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

An Automatic Methodology for Program Segment-based Compiler Optimization Search

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

CAPSL Technical Memo 71
November 14, 2006

†Intel Corporation

Abstract

The highest optimization level of a compiler, such as -O3 in GCC, does not ensure the best performance for every application. Rather, a set of carefully tailored compiler options is more likely to achieve an additional performance improvement for most programs. Furthermore, within the same program, some code segments may run faster when individually tuned compiler options are applied. The goal is to explore and use the potential of these options in order to gain an additional program performance improvement. The main challenge in reaching this goal is: *Given an arbitrary program, how to identify and process such program segments that may benefit from individually tuned compiler options?*

In this paper, we propose a methodology that automatically identifies program segments that may be positively affected by individual option tuning, and determines the best compiler optimization options for such segments. The concept behind our methodology is based on the following observations: 1) A program can be divided into segments which share similar program structure; 2) A large number of program segments are different instances of the same program structure; 3) A group of different instances of the same program structure shares the same optimized[1] sequence of optimization options; 4) A compiler can be easily modified to identify performance-sensitive program segments and to apply customized sequence of optimization options to such segments, and 5) A significant performance gain can be achieved by using only a small number of program features recognized and treated by the proposed methodology.

Our methodology describes an automated mechanism for offline dynamic program segment-based compiler optimization selection. It automatically identifies program segments that could be further optimized, and applies a customized sequence of optimization options to each of such program segments.

We have implemented this methodology over a standard GCC compiler on two experimental testbeds: Intel XScale PXA255 and Intel Pentium 4 platform. Preliminary experimental results show that, compared to the highest optimization level in GCC, our methodology significantly improves applications performance over a wide selection of benchmarks, including SPEC2000 and MediaBench on the two experimental testbeds. The average speedup is 6.0% on PXA255, and 8.1% on Pentium 4. Furthermore, our approach improves performance for 33 programs from a total of 42 in 4 benchmarks on PXA255, and 34 programs from a total of 49 in 5 benchmarks on Pentium 4 respectively.

The performance gains achieved by our approach do not appear to be affected by different input program size, as observed in the experiments discussed in this article.

[1]Term *optimized* is used to indicate an additional performance improvement over the highest optimization level of compiler.

Contents

List of Figures

List of Tables

1 Introduction

It has been known that even the highest optimization level of a compiler, such as -O3 in GCC, will not necessarily produce the best performance for every application. A significant potential for performance gains exists, which could be exploited by carefully choosing optimization options customized to a particular piece of code.

In order to take full advantage of custom selected compiler optimization options, the following question has to be addressed:

Given an arbitrary program, how to identify and process such program segments that may benefit from individually tuned compiler options?

Ideally, if a compiler did not apply a fixed sequence of optimization options to the whole program, it would be possible to identify performance sensitive segments in the code where a customized sequence of optimization options could be applied, leading to a performance improvement. A performance improvement of each program segment would contribute to the final performance improvement achieved for the whole program. A number of tools and studies have been developed on how to optimize compiler options [1, 3, 5, 7, 9, 11, 12, 15, 16].

However, to the authors' best knowledge, no method has been proposed, which can be universally applied to an arbitrary application.

This paper describes what we believe is a solution to this problem, and a comprehensive study that lead to this solution. We first give the motivation for the study by experimentally investigating the relationship between program segments and various optimization options.

Given a loop segment *L* as follows:

```
for (i==var1, i<=var2; i++) {
      A1;
      A2;
      ......
      An;
  }
```

Where Ai(i=1,...,n) is an assignment statement. In the first step of the experiment we try to find an optimized sequence of optimization options. We set n=1 and customize A1 until an optimized sequence ***S*** is found. Then we add more statements in the loop body one by one to see when the sequence ***S*** will respond to the statement adding process. We observed that the same ***S*** holds for several steps. This experimental step explores an observation that if the number of statements in the loop body belongs to a certain range, they share the same optimized sequence of optimization options. Based on the last statement adding step for which the sequence ***S*** is valid, we continue the experiment by randomly adjusting the operands and operators (add or delete) found in the statements. As predicted, the same sequence ***S*** is valid for a large number of adjustments. This experimental step reveals that a number of different instances of the same program structure (a loop structure in this experiment) share the same optimized sequence of optimization options.

The following are the observations made from the above experiment:

- **Observation 1:** A program can be divided into segments each of which represents an instance of some set of program structures (See Section 5.1);
- **Observation 2:** A large number of program segments are different instances of the same program pattern (See Section 5.3);
- **Observation 3:** A group of different instances of the same program structure shares the same optimized sequence of optimization options (See Section 5.4).

Based on our empirical knowledge in compiler development, we present two more observations:

- **Observation 4:** Most compilers can be modified easily to identify performance-sensitive program segments and apply customized sequence of optimization options to those identified segments (See Section 6);
- **Observation 5:** Only a small number of program features recognized in the mining process is sufficient to achieve significant performance gains (See Section 7.2).

Motivated by these observations, we propose a new empirical methodology for automatic exploration of compiler optimization options for program segments in an arbitrary application, which we call Program Segment-based Optimization Search (*Program-SOS* or SOS) Methodology. This Methodology describes an automated mechanism of offline dynamic program segment-based compiler optimization selection. It automatically identifies program segments that could be further optimized, and applies customized sequence of optimization options to each of the identified program segments.

The main contributions of $Program$-SOS Methodology to the field of compiler options optimization are:

- We have built a platform-independent pattern database based on the analysis of a large set of empirically selected sample programs. Similar program segments are collected via pattern creating technique. These segments are measured and some of them are removed if there is no optimized sequence of optimization options that affect them. Each remaining segment is then transformed into a *pattern*, and each patterns is stored into the pattern database.
- We have built a platform-dependent template database. A clustering algorithm is applied to cluster the instance space of each *pattern* into a set of subspaces (*templates*) under the condition that all the instances in the same subspace share the same optimized sequence of optimization options;
- We have enhanced the GCC compiler by embedding the pattern/template database and a template identification tool. For an input program P, the identification tool captures and records a set of program segments in P by matching them structurally against the records in the pattern database. The set of identified segments is further reduced by the filtering process where the contents of the candidate segments are compared to the predefined "features" contained in the corresponding template. Each remaining segment matches a concrete template in the template database.
- We have revised the original compiler code to apply the template customized optimized sequence of optimization options to the program segment that is recognized to match this template.

