. The associated profile information is as follows.

$$v = \{fn\} \tag{18}$$

$$e = \{s, t, n, tm\} \tag{19}$$

$fn(v)$ is the function name of the node. $s(e)$ identifies the caller node; $t(e)$ identifies the callee node; $n(e)$ is the number of invocations to $t(e)$ made by $s(e)$; $tm(e)$ is the time spent in $t(e)$ and its callees, when $t(e)$ is called from $s(e)$.

Ideally, we could truncate the program at any place, for example, in the beginning of a basic block, to create a tuning section. In practice, we choose tuning sections at the procedure level, since we use the call graph profile generated by *gprof*. If we had accurate basic block profiles, we could select the tuning sections at the basic-block level. The tuning section selection algorithm would be similar to the one presented in the next sections. The difference would be that a bigger graph should be used with basic blocks as the nodes instead of functions/subroutines.

4.2 A Formal Description of the Tuning Section Selection Problem

Tuning section selection aims at partitioning the program into several components. Basically, a tuning section starts with an entry function, including all the subroutines called directly or indirectly by the entry function. If one subroutine is called by two different tuning sections, it is replicated into these two tuning sections. (Such replication does not lead to code explosion, because the number of tuning sections is fixed and usually small, around three.) In the representation of the call graph in Section 4.1, tuning section selection tries to find a set of single entry-node regions (subgraphs) in G; each region is a tuning section. The entry function of a region identifies that region. One region can overlap with other regions, although it would be better if no regions overlap.

The problem of tuning section selection can be described, in a formal way, as an optimal edge cut problem. Given call graph $G = (V, E)$, find an edge cut (Θ, Ω) so as to maximize the invocation numbers and the coverage of the tuning sections. Here, Θ and Ω are a partition of the node set V, such that Θ contains the source node δ, and Ω contains the set of sink nodes in Γ. This edge cut (Θ, Ω) is a set of edges, each of which leaves Θ and enters Ω. This edge cut determines the set of tuning sections in two steps. (1) Find all the edges in this cut whose average execution times are greater than T_{lb}, the lower bound on the average execution time, that is, for each edge $e \in (\Theta, \Omega)$, put e in set S, if $tm(e)/n(e) \geq T_{lb}$. (2) The edges in set S point to the selected tuning sections, that is, make the entry-node set $T = \{v | v = t(e_i), e_i \in S\}$. Each node v in set T identifies a tuning section. (Tuning sections are the subgraphs led by the entry function v.) Figure 7 gives an example.

The tuning section selection algorithm maximizes the invocation numbers and the coverage of the tuning sections, which are computed according to aforementioned S and T as follows:

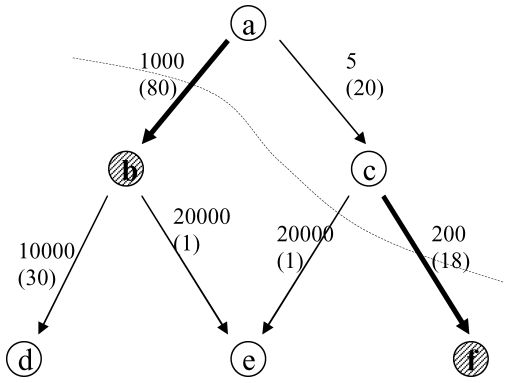


Fig. 7. An example of tuning section selection. The graph is a call graph with node a as the main function. The weights on an edge are the number of invocations and the execution time in the parentheses. The optimal edge cut is $(\Theta = \{a, c\}, \Omega = \{b, d, e, f\})$, shown by the dashed curve. Edges (a, b) and (c, f) are chosen as the S set. Edge (c, e) in the cut (Θ, Ω) is not included in S, because its average execution time is $1/20000$ less than $T_{lb} = 1e^{-4}$. There are two tuning sections led by node b and node f, $T = \{b, f\}$. The number of invocations to b and f are 1000 and 200 respectively, so, $N_{min} = 200$. The coverage of this optimal tuning section selection is $(80 + 18)/100 = 0.98$, where the total execution time, T_{total}, is 100.

(1) The number of invocations to the tuning section v is denoted as $N_t(v)$.

$$N_t(v) = \sum_{e \in S, t(e)=v} n(e). \tag{20}$$

One goal of the tuning section selection algorithm is to maximize the smallest $N_t(v)$, denoted as N_{min}.

$$N_{min} = min_{v \in T}(N_t(v)) \tag{21}$$

(2) The other goal of the tuning section selection algorithm is to maximize the execution coverage.

$$Coverage = \sum_{e \in S} tm(e)/T_{total} \tag{22}$$

T_{total} is the total execution time of the program.

The tuning section selection problem does not always have a reasonable solution. For example, suppose that a program has only one procedure, *main()*, which contains a loop consuming most of the execution time. If *main* is chosen as the tuning section, N_{min} is 1 and *coverage* is 100%. Otherwise, *coverage* is 0%. The first solution degrades to the whole-program tuning. The second solution does not find any tuning section. Neither of them is acceptable. In fact, we should use the loop body in *main* as a tuning section. Using a call graph profile, the selection algorithm cannot identify the loops within a procedure. Extra work is needed to find the loop and extract its body into a separate procedure. We call this step of the algorithm extra *code partitioning*. After this step, the loop body appears in the call graph profile, which then is chosen as a tuning section by the selection algorithm. Finding the procedures that are not selected as tuning sections but worth extra code partitioning is another task of the algorithm.

4.3 The Tuning Section Selection Algorithm

From the previous section, the tuning section selection problem can be viewed as a constrained max cut problem. (The original max cut problem is NP-complete [Karp 1972].) In this section, we will develop a greedy algorithm to select the tuning sections so as to get maximal N_{min} and *coverage*. This algorithm solves the problem in two steps. (1) We design an algorithm which aims to maximize *coverage*, under the constraint that the number of invocations to each selected tuning section is larger than a lower bound N_{lb}. (2) The final algorithm raises the N_{lb} gradually to trade off the *coverage*. It aims to achieve a large N_{min} by tolerating a small decrease in *coverage*. These two steps will be presented in Section 4.3.2 and Section 4.3.3 separately. We discuss the handling of recursive functions in Section 4.3.1.

4.3.1 *Dealing with Recursive Functions.* To call a recursive function, the program makes an *initial call* to the function. Then the function is called by itself, in the case of self-recursion, or by its callees, in the case of mutual-recursion. Both self-recursive calls and mutually-recursive calls are referred to as *recursive calls*, which are different from the initial call. Our PEAK system treats initial calls to a recursive function as normal function calls; recursive calls can be viewed as loop iterations, which are ignored by tuning section selection. In other words, the tuning section selection algorithm does not choose the call graph edges that correspond to recursive calls, but only the edges corresponding to initial calls.

In a call graph, the functions (nodes) that recursively call themselves or each other form cycles (including loops). To exclude recursive calls from tuning section selection, our algorithm identifies the cycles and ignores the edges that appear in the cycles. We do this through a call graph simplification process. This process merges the nodes involved in a common cycle into one node, removes the edges used inside a cycle, and adjusts the edges entering or leaving the merged nodes. This process uses the strongly connected components to find the nodes and edges that appear in a cycle. It adjusts the profile data as well. The pseudo-code of this simplification process is shown in Figure 8. An example is shown in Figure 9.

This simplified call graph removes the self-recursive calls and makes a new node for the functions that recursively call each other. This graph is a Directed Acyclic Graph (DAG). The call graph simplification algorithm maintains the profile information for the new nodes and edges. So, the graph has the same profile information as described in Section 4.1.

4.3.2 *Maximizing Tuning Section Coverage under N_{lb}.* This section describes a tuning section selection algorithm, which aims to achieve as large a *coverage* as possible, under the constraint that the number of invocations to each selected tuning section is larger than a lower bound N_{lb}. This algorithm selects the tuning sections and puts their entry functions into set T. It finds the functions that are worth code partitioning and puts them into set M. Besides N_{lb}, this algorithm uses two other parameters: (1) T_{lb}, the lower bound on

Subroutine G = GenerateSimplifiedCallGraph(profile)

(1) Construct call graph $G = (V, E)$ according to the profile data as described in Section 4.1. For each node $v \in V$, $v = \{fn\}$: $fn(v)$ is the function name. For each edge $e \in E$, $e = \{s, t, n, tm\}$: $s(e)$ is the caller node; $t(e)$ is the callee node; $n(e)$ is the number of invocations to $t(e)$ made by $s(e)$; $tm(e)$ is the time spent in $t(e)$ and its callees, when $t(e)$ is called from $s(e)$.

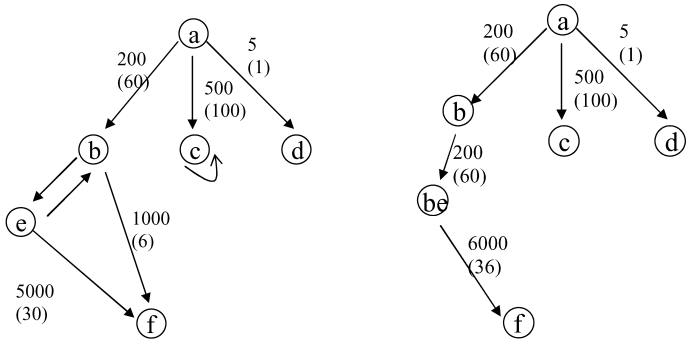
(2) Remove loops in G. (A loop identifies a self-recursive call.)

(3) For each strongly connected component SCC that contains more than one nodes, do the following to remove the cycles by merging the nodes in SCC. (Such strongly connected components identify mutually-recursive calls.)
   (a) Construct a new node u for SCC. The name of u, $fn(u)$ is the concatenation of the names of all the nodes in SCC.
   (b) Remove the inner edges, i.e., the edges starting from and ending at SCC.
   (c) Keep all the edges leaving SCC and set their starting node to be the new node u. Merge the edges that start from u and end at the same node, and sum up the profile data for the merged edges.
   (d) For each node v in SCC, remove v if there is no edge entering v.
   Otherwise, add a new edge e, which leaves v and enters the new node u. The profile information for this e is the sum of the profile data for all the edges enters v:

$$n(e) = \sum_{x \in E, t(x)=v} n(x) \tag{23}$$

$$tm(e) = \sum_{x \in E, t(x)=v} tm(x) \tag{24}$$

Fig. 8. The pseudo-code for call graph simplification. The algorithm generates a call graph from profile data, detects and discards recursive calls. Hence, the call graph is simplified to a directed acyclic graph.



