Dynamic Optimization Option Search in GCC

Haiping Wu, Eunjung Park, and Miahilo Kaplarevic
University of Delaware
{hwu,epark,kaplarev}@capsl.udel.edu

Yingping Zhang
Intel Corporation
ying.m.zhang@intel.com

Xiaoming Li and Guang R. Gao
University of Delaware
xli@ece.udel.edu, ggao@capsl.udel.edu

Abstract

A set of carefully selected compiler optimization options could provide an additional performance boost over the current best default optimization options in the GNU Compiler Collection (GCC) C compiler. However, there are more than 60 optimization options in GCC compiler, which translate to over 2^{60} possible combinations. GCC compiler developers are therefore faced with a challenge. The goal is to automate the process of optimization search, taking into consideration the properties of the program. The resulting customized set of options is aimed at out performing the best default options available in GCC.

In this paper, we present a novel machine-learning based method for dynamic optimization option search in GCC compiler, which automatically derives the best candidate set of optimization options based on the input program properties. An automated program analysis considers input program objects such as selected functions, program segments, or even the whole programs. The preliminary tests show that this method provides better performance over any default optimization level including -O3, while the additional compilation time remains minimal.

1 Introduction

GNU Compiler Collection (GCC) C compiler [5] has more than 60 available optimization options, which translate to over 2^{60} possible combinations. As a result, finding the most effective combination of compiler options represents a significant challenge to GCC users. Fully understanding even a small subset of GCC options often requires an in-depth knowledge of compiler's inner structure and organization. Therefore users typically apply default optimization such as -O3 to reduce the complexity of this task. However, even the highest default optimization level does not necessarily result in the best performance for an arbitrary input program [3, 4].

Ideally, if GCC compiler was able to automatically select a set of options according to the specific input program features, it would be possible to achieve an additional performance speedup over the best default optimization such as -O3. This is what has inspired us to developed a dynamic optimization option search method.

Our method uses a machine-learning technique to automatically derive predictive models for each program object. Predictive models are used to determine which set of optimization options will result in the best performance, given the input program object features. Program objects could be segments, functions or even the whole programs. We have developed a dynamic optimization option search method for GCC version 3.3. The preliminary results show that our method provides better performance over any standard optimization level including -O3, while introducing minimal additional compilation time.

The remaining sections of this paper are organized as follows:

We first describe the method's fundamental principles in Section 2. The detection of the program segment similarity is discussed in Section 3. In Section 4, we present the implementation skeleton that is integrated in GCC internal. Section 5 gives the preliminary experimental results. Finally, in Section 6 we discuss general observations, and introduce readers to ideas considered for future work.

2 Methodology Overview

The described dynamic optimization option search methodology automatically derives predictive models for each program object, and is based on a machine-learning technique. A predictive model is used to determine which set of optimization options will result in the best performance, given the input program object features. The automated program analysis considers input program objects such as selected program functions, program segments, and even the whole programs.

Our method first creates predictive models by training a set of selected program segments [7]. These program segments are typically loop structures. The selected program segments are compiled in GCC using different sets of optimization options and benchmarked on the underlying platform. A predictive model is created only for those combinations of optimization options that result in additional performance speedup over the best GCC default setting (usually -O3). Each predictive model consists of a feature vector, which quantifies static properties of each program segment (further called program performance behaviors), and the corresponding set of optimization options.

A selected program segment needs to be trained if a combination of optimization options which results in an additional performance speedup cannot be found. The training process is empirical. Once the predictive models are created, a machine-learning algorithm first partitions the learning space of program performance behaviors into clusters using the feature vector of the predictive models. Each cluster includes a unique predictive model. Then the machine-learning algorithm matches each input program object to a cluster using the program performance feature vector of the input program segment. The predictive model in the matched cluster is then used as input program segment's proxy to determine the set of optimization options which should be applied in compiling the program segment.

Both the predictive models and the machine-learning algorithm are integrated in GCC internal. The enhanced GCC identifies all candidate program segments, extracts the program performance behaviors of each segment and translates them into the feature vector, and finally correlates program segments with the predictive models by means of the feature vectors. In the back-end optimization GCC phase, each program segment is optimized using only the options which are selected in its matched predictive model.