We have implemented the Program-SOS Methodology over the GCC on two experimental testbeds: Intel XScale PXA255 and Intel Pentium 4 platform. Compared to the highest optimization level in GCC, our approach significantly improves performance over a wide selection of benchmarks, including SPEC2000 and MediaBench on the two experimental testbeds. The average speedup is 6.0% on PXA255, and 8.1% on Pentium 4. Furthermore, our approach improves performance for 33 programs from a total of 42 in 4 benchmarks on PXA255, and 34 programs from a total of 49 in 5 benchmarks on Pentium 4 respectively. The performance gains achieved by our approach do not appear to be affected by different input program size, as evidenced by the experiments discussed in this article.

The remaining sections of this paper are organized as follows. We first summarize related work in Section 2. Then we outline the infrastructure of our methodology in Section 3. Section 4 talks about the challenges associated to search for optimized sequences of optimization options. In Section 5, we discuss methods and tools used in detecting similar program segments, formalizing pattern abstract representations and clustering pattern instances into templates. In Section 6, we discuss methods for identifying and recording the program templates and the program segment-based optimization process. Experimental results are given in Section 7. Finally, the conclusions and future work are described in Section 8.

2 Related Work

The Methodology described in this paper is focused on the following three areas: 1) program source manipulation, 2) pattern instance clustering and, 3) optimization option selection.

Kontogiannis et al. [6] developed code-to-code matching techniques for detecting code clones and for measuring the similarity distance between two program segments. They used abstract syntax tree (AST) as the program representation scheme. Baxteret et al. [4] used standard parse analysis techniques to detect exact and near miss clones over arbitrary program segments in program source code by transforming source code into an AST. Ducassw et al. [13] used simple line-based string matching to detect duplicated code. Lee and Hall [14] developed a tool (Code Isolator) to extract ”hot spot” program segments from large scientific applications and targets tuning specially on the isolated segments.

In contrast to these studies, our program detection method operates in terms of program structure. The source code is transformed into a program structure stream representation and each program structure is associated to a weight vector. We define similarity between two program segments based on their structure similarity and through the distance between their weight vectors.

Almagor et al. [7] described an experimental study of compiler optimization sequences in a prototype adaptive compiler designed for the SPARC architecture. The paper presents various algorithms for determining the effective ordering of compiler optimizations. This approach requires a custom compiler which supports changing orders of optimization options. Our approach can be applied to an arbitrary compiler and does not change the internal order of optimizations.

Chow and Wu [5] used a fractional factorial design methodology to decide which compiler options should be used for a given application.

Cooper et al. [1, 7] used Genetic Algorithms to find an optimal sequence of optimizations. Pan et al. [15] presented the approach that does not change compiler internal structure. This group proposes three algorithms

aimed at finding compiler optimizations with a positive effect. The proposed algorithms select from the 38 optimizations used in the GCC -O3 option. However, this approach does not consider the effect of any individual option. Moreover, they also do not examine interactions between individual options.

Haneda et al. [8, 9] used an Orthogonal Array to determine the effect of optimization options. In contrast to the algorithm described in [9], our algorithm employs a different classification of compiler options and searches for a set of optimized sequences of optimization options.

Hammer et al. [2, 10] presented method of using Logical Analysis of Data to detect structural information about dataset. It looks to cluster areas of vector space by creating boolean criteria. In contrast to the method proposed in [2], we use a method of mean vector weight values to cluster similar program segments into templates. The clustering factor is based on the similarity of optimization sequence, which is a unique property of our Methodology.

3 $Program$-SOS Methodology Overview

This section presents a component view of the $Program$-SOS Methodology.

3.1 $Program$-SOS Infrastructure

The $Program$-SOS Methodology consists of three components, as shown in Figure 1. Component P_1 is composed of a group of sample programs, a program transformation tool, similar program segment detection tool, and a pattern formalization tool. The sample programs are randomly chosen from real world applications. They are used in $Program$-SOS to drive our pre-build pattern database. Component P_2 consists of pattern instance analysis and clustering algorithms. In this component, pattern instances are clustered and the templates are extracted from some of the clusters. This component also creates our pre-build template database. Component P_3 consists of template identification tools, which identify, record and optimize those program segments that match the pre-build templates. This component is integrated into the compiler.

3.2 The Working Principle of the $Program$-SOS Methodology

Component P_1 and Component P_2 are one-size-fits-all parts in the $Program$-SOS Methodology. Component P_1's responsibility is to find a set of similar program segments, to measure these segments using the option search algorithm and to formalize these segments into a set of patterns. Each of the sample programs is first transformed into a program structure stream representation by the program transformation tool. The program structure stream representation of a program is fed into a similar program segment detection tool. The tool captures similar segments and stores them into an internal pool of similar program segments (SPS-$pool$). After all of the sample programs are processed, the SPS-$pool$ stores all similar program segments that have been detected.

The similar program segment, which has an optimized sequence of optimization options, is stored into the pattern candidate pool. The pattern formalization tool transforms each element in the pattern candidate pool into a pattern representation.

