Rating Compiler Optimizations for Automatic 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

To achieve maximum performance gains through compiler optimization, most automatic performance tuning systems use a feed-back directed approach to rate the code versions generated under different optimization options and to search for the best one. They all face the problem that code versions are only comparable if they run under the same *execution context*. This paper proposes three accurate, fast and flexible rating approaches that address this problem. The three methods identify comparable execution contexts, model relationships between contexts, or force re-execution of the code under the same context, respectively. We apply these methods in an automatic *offline tuning* scenario. Our performance tuning system improves the program performance of a selection of SPEC CPU 2000 benchmarks by up to 178% (26% on average). Our techniques reduce program tuning time by up to 96% (80% on average), compared to the state-of-the-art tuning scenario that compares optimization techniques using whole-program execution.

1. INTRODUCTION

Compiler optimizations offer potentially large performance gains of high-performance computer (HPC) applications. In many programs, however, this potential remains under-exploited. One of the chief reasons is the large number of optimization options available in today's HPC compilers. Compilers are generally unable to find the best combination of techniques and apply them to all benefiting code sections – in fact, potential performance degradation from applying the "highest" optimization level is not uncommon. For users, it is extremely tedious and time-consuming to explore the large search space of optimizations by hand.

Several recent projects have begun to develop methods for analyzing the search space by experimentally comparing the execution times of code optimized under different options. The projects all face the following issue: the execution times of two invocations of a code section are only comparable if the *contexts* of the invocation (all program variables and environment parameters that influence execution time) are the same. In this paper, we present three methods for *rating*, that is evaluating the speed of, the code *version* generated under a set of compiler optimization options. The ratings of the optimized versions are compared fairly to decide which version has the best performance. We also present automatable compiler techniques for choosing and applying the most appropriate of the three techniques to a given program. We have applied the techniques to four SPEC CPU 2000 benchmarks, which we have tuned using our PEAK tuning engine on both a SPARC II and a Pentium IV platform. The results show up to 178% performance improvements (26% on average). Also, compared to a state-of-the-art tuning scenario that compares optimization techniques using whole-program execution, our techniques lead to a reduction in program tuning time of up to 96% (80% on average).

The key ideas of our optimization rating methods are as follows. *Re-execution-based rating* (RBR) directly re-executes a code section under the same context for fair comparison. *Context-based rating* (CBR) identifies and compares invocations of a code section that use the same context, in the course of the program run. *Model-based rating* (MBR) formulates the relationship between different contexts, which it factors into the comparison. These methods facilitate program tuning scenarios that use direct timing comparison of optimization variants, either in an *offline tuning* manner (applications are tuned while not running in production mode – which we used in our experiments) or in an online, *dynamically adaptive* scenario, in which the applications are tuned while in actual use. Because of their use of actual execution timings, such tuning scenarios are also referred to as empirical optimization.

Empirical optimization contrasts with model-based optimization, which attempts to find the best techniques by modeling the resulting performance at compile time. In [17] the authors have shown that, for a limited set of programs and optimization techniques, the model-based approach can yield comparable performance to empirical optimization. While the advantage is obvious – no tuning process is necessary – the approach is limited by the extreme difficulty of accurately modeling architecture and execution parameters as well as the interaction between optimization techniques. Consistent with this view, our research has shown [11] that the performance variation of compiler optimizations can be significant and unpredictable.

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

0-7695-2153-3/04 $20.00 (c)2004 IEEE

Pursuing the empirical approach, several related projects use a feedback-directed tuning process. Meta Optimization [12] uses machine-learning techniques to adjust the compiler heuristics automatically. ATLAS [16] generates numerous variants of matrix multiplication and tries to find the best one on a particular machine. Similarly, Iterative Compilation [8] searches through the transformation space to find the best block sizes and unrolling factors. Cooper [3] uses a biased random search to discover the best order of optimizations. Granston and Holler [6] developed heuristics to deterministically select PA-RISC compiler options, based on information from the user, the compiler and the profiler. Chow and Wu [2] applied fractional factorial design to optimize the selection of compiler switches. In previous work [11], we proposed an iterative algorithm to find the best optimization combination for each individual application. The Optimization-Space Exploration (OSE) compiler [13] defines sets of optimization configurations and an exploration space to find the best optimization configuration according to compile-time performance estimation. Dynamic Feedback [4] produces several versions under different synchronization optimization policies, periodically measures the overhead of each version at run time in its sampling phase, and uses, in its production phase, the one with the least overhead.

Accuracy, speed, and flexibility are important properties of optimization rating methods. If the rating is inaccurate, the tuning system will yield limited performance or even degradation. If the rating method is too slow, the tuning process may take unacceptably long to converge (days or even weeks, given realistic search spaces for optimization techniques and parameters). If the method is not flexible enough it may apply to a limited set of applications, optimization techniques, compilers, or architectures, only. Many of the proposed systems either are slow due to executing the whole program to rate one version [3, 11, 12], apply to special functions only [8, 16], or use special hardware and compilers [13]. In [13], the rating is estimated from profile information, which is transformed along with the compiler optimizations. So, the estimation is not as accurate as, although faster than, the empirical approach.