(a) Call graph before simplification (b) Call graph after simplification

Fig. 9. An example of call graph simplification. The graph is a call graph with node a as the main function. c is a self-recursive function. b and e recursively call each other. The weights on an edge are the number of invocations and the execution time in the parentheses. After simplification, the loop at node c is discarded. The strongly connected component $\{b, e\}$ is merged into one node be. The entry node b for this strongly connected component is kept. A new edge (b, be) is added. Edges (b, f) and (e, f) are merged to (be, f). The profile data on edges (b, be) and (be, f) are updated.

average execution times; (2) P_{lb}, the lower bound on the execution percentage for a code section worth code partitioning.

T_{lb} is determined by the timing accuracy of the PEAK system. We use 100 μ sec in our experiments. P_{lb} is used to determine whether a code section is worth code partitioning. We use 0.02. This means that the code section

Subroutine [T, M] = MaxCoverage(profile, N_{lb}, T_{lb}, P_{lb})
There are three thresholds used in this algorithm: N_{lb}, the lower bound on numbers of invocations; T_{lb}, the lower bound on average execution times; P_{lb}, the lower bound on the execution percentage for a code section worth partitioning. The algorithm selects the tuning sections and puts their entry functions into set T. The functions that are worth code partitioning are put into M.

(1) Construct the simplified call graph $G = (V, E)$, according to Figure 8. i.e., G = GenerateSimplifiedCallGraph(profile).
(2) Clear the selection flag for each edge $e \in E$: $f(e) = 0$. Clear the execution time due to the selected tuning sections, for each node v: $TX(v) = 0$. Empty set T and M.
(3) Mark the edges whose average execution times are less than T_{lb}. i.e., for each edge $e \in E$, set $f(e) = -1$, if $tm(e)/n(e) < T_{lb}$. (These edges will not be counted when summing the profile data of the edges for one node.)
(4) Sort all nodes in topological order: $v_1, v_2, \ldots$, i.e., if there is an edge (u, v), node u appears before node v. The nodes will be traversed in this order.
(5) For node v_i ($i = 1, 2, \ldots$), compute the total number of invocations to v_i.

$$n(v_i) = \sum_{e \in E, t(e)=v_i, f(e)=0} n(e) \tag{25}$$

If $n(v_i)$ is greater than N_{lb}, put v_i into T and set $f(e) = 1$ if $t(e) = v_i$; and update profile information of G, by calling UpdateProfile(G, v_i, TX).
(6) Put node v into M, if v is not selected but consumes a large amount of execution time, i.e., $\sum_{t(e)=v, f(e)\neq 1} tm(e) - TX(v) > P_{lb} \times T_{total}$. (These nodes may be partitioned to improve tuning section coverage.)

Fig. 10. Tuning section selection algorithm to maximize program coverage under the lower bound on numbers of TS invocations, N_{lb}. This algorithm traverses the simplified call graph from top down to find the code sections whose numbers of invocations are greater than N_{lb}. In addition, the algorithm finds the functions that may be partitioned to improve tuning section coverage.

is worth tuning if its execution time is greater than 2% of the total execution time. N_{lb} will be adjusted to trade off the tuning section *coverage* in the final tuning section selection algorithm described in Section 4.3.3. The optimal N_{lb} picked by the final algorithm usually ranges from tens to thousands.

In order to maximize the coverage, the algorithm traverses the call graph from top down to select the code sections that meet the requirements. (We use a topological order to go through the call graph.) When a tuning section is selected, the profile data are updated to reflect the execution times and invocation numbers after excluding this selected tuning section. The execution time due to the selected tuning section is deducted from the execution time of its ancestors as well. (Remember that the execution time on node v includes the time spent in v itself and the descendants of v.) After the selection process finishes, the remaining execution time on each node v is used to judge whether it is worth code partitioning. (It is worth code partitioning, if its execution time is greater than P_{lb} of the total execution time.)

Figure 10 shows the pseudo-code for the algorithm to maximize the tuning section coverage. This algorithm constructs an acyclic call graph, annotated with profile information after removing the recursive calls, according to the algorithm described in Section 4.3.1. It ignores the edges whose average execution time is less than threshold T_{lb} when computing the execution profile for a node. It goes through the call graph in a topological order to find the nodes whose numbers of invocations are greater than threshold N_{lb}. These nodes are

Subroutine UpdateProfile($G = (V, E)$, v_i, TX)
This algorithm removes the execution time due to v_i for each edge reachable from v_i, and adds the time due to v_i into TX for each ancestor of v_i. ($TX(v)$ records the time spent in v due to all the selected tuning sections. Manual code partitioning will use TX to estimate the remaining execution time after tuning section selection.)
Update the profile information of the edges that are reachable from node v_i.

(1) Set execution frequency of all nodes except v_i as 0, $q(v) = 0$; while $q(v_i) = 1.0$, which means v_i is executed 100% within the chosen tuning section.
(2) For node v_j ($j = i+1, i+2, \ldots$), compute the execution frequency of v_j based on the execution frequency of its parents according to the following equation.

$$q(v_j) = \frac{\sum_{e \in E, t(e)=v_j} q(s(e)) \times n(e)}{\sum_{e \in E, t(e)=v_j} n(e)} \tag{26}$$

(3) For node v_j ($j = i+1, i+2, \ldots$), adjust the profile information for the edges entering v_j according to the following equations, where $t(e) = v_j$.

$$n(e) = n(e) \times (1 - q(s(e))) \tag{27}$$

$$tm(e) = tm(e) \times (1 - q(s(e))) \tag{28}$$

For the nodes that reach node v_i, record how much execution time is spent due to v_i. This time will be excluded for choosing code partitioning candidates.

(1) For each node v, set the time spent in v due to v_i, $p(v)$, as 0.
(2) Set the time spent in v_i, $p(v_i) = \sum_{e \in E, t(e)=v_i} tm(e)$.
(3) For node v_j ($j = i, i-1, .., 1$), update the time spent in its parents due to the time spent in v_i. For each edge e, who enters v_j, i.e., $t(e) = v_j$, adjust $p(s(e))$ according to the following equation.

$$p(s(e)) = p(s(e)) + \frac{tm(e)}{\sum_{a \in E, t(a)=v_j} tm(a)} \times p(v_j) \tag{29}$$

(4) For node v_j ($j = i-1, i-2, .., 1$), update the time spent in v due to all the selected tuning sections: $TX(v_j) = TX(v_j) + p(v_j)$

Fig. 11. Update profile data after v_i is chosen as the entry function to a tuning section. The updated profile reflects the execution times and invocation numbers after excluding the chosen tuning section.

selected as entry functions to the tuning sections. The profile information of the relevant edges is adjusted to reflect the number of invocations and execution time spent in the rest of the program, when a node is selected. In the end, the algorithm finds the functions worth code partitioning, using the residual execution time after excluding the selected tuning sections.

When a tuning section is selected, the profile data needs to be updated in order to exclude the execution information (time and number of invocations) due to the selected tuning section. This requires a context-sensitive inclusive profile, which lists the direct and indirect callers of a function, when the execution information of this function is given. This profile should show the execution time and number of invocations of each function call; it should also split this information for each call path. Unfortunately, *gprof* does not provide such information. Instead, we use an algorithm to do an estimation, which is described in Figure 11. It handles the descendants and the ancestors of the selected node v separately.

For each descendant u, the algorithm estimates how often u is directly or indirectly called from the selected node v, out of the total invocations to u. This execution frequency of u is computed based on the execution frequency of u's

parents and the numbers of invocation to u directly from the parents, assuming the execution frequency of v is 100%. Equation (30) does the estimation, where $q(u)$ is the execution frequency of u.

$$q(u) = \frac{\sum_{e \in E, t(e)=u} q(s(e)) \times n(e)}{\sum_{e \in E, t(e)=u} n(e)} \tag{30}$$

Knowing the execution frequency of all the descendants, the profile information for all the edges reachable from v can be estimated in accordance with Eqs. (31) and (32).

$$n(e) = n(e) \times (1 - q(s(e))) \tag{31}$$

$$tm(e) = tm(e) \times (1 - q(s(e))) \tag{32}$$

The algorithm traverses the node list forwards from node v. (The nodes are sorted in a topological order.)

For each ancestor u of the selected node v, the algorithm estimates how much execution time is spent in u due to v, which is denoted as $p(u)$. The time spent in the selected node v, $p(v)$, is equal to $\sum_{e \in E, t(e)=v} tm(e)$. We assume that the time spent in u due to v is distributed in u's parent, x, proportional to the time spent in u when u is called from x. That is, $p(u)$ is distributed to $p(x)$ proportional to $tm(e)$, where e leaves x and enters u. So, $p(u)$ is distributed in accordance with to the following equation.

$$p(s(e)) = p(s(e)) + \frac{tm(e)}{\sum_{a \in E, t(a)=u} tm(a)} \times p(u) \tag{33}$$

The algorithm traverses the node list backwards from node v.

4.3.3 *The Final Tuning Section Selection Algorithm.* The previous section describes how to maximize the tuning section coverage with the constraint that the number of invocations to each selected tuning section should be larger than N_{lb}. This section aims to achieve a large N_{min} by tolerating a small decrease in *coverage*. It does this by raising N_{lb} gradually to trade against *coverage*.

Two new parameters are introduced to the final algorithm.

(1) C_{lb}, the lower bound on the tuning section coverage. If the coverage of the selected tuning sections is smaller than C_{lb}, code partitioning is necessary. We set it as 80% of the total execution time.

(2) R_{ub}, the upper bound on the coverage drop rate. The coverage drop rate is computed based on two tuning section selection solutions as follows, where N_{min2} is larger than N_{min1}.

$$R = \frac{coverage_1 - coverage_2}{N_{min2} - N_{min1}} \tag{34}$$

If, on average, after increasing N_{min} by 1, the coverage drops more than R_{ub}, the algorithm finds a trade-off point. In our experiments, we tolerate 1% decrease of the coverage, if N_{min} can be improved by 100. So, we set R_{ub} as $0.01/100 = 1e^{-4}$.

