Fast, Automatic, Procedure-Level Performance Tuning *

Zhelong Pan
Purdue University, School of ECE
West Lafayette, IN, 47907
zpan@purdue.edu

Rudolf Eigenmann
Purdue University, School of ECE
West Lafayette, IN, 47907
eigenman@purdue.edu

ABSTRACT

This paper presents an automated performance tuning solution, which partitions a program into a number of *tuning sections* and finds the best combination of compiler options for each section. Our solution builds on prior work on feedback-driven optimization, which tuned the whole program, instead of each section. Our key novel algorithm partitions a program into appropriate tuning sections. We also present the architecture of a system that automates the tuning process; it includes several pre-tuning steps that partition and instrument the program, followed by the actual tuning and the post-tuning assembly of the individually-optimized parts. Our system, called PEAK, achieves fast tuning speed by measuring a small number of invocations of each code section, instead of the whole-program execution time, as in common solutions. Compared to these solutions PEAK reduces tuning time from 2.19 hours to 5.85 minutes on average, while achieving similar program performance. PEAK improves the performance of SPEC CPU2000 FP benchmarks by 12% on average over GCC O3, the highest optimization level, on a Pentium IV machine.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Code Generation, Compilers, Optimization, Run-time Environments*

General Terms

Performance

Keywords

Performance Tuning, Optimization Orchestration, Dynamic Compilation

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

PACT'06, September 16–20, 2006, Seattle, Washington, USA.

1. INTRODUCTION

Compiler optimizations yield significant performance improvements in many programs on modern architectures. However, potential performance degradation in certain program patterns is a well-known phenomenon. Programmers are expected to deal with this problem through compiler options – when finding that an option causes performance degradation, one may switch it off. The presence of many optimization options reflects the inability of today's compilers to make the best optimization decision at compile time. In this paper, we refer to the problem 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 make this problem difficult to solve.

Several optimization orchestration algorithms have been developed [1, 4, 8, 9, 11]. (They will be discussed in Section 6.) A *feedback-directed* approach is used in many of these algorithms. They generate a series of *experimental code versions*, compiled under different optimization combinations. The performance of each experimental version is then *rated* based on execution time or estimated metrics. Using these performance ratings, the algorithms iteratively choose the next experimental optimization combinations, until some convergence criterion is satisfied. This paper makes use of the Combined Elimination (CE) algorithm [8], which is fast and accurate.

Common methods use the overall program execution as a basis for performance measurements. As typically several hundreds of executions are needed for tuning a program, tuning times amount to several hours for programs that execute in minutes, such as the SPEC CPU2000 benchmarks. The present work aims to reduce the tuning time by an order of magnitude. The key observation is that, in the course of a program execution, the same subroutines are often invoked many times; these invocations can serve as the basis for performance measurements, enabling multiple optimized variants to be rated in one run of the program. When doing so, an important challenge must be addressed: the invocations may happen under different contexts (e.g., different values of subroutine parameters). Naively comparing subroutine execution times would lead to incorrect ratings of the optimized variants. Our solution makes use of three rating methods [7] that have been designed for fair comparisons.

Building on the described techniques, our intended solution must answer two key questions: (1) How do we partition a program into appropriate sections for performance

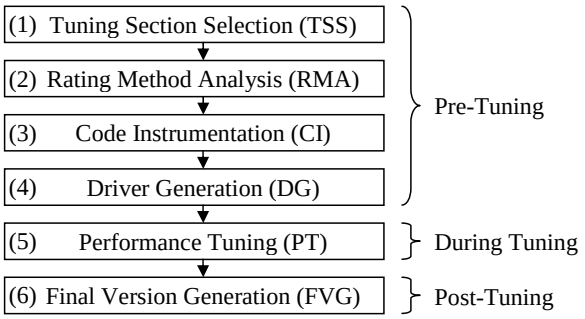

Figure 1: Work flow of the PEAK system

tuning? (2) How do we integrate the existing and new techniques into a system architecture for automatic performance tuning? Our *PEAK* (Program Evolution by Adaptive Compilation) system provides such an architecture.

Figure 1 shows the work flow of PEAK. Several pre-tuning steps must be performed. The program is partitioned into tuning sections, based on a call-graph profile. Next, our compiler tools analyze and instrument the program to employ the appropriate rating methods. The instrumented program forms a tuning driver. During tuning, the driver iteratively runs the program, under a training input data set, until optimization orchestration finishes for all the tuning sections. The feedback-directed CE algorithm is applied in this process. After tuning, the final program is assembled from the individually optimized sections.

