University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

Automatic Program Segment Similarity Detection in Targeted Program Performance Improvement

Haiping Wu
Eunjung Park
Mihailo Kaplarevic
Yingping Zhang†
Murat Bolat
Guang R. Gao

CAPSL Technical Memo 72
December 30, 2006

†Digital Enterprise Group, Intel Corporation

Abstract

Targeted optimization of program segments can provide an additional program speedup over the highest default optimization level, such as -O3 in GCC. The key challenge is how to automatically search for performance sensitive program segments in a given code, to which a customized sequence of optimization compiler options could be applied.

In this paper we propose a method based on both program syntax and architecture-dependent behavioral similarity of program segments, which addresses the above challenge. First we create a program segment pattern database and a proxy segment template database which are built in the underlying compiler. The modified compiler identifies optimization-friendly program segments in input programs using the syntax structure similarity between the candidate program segments and the pre-build program segment patterns. The identified program segments are then filtered using the architecture-dependent behavior similarity to the pre-build proxy segment templates. These identified program segments are those pieces of input code, which can be custom optimized to improve the overall program performance.

The method is evaluated and tested on the Intel XScale PXA255 platform using randomly selected benchmarks. The experimental results show that our method can provide additional speedups over the highest optimization level in GCC 3.3 (-O3) for an arbitrary set of applications.

Contents

List of Figures

List of Tables

1 Introduction

Modern compilers often have myriad options to control various aspects and degrees of optimization. The users' ability to find a good combination of compiler options has a large impact on the overall performance of the compiled code. On the other hand, even the highest default optimization level, such as -O3 in GCC, does not necessarily produce the best performance speedup for every application [2, 6, 9, 11, 12, 19]. A significant potential for further performance improvement exists, which could be exploited by carefully choosing optimization options customized to a particular performance sensitive program segments [3].

The key challenge in compiler is how to automatically identify performance sensitive program segment where a customized sequence of optimization options could be applied. To address this challenge, we propose a methodology for automated detection of performance sensitive program segments based on program segment similarity.

From a compiler point-of-view, program segment similarity appears in two ways: 1) syntax structure similarity, and 2) architecture-dependent behavior similarity. Even when the same set of optimization options is used, two program segments with the same syntax structure may exercise different optimization effects. On the other hand, two program segments may be affected in the similar way, if they have similar architecture-dependent behaviors, even if when the corresponding syntax structures are different. This phenomenon complicates the analysis of program similarity.

We address this problem through a two-stage method, which combines the syntax structure similarity and the architecture-dependent behavior similarity.

We base our methodology on the following assumptions:

- In most programs, the performance sensitive program segments can be detected, which could contribute to an additional performance improvement if custom optimized during the compiling phase.
- A set of proxy program segments can be generated based on the performance sensitive program segments by evaluating their optimized sequence of optimization options (OSO)[1]
- The compiler can apply the OSOs of the proxy program segments to the program segments with the similar characteristics (syntax structures and architecture-dependent behaviors) to those of the proxy program segments.

According to these assumptions, we first create a program segment pattern database and a proxy segment template database which are built in the underlying compiler.

A program segment pattern is an abstract representation of an optimization-friendly program segment[2]. A program segment pattern represents an infinite number of program segment instances, which have similar syntax structure.

[1] Compiled with an optimized sequence of optimization options (OSO), a program can have an additional performance speedup over the highest default optimization level of a compiler.

[2] An optimization-friendly program segment is a performance sensitive program segment for which a customized optimization sequence of options(OSO) can be defined.

A proxy segment template is a cluster of the segment instances represented by a program segment pattern. A unique OSO is associated to each proxy segment template. A proxy segment template represents program segment instances, which have similar architecture-dependent behaviors, as well as similar syntax structures.

By building the program segment pattern database and the proxy segment template database, the compiler identifies candidate program segments in the input code using the syntax structure similarity between the candidate program segments and the pre-build program segment patterns. The identified program segments are then filtered using the architecture-dependent behavior similarity between the identified program segments and the pre-build proxy segment templates. In the final phase, the identified program segments are optimized using a customized sequence of optimization options.

In this paper we describe an automated performance-friendly program segments identification technique. We explore two strategies used to compare identified program segments and the proxy segment templates with associated weights: 1) accurate-match strategy, and 2) fuzzy-match strategy. In the accurate-match strategy, the inherent architecture-dependent behaviors in each segment's statement are used in the similarity comparison. The fuzzy-match approach uses the inherent architecture-dependent behaviors of the whole program segment in the similarity comparison stage.

We evaluate our method on the Intel XScale PXA255 platform with GCC tool chain, using a random selection of benchmarks. The experimental results show that our method can provide additional speedups over the highest optimization level in GCC 3.3 (-O3) for an arbitrary set of applications.

The paper sections are organized as follows: In Section 2, we present the method for creating program segment patterns and proxy segment templates; Section 3 discusses identification of optimization-friendly program segments; Section 4 presents the experimental results; Section 5 describes related work. Finally, the conclusion and the future work are described in Section 6.