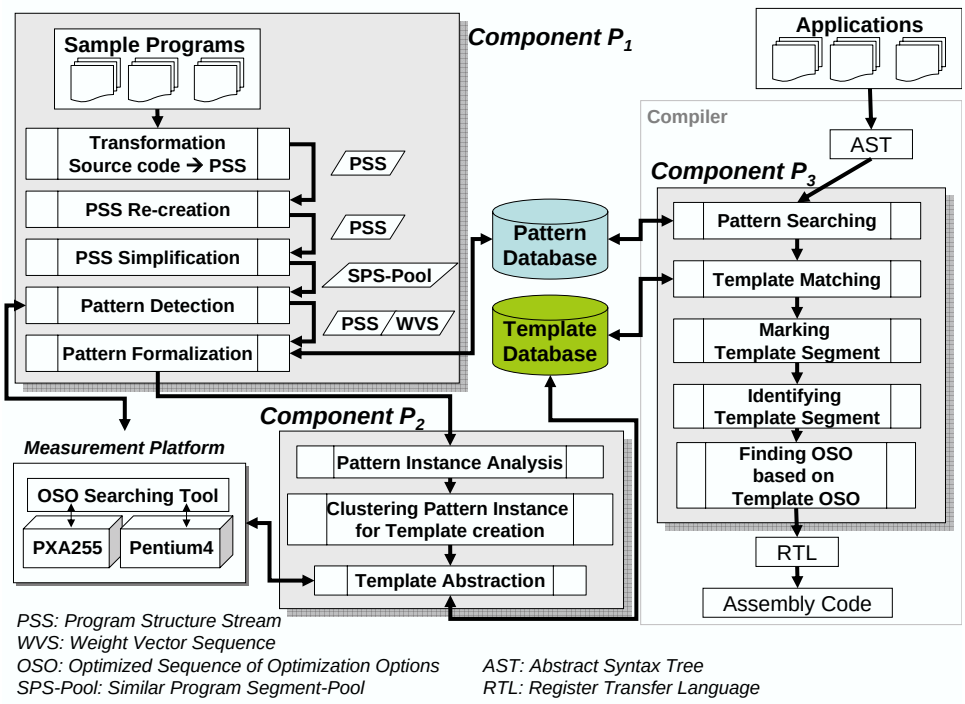

Figure 1: Infrastructure of the $Program$-SOS Methodology

Component P_2 creates templates for each pattern generated in the Component P_1. A set of empirical lower bounds and upper bounds is set for each pattern to limit the range of its instances. Based on the set of lower bounds and upper bounds, a clustering algorithm is used to cluster all the instances of a pattern and find a unique optimized sequence of optimization options for each cluster. Each cluster with an existing optimized sequence of optimization options is transformed into an abstract form of a *template*.

Component P_3 relies on the underlying compiler that integrates the $Program$-SOS Methodology. A template identification tool works on an intermediate representation of a program, captures and records the program segment which matches one of the templates generated in the Component P_2. In the optimization phase, the compiler uses a customized sequences of optimization options for the template matched program segments.

4 Searching for the Optimized Sequences of Optimization Options

This section describes the algorithm we use to search for an optimized sequences of compiler optimization options(OSOs).

If an OSO is found, the program subject to analysis can get additional performance improvement if compiled using this set of options. Additional performance improvement means improvements in speed and/or code size when compared to the $baseline$ version compiled using the highest optimization level of the compiler. Those sequences of optimization options that under perform the $baseline$ are discarded. The sequences of options that make the performance results better than the $baseline$ performance become OSOs.

Our OSO search algorithm is based on the method proposed in the paper [9]. In contrast to the algorithm described in [9], our algorithm employs a different classification of compiler options and searches for a set of optimized sequences of optimization options.

We implement the algorithm over the GCC's "chassis". The optimization options supplied by the GCC compiler are classified into three sets: basic optimization options B, global optimization options G, and local optimization options L. We choose $B \cup G$ as a default sequence of the searching algorithm. The options in $B \cup G$ are always active during the OSO search process.

Our algorithm only considers the searching space of optimization options created by L, because global optimization options force the compiler to deal with the whole function instead of just the code segments. Considering G optimizations in our methodology would call for a complex analysis of the compiler. Hence, we treat the G as a default sequence. The local optimization options L are also called OSO-$candidate$ options.

The OSO search algorithm can be described through the following three steps.

Step 1: Finding Maximal Subsequences of Positively Interacting Options

In this step the algorithm finds subsequences of optimization options, which positively interact with each other. The algorithm starts by enabling single options and by extending the subsequences based on their performance and interaction. These subsequences are included into the sequence set C.

Definition 1 The set C_k represents the sequences of positively interacting options, where k options are switched on in each sequence.

Definition 2 The set L^i is the set of individual options that do not appear in the elements of $C_1, ..., C_{i-1}$ ($2 \leq i \leq n$).

The set C_1 is constructed by testing single options of L for their performance. The first M options offering the best performance are selected and placed into the set C_1. Therefore the size of C_1 is M. The elements in the set C_2 are the combination of single compiler options from the sets L and L^2. In order for a pair of options to be added to the set C_2, the options must interact positively, i.e. they need to give better performance than if they were enabled individually. This is a rather strict condition. If two options are combined and the performance improves, that improvement should be based on a "strong-strong" cooperation instead of a "weak" option being added to a "strong" one. Therefore, the improvement is tested according to the constraint: the combination brings about a significant improvement over selecting either option without the other.

Every subsequent C_k is constructed by pairing up the options in set C_{k-1} with the individual options in set L^k. The combinations that have positive interactions are added to the set C_k.

This step finishes when $L^i = \emptyset$ and $C_i = \emptyset$. All positively interacting subsequences are stored into the C. That is, $C = \cup_{k=1}^{i} C_k$.

Step 2: Combining Subsequences in C

In this step, the elements of the set C are combined if they interact positively. If the interaction is positive, the elements are combined and included in C. The unpaired versions of the combined elements are deleted.

A new set C_{temp} is created to include the subsequences which are deleted from C and improve the performance over that of the $baseline$.

At the end of this step the subsequences in C_{temp} are stored back to the subsequences in set C.

Step 3: Selecting the Best Sequences

The performance of each subsequence in set C is tested 10 times and the mean performance is used. The mean performance is compared to the $baseline$ performance. The subsequences performing better than the $baseline$ are kept while others are discarded.

After this step, the algorithm determines a set of subsequences for a given program segment. Each of these subsequences is an OSO.

For our purpose, each OSO will be considered a candidate when searching for a common OSO for a given template.

Figure 2 shows the flowchart diagram of algorithm to search for an optimized sequences of compiler optimization options.