Subroutine [T, M] = TSSelection(profile, R_{ub}, C_{lb}, T_{lb}, P_{lb})
There are four thresholds used in this algorithm: R_{ub}, the upper bound on the coverage drop rate; C_{lb}, the lower bound on the tuning section coverage; T_{lb}, the lower bound on average execution times; P_{lb}, the lower bound on the execution percentage for a code section worth code partitioning. The algorithm selects the tuning sections and puts their entry functions into set T. The functions that are worth code partitioning are put into M.

(1) Initialization. $T_{best} = \phi$, $M_{best} = \phi$, $coverage_{best} = 0$, $N_{best} = 0$, $N_{lb} = 10$.
(2) Use the algorithm in Figure 10 to maximize the coverage.
    [T, M] = MaxCoverage(profile, N_{lb}, T_{lb}, P_{lb})
    Compute *coverage* and N_{min} of the selected tuning sections T.
(3) If *coverage* is less than C_{lb}, do the following.
    (a) If T_{best} is ϕ, code partitioning is necessary. Print T and M. Stop.
    (b) Else, T_{best} and M_{best} is the solution. Print T_{best} and M_{best}. Stop.
(4) Trade off *coverage* and N_{min} as follows.
    (a) If T_{best} is ϕ, record this solution: $T_{best} = T$, $M_{best} = M$, $coverage_{best} = coverage$, $N_{best} = N_{min}$.
    (b) Else, compute the coverage drop rate R.

$$R = \frac{coverage_{best} - coverage}{N_{min} - N_{best}} \tag{35}$$

    If the drop rate R is less than R_{ub}, record this solution: $T_{best} = T$, $M_{best} = M$, $coverage_{best} = coverage$, $N_{best} = N_{min}$.
    Otherwise, the trade-off point is found. Print T_{best} and M_{best}. Stop.
(5) Set $N_{lb} = N_{min}$ and go to Step (2).

Fig. 12. The final tuning section selection algorithm. This algorithm achieves both a large N_{min} and a high *coverage*. It iteratively uses the method shown in Figure 10 to maximize the tuning section coverage under a series of thresholds N_{lb}'s, until the optimal N_{lb} is found.

Table I. Tuning Section Selection for *mgrid*. The Best N_{lb} is 400. The Optimal *coverage* and N_{min} are 0.957 and 2000, Respectively

iteration	N_{lb}	*coverage*	N_{min}
1	10	0.998	400
2	400	0.957	2000
3	2000	0.808	2400

Figure 12 shows the pseudo code of this tuning section selection algorithm. This algorithm iteratively uses the method shown in Figure 10 to maximize the tuning section coverage under a series of thresholds N_{lb}'s. The new N_{lb} in the next iteration, N_{lb2}, is equal to the N_{min} obtained from the previous iteration. We notice that this N_{min} is greater than the old N_{lb} in the previous iteration, N_{lb1}, and that any threshold value in [N_{lb1}, N_{min}) gives the same solution to maximize the coverage. Using N_{min} from the previous iteration as the new threshold value for N_{lb} makes the trade-off process fast. This process finishes when the coverage drops below C_{lb} or the coverage drop rate is greater than R_{ub}.

For example, Table I shows the result of each iteration, via applying this algorithm to benchmark *mgrid*. The second iteration gets the optimal result, with *coverage* = 0.957 and $N_{min} = 2000$. (The initial N_{lb} is 10, which is a reasonable boundary for our rating methods to achieve faster tuning than whole-program tuning.)

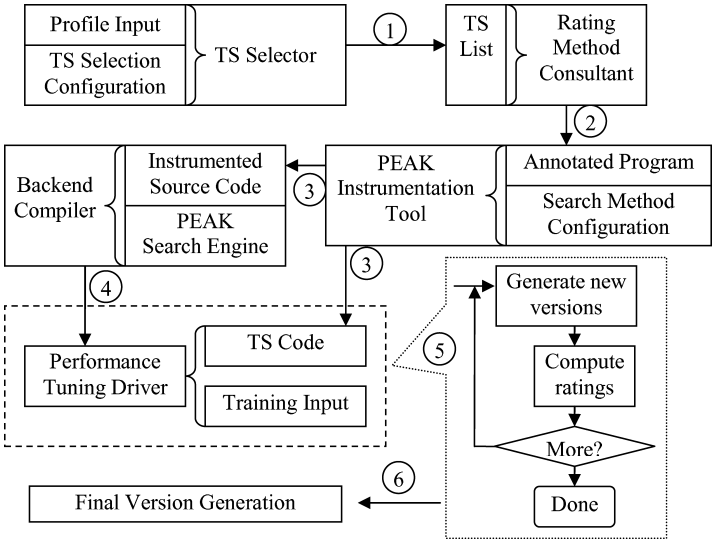


Fig. 13. Block diagram of the PEAK performance tuning system. The blocks in the diagram are the components in PEAK and the input and output of these components. Steps (1) to (6) correspond to the ones shown in Figure 1.

5. THE PEAK SYSTEM

Section 2 presented a fast and effective orchestration algorithm, which iteratively generates optimized code versions and evaluates their performance based on the execution time under a training input to search for the best version. Noticing that using a partial execution of the program may enable performance tuning in a faster way, we developed three rating methods. These methods are applied to important code sections of the program, called tuning sections. The rating methods, presented in Section 3, evaluate the performance of an optimized version of a tuning section based on a number of invocations of the tuning section. Section 4 showed an algorithm to select the tuning sections out of a program to maximize both the program execution time coverage and the numbers of invocations to the tuning sections, aiming at high-tuned program performance and fast-tuning speed. This section puts everything together to construct an automated performance tuning system, PEAK. Section 5.1 shows the design of PEAK. Special implementation problems related to runtime code generation and loading are discussed in Section 5.2. Section 5.3 gives an example of using the PEAK system.

5.1 The Design of PEAK

Figure 13 displays a block diagram of the PEAK system, showing system components. The PEAK system has two major parts: the *PEAK compiler* and the *PEAK runtime* system. The PEAK compiler is used before tuning, while the PEAK runtime system is used during tuning.

(1) The *tuning section selector* chooses the important code sections as the tuning sections, using the call graph profile generated by *gprof*. It applies the

algorithm described in Section 4. The output of this step is a list of function names, each of which identifies a tuning section.

(2) The *rating method consultant* analyzes the source program to find the applicable rating methods for each tuning section. The compiler techniques developed in Section 3 are implemented here. This tool annotates the program with the information about the context variables for CBR and the performance model for MBR.

(3) The *PEAK instrumentation tool* applies the appropriate rating method to each tuning section, after a profile run to find the major context for CBR and the model parameters for MBR according to Section 3. It adds the initialization and finalization functions to activate the PEAK runtime system and the functions to load and save the tuning state of previous runs, since the *performance tuning driver* may run the program multiple times in Step (5). Each tuning section is retrieved into a separate file, which will be compiled at Step (5) under different optimization combinations to generate optimized versions.

(4) The instrumented code is compiled and linked with the PEAK runtime library, forming the *performance tuning driver*. The PEAK runtime system implements the three rating methods described in Section 3 and the CE optimization orchestration algorithm developed in Section 2. Special functions for dynamically loading the binary code during tuning are also included in the PEAK runtime system. (The generation of the tuning driver and the optimized tuning section versions is done by the backend compiler, in this paper, the GCC compiler.)

(5) The performance tuning driver iteratively runs the program under a training input until optimization orchestration finishes for all tuning sections. At each invocation to a tuning section, the driver takes over control. It runs and times the current experimental version and decides whether more invocations are needed to rate the performance of this version. After the rating of this version is done (i.e., when the rating variance is small enough) the driver generates new experimental versions according to the orchestration algorithm. (The tuning sections are tuned independently.) The tuning process ends when the best version is found for each tuning section.

(6) After the tuning process finds the best optimized version for each tuning section, these best versions are linked to the main program to generate the final tuned code.

5.2 Dynamic Code Generation and Loading

PEAK tunes program performance via executing the program under a training input. It generates and loads the binary code at runtime. (This distinguishing feature is inherited from the ADAPT [Voss and Eigenmann 2000, 2001] infrastructure.) To do so, PEAK uses the dynamic linking facility functions: *dlopen*, *dlsym*, *dlclose* and *dlerror*. Basically, these functions enable PEAK to load binary code into memory and to resolve the address of the experimental version contained in that binary. (The binary code is generated by the backend compiler, GCC, under the given optimization options.)

To generate an optimized version for an individual tuning section, PEAK extracts the section into a separate source file. This file includes the entry

```
SUBROUTINE calc1
...
DO j = 1, n, 1
  DO i = 1, m, 1
    ...
  ENDDO
ENDDO
DO j = 1, n, 1
  ...
ENDDO
DO i = 1, m, 1
  ...
ENDDO
...
RETURN
END
```

Fig. 14. The tuning section *calc1* in *swim*.

function to the tuning section and all its direct and indirect callees; it can be compiled and optimized separately. Since callees are included in the tuning section, inlining can be performed during code generation. If a procedure is called in two tuning sections, this procedure is replicated and renamed in the corresponding source files. Such replication does not lead to code explosion, because the number of tuning sections is fixed and usually small, around three. Our PEAK compiler performs the above tasks automatically through source-to-source translation.

To load an optimized version successfully and correctly, PEAK needs to pay attention to the global variables used in the tuning sections, especially the static variables in C and the common blocks in Fortran. Because multiple versions of the same tuning section are invoked during one run of the program, PEAK ensures that each global variable used in these versions must be located at the same address, when the versions are loaded.

5.3 An Example of Using PEAK

This section uses the benchmark *swim* to illustrate the six tuning steps of PEAK.

(1) The tuning section selector uses a *gprof* output and selects four tuning sections: *calc1*, *calc2*, *calc3* and *loop3500*. Figure 14 shows the source code of the tuning section *calc1* as an example.
(2) The rating method consultant analyzes the source code to find the applicable rating methods. For *calc1*, all three rating methods are applicable. The context variables are n and m. After one profile run, PEAK finds that n and m are runtime constants. So, there is only one context for *calc1* and context-based-rating will be applied.
(3) The PEAK instrumentation tool adds PEAK runtime library calls to the source program. Each tuning section is retrieved into a separate file, which will be compiled under different optimization combinations to generate experimental versions during performance tuning. The instrumentation

```
extern void calc1_old_();
void calc1_()
{
  hrtime_t t0, t1;
  typedef void(*FunType)();
  FunType fun = NULL;
  hrtime_t tm;

  //Experiment the current optimized version
  if( (fun=(FunType)DCGetVersion(0)) != NULL ) {
    t0 = gethrtime();
    fun();                //invoke the experimental version
    t1 = gethrtime();
    tm = t1 - t0;         //time the current invocation
    DCMonRecordCBR(0, tm); //rating generation
    return;
  }
  //Optimization orchestration is done
  calc1_old_();
}
```

Fig. 15. The tuning section *calc1* instrumented by the PEAK compiler.

code is added to each tuning section and to the entry and exits of the program.