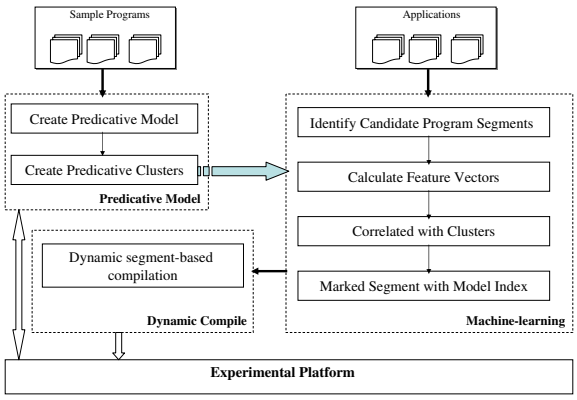


Figure 1: Infrastructure of Automatic Optimization Option Search Methodology

The creation of predictive models is a "one-size-fits-all" process, and is developed as a "plug-in" component for GCC. A flowchart description of our methodology infrastructure is shown in Figure 1.

3 Program Similarity, Predictive Model and Machine-Learning

Predictive models are used to determine applicable sets of optimization options to each input program segment. Each predictive model could represent an infinite number of program segments. Program segments that have similar program performance behaviors to their predictive models can be considered predictive model instances.

In this section, we first introduce a method used in program performance behavior analysis, then we describe the mechanism how the predictive models are created, and finally we presents the machine-learning approach.

3.1 Program Performance Behaviors Analysis

Program performance behaviors are static properties of each program segment, and are relevant in choosing compiler optimization options. In general, program performance behaviors can be determined solely based on the analysis of operations found in program segments.

To simplify the analysis we limit program segments to loop structures only. Also, we only observe the following four classes of program performance behaviors:

- Memory access
- Operation
- Conditional Statement
- Global Data and Local Data

Each class of behaviors consists of several types of program performance behaviors, depending on the underlying architectures and the complexity of the machine-learning technique. We consider 11 behaviors to describe each program segment, as shown in Table 1. Too many types would make the machine-learning space too big, and consequently the machine-learning algorithm performance would become poor.

Each type of behaviors is translated into an integer. The translation phase involves the following steps:

- Estimate the total number of instructions. Assume each operation in the program segment is transformed into a related instruction. Classify all operations and calculate the number of instructions for each type of operations.
- Calculate the total number of memory access in the program segment (LOAD/STORE operands only). A variable is treated as a STORE operand if it appears on the left side of an assignment statement. Otherwise the variable is a LOAD operand. If a variable is reused several times in the program segment, it is only counted once, except when reused in the parameter list of a function call.
- Extract the storage feature of each variable and calculate the global and the local (stack) space size.
- Extract other types of program behaviors directly from the program segment.

The behavior extraction procedure transforms program performance behaviors into a fixed-format feature vector, which always has 11 elements. For all program behaviors that are not extracted, the corresponding elements in the feature vector are set to 0.

3.2 Predictive Model Creation

Predictive models are trained from a set of sample program segments. We chose a set of accredited benchmarks for the program segment selection. The current implementation only considers loop structures as the candidates for training program segments.

The selected program segments are first benchmarked on the underlying platform. A custom made tool for optimization option combination search (options generator) repeatedly produces new combinations of options from a narrowed-down space of all possible option combinations [7]. Each selected program segment is compiled using the proposed combination of options. If this combination of options leads to an additional performance speedup over the best default option of GCC (usually -O3), a predictive model is derived from this program segment and this combination of options is stored in the model. If a combination of optimization options is stored in the predictive model, this combination of options is called an *optimized set of options* (or *OSO*) of this model.

The options generator repeatedly produces combinations of options until an *OSO* is found. If the generator has not found an *OSO* after the whole search space is exhausted, the selected program segment needs to be trained. This is a trial and error process where the number of assignment statements and the number of operations (operands and operators) is adjusted in the program segment. A selected program segment has to be skipped if no *OSO* is found after 10 training iterations.

Once an *OSO* is found, the program performance behaviors of this program segment are extracted and quantified into a feature vector. A predictive model is formed by pairing up the feature vector and the *OSO*.

3.3 Machine-Learning to Determining the *OSO*

Machine-learning techniques offer an automated framework to correlate input program segments with predictive models. The correlation is based on the program performance behaviors similarity between the input program and the predictive model. The comparison is only done between the feature vectors of the input program and the predictive model.