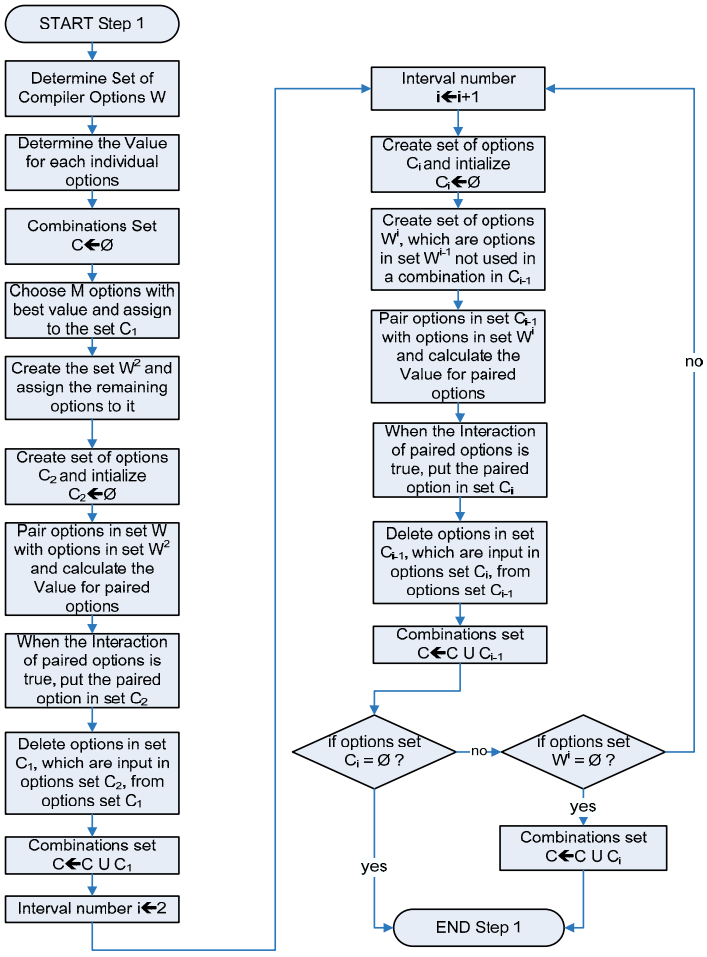

Figure 2: Flowchart of Searching for Optimized Sequences of Compiler Optimization Options

5 Program Segment Patterns and Templates

This section is divided into two parts. The first part describes methods for creating program segment patterns from arbitrary programs. The second part describes the clustering algorithm based on the mean value method to cluster pattern instances into templates.

Definition 3 A *program segment pattern* or simply $pattern$ is an abstract specification of a set of similar coarse-grain program segments for which there is at least one OSO. Each program segment represented by a pattern is called a *pattern instance* or simply $instance$.

Each pattern, theoretically speaking, covers infinite number of instances. A $pattern$ is represented by a program structure stream and a weight vector sequence. Each weight factor in a weight vector sequence describes some features found in a particular statement structure in the program structure stream representation.

An infinite number of pattern instances will be grouped into a finite number of clusters according to some criteria.

Definition 4 A *pattern instance template*, or simply $template$ originates from a pattern instances cluster in which all instances share the same OSO. The OSO is derived by measuring a selected instance in this cluster.

Each $template$ consists of the program structure stream representation of its corresponding pattern, a scaled weight vector sequence and an OSO.

A template and its corresponding pattern have the similar representation, except for the weight vector sequence in which each weight factor is a subrange of the corresponding pattern's weight vector sequence.

The Program Pattern Creation phase has four steps:

- Step 1: Transform a program into a program structure stream representation.
- Step 2: Use program structure stream matching mechanism to find similar program segments in the programs subject to analysis.
- Step 3: Measure each similar program segment to find an OSO and delete the segments which have no OSO found.
- Step 4: Formalize an abstract representation of each remaining similar program segment.

After a pattern has been created, the clustering algorithm is used to cluster the pattern instances and derive a series of templates.

The following sections give a detailed description of pattern creation and template formalization.

5.1 Program Structure Stream Representation

$Program$-SOS Methodology completely relies on static information on programs subject to analysis. More precisely, $Program$-SOS considers only statement structures and the number of operations (operands and operators). We use a simple data structure to represent a program statement and the operations (operands and

operators) of a statement, as shown in Figure $3(a)$. Each data structure is a node of a double linked list. A function reference will have individual structure if it appears as a statement. Otherwise, if it appears in an expression, it is considered to be an operand.

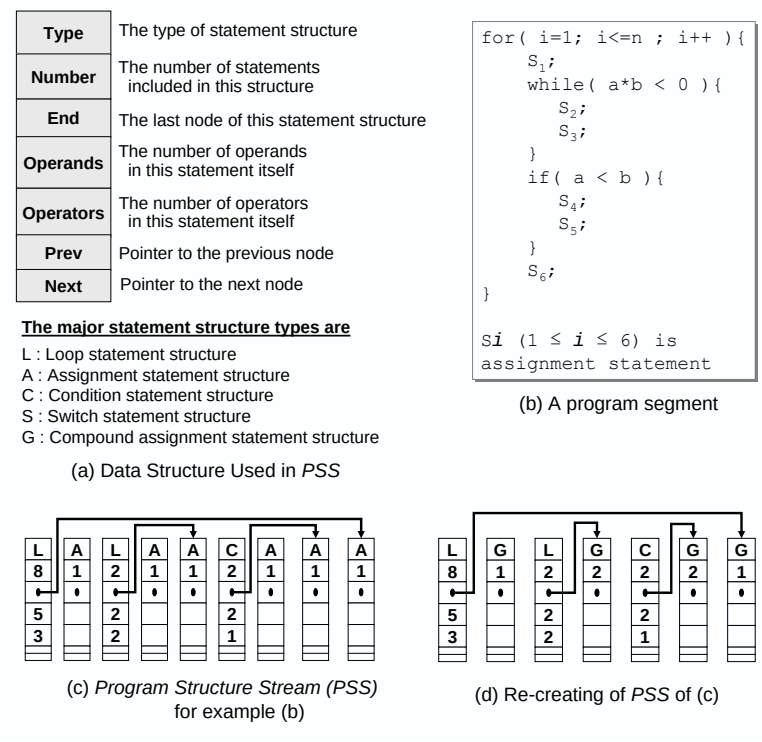

Figure 3: PSS Data Structure and Used Example

A double linked list with nodes (the structure is shown in Figure $3(a)$) is used to represent an input program. We call this structure the *program structure stream representation* of a program.

It is obvious that each node in the program structure stream representation represents a program segment. Each node can represent a large number of different program segments that have the same program structure. This property of program structure stream representation confirms the first observation made in Section 1.

One of the tools developed as a part of $Program\text{-}SOS$ Methodology transforms an input program into the program structure stream representation. Figure $3(c)$ gives the program structure stream representation for the program segment in Figure $3(b)$.

5.2 Similar Program Segments Detection

Definition 5 Two program segments S_1 and S_2 are similar if the number of nodes, node types and the node order are the same in both program structure stream representations.