2 Program Segment Pattern and Proxy Segment Template Creation

This section describes the approach for creating program segment patterns and proxy segment templates.

A program segment pattern is an abstract representation of an optimization-friendly program segment. Each program segment pattern represents an infinite number of program segment instances which have similar syntax structures.

A proxy segment template is a cluster of the program segment instances represented by a program segment pattern. A unique OSO is associated to each proxy segment template.

2.1 Program Segment Representation Data Structure

Our methodology completely relies on the static analysis of the programs subject to analysis. More precisely, we only consider statement syntax structures and the number of operations (operands and operators). The data structure used for representing program statements and the corresponding operations is designed to be intuitive, as shown in Figure 1.

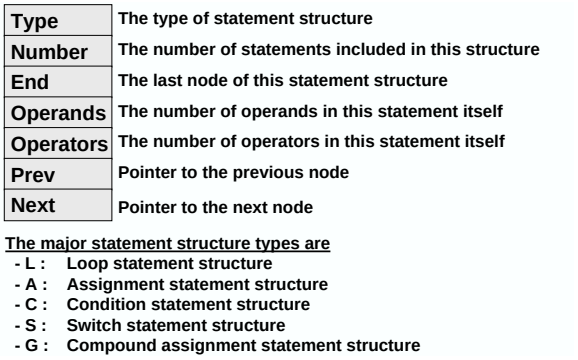

Figure 1: Data Structure used for Representing Program Segments

A double linked list with nodes (the structure is shown in Figure 1) is used to represent the input code. We call this structure the *program structure stream representation* (PSS) of a program. The nodes have the following properties:

- Each node can represent either just one program segment, or two or more different program segments that have the same program structure.
- Each node is accompanied by a data structure used for storing the program information of the represented program segment.

The format and the content of each data structure are related to the criteria of architecture-dependent behavior similarity discussed in Section 3.

One of the tools developed as a part of our methodology toolset transforms an input program into a program structure stream representation. This step is essentially a source code analysis where the program information is collected and recorded. Figure $2(b)$ gives the program structure stream representation for the program segment shown in Figure $2(a)$.

2.2 Syntax Structure Similarity

Definition 1 Two program segments S_1 and S_2 have similar syntax structure if the number of nodes, node types and the node order are the same in both program structure stream representations.

2.2.1 Simplification of the Program Stream Representation

Because we only focus on program segments, our methodology could be simplified so that the the focus is on loop segments which contain loop statements, condition statements, switch statements and assignment statements. A program stream representation is simplified to exclude all program segments not considered in the analysis.

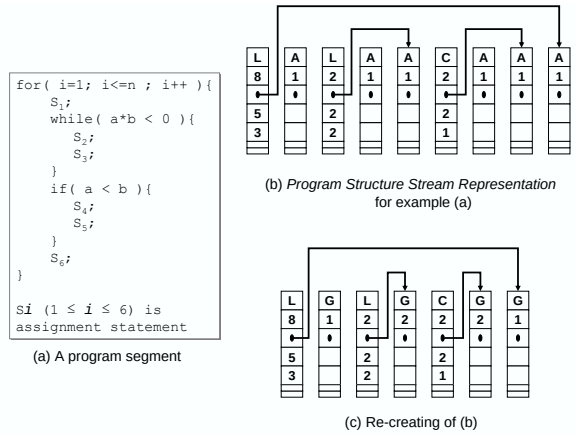

Figure 2: An Example of Program Structure Stream Representation

We first decompose the program segments that consist of nested loops. For each of these segment types, starting from the innermost loop, we extract the sequence of nodes which consist of this loop only. A new segment is created using the newly extracted sequence of nodes. This process continues until the outermost loop is reached (the one that represents the original segment). In the end, we leave this segment unchanged. In this process, a n-level nested loop will generate n-1 new segments.

The applied loop segment decomposition eliminates possible ambiguity among the similar program segments. Each similar program segment is unique and does not overlap with other program segments. After the simplification step is done, the program structure stream representation of a program contains program segments that may become syntax structure similar program segments.

The adjacent nodes of the same type are then combined to further simplify the program stream representation using the following criteria:

- A single or continuous assignment statement nodes are replaced with a special node which is called $compounded$ node, if they belong to the same level of the statement structure;
- A $null\ compounded$ statement node is inserted (the $Number$ field is 0) behind each non-$compounded$ statement node, if it is not followed by a $compounded$ node;
- Continuous conditional nodes are considered to be single conditional statement nodes if all nodes in their statement body are $compounded$ statement nodes. That is, we use a special conditional node to replace these continuous conditional nodes while keeping the original nodes;
- Continuous switch nodes are replaced by a special switch node.

Figure $2(c)$ shows the simplified program stream representation based on the representation in Figure $2(b)$. Please note that not all of the above mentioned nodes are used in this representation.