—Each entry to a tuning section is redirected to the corresponding instrumented code. Figure 15 shows the instrumented *calc1*, which calls two PEAK runtime functions, *DCGetVersion* and *DCMonRecordCBR*.

(a) *DCGetVersion* implements the optimization orchestration algorithm and dynamic code generation and loading. It returns the current experimental version. If the previous version has already been rated, this call will generate and load a new optimized version.
(b) *DCMonRecordCBR* passes the execution time obtained from a high resolution timer to the context-based-rating method. The rating method rates the current version based on a small number of invocations. The orchestration algorithm uses the ratings of previous versions to guide the generation of new optimized versions.
(c) The argument, 0, of these two functions identifies the tuning section of *calc1*.

In the code, *calc1_old_* is the original version of the tuning section.

—At the entry of the entire program, Function *init_dc* is called to set up the tuning state. It initializes the tuning state for four tuning sections, sets up the orchestrated GCC optimizations, specifies the optimization orchestration algorithm and the rating method for each tuning section, and loads the tuning state from the previous run, because multiple runs of the program may be involved during performance tuning.

—At the exits of the program, Function *exit_dc* is called to save the tuning state.

(4) The instrumented program is compiled and linked with the PEAK runtime library to generate the performance tuning driver.

(5) During performance tuning, the tuning driver continuously runs under a training input until optimization orchestration is done for all tuning sections. Its output shows the best optimization combination for each tuning section. An example output for *calc1* on a Pentium 4 machine is "g77 -c -O3 -fno-strength-reduce -fno-rename-registers -fno-align-loops calc1.f". This means that three optimizations, strength-reduce, rename-registers and align-loops, should be turned off from the baseline O3.

(6) In the final version generation stage, each tuning section uses the corresponding best optimization combination, as defined in the previous step. To generate the final program version, these optimized tuning sections are linked to the main program, which is compiled under the default optimization combination,

6. EXPERIMENTAL RESULTS

6.1 GCE Performance on SUN Forte Compilers

In Pan and Eigenmann [2006], we have compared our CE algorithm with other optimization orchestration algorithms. We have shown that CE takes the least tuning time (57% of the closest alternative), while achieving the same program performance. In this section, we conduct an experiment to evaluate the GCE algorithm for whole-program tuning using SUN Forte compilers. GCE is our generalized CE algorithm described in Section 2.2, aiming at multi-valued compiler options. We compare the performance achieved by our GCE algorithm with the one by manual tuning, which is based on the SPEC CPU2000 performance results [SPEC 2000]. In such a result, there is a *base* setting and multiple *peak* settings for optimization options. The base setting is common to all benchmarks. We use this as the baseline performance. The peak settings may be different for different benchmarks and achieves better performance than the base setting. We will compare GCE with the peak setting.

The experiments are conducted on a Sun Enterprise 450 SPARC II machine. We evaluate the GCE algorithm using the optimization options of the Forte Developer 6 compilers. The baseline option setting is "-fast -xcrossfile -xprofile". Table II lists the tuned options. The Forte compilers have some options that can be passed directly to the compiler components. These options are passed by the "-W" option for the C compiler and the "-Qoption" option for the Fortran compiler or the C++ compiler. In this table, we list the "-Qoption" only ("-W" options are similar).

Figure 16 shows the performance results. GCE tunes performance using *train* dataset as input. The final performance is evaluated with *ref* dataset. In summary, GCE achieves equal or better performance for each benchmark. On average, for floating point benchmarks, GCE achieves 10.8% improvement relative to the base setting, compared to 5.6% by the peak settings. For integer benchmarks, GCE achieves 8.1% compared to 4.1% by the peak settings.

Table II. Optimization Options Orchestrated by GCE

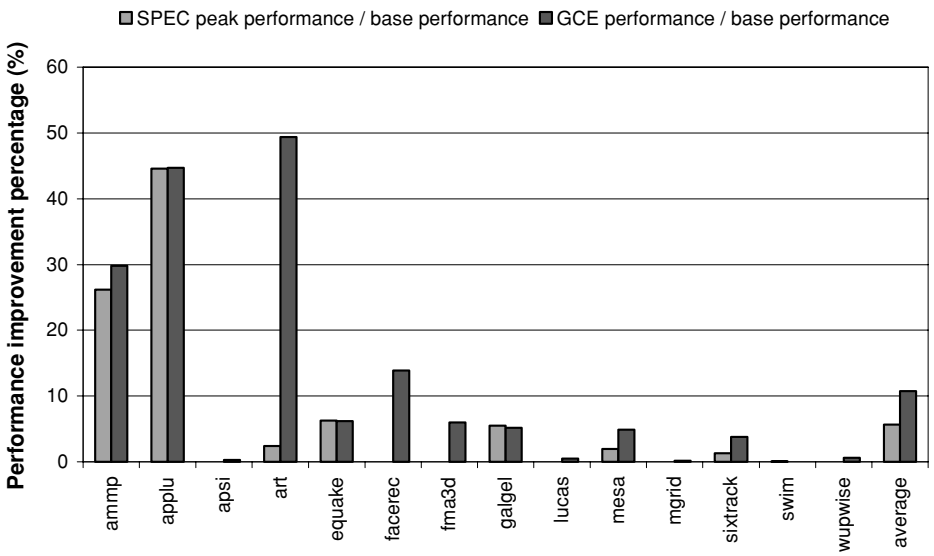
Flag Name	Experimented Values	Meaning
-xarch	v8, v8plus, generic	target architecture instruction set
-xO	3, 4, 5	optimization level
-xalias_level	std, strong, basic	alias level
-stackvar	on/off	using the stack to hold local variables
-d	y, n	allowing dynamic libraries
-xrestrict	%all, %none	pointer-valued parameters as restricted pointers
-xdepend	on/off	data dependence test and loop restructuring
-xsafe=mem	on/off	assuming no memory protection violations
-Qoption iropt -crit	on/off	optimization of critical control paths
-Qoption iropt -Abopt	on/off	aggressive optimizations of all branches
-Qoption iropt -whole	on/off	whole program optimizations
-Qoption iropt -Adata_access	on/off	analysis of data access patterns
-Qoption iropt -Mt	500, 1000, 2000, 6000, default	max size of a routine body eligible for inlining
-Qoption iropt -Mr	6000,12000,24000,40000,default	max code increase due to inlining per routine
-Qoption iropt -Mm	6000, 12000, 24000, default	max code increase due to inlining per module
-Qoption iropt -Ma	200, 400, 800, default	max level of recursive inlining

6.2 Results of Tuning Section Selection

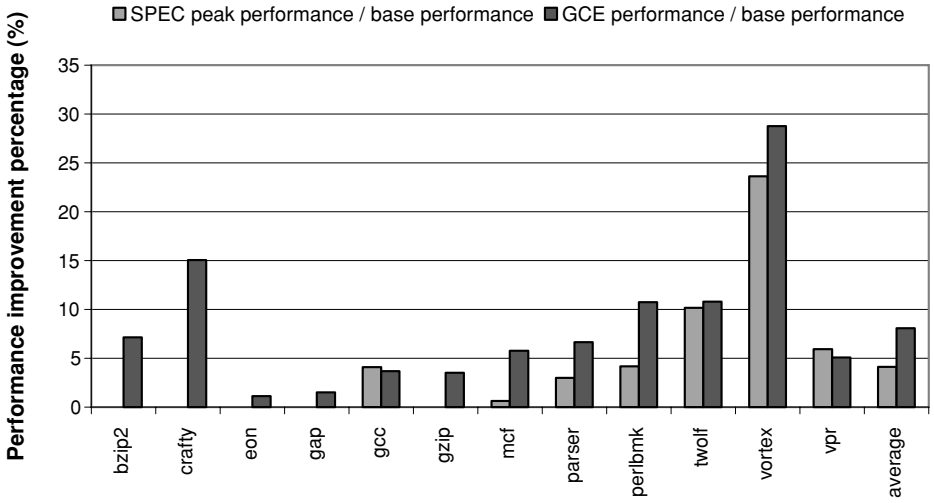
This section evaluates the tuning section selection algorithm described in Section 4. The default threshold values are used, that is, $R_{ub} = 1e^{-4}$, $C_{lb} = 80\%$, $T_{lb} = 100\ \mu$ sec, $P_{lb} = 2\%$. Again, R_{ub} is the upper bound on the coverage drop rate; C_{lb} is the lower bound on the tuning section coverage; T_{lb} is the lower bound on average execution times; P_{lb} is the lower bound on the execution percentage for a code section worth code partitioning. Focusing on the scientific programs, we analyze the FP benchmarks in Section 6.2.1 in detail. The results on the INT benchmarks are presented in Section 6.2.2. The data are listed in Table III and Table IV, respectively.

6.2.1 *SPEC CPU2000 FP Benchmarks.* Table III shows the results of our tuning section selection algorithms, demonstrating that the algorithm achieves the goal of maximizing both the program coverage and the number of invocations to the tuning sections. In most benchmarks, the coverage is 90% or higher. Three codes needed extra code partitioning.

The minimum number of invocations, N_{min}, ranges from hundreds to thousands for all the benchmarks, except *wupwise*. *Wupwise* contains many small functions with an average execution time in the order of μsecs, which are below the chosen threshold T_{lb}. When lowering the threshold to 1 μ sec, the algorithm



(a) SPEC CPU2000 FP benchmarks



(b) SPEC CPU2000 INT benchmarks

Fig. 16. Program performance achieved by the GCE algorithm vs the performance of the manually tuned results (peak setting). Higher is better. In all cases, GCE achieves equal or better performance. On average, GCE nearly doubles the performance.

finds tuning sections that are called a large number of times (22528000). This solution necessitates a high-resolution timer for accurate measurements, which is available in our system.

The three benchmarks that needed extra partitioning are *equake*, *sixtrack* and *swim*. In these codes, our tuning section selection algorithm identifies the procedures that take a large execution time but with only a few invocations. In these procedures, the extra partitioning step extracts the loop body of the

Table III. Tuning-Section Selection Results for SPEC CPU2000 FP Benchmarks. (Three benchmarks that needed extra code partitioning are annotated with '*'. The last row, wupwise+, uses a smaller $T_{lb} = 1$ μ sec.)