This paper makes the following contributions:

1. An algorithm is presented to partition a program into code segments for individual performance tuning. The objectives of the algorithm are to choose tuning sections such that they (1) cover most of (typically more than 90% of) the total program execution time and (2) are invoked many times (typically several hundreds) in one run of the program.

2. An architecture is presented that integrates the partitioning algorithm and other building blocks for program tuning into a system for automated optimization orchestration. The architecture is realized in our PEAK system, which includes a compile-time and runtime part, performing the necessary pre-tuning, actual tuning, and post-tuning steps.

3. Results are presented for the SPEC CPU2000 FP benchmarks and all options included in GCC's O3 optimization level – the highest optimization level – on a 2.8 GHz Pentium IV platform. Compared to whole-program performance tuning, our solution reduces tuning time from 2.19 hours to 5.85 minutes on average, while achieving similar program performance. Performance improves by 12% over O3 for SPEC CPU2000 FP benchmarks.

The remainder of this paper is organized as follows. Section 2 shows background on the CE algorithm and the performance rating methods. Section 3 develops an algorithm for selecting tuning sections. Section 4 shows the design of our PEAK system. Section 5 evaluates PEAK in terms of tuning time and tuned program performance. Section 6 discusses related work.

2. BACKGROUND

2.1 The Combined Elimination (CE) Algorithm

Similar to related approaches [1, 7, 9], the CE algorithm [8] tunes compiler options with the following goal:

Given a set of compiler optimization options $\{F_1, F_2, ..., F_n\}$*, find the combination that minimizes the program execution time. Each option* F_i *has two possible values:* $F_i = 0$ *means* F_i *is off;* $F_i = 1$ *means on.*[1]

CE turns on all the optimizations for the baseline performance B. The performance effect of one optimization, F_i, relative to B, can be represented by its *Relative Improvement Percentage (RIP)*, $RIP_B(F_i = 0)$, which is the relative performance difference of the baseline and the version with F_i turned off.

$$RIP_B(F_i = 0) = \frac{T(F_i = 0) - T_B}{T_B} \times 100\% \qquad (1)$$

In Equation 1, if the orchestration algorithm is applied at the whole-program level, $T(F_i = 0)$ is the execution time of the whole program with F_i turned off; T_B is the baseline execution time. Our PEAK system applies the CE algorithm at the tuning-section level. In PEAK, $T(F_i = 0)$ is the *rating* generated based on the execution times of a few invocations to the tuning-section version with F_i turned off; T_B is the baseline rating. (rating methods are discussed in Section 2.2.)

A simple algorithm would identify all the optimizations with negative effects and turn them off. While such a method is fast, it does not suffice, as it ignores possible interaction of optimizations. A better algorithm starts from the baseline and iteratively turns off the optimization with the most negative effect, one at a time. In this way, it considers interactions; however, it would be slow.

CE combines these two ideas. It iteratively turns off the optimization with the most negative effect. Moreover, in each iteration, after identifying the optimizations with negative effects, CE tries to eliminate these optimizations one by one in a greedy fashion. Therefore, when the optimizations interact weakly, CE eliminates the optimizations with negative effects in one iteration; otherwise, CE eliminates them iteratively. As a result, CE achieves both high program performance and fast tuning speed.

2.2 Fast and Accurate Performance Rating

Recall that the new idea in this paper is to apply the optimization orchestration algorithm on the basis of an invocation of a program *section*. In this way, these sections can be tuned in a single, or a few, program executions. Obtaining accurate performance ratings of the optimization variants is difficult because different executions of the same tuning section may have different input parameters – more generally, they execute under different contexts and thus have different workloads.

In our tuning system we will use three methods that have been proposed to compare multiple invocations of a program section in a fair way and to generate accurate ratings [7]. The key ideas are as follows.

[1]All options in GCC are of type "on-off optimizations". The CE algorithm can be used to tune other types of optimizations as well.

2.2.1 Context Based Rating (CBR)

Context-Based Rating (CBR) identifies and compares invocations of a tuning section that have the same workload, in the course of a program run. Our PEAK compiler implements the algorithm presented in [7] to find the *context variables*, which are the input variables that may influence the conditions of the control regions, such as `if` and `loop` constructs. The *context* of one tuning section invocation is determined by the values of all context variables in that invocation. Therefore, each context represents a unique workload.