2.2.2 Similar Program Segment Detection Algorithm

The process of detecting syntax structure similar program segments is based on a very intuitive algorithm. The algorithm initially creates a pool of similar program segment (SPS-$pool$) to store detected similar program segments. Each element in this pool consists of a program structure stream representation and a similarity weight, which records the number of similar segments found during the detection process. For each tested program P, the algorithm detects similar program segments through the following steps:

Step 1: Program P is simplified into a set of marked program segments using the above described methods. Let us denote the set of program segments as P = $\{P_1, P_2, ..., P_n\}$;

Step 2: If the SPS-$pool$ is empty, do a self matching check for each segment in P. If two segments are similar, randomly remove one segment from P and increase the similarity weight of the remaining segment by 1. This step continues until no additional similar segment can be found. P is then stored in the SPS-$pool$.

Step 3: For each segment in P, check if any node sequence matches any element in the SPS-$pool$. This matching only checks for the node order, number of nodes and node type. If there is a match, a syntax structure similar segment is found and the similarity weight value of the matched element in the SPS-$pool$ increments for 1. The syntax structure similar segment is then removed from P. This iterative process finishes when no additional similar segment can be found. The remaining segments in P, if any, are self checked and stored in the SPS-$pool$.

The number of elements in the SPS-$pool$ depends on the number of analyzed sample programs.

2.3 Program Segment Pattern Formalization

The SPS-$pool$ has program segments for its elements. These segments can not be run independently, but rather they need to be translated into a compiler friendly version, by adding the $main$ function and the standard syntax garnishments. This "formalized" version is run and measured on the underlying platform to find the OSOs.

Not every measured program has an OSO. A trial and error process is used for each examined program in which the number of assignment statements and the number of operations (operands and operators) is adjusted in each assignment statement. If a corresponding OSO is not found after 10 iterations, the program is considered to have no valid OSO assigned to it.

If an OSO is found, the program segment affected by this OSO will be used as a representative program segment and formalized into a pattern.

A pattern consists of a "pure" program structure stream representation and a weight sequence of architecture-dependent behaviors. The difference between the pure program structure stream representation and the representative program segment structure stream representation is that the former only stores the information on node's type.

During the pattern formalization, we use the syntax structure skeleton of the representative program segment to represent all program segments that have the same syntax structure. Each pattern is accompanied with a weight sequence which is defined as:

$$\{\ \omega^1, \omega^2, \cdots, \omega^n\ \}$$

N_c: Number of used cases

N_p: Number of pattern found

N_c	N_p	Characteristics of the First 11 patterns			
		All	Min	Max	Frequency
20	17	847	1	182	<528,135,108,29,10,1, 18,10,6,1,1>
35	21	1822	1	230	<1114,374,208,29,8, 42,18,12,6,10,1>

Table 1: Experiment of Pattern Creation

The n in the sequence represents the number of $compounded$ statements that appear in the pattern. Let l be the number of non-$compounded$ statement structures in the pattern, $n = 2 * l - 1$. ω^i is a pair of values.

Each pair of values describes the range of architecture-dependent behavior value (discussed in Section 3) of the corresponding $compounded$ statement in a program segment represented by the pattern.

Definition 2 A pattern with k elements in its weight sequence is called a k-$element$ pattern.

Definition 3 In a k–$element$ pattern, the instance for which each weight element has the minimum value is called the *lower bound instance* of this pattern; the instance for which each weight element has the maximum value is called the *upper bound instance*; instances for which each weight element has an arbitrary value in the range of [*minimum value*, *maximum value*] are called *sibling instances*.

In a k-$element$ pattern, any program segment represented by the pattern can have k $compounded$ statements. The $i-$th pair of values in the weight sequence of the pattern describes the minimum value and the maximum value of the architecture-dependent behaviors for the $i-$th $compounded$ statement.

To create a pattern database, we randomly select two set of sample programs that feature large fraction of loop structures. The reason for selecting this type of cases is that the current patterns are created from the loop segments.

Table 1 shows the results after the creation of the syntax structure pattern from two sets of sample programs.

The first set of sample programs consists of 20 randomly chosen programs. Using the pattern formalization algorithm, 17 different patterns are created. After analyzing 11 most frequently recognized patterns, we found that: 1) 11 patterns appeared 847 times in the set of sample programs; 2) the minimum number of recognized patterns in the sample programs is 1; and 3) the maximum number of recognized patterns is 182. The $frequency$ field stores the number of appearances for each of the 11 patterns in the set of sample programs. The second set of randomly chosen 35 sample programs includes some but not every program found in the first group. When the algorithm is applied to the second group, the results show that a few more patterns are created. Although somewhat unexpected, the first 11 patterns, are the same as the ones found in the first set.

Figure 3 shows a set of program structure stream representations of patterns. We call this set a pre-build pattern database. Each pattern has been assigned a unique order number. This is a mini pattern database used only for the evaluation purposes. In practice, the pattern database should be large enough to cover most program segments having various syntax structures. The formation of a pattern database can be very time consuming, however it is practical to do so, because the pattern database will be repeatly used when analyzing different

programs.