Benchmark	*coverage*	N_{min}	# of TS
ammp	88.6	127	3
applu	97.9	250	5
apsi	87.8	720	9
art	99.9	250	2
equake	54.6	2709	1
equake*	99.0	2709	1
mesa	96.9	4000	1
mgrid	95.7	2000	4
sixtrack	10.4	208	2
sixtrack*	97.9	1693	2
swim	83.9	198	3
swim*	99.2	198	4
wupwise	91.7	22	1
wupwise+	83.0	22528000	2

Table IV. Tuning-Section Selection Results for SPEC CPU2000 INT Benchmarks. (The benchmarks annotated with '+' use smaller T_{lb}'s.)

Benchmark	*coverage*	N_{min}	#TS
bzip2	99.9	22	6
crafty	100.0	1272	1
gap	96.6	24	3
gcc	90.7	109	1
gzip	41.7	3668	3
gzip+	84.1	4021	5
mcf	46.6	5235	1
mcf+	74.7	5235	2
parser	91.6	309	3
perlbmk	76.2	11	5
perlbmk+	92.6	11	6
twolf	98.0	120	1
vortex	78.0	6209	2
vortex+	95.5	4000	7
vpr	52.7	10746	1
vpr+	98.3	10746	2

important loop into a separate procedure. After partitioning, the new procedure will cover a significant part of the program execution time with a large number of invocations. Extra code partitioning, while automatable, was done manually in our experiments.

6.2.2 *SPEC CPU2000 INT Benchmarks.* From Table IV, our tuning section selection algorithm achieves the goal of maximizing the program coverage and the number of invocations to the tuning sections for INT benchmarks as well.

In general, the profiles of INT benchmarks are different from those of FP benchmarks: (1) The INT benchmarks have flatter profiles with more functions

Table V. 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

in the call graphs. (2) The call graphs of the INT benchmarks are deeper. (3) The functions in the INT benchmarks usually have small average execution times. Due to (3), the default threshold of the average execution time, $T_{lb} = 100$ μ sec, prohibits the selection of some important functions in *gzip*, *mcf*, *perlbmk*, *vortex*, and *vpr*. After using a smaller T_{lb} ranging from 0.01 μ sec to 1 μ sec, their program coverage is improved. (For these benchmarks, a high resolution timer is needed to generate accurate performance ratings for the selected tuning sections.)

In *bzip2*, *gap* and *perlbmk*, some important functions are only invoked tens of times, which brings down N_{min}, the minimum number of invocations to the tuning sections. For other benchmarks, N_{min} is hundreds to thousands.

6.3 PEAK Performance

6.3.1 *Experimental Environment for PEAK.* We evaluate PEAK using the optimization options of the GCC 3.3.3 compiler on two different computer architectures: Pentium 4 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 V and are described in the GCC manual [GNU 2005].

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 the whole-program tuning, to ensure accurate measurements and to 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.[3] 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.

6.3.2 *Metrics.* Two important metrics characterize the behavior of optimization orchestration:

(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 use the metric of *normalized tuning time* (*NTT*). *NTT* is the tuning time (*TT*) normalized by the time of evaluating the whole program optimized under O3, that is, one compilation time (CT_B) plus three execution times (ET_B) of the base version.

$$NTT = TT/(CT_B + 3 \times ET_B). \tag{36}$$

A good optimization orchestration method is meant to achieve both high *program performance* and short *normalized tuning time*.

6.3.3 *SPEC CPU2000 FP Benchmarks.* Figure 17 shows the normalized tuning time of the whole-program tuning and the PEAK system. (For PEAK, the tuning time includes the time spent in all the six tuning steps.) On average, the normalized tuning time is reduced from 68.3 to 3.36. So, PEAK gains a tuning speedup of 20.3. The benchmark that has a high speedup usually has a large number of invocations to the tuning sections, which are shown in Table III. This agrees with our assumption during the tuning section selection in Section 4. On average, the absolute tuning time is reduced from 2.19 hours to 5.85 minutes. Applying the rating methods in Section 3 significantly improves the tuning time.

Figure 18 shows the tuning time percentages of the six tuning steps. Most of the time is spent in Step (5), the performance tuning (PT) stage. The second largest portion of tuning time is spent in Step (1), the tuning section selection

[3]Comparing the binaries generated under two different optimization combinations via the UNIX utility, *diff*, can show whether these two binaries are identical.

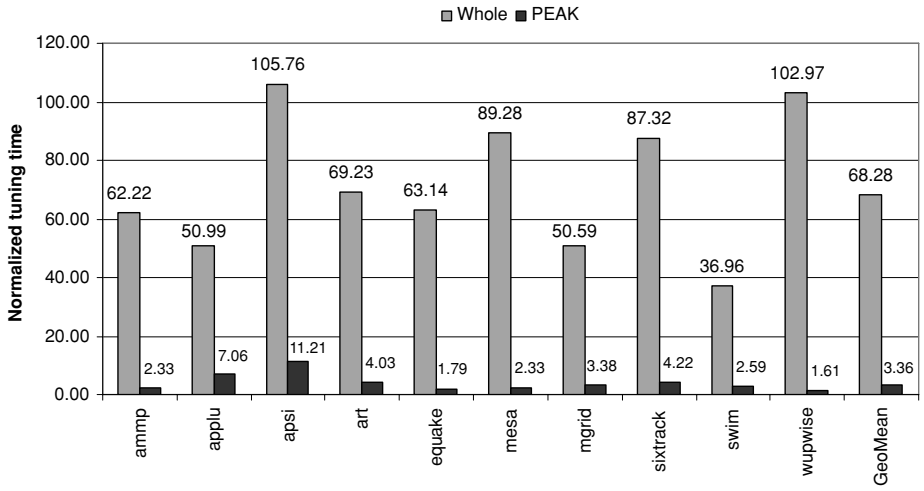


Fig. 17. Normalized tuning time of the whole-program tuning and the PEAK system for SPEC CPU2000 FP benchmarks on Pentium 4. Lower is better. On average, PEAK gains a speedup of 20.3 over the whole-program tuning.

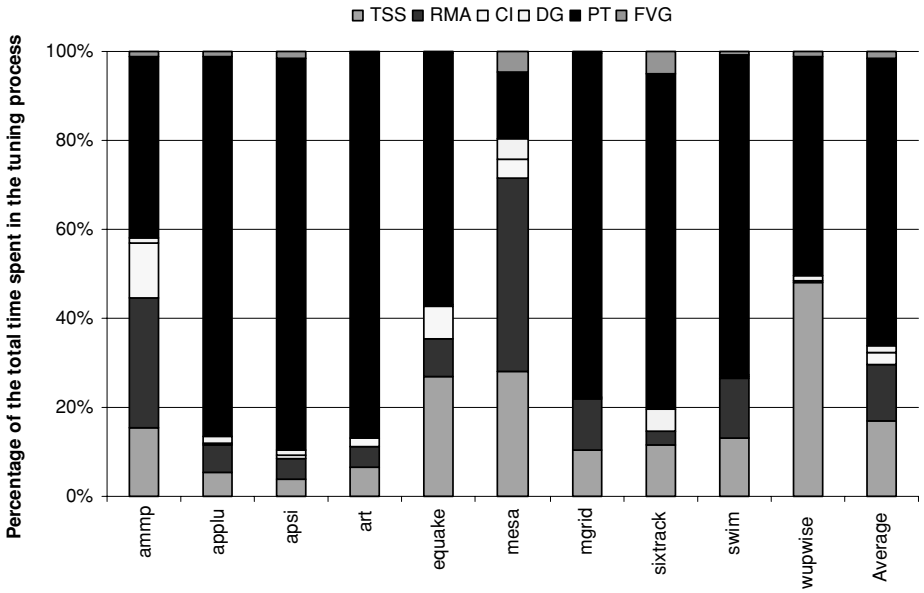


Fig. 18. Tuning time percentage of the six stages shown in Figure 1 for SPEC CPU2000 FP benchmarks on Pentium 4. (TSS: tuning section selection, RMA: rating method analysis, CI: code instrumentation, DG: driver generation, PT: performance tuning, FVG: final version generation.) The most time-consuming steps are PT, TSS and RMA.

(TSS) stage. This is because Step (1) does a profile run, which in some cases (e.g., *wupwise*) takes more time than a normal run of the program. The last large portion of the tuning time is spent in Step (2), the rating method analysis (RMA) stage. Some of the time is spent in data flow analysis; some is spent in the profile

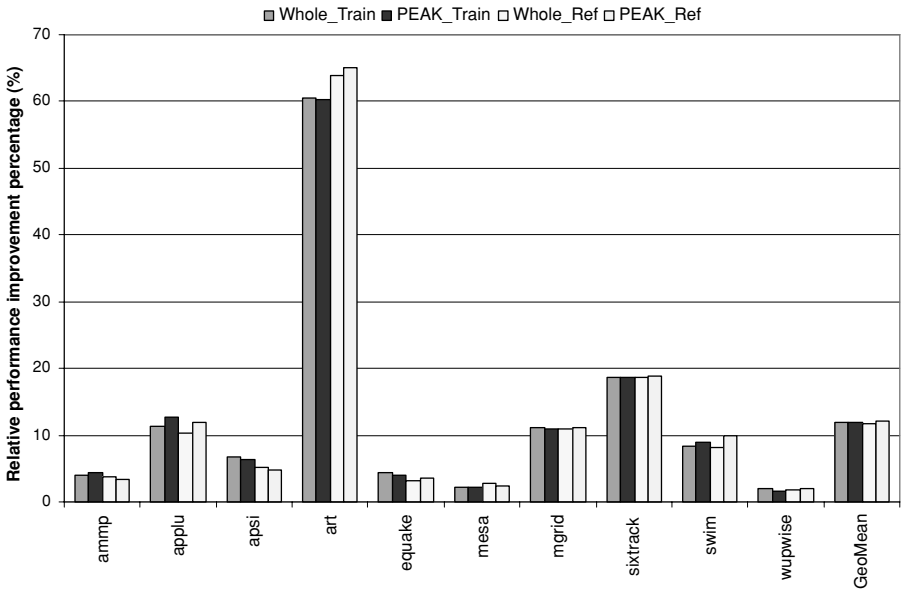


Fig. 19. Program performance improvement relative to the baseline under O3 for SPEC CPU2000 FP benchmarks on Pentium 4. Higher is better. All the benchmarks use the *train* dataset as the input to the tuning process. Whole_Train (PEAK_Train) is the performance achieved by the whole-program tuning (the PEAK system) under the *train* dataset. Whole_Ref and PEAK_Ref use the *ref* dataset to evaluate the tuned program performance, but still the *train* dataset for tuning. PEAK achieves equal or better program performance than whole-program tuning.

run to get the context parameters for CBR and execution model parameters for MBR. For programs with a large code size (e.g., *ammp* and *mesa*), compiling the source code into the internal representation of our source-to-source compiler also accounts for a big portion of the tuning time. (This compilation time comes from the compiler infrastructure we use for the implementation of our PEAK compiler.)