This paper proposes general-purpose optimization rating approaches. These methods can be applied to general, regular and irregular code sections. They can use any back-end compiler without modifying it. They do not require any special hardware environments. They can also be used for both off-line tuning and online, adaptive tuning.

The remainder of this paper is organized as follows. Section 2 presents three rating methods – CBR, MBR and RBR – along with the program analysis. Section 3 presents the use of these methods during program tuning. Section 4 introduces our automatic performance tuning system – PEAK. Section 5 shows the applicability and measures the quality of the three methods on SPEC benchmarks, together with the performance improvement by PEAK.

2. RATING METHODS AND PROGRAM ANALYSIS

This section presents three new methods for rating compiler optimizations. It also discusses the compiler analysis techniques for applying the methods to a given program. For clarity, the first subsection gives an overview of the tuning environment, in which our experiments employ the rating methods. Section 4 will provide details.

2.1 Tuning Environment

We will make use of the new rating methods in an *offline performance tuning* scenario, as follows. The application to be tuned is partitioned by a static compiler into a number of code sections, called *tuning sections (TS)*. Initially, each TS is optimized statically, as in a conventional compiler. Tuning steps can take place before or between production runs of the code. The tuning system runs the application one or several times, while dynamically swapping in and out new *code versions*, each of which is generated under a set of compiler optimization options during tuning, for the TS's. The proposed methods generate the *ratings*, which represent the speed, of the code versions based on execution times. After comparing the ratings of the versions of the same tuning section, the winning version is inserted into the improved application code. As described in the introduction, the key issue in rating these versions is to achieve fair comparisons; that is, the versions must be executed and timed under *comparable contexts*. Our rating methods achieve this goal by either identifying TS invocations[1] that use the same contexts (CBR), finding mathematical relationships between contexts (MBR), or forcing re-execution of a TS under the same context (RBR).

2.2 Context Based Rating (CBR)

Context-based rating identifies the invocations of the TS under the same context in the course of the program execution. The compiler considers the set of *context variables*, which are the program variables that influence the execution time of the TS. For example, 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 TS's work load. We define the *context* of one TS invocation as the set of values of all context variables. Therefore, each context represents one unique work load.

CBR rates one optimized version under one context by using the average execution time of several invocations of the TS. The best versions for different contexts may be different, in which case CBR reports the context-specific winners. The experiments described in Section 5 make use of only the best version with the most important context for a TS, whereas an adaptive tuning scenario would make use of all versions.

For CBR to be accurate, a TS needs to be invoked a significant number of times with the same context (typically 10s of times) in the course of a program execution. The number of contexts plays an important role in CBR. When it is large, we can only make use of a small percentage of TS invocations in one run of the application to rate the code version under a certain context. To keep the number of contexts reasonable, CBR is applied only to TS's whose context variables only consist of scalars. Furthermore, if the number of contexts of a TS is large, the MBR method (described next)

[1]One invocation is one execution of the tuning section, which may or may not be a subroutine.

is preferred. We gather the number of contexts by a profile run in the offline tuning scenario.

```
//ContextSet: the set of context variables.
VariableSet ContextSet;

//Return value: applicability of CBR on TS
Boolean GetContextSet(TuningSection TS)
{
  ContextSet = {};
  Set the state of each statement as "undone";
  For each control statement s in TS {
    For each variable v used in s {
      if( GetStmtContextSet(v, s) == false )
        return false;
    }
  }
  Remove the constant variables from ContextSet;
}

Boolean GetStmtContextSet(Variable v, Statement s)
{
  StatementSet SSet = Find_UD_Chain(v, s);
  Set s as "done";
  For each statement m in SSet {
    if( m is the entry statement ) {
    //v is in Input(TS).
      if( v is scalar ) {
        put v into ContextSet;
      }
      else
        return false;
    }
    if( m is "done" ) {//avoid loop.
      continue;
    }
    For each variable r used in m {
      if( GetStmtContextSet(r, m) == false )
        return false;
    }
  }
  return true;
}
```

Figure 1: Context variable analysis

Figure 1 shows the compiler analysis to determine the applicability of CBR and to find the context variable set. The algorithm traverses each control statement and recursively finds the input variables that may influence the values used in control statements. All these variables are considered as context variables. If there exist one or more non-scalar context variables, CBR is not applied. Three types of variables are regarded as scalars: (1) plain scalars; (2) array references with constant subscripts; (3) memory references by pointers that are not changed within the tuning section. We found that simple points-to analysis is sufficient for that purpose.

We eliminate unnecessary context variables, if they are *run-time constants*; that is, values that are the same for all invocations of the TS. (Run-time constants are widely used[7, 9, 10] in compiler optimizations.)

2.3 Model Based Rating (MBR)

Model-based rating formulates mathematical relationships between different contexts of a TS and adjusts the measured execution time accordingly. In this way, contexts that are incompatible under CBR become comparable.

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

$$T_{TS} = \sum (T_b * C_b) \tag{1}$$

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

Furthermore, if the numbers of entries of 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.

$$T_{TS} = \sum (T_i * C_i) \tag{2}$$

T_{TS} consists of several components, each of which has a *component count* C_i and a *component time* T_i. We assume that there is always a constant component T_n, with $C_n = 1$ for all TS invocations.