Performance Behaviors	Meaning
Global Space	Number of Bytes allocated for globals
Local Space	Number of Bytes allocated for locals
Load	Number of Memory load
Store	Number of Memory store
Boolean	Number of Boolean operations
Arithmetic	Number of Arithmetic operations
Logical	Number of Logical operation
If Condition	Number of IF statements
Switch	Number of Switch statements
Function	Number of Function call
Parameter	Number of actual parameters in all functions

Table 1: Performance behaviors of a Program Segment

3.3.1 Cluster Creation

It is rarely the situation that the feature vector of an arbitrary program segment and the predictive model match exactly. Fortunately, our machine-learning technique supports an approximate match. That is, the learning-space (program performance behaviors space) is partitioned into a series of clusters. Any two points are considered to be matched if they belong to the same cluster. Accordingly, a program performance feature space can be partitioned into a series of clusters, using the predictive models as a representative point in a cluster. The goal is to find a unique mapping between the feature vector of an input program segment and a cluster.

Partitioning a learning space can be done in many ways, such as Logistic Regression approach [2], Decision Tree approach [1, 6] and Clustering [7]. The current implementation uses the mean-value clustering approach to partition the program performance features space.

We have observed that if an *OSO* applies to a program segment, the same *OSO* also applies to its variations (for instance, decrease or increase in the number of operations found in the segment) in certain ranges [8]. In other words, we treat the feature vector of a predictive model as a point (we call it a model point) in the learning space. The points near the model point can also converge to the same *OSO* as the predictive model.

Our goal is to find the points that are within a close proximity of a model point, and to partition them into a cluster. To find these points, we developed the following partitioning algorithm:

Learning-Space Partition Algorithm

Step 1: Create two bound vectors V_L and V_U by means of the feature vector of the predictive model. The ith elements of V_L and V_U are $0.5m_i$ and $2m_i$ (m_i is the ith element of the feature vector of the predictive model), respectively.

Step 2: Using the bound vectors V_L and V_U, manually create two new program segments PI_L and PI_U. These two segments, in source structure, should be similar to the program segment that converges to the predictive model. The segments are then measured to find their *OSO*s. If no *OSO* is found or the *OSO* is not the same as the *OSO* of the predictive model for segment PI_L, each element of V_L will be adjusted by increasing it for a half of the original value. A new PI_L is created and again measured iteratively. Similarly, if no *OSO* is found or the *OSO* is not the same as the *OSO* of the predictive model for the segment PI_U, each element in V_U will be adjusted by reducing it for a half of the original value, and a new PI_U is created and measured iteratively.

Step 3: If the *OSO* is not the same as the *OSO* of the predictive model for the segment PI_L or PI_U, a new predictive model is created by the segment PI_L or PI_U and a new cluster is created by repeating the algorithm.

After step 2 is completed, the final bound vectors V_L and V_U create a cluster of learning space based on the feature vector of the predictive model. A cluster contains infinite number of instances. If both V_L and V_U are the same as the feature vector of the predictive model, the cluster

contains only one point, which is the predictive model itself.

Although only a few points are measured in a cluster, it is possible to assert that all instances have the same *OSO* if they belong to the same cluster. However, the formal proof is beyond the scope of the paper.

3.3.2 Mapping Program Segment to Cluster

The machine-learning algorithm tries to correlate an input program segment to a predictive model by assigning the feature vector of the input program segment to a cluster. That is, the algorithm determines which cluster includes the feature vector of the program segment. Once it finds the cluster that includes the program segment's feature vector, the *OSO* of the predictive model in the cluster will be applied to the program segment.

The program performance behaviors of an input program segment are first extracted and transformed into a feature vector. The machine-learning algorithm then searches the corresponding cluster using the feature vector. If each feature vector element is within the range of the corresponding bound vector of a cluster, the input program segment is considered an instance of this cluster. In other words, the cluster includes the feature vector of the program segment. Therefore, the predictive model of each cluster determines the *OSO* applied to the program segment.

Not all elements of the program segment's feature vector fall within the range of the same cluster. Instances that experience this scenario are assigned to a universal cluster *UC*, which has no specific predictive model assigned to it. These instances will be compiled using a default -*O*3 option.

4 Implementation in GCC Compiler

The integration of the presented methodology is fully transparent from the user's point of view. This section describes the required alterations to GCC compiler.

The complete modification is done in two steps: 1) a front-end integration of the predictive models and the machine-learning tool, after an intermediate representation is generated and the inlining has been finished, and 2) implementation of the program segment-based *OSO* selection in the optimization phase.