CBR rates one optimized version under a certain context by using the average execution time of several invocations. The best versions for different contexts may be different, in which case CBR could report the context-specific winners. PEAK makes use of the best version under the most important context, which covers most (e.g. more than 80%) of the execution time spent in the tuning section. This major context is determined by one profile run of the program. The rating of a version v, $R(v)$, is computed according to Equation 2, 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 number of invocations.

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

2.2.2 Model Based Rating (MBR)

If a tuning section has no major context or the number of invocations of the major context is small, the Model-Based Rating (MBR) method is preferred. MBR formulates mathematical relationships between different contexts of a tuning section and adjusts the 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:

$$T_{TS} = \sum (T_b \times C_b) \tag{3}$$

T_{TS} is the execution time in one tuning-section invocation; 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 tuning-section invocation.

The tuning system collects the execution times (T_{TS}) and the numbers of entries (C_b) for a number of invocations. By linear regression, MBR computes the *component times* (T_b). The rating of a version, $R(v)$, is computed based on the T_b's according to Equation 4, where C_{bavg} is the average C_b during one whole run of the program.

$$R(v) = \sum (T_b \times C_{bavg}) \tag{4}$$

2.2.3 Re-execution Based Rating (RBR)

If there are many (e.g., ten) components in the above execution time model, a large number of invocations need to be measured in order to perform an accurate linear regression. MBR would lead to a long tuning time in this case and hence is not applied. Instead, Re-execution-Based Rating (RBR) can be employed. It enables fair comparison by re-executing a tuning section under the same input.

Suppose that the execution times of two versions ($v1$ and $v2$) under the same input are T_{v1} and T_{v2}. Then, the performance of $v2$ relative to $v1$ is $R_{v2/v1}$:

$$R_{v2/v1} = T_{v1}/T_{v2} \tag{5}$$

If $R_{v2/v1}$ is larger than 1, $v2$ performs better than $v1$. Otherwise, $v2$ performs worse. RBR uses the average $R_{v2/v1}$ over a number of invocations as the relative performance rating. In [7], details have been provided about how to re-execute the optimized versions under the same input and how to generate an accurate rating.

2.2.4 Applying the Rating Methods to Our Solution

Before tuning, the original source program is analyzed according to the techniques described in [7]. One profile run yields the major context, if it exists, for CBR, and the execution time model for MBR. From the static analysis and profile information, the applicable rating method is chosen for each tuning section, in the priority order of CBR, MBR and RBR. For the PEAK system, we have developed a set of compiler tools to perform these tasks.

During the actual tuning process, the *rating*, $R(v)$, and the *rating variance*, $Var(v)$, are generated across a number of invocations of the tuning section, which we call a *window* [7]. PEAK's orchestration algorithm uses $R(v)$ as the performance of each version. The window size is determined by $Var(v)$. Basically, the version v is executed and rated until the rating variance $Var(v)$ falls below a threshold. Performing this task is the responsibility of the PEAK runtime system.

3. TUNING SECTION SELECTION

While the techniques presented in Section 2 are important building blocks of our auto-tuning solution, the key open question is how to partition a program into sections that can be tuned effectively. A good algorithm must select tuning sections that cover a large percentage of the program's execution time, so that improvements of the sections translate into overall performance gain. The algorithm must also define the tuning sections in a way that they are invoked many times in the course of a program execution; this enables the tuning algorithm to try many code variants in one run of the program and hence to complete quickly. Furthermore, the algorithm needs to form tuning sections that execute sufficiently long, allowing accurate time measurements.

3.1 Profile Data for Tuning Section Selection

Our algorithm selects a tuning section based on its number of invocations and its execution time. These data are collected from a profile run (using *gprof* [3]) allowing the construction of a call graph $G = (V, E)$; the graph is annotated with execution time spent in a procedure and its descendants, how many times a procedure is called and how many times a procedure calls its children.

The call graph is a directed graph. It has one source (root) node, δ, and a set, Γ, of sink (leaf) nodes. Each node $v \in V$ identifies a procedure. δ identifies subroutine `main()`. Each edge $e \in E$ identifies a procedure call. The associated profile information is as follows.

$$v = \{fn\} \tag{6}$$

$$e = \{s, t, n, tm\} \tag{7}$$

$fn(v)$ is the procedure 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 $s(e)$.