Our compiler uses compile-time analysis to get the expression determining the number of entries to the basic block b, C_b, if the code structure is regular, such as the loop body of a perfectly nested loop. Otherwise, it instruments the relevant blocks with counters. (These counters do not add control dependences or data dependences to the original codes and thus have little influence on compiler optimizations and the rating accuracy.) After a profile run, our compiler finds the relationships among C_b's, thus to merge them into components. Next, the unnecessary instrumentation code for the merged blocks is removed. The remaining instrumentation will supply the component counts C_i's to the rating process during performance tuning.

During tuning, the run-time system collects the timings of a number of invocations of the code version until the rating error is small, which we will discuss in Section 3. The performance rating system gathers the TS-invocation-time vector, Y, and the component-count matrix, C, in which $Y(j)$ is the T_{TS} in the j'th invocation and $C(i, j)$ is the i'th component count C_i in the j'th invocation. Solving the following linear regression will yield the component time T_i's.

$$Y = T * C \tag{3}$$

Here, $T = (T_1, T_2, ..., T_n)$ represents the component-time vector of one particular version. The version with smaller T_i's performs better. Hence, MBR may compare different versions based on their T vectors. There are two ways of rating based on the T vector under MBR. (a) Directly, use T_i of the dominant component, if this component consumes a significant portion, e.g. 90%, of the execution time. (b)

Use the execution time estimation T_{avg}.

$$T_{avg} = \sum (T_i * C_{avgi}) \quad (4)$$

C_{avgi} is the average count of component i during one whole run of the program. These data are obtained from the profile run.

Figure 2 (a) shows an example code with two components. The first component is the loop body with a variable number, N, of entries during one invocation of the tuning section. The second component is the tail code with one entry per invocation. Figure 2 (b) shows the Y and C gathered by the performance rating system during tuning. Each column of Y and C corresponds to the data in one invocation of the tuning section. Linear regression generates the component-time vector T, shown in Figure 2 (c). The first component dominates, so, this version has a rating of $T_1 = 110.05$.

```
DO I = 1, N
...loop body...
ENDDO
...tail code...
```

(a) A tuning section with two components

$$Y = \begin{bmatrix} 11015 & 5508 & 6626 & 6044 & 8793 \end{bmatrix}$$
$$C = \begin{bmatrix} 100 & 50 & 60 & 55 & 80 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

(b) TS-invocation-time vector Y and component-count matrix C collected during tuning

$$T = \begin{bmatrix} 110.05 & 3.75 \end{bmatrix}$$

(c) Component-time vector T by linear regression

Figure 2: A simple example of MBR

If there are many components in the estimation model, a large number of invocations of this TS need to be measured in order to perform an accurate linear regression. MBR would lead to a long tuning time in this case and so is not applied. A number of simplifications increase the efficiency of MBR. (1) If the two branches in a conditional statement have the same work load, the components representing the branches are merged. (2) If the work load in conditional statements is small, they are treated as normal statements. For example, the if-statement with a simple increment statement is not treated as a basic block, but as an increment statement. (3) Components that exhibit constant behavior, are put into the constant component. MBR is not as accurate as CBR due to these simplifications and the side effect of the inserted counters. However, in our experiments, this inaccuracy was negligible.

2.4 Re-execution Based Rating (RBR)

Re-execution-based rating forces a roll-back and re-execution of a TS under the same context. It is applicable to most TS's; however, it also generally has the largest overhead. We first present a basic re-execution method, followed by a method that reduces inaccuracies caused by cache effects.

2.4.1 *Basic RBR Method*

Figure 3 shows the basic idea. In each invocation of the TS, the input data to the TS is saved, then Version 1 is timed, the input is restored, and Version 2 is executed. The input set $Input(TS)$ can be obtained through liveness analysis. $Input(TS)$ is equal to $LiveIn(b1)$, the live-in set of the first block in TS. Eligible TS's must not call library functions with side effects, such as `malloc`, `free`, `rand`, and I/O operations. (TS may be partitioned to exclude these functions.) Since the input to the two versions is the same, the second execution has the same workload as the first one.

Step 1. Save the $Input(TS)$
Step 2. Time Version 1 (the current best version)
Step 3. Restore the $Input(TS)$
Step 4. Time Version 2 (the experimental version)
Step 5. Return the two execution times

Figure 3: Basic re-execution-based rating method (RBR)

RBR directly generates a *relative performance improvement* based on the execution times of the two versions, which are executed during one TS invocation. Suppose that the execution time of Version 1 is T_{v1}, and that the execution time of Version 2 is T_{v2}. Then, the performance improvement of Version 2 relative to Version 1 is $R_{v2/v1}$.

$$R_{v2/v1} = T_{v1}/T_{v2} \quad (5)$$

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 improvements 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.

2.4.2 *Improved RBR Method*

Even under the same input, two executions of the same version within one TS invocation may result in different execution times. The first execution may precondition the cache, affecting the second one. To address this problem, the improved RBR method

1. inserts a *precondition version* before Version 1 to bring the data into the cache, and
2. swaps Version 1 and Version 2 at each invocation, so that their order does not bias the result.