4.1 Implementation Principle

The input code is first transformed into the Abstract Syntax Tree (AST) intermediate representation in GCC's front-end. This is where the machine-learning algorithm is integrated. The compiler enters the machine-learning routine before it traverses the AST to generate the Register Transfer Language (RTL) representation. The machine-learning routine searches for the candidate program segments and records their positions in the compiled program.

During the candidate program segment searching phase, the machine-learning routine correlates each candidate program segment to a predictive model. Upon the completion of this phase, the compiled program is divided into a sequence of program segments. The program segments that do not correlate to any predictive model are assigned a default -O3 option. The predictive model correlated program segments are usually represented in the RTL format and are passed to the GCC's back-end.

Unmodified GCC uses fixed set of optimization options to optimize the whole compiled program. The modified GCC 'forces' the compiler to use a customized set of optimization options for each program segment. The set of optimization options for each program segment comes from the *OSO* of the corresponding predictive model. The candidate program segments search and the machine-learning phase are executed after the inlining has been performed in order to keep the program segment invariance.

4.2 Machine-Learning Implementation

In this section we present the implementation process of the machine-learning approach which correlates program segments to predictive models.

The machine-learning routine first extracts performance behaviors for each program segment in the compiled program. This is done in a single pass, which typically increases the compilation time for less than 1%. The extracted behaviors are quantified and translated into a feature vector. The machine-learning algorithm then determines the feature cluster of the observed program segments using the feature vector. The *OSO* of the predictive model of this cluster is used to compile the program segment. If the program segment's feature vector relates

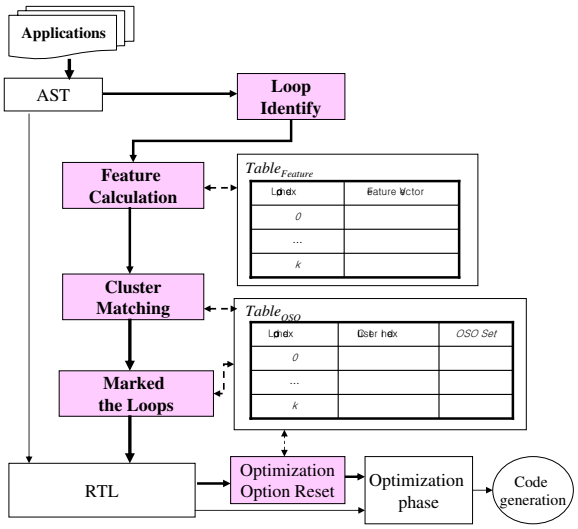


Figure 2: Flowchart of Machine-Learning Approach

to more than one cluster, it is considered a non matched segment.

The current implementation processes each loop structure in the program in the described way. A predictive model table is then used to map loop indexes (the order of the loop in the compiled program) to the order number of the correlated predictive model. A special predictive model order number 0 matches every loop structure and program segments which have no determined correlating predictive model.

Figure 2 shows the flowchart and the data structures used in the implementation of the machine-learning approach.

4.2.1 Feature Vector Creation

The procedure of identifying candidate program segment and the feature vector calculation is described as follows:

1. Traverse AST until a loop structure is determined.
2. For each loop structure,
   (a) Assign an index $Index_L$(initial 0) and initialize the predictive model record table $Table_{Feature}$ with 0
   (b) For each statement in the loop body, extract the information which contributes in the calculation of the program performance behaviors. Repeat until the end of the loop structure is reached.
   (c) Calculate the feature vector and store it to the $Table_{Feature}$.
3. Repeat the above until the end of AST is reached.

In the end, each item in the $Table_{Feature}$ represents a correlation between a program segment (loop index) and its corresponding feature vector.

4.2.2 Predictive Model Correlation

After calculating the feature vectors of all candidate program segments, the machine-learning algorithm attempts to match segments to clusters. The following procedure describes the matching process:

1. For each feature vector in $Table_{Feature}$
   (a) Compare the feature vector with the feature vectors of all clusters.
      i. If the feature vector is an instance of a cluster, correlate this segment to the predictive model of this cluster and store the corresponding OSO into $Table_{OSO}$.
      ii. If the feature vector is not an instance of any cluster, store 0 into $Table_{OSO}$
2. Repeat the above for all elements in $Table_{Feature}$

In the end, each item in $Table_{OSO}$ contains the correlation between each loop structure in the program (loop index) and a matched predictive model's OSO. $OSO = 0$ designates the loop structures that have no corresponding predictive model. The following optimization phase will replace 0 with -$O3$ for these segments.