5.2.1 Re-creation of the Program Stream Representation

Similar program segments can appear in the same program or in different programs. It is rarely the case that two program segments are exactly the same. To narrow down the focus of a similar program segment for our purposes, we recreate the program stream representation using the following new nodes:

- A single or continuous assignment statement nodes are replaced with a compound statement node, if they belong to the same level of a statement structure.
- Insert a $null$ compound statement node (the $Number$ field is 0) behind each non-compound statement node, if it is not followed by a compound node.
- Continuous conditional nodes are considered to be a special conditional statement node if all nodes in their statement body are compound 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 $3(d)$ shows the recreated program stream representation from the representation in Figure $3(c)$.

5.2.2 Simplification of the Program Stream Representation

The constraints of our $Program$-SOS Methodology could be loosened so that the program segments, the subject of our interest, are loop sections which contain only loops, condition statements, switch statements and/or assignment statements. Based on the constraint, a simplification process is applied to the recreated program stream representation of a program to exclude all program segments not included in the further analysis.

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 loop segment decomposition eliminates any 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 similar program segments.

5.2.3 Similar Program Segment Detection Algorithm

The process of detecting similar program segments is based on a very intuitive algorithm. The algorithm initially creates a similar program segment pool (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 in 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 $SPS\text{-}pool$ 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 $SPS\text{-}pool$.

Step 3: For each segment in P, check if any node sequence matches any element in the $SPS\text{-}pool$. This matching only checks for the node order, number of nodes and node type. If there is a match, a similar segment is found and the similarity weight value of the matched element in the $SPS\text{-}pool$ increments for 1. The 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 $SPS\text{-}pool$.

The number of elements in the $SPS\text{-}pool$ depends on how many sample programs are tested.

5.3 Pattern Formalization

Here we present the process of translating similar program segments into patterns.

The $SPS\text{-}pool$ has program segment for its elements. These segments can not be run as separate programs, so 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.

A similar program segment will be further formalized into a pattern if an OSO is found. A pattern consists of a "pure" program structure stream representation and a weight vector sequence. The difference between the pure program structure stream representation and the original (simplified) program structure stream representation is that the former only keeps the type messages for all nodes.

During the pattern formalization, we use the skeleton of the representative similar program segment to represent all program segments that have the same skeleton. The characteristics of each program skeleton are described by weight vector sequence.

A weight vector sequence of a pattern is an ordered vector sequence defined as:

$$\{\ \omega^1, \omega^2, \cdots, \omega^n\ \}\ (\omega^i = < v_n^i, v_o^i > (1 \leq i \leq n))$$

In the sequence, n is the number of compound statements that appear in the pattern. Let l be the number of non-compound statement structures in the pattern, $n = 2 * l - 1$. v_n^i is the range of number of assignment statements that can appear in ith compound statement and v_o^i is the range of the sum of all operations (operands and operators) in the ith compound statement.

Definition 6 A pattern with k elements in its weight vector sequence is called a $k\text{-}element$ pattern.

A $k\text{-}element$ pattern corresponds to a $2k$ dimensional instance space, which is created by its weight vector sequence. This pattern represents a number of program segments that have the same program structure stream representation and have the same number of weight vectors as the pattern. This pattern property reveals the second observation in Section 1.

Figure 4 shows the program structure stream representations for some patterns generated in this study. We call this a pre-build pattern database. Each pattern has been assigned a unique order number.

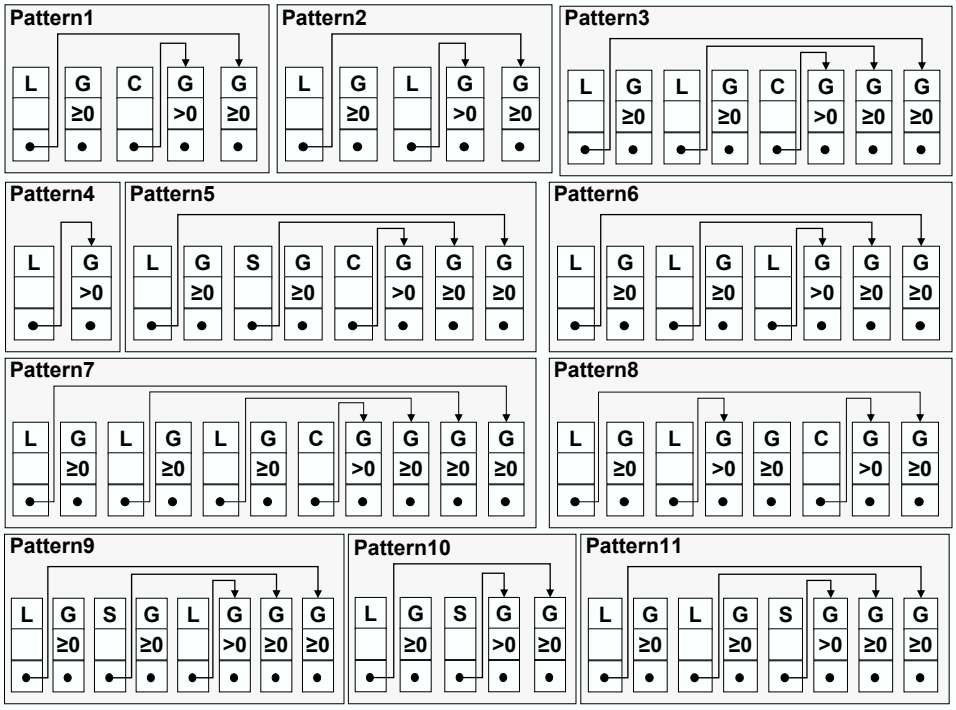

Figure 4: Elements in a Pre-Build Pattern Database

5.4 Templates Creation through the Clustering Algorithm

This section describes the clustering algorithm used in the $Program\text{-}SOS$ Methodology.

As described in Section 5.3, each program pattern is associated to a weight vector sequence and each weight vector in this sequence corresponds to a compound assignment statement structure in the pattern.

The clustering algorithm groups a large number of pattern instances into several clusters, and then derives templates for those clusters.

Definition 7 A $cluster$ of a pattern consists of pattern instances that meet the following requirements:

- They have the same program structure stream representation as the pattern.