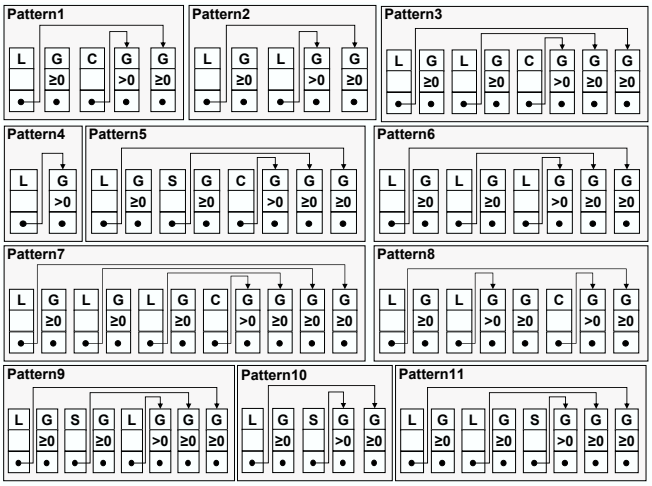

Figure 3: A Pre-Build Pattern Database

2.4 Creation of Proxy Segment Template

A proxy segment template represents a subset of all program segment instances represented by the corresponding pattern. The minimal value and the maximal value of the architecture-dependent behavior in the corresponding $compounded$ statement is determined by the corresponding pair of values in the weight sequence of the proxy segment template.

The differences between a proxy segment template and a program segment pattern are:

- The weight sequence of a proxy segment template is a subrange of the weight sequence of a corresponding pattern.
- Each proxy segment template has a unique OSO and all instances represented by the proxy segment template share this OSO.

The value range of the weight sequence of a proxy segment template is determined through clustering. For each pattern, the clustering algorithm groups all program segment instances that share the same OSO. The minimum weight value and the maximum weight value of the grouped instances consist of the corresponding pair of values found in the weight sequence of the proxy segment template.

The clustering algorithm is based on the following hypothesis:

Given a proxy segment template, if its lower bound instance and its upper bound instance have the same OSO, any sibling instance would have the same OSO.

Based on this hypothesis, the clustering algorithm divides the instance space of a pattern into a set of sub-spaces.

Pattern Index	Number of Created Templates	Performance Speedup(%)	
		Min	Max
1	4	2	25
2	2	4	24
3	2	10	14
4	2	7	13
5	2	3	25
6	2	9	22
7	2	7	13
8	2	3	15
9	2	9	17
10	3	1	15
11	2	1	15

Table 2: Characteristics of Pre-Build Proxy Segment Templates

The criterion that constraints the clustering is that the lower bound instance and the upper bound instance in a subspace should have same OSO.

We use mean value method to cluster the instance space of a pattern to create the proxy segment templates. Based on the pre-build pattern database shown in Figure 3, we create 25 proxy segment templates. Table 2 gives the features of the proxy segment templates derived from the corresponding pattern by the applied mean value clustering algorithm. This table shows the number of proxy segment templates derived from each pattern. The performance improvement field gives the minimal and the maximal performance speedup over the GCC -O3 among the proxy segment templates in each pattern.

There is only a few proxy segment templates derived from each program segment pattern. Clearly, having more proxy segment templates would result in a better performance speedup, but this does not affect the evaluation of the real benefits of our methodology.

3 Identifying Optimization-Friendly Program Segments

By building a program segment pattern database and a proxy segment template database, a compiler can automatically identify the optimization-friendly program segments in the processed code. The identification of optimization-friendly program segments is based on 1) the syntax structure similarity, and 2) the architecture-dependent behavior similarity.

The approach for detecting the syntax structure similarity is presented in Section 2. In this section, we focus on architecture-dependent behavior similarity. We first examine the correlation between architecture-dependent behaviors and program segments. Then we present a quantitative approach to transform the architecture-dependent behaviors of a program segment into a weight sequence. Finally, we present an added compiler technique for automatic identification of optimization-friendly program segments.

3.1 Architecture-Dependent Behaviors

In this paper we focus on the correlation between optimization-friendly program segments and the architecture-dependent behaviors. This correlation is best analyzed using a unique OSO as a criterion. Architecture-dependent behaviors are solely based on the operations found in program segments.

Given an optimization-friendly program segment S_i = $\{S_i^1, S_i^2, ..., S_i^m\}$, let B_i = $\{B_i^1, B_i^2, ... B_i^n\}$ represent the architecture-dependent behaviors of S_i, and O_i represent the OSO. The OSO correlation between S_i and B_i can be expressed by function Φ in the following way:

$$\Phi(S_i, B_i) = O_i$$

Let us use a concrete optimization-friendly program segment to investigate the correlation between this segment and the architecture-dependent behaviors.