The results on SPARC II are similar. The normalized tuning time is reduced from 63.42 to 4.88, with a tuning speedup of 13.0. The absolute tuning time is reduced from 9.83 hours to 43.7 minutes. (Our SPARC II machine is slower than Our Pentium 4 machine.)

Figure 19 shows the program performance achieved by the whole-program tuning and the PEAK system on Pentium 4. We use the *train* dataset as the input to the tuning process.

The first two bars show the performance of the final tuned version under the same *train* dataset for the whole-program tuning and the PEAK system. For all the benchmarks, PEAK achieves equal or better performance. PEAK outperforms whole-program tuning by 1.5% on *applu*. Some benchmarks, such as *equake* and *mesa*, have only one tuning section. Some benchmarks, such as *art*, *swim* and *mgrid*, have similar code structure in the selected tuning sections, which leads to the fact that the tuning sections favor similar optimizations. The above two observations explain why PEAK does not outperform the whole-program tuning significantly in terms of tuned program performance.

A fair performance evaluation should use an input different from the training input. To this aim, we use the *ref* dataset to evaluate the performance of the tuned version. (Still, the *train* dataset is the input to the tuning process.) The results are shown by the last two bars. We can see that using different inputs still achieves similar performance to the first two bars. So, our tuning scenario does find an optimal combination of the compiler optimizations, which performs much better than the default optimization configuration.

On average, PEAK improves the performance by 12.0% and 12.1% with respect to the *train* dataset and the *ref* dataset, while the whole-program tuning by 11.9% and 11.7%. The results on SPARC II are similar: PEAK improves the performance by 4.1% and 3.7% with respect to *train* and *ref*; while the whole-program tuning by 4.1% and 3.7% as well. So, PEAK achieves equal or better program performance than the whole-program tuning.

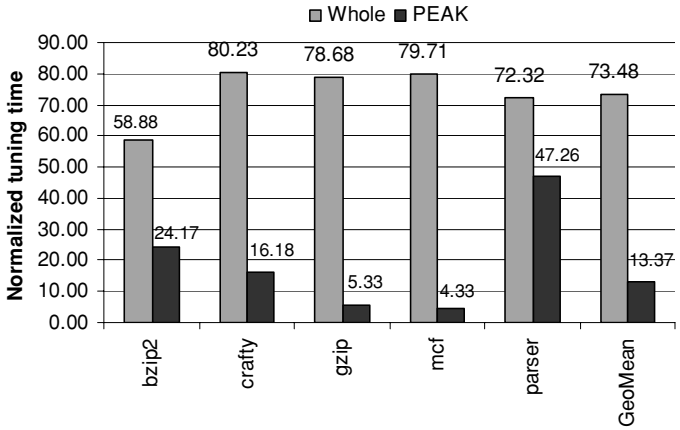
6.3.4 *SPEC CPU2000 INT Benchmarks.* Integer benchmarks generally have irregular code structures with many conditional statements. Moreover, the use of pointers complicates rating-method analysis. As a result, only re-execution-based rating could be applicable. On the other hand, the selected tuning sections in some benchmarks use library functions with side effects, for example, the memory allocation functions. For these benchmarks, our current implementation fails tuning, unable to roll back the execution of the tuning section.

To illustrate the performance of PEAK on INT benchmarks, we experiment with five benchmarks, *bzip2*, *crafty*, *gzip*, *mcf* and *parser*, which do not use the functions with side effects in their tuning sections.

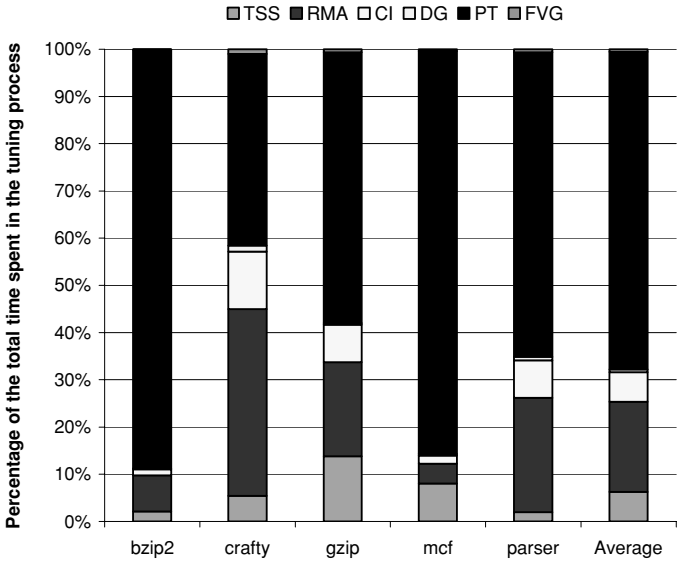
Figure 20(a) shows the normalized tuning time. PEAK speeds up the tuning process for INT benchmarks as well. On average, the normalized tuning time is reduced from 73.48 to 13.37, and the absolute tuning time is reduced from 71.64 minutes to 14.67 minutes. *bzip2* does not gain a high speedup due to the small N_{min} of 22. Besides, there are three major reasons for the fact that the speedup for INT benchmarks is not as high as the one for FP benchmarks: (1) Due to the irregularity of the code, INT benchmarks can only use re-execution-based rating, which introduces more overhead than the other two rating methods used in FP benchmarks. (2) PEAK compiler spends more time in analyzing INT benchmarks, which generally have more code, especially in *crafty* and *parser*. (3) Unlike FP benchmarks, the tuning sections in INT benchmarks generally do not have a similar code structure. Different tuning sections in INT benchmarks may favor different optimizations, which leads to more experimental versions, especially in *parser*.

Figure 20(b) shows the tuning time percentages of the six tuning steps. The most time-consuming components are performance tuning, rating method analysis, and tuning section selection. Due to the larger source code size, rating method analysis and code instrumentation spend more time on INT benchmarks than on FP benchmarks.

Figure 21 shows the tuned program performance. On average, PEAK improves the performance by 4.6% and 4.2% with respect to the *train* dataset and the *ref* dataset, while the whole-program tuning improves the performance by



(a) Normalized tuning time of the whole-program tuning and the PEAK system for SPEC CPU2000 INT benchmarks on Pentium 4. Lower is better. On average, PEAK gains a speedup of 5.5.



(b) Tuning time percentage of the six stages for SPEC CPU2000 INT benchmarks on Pentium 4. (TSS: tuning section selection, RMA: rating method analysis, CI: code instrumentation, DG: driver generation, PT: performance tuning, FVG: final version generation.) The most time-consuming steps are PT, RMA and TSS.

Fig. 20. PEAK tuning time for INT benchmarks on Pentium 4.

4.4% and 4.2% respectively. PEAK achieves less performance for *gzip* than the whole-program tuning, because of the low program coverage of 84%. For *parser*, PEAK achieves much better performance than the whole-program tuning.

7. RELATED WORK

Our work is motivated by the observation that, in some situations, compiler optimizations may degrade performance. Attempts to avoid such degradations

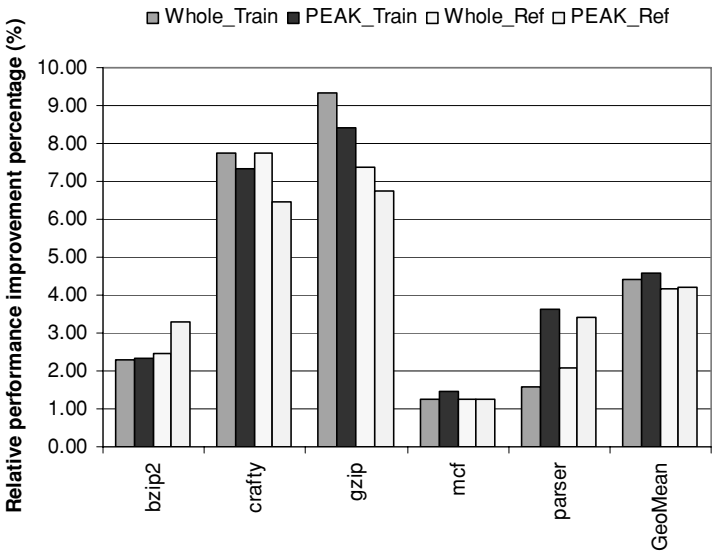


Fig. 21. Program performance improvement relative to the baseline under O3 for SPEC CPU2000 INT benchmarks on Pentium 4. Higher is better. All the benchmarks use the *train* dataset as the input to the tuning process. Whole_Train (PEAK_Train) is the performance achieved by the whole-program tuning (the PEAK system) under the *train* dataset. Whole_Ref and PEAK_Ref use the *ref* dataset to evaluate the tuned program performance, but still the *train* dataset for tuning.

at their source are important related work. Doing so is difficult, as compile-time performance models are intrinsically limited by the knowledge available about input data and characteristics of the execution platform. Even more difficult is to model optimization interactions, which can be subtle and unexpected. In addition to compile-time methods, some work has made use of runtime techniques.

One attempt to alleviate the problem of optimization orchestration is to improve compiler optimizations so as to reduce their potential performance degradation. However, this approach has two problems: (1) Compilers can hardly consider the interaction between all the optimizations, given the large number of optimizations and the complexity of their interactions; (2) The compile-time performance models used in the compiler are limited by the unavailability of program input data and insufficient knowledge of the target architecture. Many projects try to improve optimization performance in these two aspects, using either compile-time or runtime techniques.