In addition, the improved RBR method saves and restores only part of the input set, the $Modified_Input(TS)$.

$$Modified_Input(TS) = Input(TS) \cap Def(TS) \quad (6)$$

$Def(TS)$ is the def set of the TS.

Figure 4 shows the improved RBR. This method incurs three overheads: (1) save and restore of the $Modified_Input(TS)$;

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

Figure 4: Improved re-execution-based rating method

(2) execution of the precondition code; and (3) execution of the second code version.

Compile time analysis may not be able to determine the exact $Modified_Input(TS)$ set. Before irregular array and pointer write references to variables in this set, inspector code that records the addresses and values of the write references is inserted into the precondition version.

The overhead of the save, restore and precondition code can be reduced through a number of compiler optimizations. For example, the save and restore overhead can be reduced by accurately analyzing the $Modified_Input(TS)$ set. This can be achieved using symbolic range analysis [1] for regular data accesses. Other optimizations include the combination of a number of experimental runs into a batch, and the elimination of instructions from the precondition code that do not affect the cache.

3. DISCUSSION AND APPLICATION TO AUTOMATIC PERFORMANCE TUNING

We have presented three approaches for rating, that is evaluating the speed of code versions under different optimizations: RBR, CBR and MBR. Re-execution-based rating (RBR) can be applied to almost all tuning sections; however, the overhead is the highest among the three. Context-based rating (CBR) has the least overhead but is not applicable to irregular codes or codes with many contexts. Model-based rating (MBR) is applicable to irregular codes, but may reduce the rating accuracy. Generally, the applicability of these three rating approaches increases in the order of CBR, MBR and RBR; so does the overhead.

At compile time, the program is divided into TS's to be tuned individually. Next, the necessary program analysis and insertion of instrumentation code is performed: (1) Code is inserted for saving and restoring the modified input set, and the precondition code for RBR; (2) context variables are determined, if CBR is applicable; (3) counters and the corresponding performance model are inserted in TS's, if MBR is applicable; (4) instrumentation is added to measure the TS's execution time and trigger the performance rating; (5) the main program is instrumented to activate the performance tuning.

For each optimized version of TS, the tuning system generates the *rating*, $EVAL$, and the *rating variance*, VAR, across a number of TS invocations, which is called a *window*. The tuning system compares $EVAL$'s of different versions to know which version is the best one. We use different methods to compute $EVAL$ and VAR under CBR, MBR, and RBR. (1) CBR: Suppose that $T(i, x)$ is the execution time of the i'th invocation with context x. $EVAL$ and VAR of context x are the mean and the variance of $T(i, x)$, $i = 1...w$, where w is the window size. (2) MBR: After the linear regression, T_{avg} or the T_i of the dominant component i is the $EVAL$ of the corresponding version. VAR is computed as the ratio of the sum of squares of the residual errors of the regression to the total sum of squares of the TS execution times. (3) RBR: Suppose that $R_{v1/vb}(i)$ is the relative performance improvement of the experimental version $v1$ over the base version vb at the i'th invocation. Under RBR, $EVAL$ and VAR are the mean and the variance of $R_{v1/vb}(i)$, $i = 1...w$.

The tuning engine also identifies and eliminates measurement outliers, which are far away from the average. Such data may result from system perturbations, such as interrupts. Also, as VAR decreases with increasing size of the window, the system continually executes and rates the version until VAR falls below a threshold. In this way, consistent ratings are produced.

For off-line tuning, our compiler chooses the appropriate rating method by doing a profile run using the tuning input. For example, we need to know the number of contexts for CBR and the most important component or the C_{avgi}'s for MBR. Then, our compiler chooses the applicable rating approach with the least overhead estimated from the profile.

Our compiler picks the initial rating approach for each tuning section in the order of CBR, MBR, and RBR, if they are applicable. At tuning time, the system gathers the execution time of the experimental versions and relevant parameters to rate the code version. If the system cannot achieve enough accuracy, i.e. get a small VAR, within some number of invocations, it switches to the next applicable rating method.

4. EXPERIMENTAL SETUP

4.1 Program Partitioning into Tuning Sections

In this paper, we choose as TS's the most time-consuming functions and loops, according to the program execution profiles in the SPEC benchmarks. We call the generated code for a TS under one set of optimization options one *version*.

For compilation, each TS is extracted into a subroutine so that it can be compiled and optimized separately. We use the Gnu compiler collection (GCC) and initially compile all programs with the "-O3" optimization option.

4.2 Overview of the PEAK Tuning System

The PEAK tuning system uses an *offline tuning* scenario. Tuning happens at runtime but not during production runs. The result of the tuning process is a new, improved code version, which is absent of any instrumentation code, and thus overheads, that are incurred during tuning. PEAK builds

on the ADAPT infrastructure [14], which was designed to perform online, adaptive optimizations. The comparison of online and offline tuning is beyond the scope of this paper. Briefly, a main advantage of offline tuning is the elimination of tuning system overheads during production runs. On the other hand, online, adaptive tuning is a necessity for continuously running applications and for applications whose TS's exhibit very different behavior in different invocations.