Given an optimization-friendly loop segment L_0 as follows (it is manually examined to find an OSO):

```
for (i=var1, i<=var2; i++) {
      S1;
      S2;
      ......
      Sn;
  }
```

Si(i=1,...,n) is an assignment statement. Let us assume the OSO is O. We randomly duplicate Si in the loop body and adjust the operands and operators (add or delete) found in the statements one by one, so that we can experiment when the sequence O will change during this process. We observed that the same *O* will hold for several duplication/adjustion steps. This experiment disclosed that a number of different instances of the same program structure (a loop structure in our experiment) share the same OSO.

From the beginning until the last step, for which the *O* is valid in the duplication/adjustion process, the original segment L_0 is transformed into a series of new segments L_1, L_2, ..., L_{last}. Similarly, the architecture-dependent behaviors B_{L_0} are transformed into a set of B_{L_1}, B_{L_2}, ..., $B_{L_{last}}$. For these segments, the following equation holds:

$$\Phi(L_i, B_{L_i}) = \Phi(L_j, B_{L_j}) \ (0 \leq i, j \leq last \text{ and } i \neq j)$$

According to Definition 1 given in Section 2.2, these segments have the same syntax structure.

Based on the experiment observations, the architecture-dependent behavior similarity is defined as:

Definition 4 Let B_1 and B_2 are two set of the architecture-dependent behaviors for program segments S_1 and S_2, respectively, S_1 and S_2 have similar architecture-dependent behavior if:

$$\Phi(S_1, B_1) = \Phi(S_2, B_2)$$

The similar architecture-dependent behavior program segments have the following features: 1) they are optimization-friendly program segments, 2) they have similar syntax structure, and 3) they share the same OSO.

3.2 Quantifying the Architecture-Dependent Behaviors

In this paper, we do not investigate how a compiler can speedup a program segment by fully utilizing the set of the architecture-dependent behaviors. The reason is that our method captures program segments and relates them to a set of proxy program segments. These proxy program segments are manually selected and examined. Indeed, we only need to investigate the correlation between a candidate program segment and a proxy program segment based on the architecture-dependent behaviors. We trust the compiler to fairly process the candidate program segment by the same criteria used in processing proxy program segments. Simply said, a fair judge will draw a verdict guided by the same criteria used in the previous similar cases.

We have addressed the problem of determining the syntax structure similarity between program segments in Section 2. In this section we explain how to determine that two program segments with similar syntax structure also have similar architecture-dependent behavior.

To simplify the analysis we will only focus on the following architecture-dependent behaviors:

- Instruction latency
- Memory access latency
- Register set
- Data cache and instruction cache

Each type of architecture-dependent behavior is translated into an integer value according to the architecture-dependent characteristics. Table 3 describes the correlation between the architecture characteristics and the quantified values.

The analysis of program segment syntax components can estimate the total integer value for each architecture-dependent behavior. The translation of an architecture-dependent behavior into an integer number involves the following steps:

- Estimate the total instruction latency in the program segment. Assume each operation in the program segment is transformed into a related instruction. Simply collect and classify all operations and calculate the total number of instructions. The total number of instruction latencies is calculated using the architecture characteristics lookup table.
- Estimate the total number of memory access points in the program segment. We only look for the load/store operands. A variable is treated as a store operand if it appears in the left side of an assignment statement. Otherwise the variable is a load operand. If a variable appears in the left side of several assignment statements in the program segment, it is only counted as one store operation. Through the use of the architecture characteristics table, we can calculate the total number of memory access latencies for each segment.
- Estimate possible register allocation for the operands found in the segment. Assume n to be the number of registers. The first n load variables that appear more times than other variables, if they do not appear in the

Architecture Characteristics	Quantified Value
Multiply (short)	3
Multiply (long)	6
Multiply-ADD (short)	3
Multiply-ADD (long)	6
Compare	1
Move	1
Arithmetic	1
Logical	1
Shift/rotate	1
Branch	1
Load	6
Store	6
Register Set	14
Data cache	8*1024+512(mini data cache)
Instruction cache	8*1024

Table 3: Intel XScale Architecture Characteristics and Quantified Value

left side of assignment statements, they are counted as load operations - the first time they appear. That is, these variables are assigned to the registers. Therefore the succussing appearances of these variables are treated as register variables. We assume there is no latency for register variables. Other variables are still treated as memory access operands no matter how many times they appear in the code.

- Estimate the correlation between the total number of operations, the data cache size, and the instruction cache size. Since we have estimated the total number of instructions and operands in a program segment, the number of instructions of the program segment to the instruction cache size ratio can be calculated. The same can be done for the number of operands to the data cache size ratio.

Based on the above procedures, program segment architecture-dependent behaviors have been transformed into integer values. The architecture-dependent behavior similarity of program segments is determined based on these values.

3.3 Optimization-Friendly Program Segments Detection

In this section, we describe an automated compiler technique for optimization-friendly program segments detection based on the pre-build pattern database and the proxy segment template database.