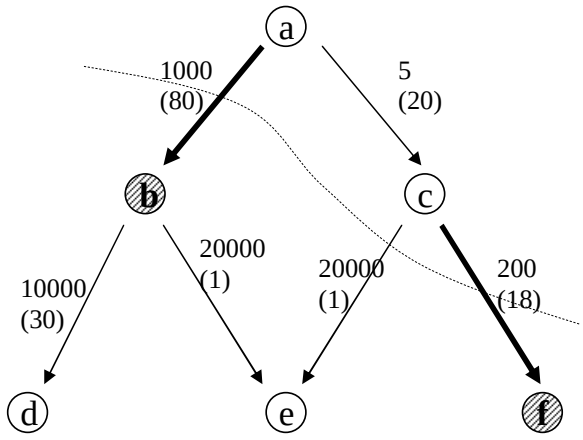

Figure 2: An example of tuning section selection. The graph is a call graph with node a as the main procedure. The weights on each edge are the number of invocations and the execution time, which is in the parentheses. The optimal edge cut is $(\Theta = \{a, c\}, \Omega = \{b, d, e, f\})$, shown by the 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 numbers of invocations of b and f are 1000 and 200 in respect, so, $N_{min} = 200$. $coverage = (80 + 18)/100 = 0.98$, where the total execution time, T_{total}, is 100.

3.2 A Formal Description of the Tuning Section Selection Problem

Tuning section selection aims at finding a set of single entry regions, each of which is a tuning section. The entry procedure in a region identifies the tuning section. All other procedures in the region are the direct or indirect callees. If one procedure is used in two different tuning sections, it is replicated into these two tuning sections.

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. i.e., for each edge $e \in (\Theta, \Omega)$, put e in a set S, if $tm(e)/n(e) \geq T_{lb}$. (2) The edges in set S point to the selected tuning sections. i.e., 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 procedure v.) Figure 2 gives an example.

The tuning section selection algorithm maximizes the coverage and the number of invocations of the tuning sections. These two goals are computed as follows:

1. The primary goal of the tuning section selection algorithm is to maximize the execution time coverage.

$$coverage = \sum_{e \in S} tm(e)/T_{total} \tag{8}$$

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

2. The secondary goal deals with the number of invocations to the tuning section v:

$$N_t(v) = \sum_{e \in S, t(e)=v} n(e). \tag{9}$$

   The goal is to maximize the smallest $N_t(v)$, denoted as N_{min}.

$$N_{min} = min_{v \in T}(N_t(v)) \tag{10}$$

We use a two-step algorithm to maximize the two goals *coverage* and N_{min}. In the first step, the algorithm maximizes the primary goal, *coverage*, under the constraint that the number of invocations to each selected tuning section exceeds a lower bound N_{lb}. In the second step, the algorithm trades off large gains of N_{lb} for small losses in *coverage*; this is done as long as the *coverage drop rate* (the loss/gain ratio) is small and the *coverage* remains above a lower bound. The two steps and chosen thresholds will be described formally in Section 3.4 and Section 3.5, respectively. We briefly discuss the handling of recursive functions in Section 3.3.

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 whole-program tuning. The second solution does not find any tuning section. Neither of them is acceptable. The right solution is to choose 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.

3.3 Dealing with Recursive Functions

Recursive functions cannot be easily partitioned. To call a recursive function, the program makes an *initial call* to the function; then the function will be 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; while 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.

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 call-graph simplification algorithm uses the strongly connected components to find the nodes and edges that appear in a cycle. It merges the nodes involved in a common cycle into

Subroutine
[T, M] = MaxCoverage($G = (V, E)$, N_{lb}, T_{lb}, P_{lb})
Input: $G = (V, E)$, a simplified call graph, which is a DAG; 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 code partitioning.
Output: The algorithm selects the tuning sections and puts their entry procedures into set T. The procedures that are worth extra code partitioning are put into M.