4.3 Dynamic Program Segment Compilation

The optimization options supplied by GCC compiler are classified into three sets: basic optimization options B, global optimization options G, and local optimization options L. Our methodology only considers local optimization options L, because global optimization options force GCC to optimize the whole program instead of just the code segments. Considering G optimizations in our methodology would call for a complex analysis of GCC compiler. The basic optimization options B, typically activated by -$O1$ in GCC, do not contribute to our

Index	Optimization Option
o1	fforce-addr
o2	fmove-all-movables
o3	fprefecth-loop-arrays
o4	freduce-all-givs
o5	fsched-spec-load
o6	fsched-spec-load-dangerous
o7	funroll-loops
o8	funroll-all-loops
o9	fbranch-count-reg
o10	fpeephole
o11	fforce-mem
o12	strength-reduce
o13	fcse-follow-jumps
o14	fcse-skip-blocks
o15	freturn-loop-opt
o16	fsched-spec
o17	fpeephole2
o18	frename-registers

Table 2: GCC Local Optimization Options

method. Hence, we choose $B \cup G$ as a default set of options in the current implementation. That is, the options in $B \cup G$ are always active.

Table 2 lists the local optimization options L which is used in our method.

The original GCC optimization phase is modified by adding an additional step where the customized set of optimization options for each program segment is selected dynamically. The set of optimization options for each segment comes from the corresponding predictive model *OSO*.

In the GCC optimization phase, each option applies to the whole input program object (program or function). A customized set of options can not be set for program segment due to GCC internal architecture constrains. Alternatively, we dynamically turn on and off any option when it travels the input program object. That is, the application of an option o to a program component C depends on whether the option o is in the *OSO* of the program segment to which the program component C belongs.

This is the phase that limits the number of options applicable in our methods, and generates the majority of added compilation time. For example, in the current implementation, only 10 among the total of 18 local optimization options shown in Table 2 have passed the test.

If GCC compiler was able to do region-based processing instead of a whole program based processing, we could dynamically set the customized set of options in the begin of each region, because the program segments could be transformed into regions. This issue is addressed by the ongoing efforts.

4.4 Dynamic Program/Function Based Compilation

The presented method also covers the automated optimization option search applied to both program functions and to the whole programs. This implementation is integrated in GCC toolchain instead of GCC internal.

The whole program search is based on the dynamic option search applied to program functions only. The later is in turn, derived from a dynamic option search for program segments in the function.

There are four steps in the function based dynamic option search implementation:

Step 1: Identify all candidate program segments in each function found in the input program and search their *OSO*s. This step is the same in the segment based search.

Step 2: Assign a weight to each *OSO*s of a function. The weight is equal to the product between the number of times the same *OSO* appeared in the function, and the sum of the *OSO*'s feature vector elements.

Step 3: Each function is assigned the *OSO* with the maximum weight.

Step 4: Spill the input program into functions and involve GCC to compile each function with the selected *OSO*. The generated object files of these functions are then linked and executed.

The implementation of the program based dynamic option search is somewhat similar to the function based implementation, with certain exceptions. After determining the *OSO*, each function's *OSO* is assigned a new weight. The new weight is the sum of all function's weight in which the same *OSO* appears. The *OSO* with the maximum new weight is selected as the program's *OSO*.

This is just one of the possible implementations for function based and program based dynamic option search methods, which is not necessarily the most efficient one. We are aware of at least one important shortcoming:

the selected OSO may not be a dominant OSO for all program segments found in the function. In other words, some program segments may introduce negative performance speedup when compiled with this OSO.

Our research is partially focused with exploring alternative methods, which could better address the observed issues with the current implementation.

5 Experimental Results and Analysis

The current implementation supports 1) the Intel embedded XScale PXA255, and 2) a general-purpose Intel Pentium 4 platform running Linux. Both platforms use GCC 3.3.

5.1 Experimental Results

A total of 13 cases have been tested. The 13 cases are selected from 4 accredited benchmark packages - CommBench, Mediabench, Mibench and SPEC2000. The test cases and the predictive model training cases are coming from two separate suites. The training cases are tuned, while the test cases are unknown to the compiler. We believe that this makes the evaluation of our method fair and accurate.