We have modified compiler's front-end so that the detection is executed in an automated fashion. An extra pass is added in the compiler to transform the input program into program segments using the *program structure stream representation* data structure. For each program segment, the modified compiler first does syntax structure similarity analysis to determine which program segment pattern represents the current program segment. In the next step, the architecture-dependent behaviors of this program segment are collected and translated into integer

weight values based on the weight sequence of the matched program segment pattern. The modified compiler then searches the proxy segment template database that corresponds to this pattern using a concrete weight sequence. If the program segment weight sequence is a child of the weight sequence of a proxy segment template, this program segment is considered to be the match and a flag is set which triggers the modified compiler to custom optimize this program segment.

The modified compiler identifies optimization-friendly program segments in two stages:

- Stage 1: Syntax Structure Pattern Matching and Weight Sequence Creation
- Step 2: Proxy Template Matching

Figure 4 shows the process of optimization-friendly program segments detection in the input code.

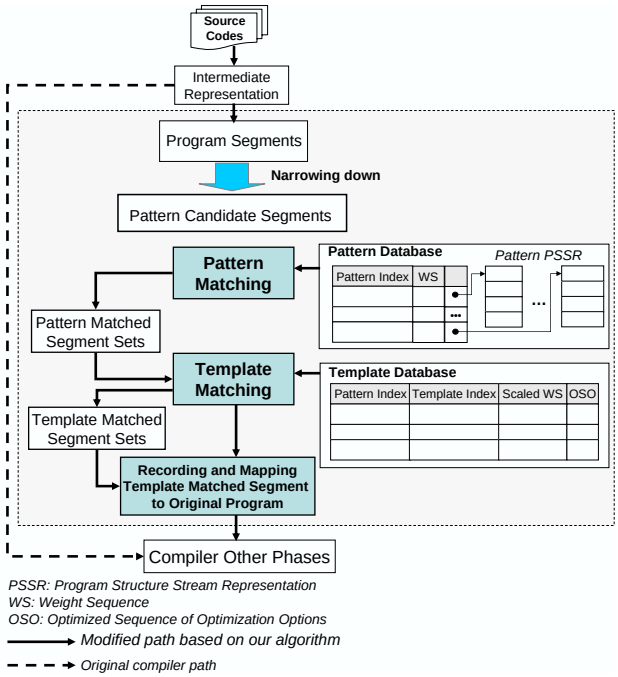

Figure 4: Optimization-friendly Program Segments Detection Flowchart

We use two strategies to check for the match between an identified program segment and the proxy segment template using a weight sequence: 1) an accurate-match, and 2) a fuzzy-match.

In the accurate-match strategy, the inherent architecture-dependent behaviors in each segment's statement are used in the similarity comparison. The fuzzy-match approach uses the inherent architecture-dependent behaviors of the whole program segment in the similarity comparison.

In the accurate-match strategy, each element of the program segment weight sequence must be in the range of the corresponding proxy template's weight sequence. This is a very strict condition. The advantage of the

accurate-match strategy is that it can find a true proxy segment template if the match is successful. The shortcoming of this strategy is that it needs a large proxy segment template database in order to cover majority of the weight values combinations.

The fuzzy-match strategy adds all elements of the weight sequence into a weight value and determines if this value is in the range of the weight value of a proxy segment template. Obviously, the prerequisite is that, when creating a proxy segment template, the weight sequence of this proxy segment template deteriorates to a single element.

No matter how many optimization-friendly segments can be recognized in the input code, the amplitude of the performance improvement depends on the three factors:

- A similar segment recognition procedure must be applied after of all program transformations, such as function inline, loop transformation, etc.
- A compiler supports ”region-based” optimization mechanism. That is, a compiler repartitions compilation function into more optimization and scheduling friendly compilation units.
- The optimization-friendly segments are targeted ”hot-spots”.

4 Experiment

Our methodology is applicable to any platform, and should provide improved performance for a randomly chosen input C program. In this section, we evaluate the methodology on the Intel embedded XScale PXA255 architecture. The creation of the pattern database and the proxy template database is architecture dependent but it is done only once. In the experiment, we select 4 benchmark packages - CommBench, DSP kernel suite, Mediabench, Mibench (both large and small input data set are used in the experiment).

The experiment evaluates the following characteristics:

- Pattern and template match rates through: 1) The number of program segments that can be identified, and 2) How many pattern matched segments also match proxy templates? The results help estimate the pattern database size and the proxy template database size.
- Performance speedup. In this part of the experiment we test the ability to automatically identify optimization-friendly program segments.

The benchmark programs are first compiled using the -O3 option in GCC 3.3. Only those cases that pass the execution (some of them cannot compile or execute correctly) are used in the testing phase. Each tested program is run **5** times, and the average performance speedup is used as a reference in the further analysis.