1. Clear the selection flag for each edge $e \in E$: $f(e) = 0$. Empty the result sets: $T = \phi$ and $M = \phi$.
2. 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 be ignored when summing the profile data of the edges for one node.)
3. Sort all the nodes into a topological order: v_1, v_2, ..., i.e., if there is an edge (u, v), node u appears before node v. The nodes will be traversed in this order.
4. For node v_i ($i = 1, 2$, ...), compute the total number of invocations to v_i.
$$n(v_i) = \sum_{e \in E, t(e)=v_i, f(e)=0} n(e) \quad (11)$$
If $n(v_i)$ is greater than N_{lb}, v_i is selected: (1) Put v_i into T; (2) set $f(e) = 1$, if $t(e) = v_i$; (3) update the profile.
5. Put node v into M, if v is not selected but consumes a large amount of execution time, i.e., $\sum_{t(e)=v, f(e)\neq 1} tm(e) > P_{lb} \times T_{total}$. (These nodes may be partitioned to improve tuning section coverage.)

Figure 3: A TS selection algorithm to maximize execution time coverage given a 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 procedures that may be partitioned to improve tuning section coverage.

one node, removes the edges used inside a cycle, adjusts the edges entering or leaving the merged nodes, and updates the associated profile data. After this simplification process, the call graph is a Directed Acyclic Graph (DAG).

3.4 Maximizing Coverage under N_{lb}

Figure 3 describes an algorithm to achieve a large *coverage*, 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 nodes into set T. It finds the nodes that are worth extra code partitioning and puts them into set M. The call graph $G = (V, E)$ is a DAG, simplified by the algorithm described in Section 3.3. Besides N_{lb}, two other parameters are used: (1) T_{lb}, the lower bound on average execution times; (2) P_{lb}, the lower bound on the execution percentage for a code section worth extra partitioning.

T_{lb} is determined by the timing accuracy of the PEAK system. By default, we set $T_{lb} = 100\mu sec$ in our experiments. We set $P_{lb} = 0.02$, meaning that a code section is worth extra partitioning if its execution time is greater than 2% of T_{total}. N_{lb} will be adjusted to trade off the tuning section *coverage* in the final tuning-section selection algorithm described in Section 3.5. The optimal N_{lb} picked by the final algorithm usually ranges from hundreds to thousands.

The max-coverage algorithm in Figure 3 traverses the call graph from top down in a topological order. The procedure calls whose average execution time is less than T_{lb} are ignored. The nodes that are invoked more than N_{lb} times are selected as tuning sections. When a tuning section is selected, the profile data are updated to reflect the execution times and invocation numbers after excluding this tuning section. [2] After the whole selection process finishes, if the remaining execution time on node v is greater than $P_{lb} \times T_{total}$, v is worth extra partitioning.

3.5 The Final TS Selection Algorithm

In order to achieve a large N_{min}, we raise the N_{lb} gradually to tolerate a small drop of *coverage*. Two new parameters are introduced to this 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}, extra 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}.
$$R = \frac{coverage_1 - coverage_2}{N_{min2} - N_{min1}} \quad (12)$$
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 $0.01/100 = 1e^{-4}$.

The final tuning-section selection algorithm iteratively uses the method shown in Figure 3 to maximize *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 $[N_{lb1}, N_{min} - 1]$ 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 1 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 fast tuning.)

[2] To update the profile, we need a context-sensitive inclusive profile. This profile should show the execution time and number of invocations of each procedure call; it should also split this information for each call path. Note that *gprof* does not provide such information. Instead, our algorithm estimates it by evenly distributing the profile data of the selected tuning section to its ancestors and descendants.

Table 1: Tuning section selection for *mgrid*. The best N_{lb} is 400. The optimal *coverage* and N_{min} are 0.957 and 2000.

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

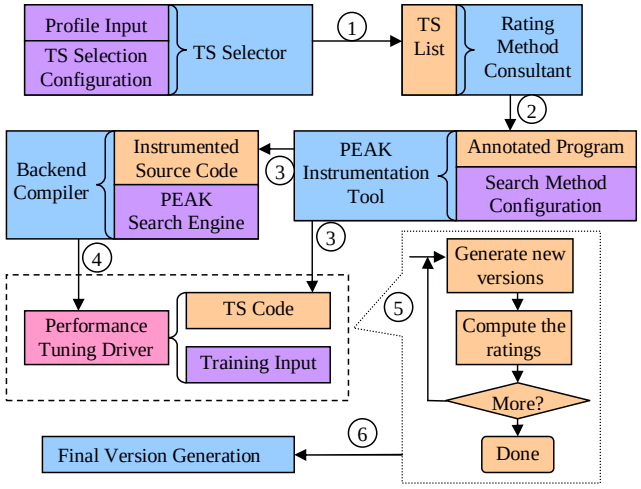

Figure 4: Block diagram of the PEAK performance tuning system. The blocks in the diagram are the components in PEAK and the input and output of these components. Steps 1 to 6 correspond to the ones shown in Figure 1.