The testing phase consists of the following 4 arrangements:

- **Baseline Test** This test searches for the best result for each test case, when the default GCC options are used such as -*O*1, -*O*2 and -*O*3. The best result is set as a baseline, and is compared to the performance achieved through applying the new methodology.
- **Program segment Based Test** This test evaluates our dynamic optimization options search method, during its application to program segments. This test is done automatically.
- **Function Based Test** This test evaluates the application of our method to program functions. Each function is compiled separately through its *OSO*. The individually compiled files are linked together and measured as a whole. This test step is currently done manually.
- **Whole Program Based Test** This test evaluates the application of our method to a whole program. The tested case is compiled solely through the corresponding *OSO*. This step is also done manually.

Figure 3-(a) shows the performance speedup results on a general-purpose Intel Pentium 4 platform running Linux and GCC 3.3. The presented results cover the performance speedup of the whole program based, the function based, and the program segment based approach over the best default GCC option (*baseline*), for each test case.

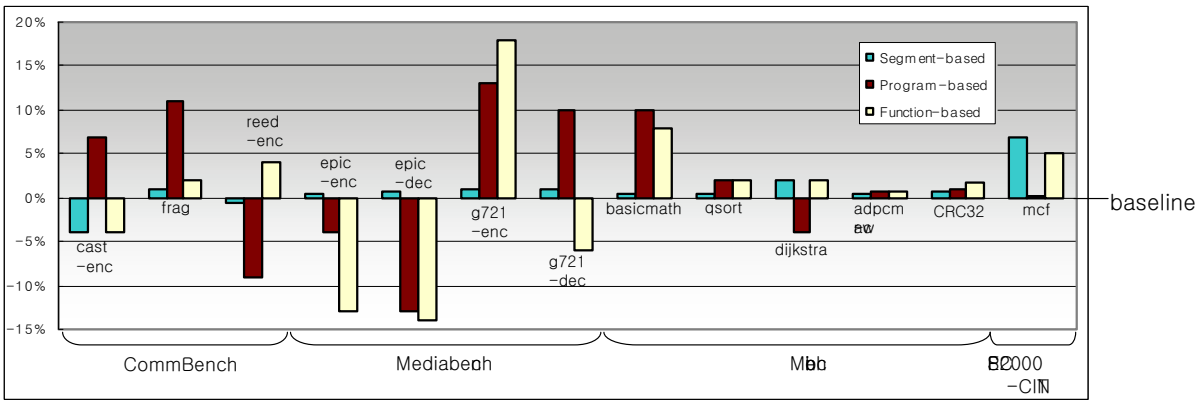
Figure 3-(b) shows the compilation overhead over -O3 default option in GCC 3.3, for the tested cases.

5.2 Analysis of Experimental Results

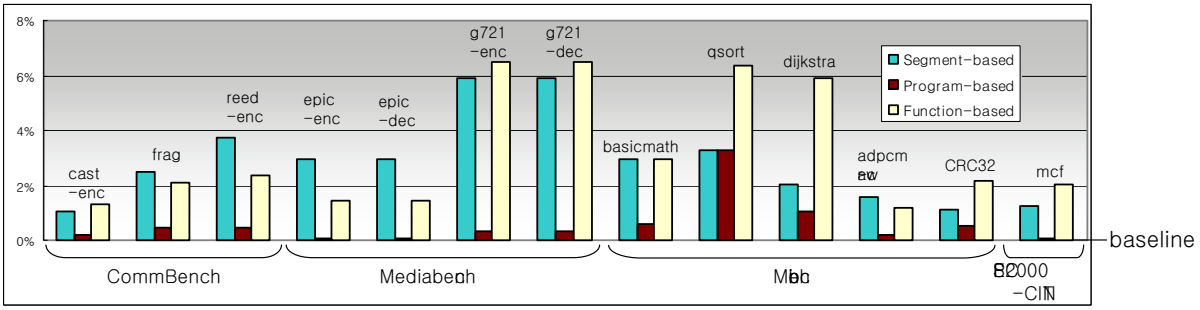
From Figure 3-(a), we find that the additional performance speedups over the best GCC default result for different program objects of: the whole program based, the function based, and the program segments based approach is on average 2.2%, 1.2% and 1.7%, respectively. The Maximum speedups are 13%, 18% and 7% respectively.