4.1 Pattern and Template Match Rate Evaluation

Table 4 shows the match rates for the candidate program segments (matches to the program segment patterns and matches to the proxy segment templates). We recorded the total of 3514 candidate program segments,

Template Matching Strategy	Number of Candidate Segments	Number of Pattern Matched	Number of Template Matched
Accurate-match	3514	594	441
Fuzzy-match	3514	594	516

Table 4: Pattern Matched Rate and Template Matched Rate

where 594 match patterns. The match rate is 16.9%. Among these pattern matched segments, 441 and 546 segments are proxy segment template matched when using the accurate-match strategy and the fuzzy-match strategy, respectively. The proxy segment template match rates are 74% and 87%, respectively.

4.2 Performance Speedup Evaluation

Among 30 tested cases originating from 4 benchmarks, there are 9 cases that show performance speedup over -O3. We believe that the improvement would be signifi cantly better if there were more proxy segment templates to compare against. Our group continuously work on extending the proxy segment template database.

Figure 5 gives the performance speedups over the default -O3 option among 9 programs using the accurate-match strategy and the fuzzy-match strategy, respectively. The average speedup is 1% and the maximum speedup is 3% for the accurate-match strategy. The fuzzy-match strategy has the average speedup of 2% and the maximum speedup of 7%.

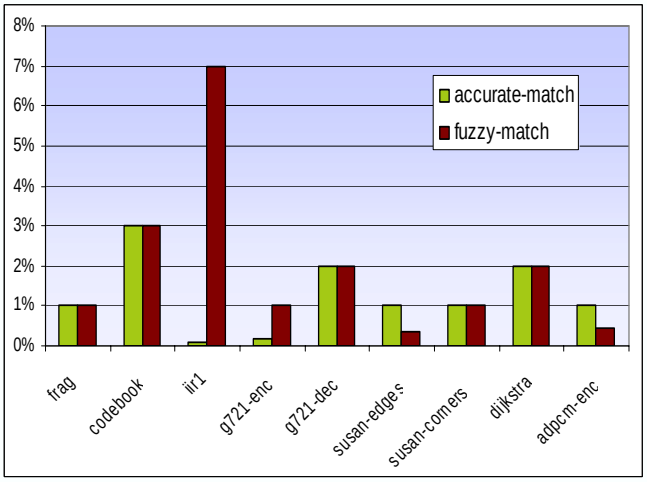

Figure 5: Performance Speedup over GCC -$O3$

4.3 Analysis and Estimation

Based on the experimental results, the fuzzy-match strategy shows better performance speedup than the accurate-match strategy, as well as a higher proxy segment template matching rate. This means that the detection of

program segment similarity is better if program segments coarse-grain architecture behaviors are observed, instead of the fine-grain.

The architecture behaviors of some statement components may be combined with other statement components by the compiler optimization. Therefore the weight values of statement component in two similar program segments may be completely different. On the other hand, the weight values of architecture behaviors of two similar program segments will be similar no matter how the compiler optimizes them.

Finally, there is no significant performance speedup observed for randomly selected programs. Besides the fact that the small fraction of segments are identified based on our experimental proxy segment template database, we have to ignore certain optimization opportunities, when the optimization sequence for this code relates to the outside codes. The solution to this problem is also a part of an ongoing effort to improve the methodology results.

5 Related Work

Sree et al. [13] presented a method for architecture resources management through compiler analysis. According to this method, a compiler is used to identify frequently executed code regions, which are then directed to a memory system. A memory system is used to prioritize access to data and instructions from these regions. Tom et al. [15, 16, 17] presented a region formation algorithm which eliminates high compiler-time memory costs caused by aggressive inlining pre-passing. This method is based on individual regions processing, which takes place during inter-procedural region formation. This work describes an architecture-independent compilation mechanism. Liu et al. demonstrated a region-based compiler infrastructure. In contrast to these studies, our method completely relies on static information gathered from programs subject to analysis. In our method, only a compiler is required to do an ”one-size-fits-all” optimization for the proxy program segments.

Annavaram [10] and Lau et al. [5] discuss the correlation between program source code and performance. They examined the use of code signatures obtained through periodic sampling to predict performance for database applications and SPEC2000. Hoste et al. [1, 7] propose a methodology to predict program performance on any architecture. They measure the architecture-independent characteristics of programs and then relate measured information to pre-profiled benchmarks. In contrast to these studies, our method employs different criteria to determine program similarity. Furthermore, our method can be used not only to identify similar program segments, but also to direct compiler to generate a custom highly optimized sequence of optimization options, which best fits each detected programs segment.

Kontogiannis et al. [8] developed code-to-code matching techniques for detecting code clones and for estimating similarity distances between two program segments. They used abstract syntax tree (AST) as the program representation scheme. Baxteret et al. [4] use standard parse analysis techniques to detect exact and near miss clones over arbitrary program segments in a program source code by transforming source code into an AST. Ducassw et al. [14] use simple line-based string matching to detect duplicated code. Lee and Hall [18] developed a tool (Code Isolator) to extract ”hot spot” program segments from large scientific applications. This tool also targets tuning to isolated segments. What makes our detection method different is that it operates in terms of program structure. A source code is transformed into a program structure stream representation and each program structure is associated to its weight sequence.