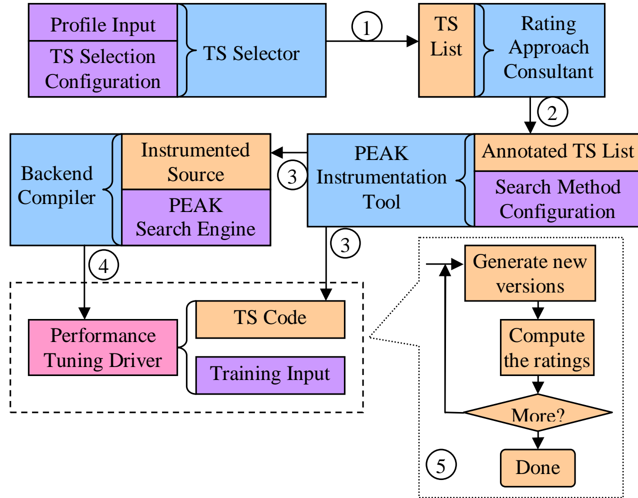


Figure 5: Overview of the PEAK tuning system

Figure 5 gives an overview of PEAK. PEAK performs the following steps. (1) The TS Selector chooses the tuning sections, as described in Section 4.1. (2) The Rating Approach Consultant annotates each TS with applicable rating methods. (3) The PEAK Instrumentation Tool extracts each TS into a separate file, which will be compiled in step 5 under different optimization options. It inserts the instrumentation code, as described in Section 3. (4) The instrumented code is compiled and linked with the PEAK Search Engine to form the Performance Tuning Driver. (5) The Performance Tuning Driver iteratively finds the best version for each TS. It tunes the TS by generating new optimized versions, comparing their ratings, and finally picking the best among these versions for the resulting code. An important issue in PEAK is the method for traversing the search space of optimization options. We use the *Iterative Elimination* algorithm, developed in previous research [11].

PEAK uses a number of distinguishing features of the underlying ADAPT infrastructure. Details are described in [14, 15].

- Optimized versions are compiled dynamically and inserted into the code using dynamic linking options.
- Figure 6 shows the underlying mechanism. Both a "best" and an "experimental" code version for each TS are kept and dynamically swapped in and out. The *Remote Optimizer* can be any compiler, which may run on the local or a remote processor.

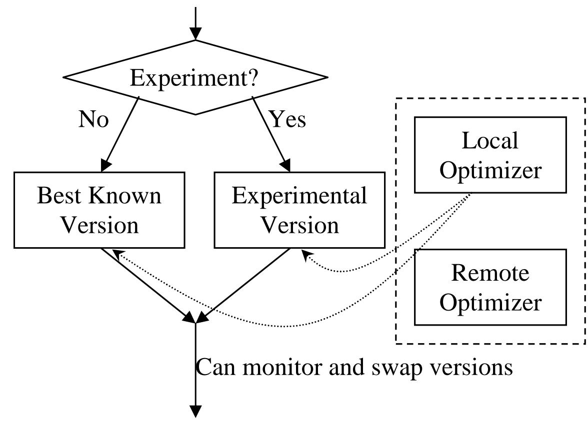


Figure 6: Underlying tuning mechanism of PEAK/ADAPT

5. RESULTS

5.1 Consistency of Rating Methods

An important question is the consistency of measurements achieved by the rating methods; that is, the reliability that a decision on an improved optimization version is correct. We tested the consistency of the three rating methods by measuring the mean and standard deviation of the ratings under different *window sizes*. A window is the number of invocations over which the measured TS's are averaged before making a decision.

Table 1 shows the most important TS for each benchmark and consistency metrics. The upper half shows the integer benchmarks; the lower half shows the floating point benchmarks. The integer codes exhibit a large number of conditional statements, leading to highly irregular behavior. Because of this, our algorithm applies the re-execution-based methods (RBR) to all these codes. The floating point benchmarks are more regular. The context-based rating (CBR) and the model-based rating (MBR) methods are applicable to these benchmarks.

To obtain a measure of rating consistency, our experimental system uniformly sampled the ratings throughout the execution, with the training data set as the input. In this way, it gathered a vector of ratings, $[V_1, V_2, ..., V_n]$, where V_i is the $EVAL$ computed at sampling time i, as described in Section 3. (Each rating V_i is based on timing measurement of w invocations of the TS.) For this measurement, we use only one experimental version, which is compiled under "-O3". So, we assume that the ideal rating is the average of V_i, $\overline{V}$, for CBR and MBR. The ideal rating for RBR is 1, since the experimental version is the same as the base version.

$$\overline{V} = \sum(V_i)/n \tag{7}$$

We compute the rating error, X_i, at sampling time i.

$$X_i = \begin{cases} V_i/\overline{V} - 1 & \text{for CBR and MBR} \\ V_i - 1 & \text{for RBR} \end{cases} \tag{8}$$

Table 1 shows the statistic characteristics of the rating errors, the *Mean*, μ, and the *Standard Deviation*, σ, which are

Table 1: Consistency of rating approaches for selected tuning sections.
The columns from left to right show the benchmark name, the tuning section name, the applicable rating approaches, the number of invocations of the tuning section during one run of the benchmark, and the rating consistency under different window sizes. The numbers for the consistency columns are multiplied by 100 for readability. (For CBR, multiple rows are used for each tuning section, if there are multiple contexts.)