Some try to combine a few optimizations into one big pass, which considers the interactions between the optimizations. For example, Wolf et al. [1996] develop an algorithm that applies fission, fusion, tiling, permutation and outer loop unrolling to optimize loop nests. Similarly, Click and Cooper [1995] show that combining constant propagation, global value numbering, and dead code elimination leads to more optimization opportunities. Nevertheless, it would be very difficult, or even impossible, to combine all optimizations into one pass that removes all possible performance degradation, because of the complexity of the compilation task and the subtle interactions between optimizations.

Some postpone optimizations until runtime, when accurate knowledge about the target architecture and the program input can be supplied to the compiler:

(1) Similar to JVM JIT compilers [Adl-Tabatabai et al. 1998; Arnold et al. 2000, 2002; Cierniak et al. 2000], several projects aim at achieving both portability and performance. DCG [Engler and Proebsting 1994] proposes a retargetable dynamic code generation system; VCODE [Engler 1996] provides a machine-independent interface for native machine code generation; DAISY [Ebcioglu and Altman 1997] generates VLIW code on-the-fly to emulate the existing architectures on a VLIW architecture.
(2) Several projects generate optimized code using runtime input via Runtime Specialization [Consel and Noël 1996]. RCG and Fabius [Lee and Leone 1996; Leone and Lee 1998] automatically translates ML programs into code that generates native binary code at runtime. Calpa [Mock et al. 2000] and DyC [Grant et al. 1999] form a staged compiler [Auslander et al. 1996]: Calpa annotates the program at compile-time; DyC creates a runtime compiler from the annotated program; this runtime compiler in turn generates the executable using runtime values.
(3) Some try to re-optimize binaries based on runtime information. Dynamo [Bala et al. 2000; Duesterwald and Bala 2000] (for HP workstations) and DynamoRIO [Bruening et al. 2003] (for IA-32 machines) find and optimize hot traces for statically generated native binaries at runtime. Similarly, in Merten et al. [1999, 2001] and Nystrom et al. [2001], techniques are developed to detect hot spots and to generate new traces for runtime optimization via hardware support. ADAPT [Voss and Eigenmann 2000, 2001] compiles code intervals under different optimization configurations at runtime, and chooses the best version for each interval. Continuous Program Optimization [Kistler and Franz 2003] continually adjusts the storage layouts of dynamic data structures to enhance the data locality and re-schedules the instructions based on runtime profiling.
(4) Besides the above runtime compilation systems, some develop specific runtime optimization techniques. For example, a runtime data and iteration reordering technique is applied to improve data locality in Strout et al. [2003]. LRPD test [Rauchwerger and Padua 1999; Dang et al. 2002] speculatively executes candidate loops in a parallel form, and re-executes the loops serially when some data dependence is detected at runtime. Rus et al. [2002] analyze the memory reference at both compile-time and runtime to help parallelization. Runtime path profiling is proposed to help compiler optimizations in Nandy et al. [2003]. Dynamic Feedback [Diniz and Rinard 1997] produces several versions under different synchronization optimization policies and automatically chooses the best version by periodically sampling the performance of each version.

The above techniques defer the optimizations to runtime; hence they need to reduce or amortize additional overhead introduced to program execution. In some systems, the baseline performance is much worse than the optimized code, therefore, there is an opportunity to amortize the overhead by performance

improvement from the optimizations. For example, Fabius [Lee and Leone 1996; Leone and Lee 1998] improves ML code, JIT [Adl-Tabatabai et al. 1998; Arnold et al. 2000, 2002; Cierniak et al. 2000] improves byte code, and Dynamo [Bala et al. 2000; Bruening et al. 2003] improves un-optimized library code. Some [Voss and Eigenmann 2000, 2001] off-load the compilation job to another processor. Some [Consel and Noël 1996; Lee and Leone 1996; Leone and Lee 1998; Mock et al. 2000; Grant et al. 1999; Auslander et al. 1996] generate a small code generator before the production run, so as to reduce the compilation overhead at runtime. Others [Merten et al. 1999, 2001; Nystrom et al. 2001] use hardware to reduce the profiling overhead.

The major goal of this article is to find the best compiler optimization combination for scientific programs. These programs are mostly written in C or Fortran, and the baseline is compiled efficiently under the default optimization setting. By contrast, a large number of optimization options, 38 GCC flags in our experiments, are involved in advanced optimizations. As the time for navigating this space is substantial, our work takes an offline optimization approach. Similar to profile-based optimizations, this article tunes program performance under a training input, creating an optimized, final version, which will be used at runtime. The tuning process can also be applied in-between the production runs.

This article takes the commonly used *feedback-directed optimization* approach. In this approach, many different binary code versions generated under different *experimental optimization combinations* are 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. In the end, the optimization orchestration algorithm gives the final optimal version for the entire program or important code sections in the program.

Many performance tuning systems use this feedback-directed approach. For example, ATLAS [Whaley and Dongarra 1998] generates numerous variants of matrix multiplication to search for the best one for a specific target machine. Similarly, Iterative Compilation [Kisuki et al. 1999] searches through the transformation space to find the best block sizes and unrolling factors. Meta optimization [Stephenson et al. 2003] uses machine-learning techniques to adjust several compiler heuristics automatically.

The above three projects [Whaley and Dongarra 1998; Kisuki et al. 1999; Stephenson et al. 2003] have focused on a relatively small number of optimization techniques, while this paper tunes all optimizations that are controlled by compiler options (e.g., all 38 GCC O3 optimization options in our experiments).

Several projects target the same optimization orchestration problem as this paper. Random search [Haneda et al. 2005] and machine learning [Agakov et al. 2006; Cavazos and O'Boyle 2006] are introduced to solve this problem. The Optimization-Space Exploration (OSE) compiler [Triantafyllis et al. 2003] defines sets of optimization configurations and an exploration space, which is traversed to find the best configuration for the program using compile-time performance estimates as feedback. Statistical Selection (SS) in Pinkers et al.

[2004] uses orthogonal arrays [Hedayat et al. 1999] to compute the performance effects of the optimizations based on a statistical analysis of profile information, which, in turn, is used to find the best optimization combination. Compiler Optimization Selection [Chow and Wu 1999] applies fractional factorial design to optimize the selection of compiler options. Option Recommendation [Granston and Holler 2001] chooses the PA-RISC compiler options intelligently for an application, using heuristics based on information from the user, the compiler and the profiler. (Different from finding the best optimization combination, Adaptive Optimizing Compiler [Cooper et al. 2002] uses a biased random search to discover the best order of optimizations.[4] In Kulkarni et al. [2004, 2006], a genetic algorithm is used to solve the above problem. Genetic algorithms are powerful but are converging slowly; two approaches are proposed to make the genetic algorithm faster by avoiding unnecessary execution and reducing the number of generations.)

Two major issues remain for optimization orchestration.

(1) How do we search through the optimization space fast and effectively, given that the search space is huge due to the interactions between compiler optimizations? Complex algorithms, such as the exhaustive search in ATLAS [Whaley and Dongarra 1998] and the automatic theorem prover in Denali [Joshi et al. 2002], would be prohibitively slow to solve this problem for a real application. Also, while heuristics for special environments can be found [Granston and Holler 2001], this article looks for a more general solution.

(2) How do we evaluate the optimized versions fast and accurately? The most accurate method is to use the real execution time to evaluate the performance of the experimental versions. However, given the large number of experimental versions and the execution time of a real application, this method takes excessively long. Attempts to develop performance models that reduce this time [Triantafyllis et al. 2003] came at the cost of significantly less program performance. In Cooper et al. [2005], a "virtual execution" technique is introduced to derive instruction counts from branch profile information. This method is fast but not as accurate as using real execution times, because instruction counts cannot fully represent machine cycles. For example, this method can hardly reflect the effectiveness of code scheduling. Moreover, this method may fail when the control flow of a program is significantly changed. In Lau et al. [2006], a system similar to our PEAK is implemented for JVM. It simply uses average execution time to evaluate performance, hoping that with a large number of samples the evaluation may be accurate.

The goal of our PEAK system is to tune the performance of the important code sections in a program, in a fast and effective way, via orchestrating the optimizations controlled by compiler options. PEAK is an automated system,

[4]Usually, a compiler does not have the option to specify the order of the optimizations. So, this article does not compare to Cooper et al. [2002], although the techniques developed in this article can be extended to search for the best order of optimizations.

targeting at the above two issues. First, a fast and effective feedback-directed algorithm is designed to search through the optimization space, considering the interactions between the optimizations. The effectiveness of our algorithm stems from its ability to navigate the search space rapidly where optimizations do not interact, but consider interactions where they exist. Second, PEAK uses fast and accurate performance evaluation methods based on a partial execution of the program.

8. CONCLUSIONS

The techniques developed in this paper have led to the creation of an automated performance tuning system called PEAK. PEAK searches for the best compiler optimization combinations for important tuning sections in a program, using a fast and effective optimization orchestration algorithm – combined elimination. Three fast and accurate rating methods – CBR, MBR and RBR – are developed to evaluate the performance of the optimized versions based on a partial execution of the program. The PEAK compiler selects the important code sections for performance tuning, analyzes the source program for applicable rating methods, and instruments the source code to construct a tuning driver. The tuning driver adopts a feedback-directed tuning approach. It continually runs the program, loads experimental versions generated under different optimization combinations, rates these versions based on the execution times, and explores the optimization space according to the generated ratings, until the best optimization combination is found for each tuning section. In addition to the above functionalities, the PEAK runtime system provides the facilities to dynamically load executables at runtime.

PEAK achieves fast tuning speed and high tuned program performance. When our Combined Elimination (CE) algorithm is applied to the whole program, the program performance is improved by 12% over GCC O3 for SPEC CPU2000 FP benchmarks and 4% for INT benchmarks. Using SUN Forte compilers, CE improves performance by 10.8%, compared to 5.6% improved by manual tuning for FP benchmarks, and 8.1% to 4.1% for INT benchmarks. After applying the rating methods to individual tuning sections, PEAK reduces tuning time from 2 hours to 4.9 minutes for FP benchmarks, and from 1.2 hours to 13 minutes for selected INT benchmarks, while achieving equal or better program performance.

REFERENCES

ADL-TABATABAI, A.-R., CIERNIAK, M., LUEH, G.-Y., PARIKH, V. M., AND STICHNOTH, J. M. 1998. Fast, effective code generation in a just-in-time java compiler. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*. ACM, New York, 280–290.