- The weight vector sequence of an instance with the lowest values and the weight vector sequence of an instance with the highest values in a cluster are combined into a scaled weight vector sequence. All other instances in this cluster have their weight vector sequence values within the range of this scaled weight vector sequence.

In order to derive a template from a cluster, all instances in the cluster must have the same OSO. The uniqueness of cluster's OSO can be verified by comparing the performance when instances in the cluster are compiled using this OSO to the $baseline$ performance. However, the exhaustive comparison is impractical because the total number of instances is large, hence this process is quite time consuming. However, we have significantly shorten the clustering time by making several assumptions:

Definition 8 In a $2k$ dimensional weight vector space of a k-$element$ pattern, the instance for which each weight has the minimum value in this space is called the *lower bound instance* of this space; the instance for which each weight has the maximum value in this space is called the *upper bound instance*; instances for which each weight has an arbitrary value in this space are called *sibling instances*.

Hypothesis *For any $2k$ dimensional weight vector space of a k-$element$ pattern, if its lower bound instance and its upper bound instance have the same OSO, any sibling instance would have the same OSO.*

According to this hypothesis, we have developed a clustering algorithm based on the iterative mean factor value method, to scale the value ranges of each weight factor of pattern's weight vector sequence. The scaled weight vector sequence is later used to derive templates.

Definition 9 To average a sequence of n factors
$V = \{v_1, v_2, ..., v_n\}$, each time one factor v_i $(1 \leq i \leq n)$ is being averaged, all other factors remain unchanged until v_i averaging is done. This process is iteratively applied to all factors until no further averaging is possible for any factor. We call this method *iterative mean factor value method*.

The iterative mean factor value method applied in our algorithm can be described as a top-down approach. In the first step, we constrain the $2k$ dimensional weight space using a set of empirical upper bounds for each weight factor. Then we create a *lower bound instance* and an *upper bound instance* for this space. A cluster factor OSO is used to orchestrate further clustering. We measure the *lower bound instance* and the *upper bound instance* to find the shared OSO. If the same OSO is found for both bounds, the clustering process finishes. Otherwise, the weight space is divided into two subspaces using the iterative mean factor value method. That is, one subspace keeps all the lower bounds and uses all the mean weight values as its upper bounds, while the other subspace uses all the mean weight values as its lower bounds and keeps the original upper bounds. The mean bound values must be adjusted so that the subspaces do not overlap. This step is repeated for each subspace until a subspace-unique OSO is found or the clustering can not be continued. Finally, the $2k$ dimensional weight space of a k-$element$ pattern is clustered into a finite number of subspaces. Each subspace corresponds to a template if it has an OSO associated to it. All program segments that belong to this subspace share the same OSO. This Characteristic confirms our observation 3 described in Section 1.

Figure 5 shows an intuitive view of templates derived from *pattern 1*. It shows a subspace of pattern 1, and further clustering creates 3 new subspaces which correspond to templates T1, T2 and T3. For example, template T3 includes those instances for which the number of assignment statements is from 1 to 7 and the number of operations is from 9 to 13.

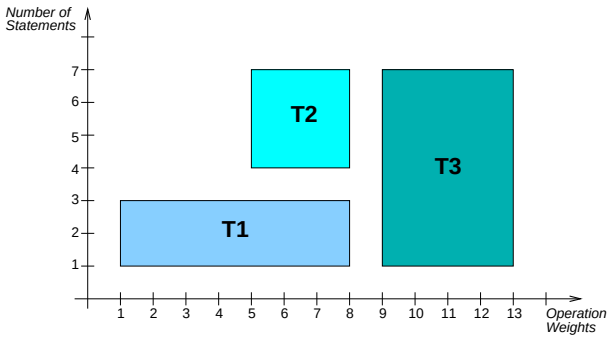

Figure 5: An Intuitive Template Scene

Figure 6 presents a pseudo code description of our clustering algorithm through a step-by-step description of the clustering algorithm (processing of a k-$element$ pattern).

Step 1: The algorithm initially creates two special weight vector sequences W_L and W_U, which store the lowest and the highest weight values, respectively. The initial value for each weight factor of W_L is set to 0 if the $Number$ field of the corresponding compound structures is defined as "≥ 0", otherwise it is set to 1. The weight factor of W_U is set to a different empirical value m^i $(1 \leq i \leq k)$. Value m^i should not be too small, otherwise the clustering algorithm will converge too quickly, leading to creation of very few templates.

Step 2: Using the weight values of W_L and W_U, we manually create two pattern instances PI_L and PI_U. These instances are measured to find their OSOs. If no OSO is found for the instance PI_L, the weight factors in W_L will be adjusted, and a new PI_L is created and again measured iteratively. Similarly, if no OSO is found for the instance PI_U, the weights in W_U will be adjusted, and a new PI_U is created and measured iteratively.

This is a trial and error process of adjusting the W_L and W_U which guarantees the finding of OSOs for both instances PI_L and PI_U. All found OSOs are then stored in C_L and C_U, respectively.

$C_L \cap C_U \neq \emptyset$ indicates that the same OSOs are found for both instance PI_L and PI_U. The clustering of the current weight vector sequence finishes and the algorithm skips to step 5, where a template is created. If $C_L \cap C_U = \emptyset$, the algorithm continues with step 3.

Step 3: The OSO search algorithm described in Section 4 does not find all OSOs. Some OSO in C_L may be the correct OSOs for PI_U and vice versa for C_U. This step does an "intercrossing" measurement of instances PI_L and PI_U over C_U and C_L respectively. That is, pair up and measure PI_L with each OSO in C_U and PI_U with C_L. If the "intercrossing" measurement successfully finds new OSOs, they are added to the instance's set C_L or C_U and the algorithm advances to step 5. If the "intercrossing" does not detect any new OSO, the algorithm continues with step 4.

Step 4: If no OSO sequence is found for a scaled weight vector sequence, new weight vector sequences are created by splitting the weight range based on the method of iterative mean weight values. The algorithm is invoked recursively over the new weight vector sequences until an OSO is found.

Step 5: For all OSOs that are found in the clustering phase, the algorithm determines the best one for both instances PI_L and PI_U and denote it as a shared OSO. A new scaled weight vector sequence is created accordingly. The new scaled weight vector sequence and the shared OSO along with the pattern's program structure stream representation form a template.