Benchmark Name	Tuning Section	Rating Approach	#invocations	Rating Consistency: *Mean* (*Standard Deviation*) $*$ 100 w=10	w=20	w=40	w=80	w=160
BZIP2	fullGtU	RBR	24.2M	0.95(2.6)	0.5(1.9)	0.27(1.3)	0.09(1.0)	0.07(0.7)
CRAFTY	Attacked	RBR	12.3M	-0.91(2.3)	-0.43(1.7)	-0.25(1.5)	-0.33(1.2)	-0.16(0.8)
GZIP	longest_match	RBR	82.6M	-1.0(2.7)	-0.14(1.2)	-0.08(1.1)	-0.1(0.9)	-0.05(0.7)
MCF	primal_bea_mpp	RBR	105K	-0.23(0.92)	-0.18(0.71)	-0.16(0.48)	-0.09(0.36)	-0.11(0.31)
TWOLF	new_dbox_a	RBR	3.19M	-0.56(1.9)	-0.45(1.3)	-0.36(1.0)	-0.23(0.58)	-0.13(0.37)
VORTEX	ChkGetChunk	RBR	80.4M	-0.12(3.0)	0.26(1.6)	0.18(1.2)	-0.16(0.97)	-0.11(0.76)
APPLU	blts	CBR	250	0(0.71)	0(0.65)	0(0.57)	0(0.49)	0(0.18)
APSI	radb4(Context 1)	CBR	1.37M	0(2.2)	0(2.6)	0(3.0)	0(2.7)	0(1.4)
	radb4(Context 2)	CBR		0(0.7)	0(0.7)	0(0.7)	0(0.7)	0(0.5)
	radb4(Context 3)	CBR		0(0.5)	0(0.4)	0(0.3)	0(0.3)	0(0.2)
ART	match	RBR	250	-0.06(0.28)	-0.07(0.17)	-0.08(0.11)	-0.1(0.07)	-0.09(0.04)
MGRID	resid	MBR	2410	0(1.0)	0(0.82)	0(0.76)	0(0.63)	0(0.48)
EQUAKE	smvp	CBR	2709	0(2.7)	0(2.5)	0(2.4)	0(2.1)	0(1.6)
MESA	sample_1d_linear	RBR	193M	-0.05(1.3)	0.07(1.0)	0.03(0.78)	0.07(0.57)	0.02(0.36)
SWIM	calc3	CBR	198	0(0.33)	0(0.29)	0(0.19)	0(0.06)	0(0.01)
WUPWISE	zgemm(Context 1)	CBR	22.5M	0(1.3)	0(1.1)	0(1.1)	0(0.94)	0(0.86)
	zgemm(Context 2)	CBR		0(1.5)	0(1.6)	0(1.6)	0(1.7)	0(1.5)

the measure of rating consistency.

$$\mu = \sum(X_i)/n \quad (9)$$

$$\sigma = \sqrt{\sum(X_i-\mu)^2/(n-1)} \quad (10)$$

High rating consistency requires the *Mean*, μ, be close to zero and a small *Standard Deviation*, σ.

The last column in Table 1 shows the *Mean* and the *Standard Deviation* under different window sizes. Generally, both metrics decrease with increasing window size. RBR achieves a very small mean (< 0.002) and a small standard deviation (< 0.016) with a reasonable window size for all cases. EQUAKE has a relatively high variation, which we attribute to its irregular memory access behavior, resulting from sparse matrix operations. We conclude that our rating methods are consistent. Small tuning sections exhibit more measurement variation but also tend to have higher numbers of invocations. In these cases, consistency is achieved through larger window sizes.

The fourth column in Table 1 shows the number of invocations of the tuning section under the training data set. For some benchmarks, the number of invocations exceeds one million, while for others it is only several hundred. In all benchmarks, the system may rate multiple versions during one run of the application. The total number of invocations needed in one tuning is roughly *window size* $*$ *number of versions*. Some benchmarks fit in one run, others fit in multiple runs. So, PEAK can reduce the tuning time by a significant amount, which we will show in the next section.