4. THE PEAK SYSTEM

PEAK has two major parts: the *PEAK compiler* and the *PEAK runtime* system. The PEAK compiler is used in the pre-tuning steps, while the PEAK runtime system controls the actual tuning process. Figure 4 shows a block diagram of the PEAK system, which lists all the components in PEAK and all the performance tuning steps. Steps 1 to 4 are taken before tuning to construct a tuning driver for a given program. In these steps, the PEAK compiler analyzes and instruments the source code. 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 PEAK runtime system is involved in dynamically generating and loading optimized versions, rating these versions, and feeding new optimization combinations to the tuning driver. 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. Specifically, PEAK takes the following steps.

1. The *tuning section selector* chooses the important code sections as the tuning sections, applying the algorithm described in Section 3.

2. The *rating method consultant* analyzes the source program to find the applicable rating methods for each tuning section. The compiler techniques developed in [7] are implemented here.

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. It inserts 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 system, which is provided in a library format, to construct the *performance tuning driver*. The PEAK runtime system implements the three rating methods developed in [7] and the CE optimization orchestration algorithm developed in [8]. Special functions for dynamically loading the binary code during tuning are also included in the PEAK runtime system.

5. The performance tuning driver iteratively runs the program under a training input until optimization orchestration finishes for all the tuning sections. At each invocation to a tuning section, the driver takes over the 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 version. Here, the main program is the original source program with the tuning sections removed. The final version is the one to be delivered to the end users. This completes the tuning process.

PEAK evaluates the performance of optimized code versions via executing the program under a training input. It generates and loads the binary code during the program execution.

To generate an optimized version for a tuning section separately, excluding other unrelated code, PEAK extracts each tuning section into a separate source file via a source-to-source transformation. This source file includes the entry procedure to the tuning section and all its direct and indirect callees. So, this source file can be compiled and optimized separately.

Callees are included in the tuning section, so, inlining and interprocedural analysis can be performed during code generation, which is important to performance improvement in some cases. If a procedure is called in two tuning sections, this procedure is replicated and renamed in the corresponding source files to avoid name conflicts at link time. Different from cloning [2], such replication in our PEAK system does not have the problem of code explosion, because the number of replicas for a procedure is never more than the number of tuning sections, which is fixed and usually small, around three.

Table 2: 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} = $1\mu sec$.)

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

An important implementation detail is the mechanism for dynamically loading an optimized version. PEAK must pay attention to the global variables used in the tuning sections, especially static variables in C and common blocks in Fortran. Multiple versions of the same tuning section are invoked during one run of the program. Each global variable used in these versions will be located to the same address, when the versions are loaded. We developed a special binary update tool to achieve this effect.

5. EXPERIMENTAL RESULTS

5.1 Results of Tuning Section Selection

Table 2 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 $1\mu sec$, the algorithm finds tuning sections that are called a large number of times (22528000). This solution necessitates a high-resolution timer for accurate measurements, which is implemented 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 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.

5.2 PEAK Performance

5.2.1 Experimental Environment for PEAK

We evaluate PEAK's performance in orchestrating the 38 optimizations that are included in the *O3*-level of the GCC 3.3.3 compiler. To verify PEAK performs well on different platforms, we experimented with a Pentium IV (2.8 GHz) and a SPARC II (400 MHz) system. The test applications include all SPEC CPU2000 FP benchmarks written in F77 and C, which are amenable to GCC.

We compare PEAK with a system that performs whole-program tuning. This reference system applies the same CE algorithm, but uses overall program execution for timing measurements [8].

Similar to [8], we apply two experiment techniques. (1) We use a code repository to memorize and reuse the performance results. (2) To ensure accurate measurements, we execute each code version multiple times under a single-user environment. In the reference system, the whole program is re-executed until the three least execution times are within a range of [−1%,1%]. In PEAK, each version of a tuning section is invoked a number of times controlled by the techniques presented in Section 2.2.