6 Conclusion and Future Work

We propose an automated methodology for performance-friendly program segments detection based on similarity between identified program segments and the records stored in two pre-built databases: 1) segment pattern database, and 2) proxy segment template database.

First we generate an experimental program segment pattern database and a proxy segment template database. We present a compiler technique to automatically capture the performance-friendly program segments detected in arbitrary input programs by the use of syntax structure and architecture-dependent behavior similarity analysis.

We evaluate the applicability and performance of our methodology on Intel XScale PXA255 platform by integrating it into GCC 3.3 compiler. The experimental results show that our method can provide additional performance improvement over the highest optimization level in GCC 3.3 (-O3) for an arbitrary set of applications.

Several research topics are raised based on the observations and the analysis of the experimental results. We focus our ongoing work on some of those observations:

- Build the practical program segment pattern database and proxy segment template database by extending the range of the statement type of a *program structure stream representation* data structure, to represent more statement components.
- Develop a more accurate proxy segment template clustering algorithm.
- Revise compiler so that it can fully support our methodology.
- Test our approach over a larger set of randomly chosen applications.

Acknowledgments

We wish to acknowledge our sponsors from DOD, DOE(Award No. DE-FC02-01ER25503), and NSF(Award No. CCF-0541002 and CNS-0509332).

References

[1] A. Phansalkar, A. Joshi, L. Eeckhout and L. K. John. Measuring program similarity: Experiments with spec cpu benchmark suites. In *Performance Analysis of Systems and Software*, 2005.

[2] K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Workshop on Languages, Compilers, and Tools for Embedded Systems*, May 1999.

[3] H P. Wu, L. Chen, J. Cuvillo and G. R. Gao. A user-friendly methodology for automatic exploration of compiler options. In *The 2006 International Conference on Programming Languages and Compilers*, Las Vegas, US, June 2006.

[4] Ira D. Baxter and Andrew Yahin and Leonardo M. De Moura and Marcelo Sant'Anna and Lorraine Bier. Clone detection using abstract syntax trees. In *ICSM*, pages 368–377, 1998.

[5] J. Lau, J. Sampson, E. Perelman, G. Hamerly and B. Calder. The strong correlation between code signatures and performance. In *IEEE International symposium on Performance analysis of systems and Software*, 2005.

[6] K. Chow and Y. Wu. Feedback-directed selection and characterization of compiler optimizations. In *2nd ACM Workshop on Feedback-Directed Optimization (FDO)*, Haifa, Israel, November 1999.

[7] K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L. K. John and K. D. Bosschere. Performance prediction based on inherent program similarity. In *Proceedings of PACT2006*, 2006.

[8] K. Kontogiannis, R. DeMori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. In *Automated Software Engineering*, pages 3(1–2):77–108, 1996.

[9] L. Almagor, K. D. Cooper, A. Grosul, T. J. Harvey, S. W. Reeves, D. Subramanian, L. Torczon, and T. Waterman. Finding effective compilation sequences. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 231–239, 2004.

[10] M. Annavaram, R. Rakvic, M. Polito, J. Bouguet, R. Hankins and B. Davis. The fuzzy correlation between code and performance predictability. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO-37)*, 2004.

[11] M. Haneda, P.M.W. Knijnenburg and H.A.G. Wijshoff. Automatic selection of compiler options using non-parametric inferential statistics. In *Proceedings of the 14th international conference on Parallel architectures and compilation techniques*, 2005.

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

[13] R. Sree, A. Settle, I. Bratt and D. Connors. Compiler-directed resource management for active code regions. In *In Proceedings of the 7th Workshop on Interaction between Compilers and Computer Architecture*, February 2003.

[14] Stéphane Ducasse and M. Rieger and S. Demeyer. A language independent approach for detecting duplicated code. In *Proceedings ICSM'99 (International Conference on Software Maintenance)*, pages 109–118, 1999.

[15] T. Way and L. L. Pollock. A region-based partial inlining algorithm for an ilp optimizing compiler. In *The 2002 International Conference on Paral lel and Distributed Processing Techniques and Applications*, page 552556, 2002.

[16] T. Way and Lori Pollock. Evaluation of a region-based partial inlining algorithm for an ilp optimizing compiler. In *IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2002)*, 2002.

[17] T. Way, B. Breech and L. L. Pollock. Region formation analysis with demand-driven inlining for region-based optimization. In *In Conference on Paral lel Architectures and Compilation Techniques (PACT)*, page 2436, 2000.

[18] Yoon-Ju Lee and Mary Hall. A code isolator: Isolating code fragments from large programs. In *Proceedings of the LCPC'04*, Sept. 2004.

[19] 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.