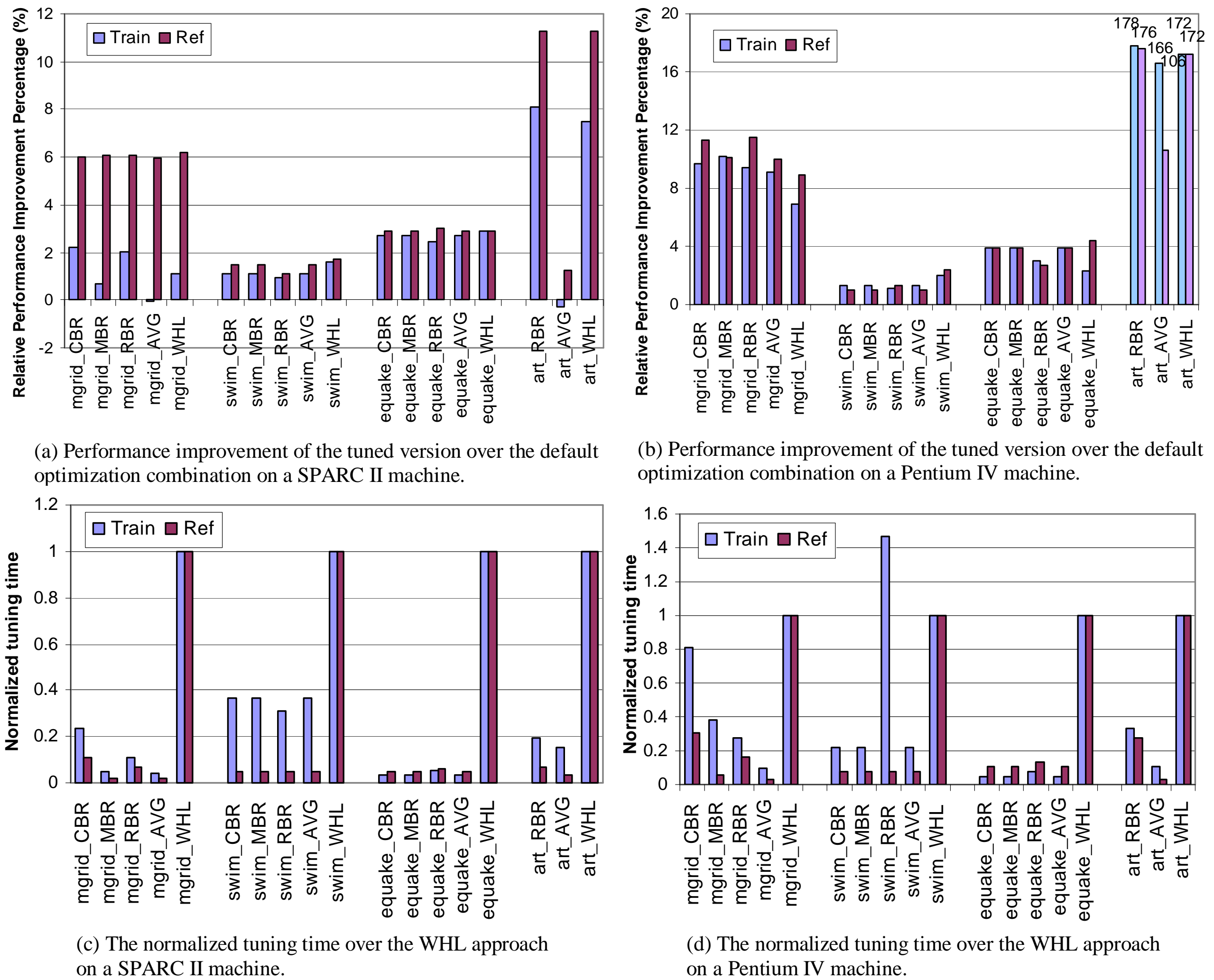
5.2 Performance Results

To measure the performance improvement obtained by the tuning process, we explore all $n = 38$ optimization options implied by "-O3" of the GCC 3.3 version [5]. An exhaustive search for the best combination has a complexity of $O(2^n)$, which is unacceptably time-consuming. Hence, we use the *Iterative Elimination* algorithm [11], which reduces the complexity to $O(n^2)$. It starts with O3 and iteratively removes the optimizations with the largest negative effects. Alternative pruning algorithms [2, 13] could also be plugged into our system.

We perform the experiments on both a SPARC II and a Pentium IV machine, with two floating point (SWIM and MGRID) and two integer (ART and EQUAKE) SPEC CPU 2000 benchmarks.

Figure 7 (a) and (b) show the performance improvement over the version compiled under "O3" on two machines, respectively, using the full (i.e., *ref*) data set. Like in profile-based optimization, for reporting fair results of an offline tuning system, a program input must be used that is different from the production run's input. We use the SPEC benchmarks' *train* and *ref* data sets for that purpose. The performance of the tuned application is measured using the *ref* data set whereas we use the *train* data set during the tuning process. For each entry, the left bar uses the *train* data set, which is the appropriate data set to use during the tuning process. The right bar is for comparison and shows the performance that would be achieved if one used the same (*ref*) data set for tuning and production runs.

The graphs show the performance that would be achieved using all applicable rating methods. IF CBR is applicable, then MBR is also applicable; if MBR is applicable, RBR is also applicable. The PEAK compiler chooses MBR for MGRID, CBR for SWIM, CBR for EQUAKE, and RBR for ART.



(a) Performance improvement of the tuned version over the default optimization combination on a SPARC II machine.

(b) Performance improvement of the tuned version over the default optimization combination on a Pentium IV machine.

(c) The normalized tuning time over the WHL approach on a SPARC II machine.

(d) The normalized tuning time over the WHL approach on a Pentium IV machine.

Figure 7: Performance improvement by PEAK and normalized tuning time.
All performance measurements are taken using the *ref* data sets. Left bars show tuning using the *train* data sets; right bars show tuning using the *ref* data sets. (The significant performance improvement of ART in (b) is rescaled to fit in.)