Two important metrics characterize the behavior of optimization orchestration:
(1) The *program performance* of the final, tuned version. We define it as the performance improvement percentage of the final version relative to the base version under the highest optimization level O3.
(2) The total *tuning time*. Because the execution times of different benchmarks are not the same, we normalize the tuning time (TT) by the time of evaluating the whole program optimized under O3, i.e., one compilation time (CT_B) plus three execution times (ET_B) of the base version.

$$NTT = TT/(CT_B + 3 \times ET_B) \tag{13}$$

The goal is the achieve high *program performance* and short *tuning time*.

5.2.2 Results of PEAK

Figure 5 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 six tuning steps.) On average, the normalized tuning time is reduced from 68.3 to 3.36., amounting to a tuning speedup of 20.3. The benchmark that has a high speedup usually has a large number of invocations to the tuning sections. On average, the absolute tuning time is reduced from 2.19 hours to 5.85 minutes. Hence, applying the rating methods at the tuning-section level significantly improves the tuning time.

Figure 6 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 is spent in Step 1, the tuning section selection (TSS) stage. This is because TSS does a profile run, which in some cases (e.g., *wupwise*) takes more time than a normal run of the program. The third largest portion is spent in Step 2, the rating method analysis (RMA) stage. RMA does data flow analysis and may need a profile run to get the context parameters for CBR and execution model parameters for MBR. For programs with a large source code (e.g., *ammp* and *mesa*), compilation time is also significant.

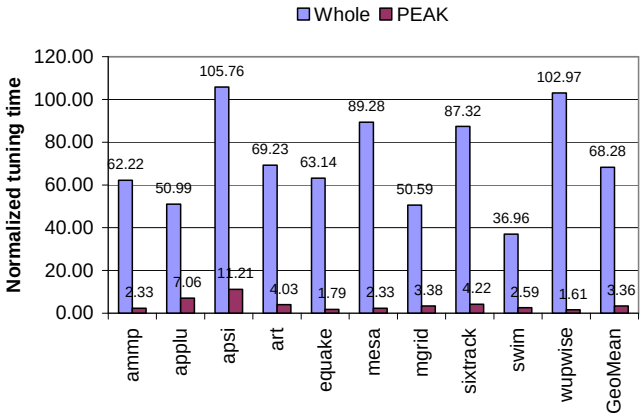

Figure 5: Normalized tuning time of the Whole-Program Tuning and the PEAK system for SPEC CPU2000 FP benchmarks on Pentium IV. On average, PEAK gains a speedup of 20.3 over the Whole-Program Tuning.

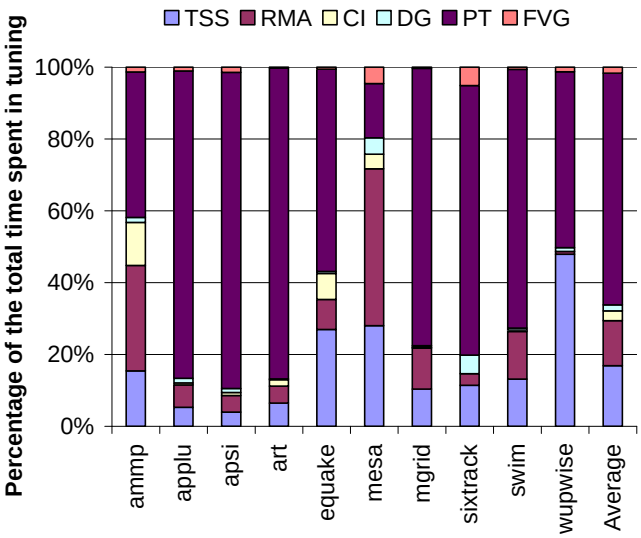

Figure 6: Tuning time percentage of the six stages for SPEC CPU2000 FP benchmarks on Pentium IV. (TSS: tuning section selection, RMA: rating method analysis, CI: code instrumentation, DG: driver generation, PT: performance tuning, FVG: final version generation.)

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. (Note that our Pentium machine is much faster than the SPARC machine.)

Figure 7 shows the program performance achieved by the Whole-Program Tuning and the PEAK system on Pentium IV. 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 system from those caused by the input sets, we measure program performance under both the *train* and *ref* datasets.

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 similar performance. PEAK outperforms Whole-Program Tuning by 1.5% on *applu*. The benchmarks *equake* and *mesa* have only one tuning section. *Art*, *swim* and *mgrid* have similar code structure in the selected tuning sections, hence they favor similar optimizations. These two observations explain why PEAK outperforms Whole-Program Tuning insignificantly in terms of tuned program performance.