AGAKOV, F., BONILLA, E., CAVAZOS, J., FRANKE, B., FURSIN, G., O'BOYLE, M. F. P., THOMSON, J., TOUSSAINT, M., AND WILLIAMS, C. K. I. 2006. Using machine learning to focus iterative optimization. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization* (Washington, DC). IEEE Computer Society Press, Los Alamitos, CA, 295–305.

ARNOLD, M., FINK, S., GROVE, D., HIND, M., AND SWEENEY, P. F. 2000. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, New York, 47–65.

ARNOLD, M., HIND, M., AND RYDER, B. G. 2002. Online feedback-directed optimization of java. In *Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, New York, 111–129.

AUSLANDER, J., PHILIPOSE, M., CHAMBERS, C., EGGERS, S. J., AND BERSHAD, B. N. 1996. Fast, effective dynamic compilation. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, 149–159.

BALA, V., DUESTERWALD, E., AND BANERJIA, S. 2000. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*. ACM, New York, 1–12.

BRUENING, D., GARNETT, T., AND AMARASINGHE, S. 2003. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Computer Generation and Optimization (CGO '03)*. IEEE Computer Society Press, Los Alamitos, CA.

CAVAZOS, J. AND O'BOYLE, M. F. P. 2006. Method-specific dynamic compilation using logistic regression. In *OOPSLA '06: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM, New York, 229–240.

CHOW, K. AND WU, Y. 1999. Feedback-directed selection and characterization of compiler optimizations. In *Proceedings of the 2nd Workshop on Feedback Directed Optimizations*. Israel.

CIERNIAK, M., LUEH, G.-Y., AND STICHNOTH, J. M. 2000. Practicing JUDO: Java under dynamic optimizations. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*. ACM, New York, 13–26.

CLICK, C. AND COOPER, K. D. 1995. Combining analyses, combining optimizations. *ACM Trans. Prog. Lang. Syst. (TOPLAS) 17*, 2, 181–196.

CONSEL, C. AND NOËL, F. 1996. A general approach for run-time specialization and its application to c. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, 145–156.

COOPER, K. D., GROSUL, A., HARVEY, T. J., REEVES, S., SUBRAMANIAN, D., TORCZON, L., AND WATERMAN, T. 2005. Acme: adaptive compilation made efficient. In *LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. ACM, New York, 69–77.

COOPER, K. D., SUBRAMANIAN, D., AND TORCZON, L. 2002. Adaptive optimizing compilers for the 21st century. *J. Supercomput. 23*, 1, 7–22.

DANG, F., YU, H., AND RAUCHWERGER, L. 2002. The R-LRPD test: Speculative parallelization of partially parallel loops. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS '02)*. IEEE Computer Society Press, Los Alamitos, CA.

DINIZ, P. C. AND RINARD, M. C. 1997. Dynamic feedback: An effective technique for adaptive computing. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, 71–84.

DUESTERWALD, E. AND BALA, V. 2000. Software profiling for hot path prediction: Less is more. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, 202–211.

EBCIOGLU, K. AND ALTMAN, E. R. 1997. DAISY: Dynamic compilation for 100. In *Proceedings of the Annual International Symposium on Computer Architecture*. 26–37.

ENGLER, D. R. 1996. VCODE: a retargetable, extensible, very fast dynamic code generation system. *SIGPLAN Notes 31*, 5, 160–170.

ENGLER, D. R. AND PROEBSTING, T. A. 1994. DCG: An efficient, retargetable dynamic code generation system. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, 263–272.

GNU. 2005. *GCC online documentation*. http://gcc.gnu.org/onlinedocs/.

GRAHAM, S. L., KESSLER, P. B., AND MCKUSICK, M. K. 1982. GPROF: A call graph execution profiler. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*. ACM, New York, 120–126.

GRANSTON, E. D. AND HOLLER, A. 2001. Automatic recommendation of compiler options. In *Proceedings of the 4th Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*.

GRANT, B., PHILIPOSE, M., MOCK, M., CHAMBERS, C., AND EGGERS, S. J. 1999. An evaluation of staged run-time optimizations in dyc. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*. ACM, New York, 293–304.

HANEDA, M., KNIJNENBURG, P. M. W., AND WIJSHOFF, H. A. G. 2005. Generating new general compiler optimization settings. In *ICS '05: Proceedings of the 19th Annual International Conference on Supercomputing*. ACM, New York, 161–168.

HEDAYAT, A., SLOANE, N., AND STUFKEN, J. 1999. *Orthogonal Arrays: Theory and Applications*. Springer-Verlag, New York.

JOSHI, R., NELSON, G., AND RANDALL, K. 2002. Denali: A goal-directed superoptimizer. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*. ACM, New York, 304–314.

KARP, R. 1972. Reducibility among combinatorial problems. In *Proceedings of the Symposium on the Complexity of Computer Computations*. Plenum Press, New York, 85–103.

KISTLER, T. AND FRANZ, M. 2003. Continuous program optimization: A case study. *ACM Trans. Program. Lang. Syst. 25*, 4, 500–548.

KISUKI, T., KNIJNENBURG, P. M. W., AND O'BOYLE, M. F. P. 2000. Combined selection of tile sizes and unroll factors using iterative compilation. In *Proceedings of the IEEE PACT*. IEEE Computer Society Press, Los Alamitos, CA, 237–248.

KISUKI, T., KNIJNENBURG, P. M. W., O'BOYLE, M. F. P., BODIN, F., AND WIJSHOFF, H. A. G. 1999. A feasibility study in iterative compilation. In *Proceedings of the International Symposium on High Performance Computing (ISHPC'99)*. 121–132.

KULKARNI, P., HINES, S., HISER, J., WHALLEY, D., DAVIDSON, J., AND JONES, D. 2004. Fast searches for effective optimization phase sequences. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*. ACM, New York, 171–182.

KULKARNI, P. A., WHALLEY, D. B., TYSON, G. S., AND DAVIDSON, J. W. 2006. Exhaustive optimization phase order space exploration. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization* (Washington, DC). IEEE Computer Society Press, Los Alamitos, CA, 306–318.

LAU, J., ARNOLD, M., HIND, M., AND CALDER, B. 2006. Online performance auditing: Using hot optimizations without getting burned. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, New York.

LEE, P. AND LEONE, M. 1996. Optimizing ML with run-time code generation. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, 137–148.

LEONE, M. AND LEE, P. 1998. Dynamic specialization in the fabius system. *ACM Comput. Surv. 30*, 3es, 23.

MERTEN, M. C., TRICK, A. R., BARNES, R. D., NYSTROM, E. M., GEORGE, C. N., GYLLENHAAL, J. C., AND MEI W. HWU, W. 2001. An architectural framework for runtime optimization. *IEEE Trans. Comput. 50*, 6, 567–589.

MERTEN, M. C., TRICK, A. R., GEORGE, C. N., GYLLENHAAL, J. C., AND HWU, W. W. 1999. A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*. IEEE Computer Society Press, Los Alamitos, CA, 136–147.

MOCK, M., CHAMBERS, C., AND EGGERS, S. J. 2000. CALPA: A tool for automating selective dynamic compilation. In *Proceedings of the International Symposium on Microarchitecture*. 291–302.

NANDY, S., GAO, X., AND FERRANTE, J. 2003. TFP: Time-sensitive, flow-specific profiling at runtime. In *Proceedings of the Workshop on Languages and Compiling for Parallel Computing (LCPC)*.

NYSTROM, E. M., BARNES, R. D., MERTEN, M. C., AND MEI W. HWU, W. 2001. Code reordering and speculation support for dynamic optimization systems. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*.

PAN, Z. AND EIGENMANN, R. 2004a. Compiler optimization orchestration for peak performance. Tech. Rep. TR-ECE-04-01, School of Electrical and Computer Engineering, Purdue University.

PAN, Z. AND EIGENMANN, R. 2004b. Rating compiler optimizations for automatic performance tuning. In *SC2004: Proceedings of the High Performance Computing, Networking and Storage Conference*. (10 pages).

PAN, Z. AND EIGENMANN, R. 2006. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *Proceedings of the 4th Annual International Symposium on Code Generation and Optimization (CGO)*. (12 pages).

PINKERS, R. P. J., KNIJNENBURG, P. M. W., HANEDA, M., AND WIJSHOFF, H. A. G. 2004. Statistical selection of compiler options. In *Proceedings of the IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'04)* (Volendam, The Netherlands). IEEE Computer Society Press, Los Alamitos, CA, 494–501.

RAUCHWERGER, L. AND PADUA, D. A. 1999. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Trans. Para. Distrib. Syst. 10*, 2 (Feb.), 160–180.

RUS, S., RAUCHWERGER, L., AND HOEFLINGER, J. 2002. Hybrid analysis: Static & dynamic memory reference analysis. In *Proceedings of the 16th International Conference on Supercomputing*. ACM, New York, 274–284.

SPEC. 2000. *SPEC CPU2000 Results*. http://www.spec.org/cpu2000/results.

STEPHENSON, M., AMARASINGHE, S., MARTIN, M., AND O'REILLY, U.-M. 2003. Meta optimization: improving compiler heuristics with machine learning. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. ACM, New York, 77–90.

STROUT, M. M., CARTER, L., AND FERRANTE, J. 2003. Compile-time composition of run-time data and iteration reorderings. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, New York.

SUN. 2000. *Forte C 6 /Sun WorkShop 6 Compilers C User's Guide*. http://docs.sun.com/app/docs/doc/806-3567.

TRIANTAFYLLIS, S., VACHHARAJANI, M., VACHHARAJANI, N., AND AUGUST, D. I. 2003. Compiler optimization-space exploration. In *Proceedings of the International Symposium on Code Generation and Optimization*. 204–215.

VOSS, M. AND EIGENMANN, R. 2000. ADAPT: Automated de-coupled adaptive program transformation. In *Proceedings of the International Conference on Parallel Processing*. IEEE Computer Society Press, Los Alamitos, CA, 163–170.

VOSS, M. J. AND EIGEMANN, R. 2001. High-level adaptive program optimization with ADAPT. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*. ACM, New York, 93–102.

WHALEY, R. C. AND DONGARRA, J. 1998. Automatically tuned linear algebra software. In *Proceedings of the SuperComputing 1998: High Performance Networking and Computing*.

WOLF, M. E., MAYDAN, D. E., AND CHEN, D.-K. 1996. Combining loop transformations considering caches and scheduling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*. ACM, New York, 274–286.

Received May 2006; revised November 2006; accepted May 2007