The program segment based option search performance is not as expected, and in few cases we even experience a negative speedup. We believe that this is due to not enough identified program segments, and due to the fact that we currently train only a small number of predictive models. The total of 11 program performance behaviors is currently used. However, the machine-learning space is very large. A higher learning performance would essentially require a lot more data. The second reason is that the execution time for the identified program segments is only a small fraction of the program execution time. Also, we have currently implemented only 10 local options in GCC, while the other 8 options are in the debugging phase. That is, the *OSO* for each segment is searched from a subset of the local option set *L*.

From Figure 3-(b), the compilation overhead over -O3 for segment based option search is on average 2.87%



(a) Performance Improvement over the best GCC default option



(b) Compilation Time Overhead over GCC –O3 option

Figure 3: Experimental Results on Pentium 4

(for program based and function based option search, the compilation overhead is on average 0.58% and 3.25%, respectively). The major part of the compilation overhead is due the dynamic option search phase. Limited by the structure of GCC 3.3, for every operation, we have to locate which segment does it belong to. We are confident that the compilation overhead can be reduced on average to less than 2% as we go deeper into the GCC structure.

The speedups found in function based option search show interesting phenomena. In some cases we experienced impressive speedups, while in others have negative speedups. If the speedup of a certain function is positive, the selected *OSO* from one of its program segments is one of the dominant *OSO*s for all other program segments in the function. We were not able to find a dominant *OSO*s for the functions that experience negative speedup. The presented test results show only an assertion that there is a dominant *OSO* for each function. However, how to find dominant *OSO*s remains an open topic.

The speedups of program based option search are relatively comparable to function based speedups. Therefore, any improvement in function based option search will consequently improve the program based option search.

6 Conclusion and Future Work

In this paper, we have described an automated optimization options search method, discussed certain implementation challenges, and presented preliminary experimental results.

The final goal of our work is to find the optimal or near-optimal set of GCC compiler inherent optimizations for an arbitrary application in an automated fashion. We believe that defining a proper methodology is only the first step toward reaching this goal.

Currently we are focused to the following tasks:

- Develop a new clustering strategy of partitioning the machine-learning space

- Study the relationships between the program segment performance behaviors and the underlying architecture features and develop a more precise description of the program segment performance behaviors.
- Improve the efficiency of the current non region based implementation of our method, so that the overhead of compilation is further reduced.
- Extend our method to apply multi-core architectures.

We believe that the current implementation meets the two initially set criteria: (1) achieve increased performance over the best default optimization setting, and (2) additional compilation time should be minimal and should not interfere with the overall GCC performance.

Acknowledgments

We sincerely thank our colleague Murat Bolat for his valuable role in developing the options combination search tool.

This project was made possible by the NSF awards CCF-0541002 and CNS-0509332, for which we are sincerely thankful.

References

[1] Francois Bodin Antoine Monsifrot and Rene Quiniou. A machine learning approach to automatic production of compiler heuristics. In *In Artificial Intelligence: Methodology, Systems, Applications, pp 41-50, 2002*.

[2] John Cavazos and M.F.P. O'Boyle. Method-specific dynamic compilation using logistic regression. In *OOPSLA'06, Portland OR, US, Oct. 2006*.

[3] M. Haneda, P.M.W. Knijnenburg, and H.A.G. Wijshoff. Optimizing general purpose compiler optimization. In *CF'05*, May 2005.

[4] Z. Pan and R. Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2006.

[5] Richard Stallman. *Using and Porting the GNU Compiler collection (GCC)*. Free Software Foundation, Inc., 2000.

[6] S. Kasif S. K. Murthy and S. Salzberg. A system for induction of oblique decision trees. In *Journal of Artificial Intelligence Research, 2:1-32, 1994*.

[7] Hai P. Wu, E. Park, Murat Bolat, Mihailo Kaplarevic, Ying P. Zhang, Xiao M. Li, and Guang R. Gao. An automatic methodology for program segment-based compiler optimization search. In *Technical Memo071, CAPSL, Unive. of Delaware, Nov. 2006*.

[8] Hai P. Wu, E. Park, Mihailo Kaplarevic, Ying P. Zhang, Murat Bolat, Xiao M. Li, and Guang R. Gao. Automatic program segment similarity detection in targeted program performance improvement. In *2007 Workshop on Performance Optimization for High-Level Languages and Libraries (POHLL07)*, March 2007.