The last two bars show the performance under the *ref* dataset. (Still, the *train* dataset is used during tuning.) For these benchmarks, the performance is similar to the one under the *train* dataset. This similarity shows that our off-line tuning scenario does find a good combination of compiler optimizations, for the chosen benchmarks, that is of general applicability.

On average, PEAK improves performance by 12.0% and 12.1% for *train* and *ref*, respectively, while Whole-Program Tuning improves performance by 11.9% and 11.7%. On SPARC II, both PEAK and Whole-Program Tuning improve performance by 4.1% and 3.7% for *train* and *ref*, respectively. Hence, we find that PEAK achieves similar program performance than Whole-Program Tuning.

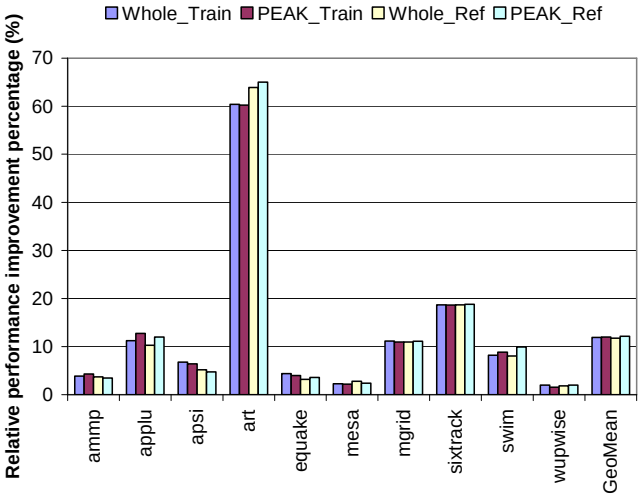

Figure 7: Program performance improvement relative to O3 for SPEC CPU2000 FP benchmarks on Pentium IV. All the benchmarks use the *train* dataset as the input to the tuning process. Whole_Train (PEAK_Train) is the performance achieved by the Whole-Program Tuning (the PEAK system) under the *train* dataset. Whole_Ref and PEAK_Ref use the *ref* dataset to evaluate the tuned program performance.

6. RELATED WORK

Many performance tuning systems use a feedback-directed approach. For example, ATLAS [12] generates numerous variants of matrix multiplication to search for the best one for a specific target machine. Similarly, Iterative Compilation [6] searches through the transformation space to find the best block sizes and unrolling factors. Meta optimization [10] uses machine-learning techniques to adjust several compiler heuristics automatically.

The above three projects [6, 10, 12] 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 the 38 GCC O3 optimization options in our experiments).

Several projects address the same optimization orchestration problem as this paper. The Optimization-Space Exploration (OSE) compiler [11] 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 [9] uses orthogonal arrays [5] 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 [1] applies fractional factorial design to optimize the selection of compiler options. Option Recommendation [4] 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 [4] uses a biased random search to discover the best order of optimizations.)

Many of the above systems [1, 4, 9] tune the performance at the whole-program level. By contrast, we have developed a system that tunes at the procedure level. To this end, we have designed a new algorithm to partition a program into multiple sections for individual performance tuning. Another important difference is that OSE [11] uses compile-time performance estimates as feedback, whereas our system uses accurate execution times.

7. CONCLUSIONS

This paper has described an automated performance tuning system, called PEAK. We have presented a novel algorithm that selects the important code sections in a program for individual performance tuning. PEAK searches for the best optimization combination for each such tuning section, leveraging a fast optimization orchestration algorithm and accurate rating methods, presented in related work. Instead of measuring overall performance, PEAK uses partial program executions for tuning feedback, leading to a reduction in tuning time from 2.19 hours to 5.85 minutes, while achieving similar performance.

8. REFERENCES

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

[2] K. D. Cooper, M. W. Hall, and K. Kennedy. A methodology for procedure cloning. *Computer Languages*, 19(2):105–117, 1993.

[3] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a call graph execution profiler. In *SIGPLAN Symposium on Compiler Construction*, pages 120–126, 1982.

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

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

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

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

[8] Z. Pan and R. Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *The 4th Annual International Symposium on Code Generation and Optimization (CGO)*, page (12 pages), March 2006.

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

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

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

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