For comparison, two simplistic rating methods are also shown, WHL and AVG. WHL averages the TS's execution times over the entire applications. It represents the best that can be achieved by static tuning. (In some cases, other rating methods may achieve slightly better performance than WHL. The reason is that they identify the measurement outliers to increase the rating accuracy.) The chief disadvantage of WHL is extremely long tuning times, because every trial needs a full application run. AVG simply takes the timing average of a number of invocations, regardless of the TS's context. It can be viewed as a naive attempt to avoid WHL's disadvantage. It is worth noting that AVG does not generally produce consistent ratings as the other approaches do, because it ignores the context of each invocation.

Figure 7 (a) and (b) show that all applicable approaches achieve similar performance improvement, which is close to the one from WHL. AVG achieves lower improvement than other approaches, and may even cause degradation. In SWIM and EQUAKE, there is only one context and, hence, AVG and MBR are equivalent to CBR. AVG improve MGRID on Pentium IV, because it has a periodic execution time pattern. In ART on Pentium IV, AVG is able to pick out the optimization that significantly hurts performance.

Comparing the left bar with the right bar, we find that, except for MGRID and ART on SPARC II, tuning with a training data set comes close to the performance achievable with the production data set.

Figure 7 (c) and (d) show to what value the tuning time is reduced, compared to the state-of-the-art approach of using full application, that is, using the WHL method. In most cases, tuning time is reduced by more than a factor of ten. The figures also show that using the wrong rating approach may increase tuning time. MGRID_CBR has too many contexts, so it is worse than MGRID_MBR. SWIM_RBR on Pentium IV is significantly worse than SWIM_CBR. The tuning time based on the *ref* data set improves more than the one using the *train* data set. This is because there are

more invocations of each tuning section. So, on average, each run of the program rates more code versions.

Using the rating methods suggested by PEAK, the tuning system achieves up to 178% performance improvements (26% on average). Also, compared to the WHL approach that rates optimization techniques using whole-program execution, our techniques lead to a reduction in program tuning time of up to 96% (80% on average).

There are many optimizations with potential harmful effects. Due to the interaction between the compiler optimizations, it is very difficult to analyze the reasons for performance degradation, [11] analyzes several interesting optimizations, for example, global common subexpression elimination and if conversion. Of particular interest, here, we discuss ART. The significant performance improvement of ART on the Pentium IV machine comes from turning off "Strict Aliasing". With strict aliasing, the live ranges of the variables become longer, leading to high register pressure and spilling. This spill code generates substantial memory accesses during the execution, which in term causes the performance degradation. However, the SPARC II machine has more general purpose registers than the Pentium IV machine, so, the SPARC II machine can tolerate higher register pressure. Strict aliasing improves the performance of ART on SPARC II.

6. CONCLUSIONS AND FUTURE WORK

We have proposed three methods, context-based (CBR), model-based (MBR), and re-execution-based (RBR) rating, for identifying and comparing the execution time of code sections that execute under comparable execution contexts. Such rating methods are an important part of any offline or online/adaptive performance tuning system. RBR is a generally applicable approach that forces the re-execution of a code section under the same context; however, it has the biggest overhead. CBR minimizes the overhead by picking invocations that have the same context. Its applicability is limited. MBR finds mathematical relationships between contexts. It is applicable to irregular codes but may reduce accuracy. We presented related compile-time program analysis methods and an algorithm to select the most appropriate rating method.

We have demonstrated that these approaches generate consistent ratings. Applied in the PEAK performance tuning system, we have shown that they facilitate both improved performance and significantly reduced tuning times.

In an ongoing project, we are developing the compiler framework for PEAK, to fully automate performance tuning. We will also take the memory access behavior into consideration to improve rating accuracy.

While we have demonstrated an offline tuning process in this paper, the presented rating methods are also applicable to an online, adaptive optimization scenario. In a related project we are pursuing this goal, thus facilitating dynamic tuning of applications that are very long running, or that exhibit different behavior across their execution time.

7. REFERENCES

[1] William Blume and Rudolf Eigenmann. Symbolic range propagation. In *the 9th International Parallel Processing Symposium*, pages 357–363, 1995.

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

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

[4] Pedro C. Diniz and Martin C. Rinard. Dynamic feedback: An effective technique for adaptive computing. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 71–84, 1997.

[5] Free Software Foundation, http://gcc.gnu.org/onlinedocs/gcc-3.3.3/gcc/. *GCC online documentation*, 2003.

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

[7] Brian Grant, Matthai Philipose, Markus Mock, Craig Chambers, and Susan J. Eggers. An evaluation of staged run-time optimizations in dyc. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 293–304. ACM Press, 1999.

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

[9] Peter Lee and Mark Leone. Optimizing ML with run-time code generation. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 137–148, 1996.

[10] Markus Mock, Craig Chambers, and Susan J. Eggers. Calpa: a tool for automating selective dynamic compilation. In *International Symposium on Microarchitecture*, pages 291–302, 2000.

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

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

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

[14] Michael Voss and Rudolf Eigenmann. ADAPT: Automated de-coupled adaptive program transformation. In *International Conference on Parallel Processing*, pages 163–170, 2000.

[15] Michael J. Voss and Rudolf Eigemann. High-level adaptive program optimization with ADAPT. *ACM SIGPLAN Notices*, 36(7):93–102, 2001.

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

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