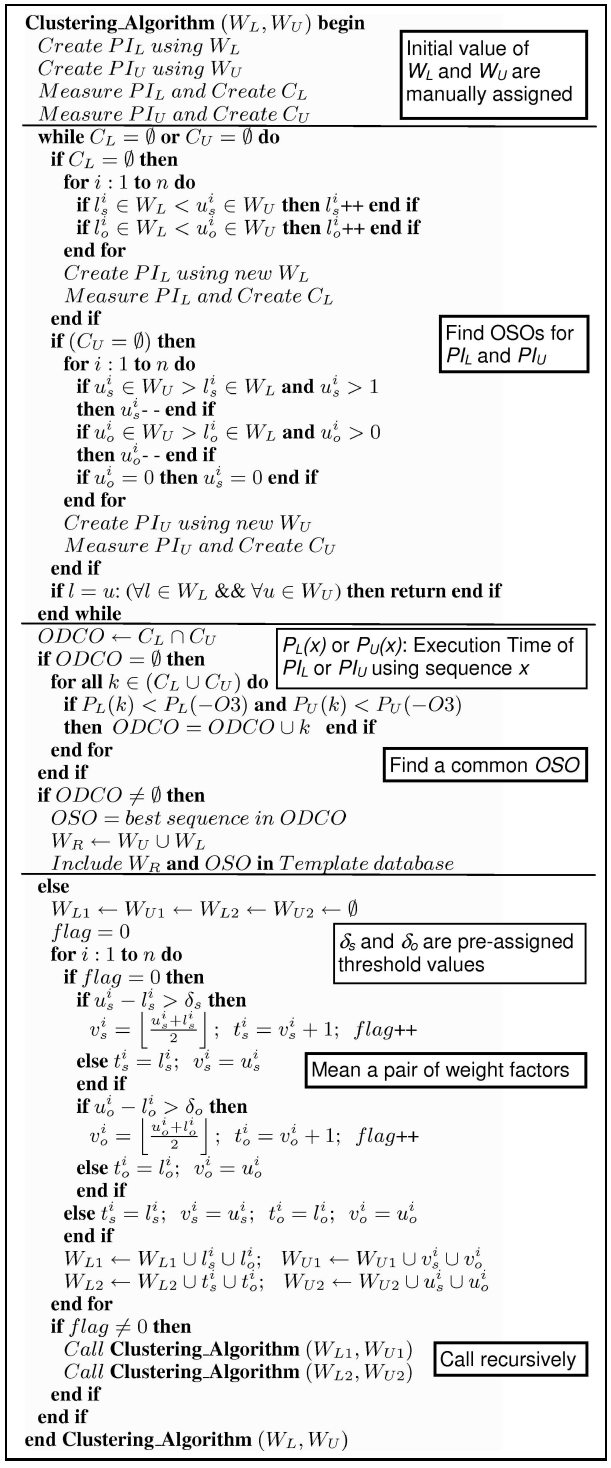

Figure 6: Template Clustering Algorithm

Steps 2 through 5 are recursively repeated for each mean value weight vector sequence, until either of the following conditions is met: 1) every clustering weight vector sequence gets a shared OSO, or 2) the algorithm converges. That is, the weight vector sequences can not be further scaled. For these weight vector sequences, no

shared OSO exists. Therefore, they are excluded from the template formalization phase.

The proposed clustering algorithm will eventually find one or more pattern clusters or templates, due to its recursive nature. After the algorithm is applied over different patterns, a template database is built to store detected templates.

This part of the $Program$-SOS Methodology implements a "one-size-fits-all" approach in creating the pattern database and template database. Given this new information, the underlying compiler can be improved to fully exercise program segment based optimization.

6 Embedding $Program$-SOS into Compiler

The process of embedding the $Program$-SOS Methodology into any platform compiler is rather simple. This process is fully transparent from the user's point of view. In this section we describe the necessary alterations to GCC compiler required for successful embedding of the $Program$-SOS Methodology.

There are only two steps required for GCC modification: 1) integration of a template identification tool in the front-end after an intermediate representation is generated, and 2) implementation of the segment-based OSO in the optimization phase.

Because most of modern compilers have similar architectural skeleton, such as front-end, middle-end and back-end, the simplicity of embedding the $Program$-SOS Methodology into the GCC compiler applies to other compilers (please refer to observation 4 in Section 1).

6.1 Implementation Principle

In the front-end of the GCC compiler, an input code is transformed into the AST (Abstract Syntax Tree) intermediate representation, and this is where the template identification tool is integrated. The compiler enters the tool before it traverses the AST to generate the RTL (Register Transfer Language) representation. The identification tool searches for the templates and records the position of each template in the compiled program.

After the template identification step is completed, the compiled program is divided into a sequence of templates. The program segments that are not covered by any pre-build template are treated as a special template, which has the OSO equal to -O3. The template sequence is normally represented in the RTL format and it is passed to the back-end of GCC.

We add one step to the original optimization phase. Unaltered GCC uses fixed optimization sequence to optimize the whole compiled program. The modified version 'forces' compiler to use a customized set of options for each program segment. The optimization sequence for each segment comes from the corresponding template OSO.

6.2 Template Identification Algorithm

In this section we present the algorithm for identifying and recording of the templates in the compiled program.

For each program segment in the compiled program, the algorithm first determines a corresponding pattern. In the next step, the format and definition of the weight vector sequence of this pattern is used to create the concrete weight vector sequence of the program segment. The algorithm then searches the template database of this pattern using the weight vector sequence. If the weight vector sequence is a child of the weight vector sequence of a template, this is considered to be a match.

In summary, the algorithm identifies a template in two steps:

- Step 1: Pattern Matching and Weight Vector Sequence Creation
- Step 2: Template Identification

In Step 1, the algorithm does the pattern matching and calculates weights by traversing the AST representation, records information on found patterns and calculates their weight vector sequences. In Step 2, the algorithm identifies template based on the output of the previous step. If a matching template is identified, the template order number is recorded to the template table. Otherwise zero is recorded, which denotes the special template case.

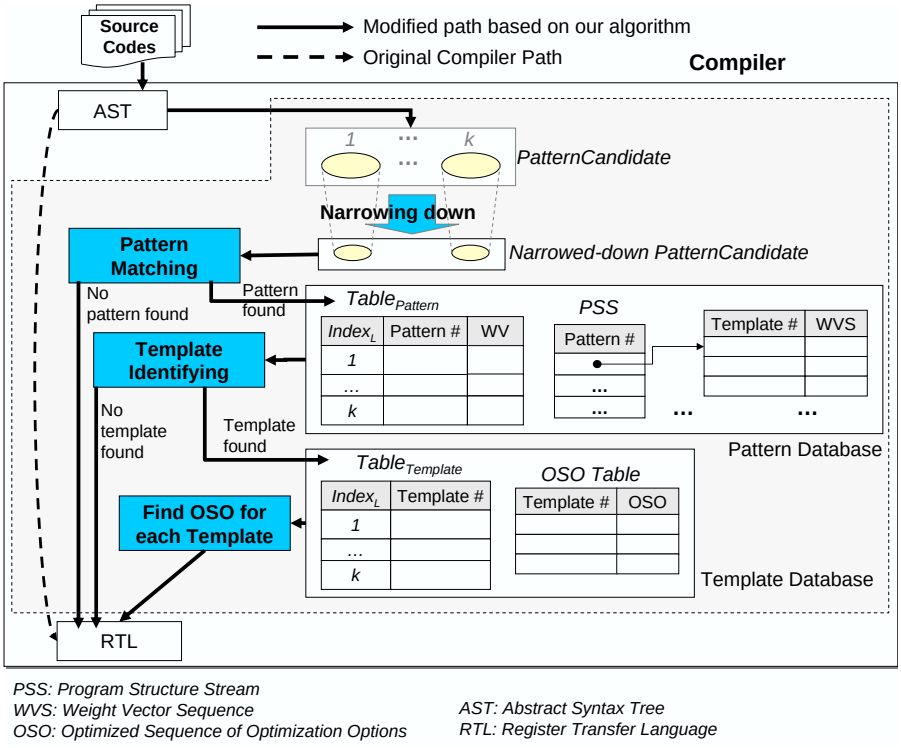

Figure 7: Flowchart of Template Identification

After completing all steps for each loop structure in the program, the template table creates a mapping between loop index (the order in the compiled program) and the order number of the identified template. Template order number 0 matches every loop structure which has no corresponding pre-build template.

Figure 7 shows the control flow and the data structures used in each step.

6.2.1 Pattern Matching and Weight Vector Sequence Creation

The procedure of pattern matching and WVS calculation presents in the following:

1. Traverse AST until a loop structure appears.
2. For the loop structure,
   (a) Assign $Index_L$ and initialize $PatternCandidate(Index_L)$
   (b) Go to next statement (in the loop body)
      i. If it is an expression, calculate weight and fill WV column in $Table_{Pattern}$. If it is another loop structure, re-assign $Index_L$.
      ii. Check what patterns are possible from $Index_L$ to this structure. Delete pattern order numbers from $PatternCandidate(i)$ which is not possible to derive from i, when i = 1 to largest $Index_L$ in this step.

   Repeat above until reach last node in outermost loop structure.
3. Repeat above until reach the end of AST.
4. Check $PatternCandidate(i)$ (i = 1 to largest $Index_L$) in which has one element P and fill the $Table_{Pattern}$ with
   (a) 0 for index 1 $\sim$ i-1;
   (b) P for index $i \sim$ largest $Index_L$.

After this procedure, the $Table_{Pattern}$ has all pattern order numbers and corresponding WVS calculated from source codes. The reason why we choose $PatternCandidate(i)$ which has only one element is to remove all ambiguity of decision.

6.2.2 Template Identifying

Based on the table $Table_{Pattern}$, template identifying procedure is following.

1. Go to the first element in $Table_{Pattern}$
   (a) If pattern order number is 0, put 0 in $Table_{Template}$.
   (b) If pattern order number is P, find the PSS representation of pattern P. Compare WVS in $Table_{Pattern}$ with that in the template table of P.
      i. If WVS in $Table_{Pattern}$ is a *child WVS* of one template in the template table of P, get this template order number and fill into $Table_{Template}$.
      ii. If we fail to find any *child WVS* in the template table, fill 0 in $Table_{Template}$

2. Repeat above for all elements in $Table_{Pattern}$

After finishing all steps, for each loop structure in the program, the $Table_{Template}$ contains a identified template order number. Template order number 0 matches all the loop structures that have no corresponding pre-designed templates.

7 Experimental Phase

To evaluate the abilities and effects of the $Program$-SOS Methodology, we have implemented the whole package for the GCC compiler on two platforms: 1) the Intel embedded XScale PXA255, and 2) a general-purpose 2GHz Intel Pentium 4 platform running Linux.

In the experiment of performance improvement, each tested program is run **5** times on PXA255 (**10** times on Pentium 4) and the mean performance is used.

7.1 Benchmarks used for the Experiment

We have randomly collected 5 benchmark packages - CommBench, DSP kernel suite, Mediabench, Mibench (we use both large and small input data set in the experiment) and CINT2000 of SPECCPU 2000 (we use the test input data set in the experiment), for our experiments. CINT2000 does not test on PXA255 because PXA255 has the memory capacity of only 64MB while CINT2000 requires at least 256 MB.

Brief description of each benchmark package is following:

- **CommBench**: Telecommunication benchmark for evaluating network processors including 8 programs for packet header processing and payload processing.
- **DSP kernel suite**: Benchmark for digital signal processing including vector multiply, Dot product, FIR filter, Lattice Synthesis, IIR Filter and Vocoder codebook search and JPEG discrete cosine transform
- **Mediabench**: Benchmarks to evaluate multimedia and communication systems including image processing, communications and DSP applications
- **Mibench**: Benchmark suites for embedded systems including applications for automotive and industrial control, network, security, consumer devices, office automation and telecommunications
- **CINT2000 of SPECCPU 2000**: Benchmark for measuring and comparing performance with strictly defined set of operations on different kind of practical hardware platform.

When we build the platform-independent pattern database, we randomly select a set of sample programs that have a large fraction of loop structures. The reason for selecting this type of cases is that the current patterns are created from loop segments. The set of patterns described in 4 are driven from these sample programs.

To test the $Program$-SOS Methodology, we first test all cases in the benchmark packages 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.

N_c: Number of used cases

N_p: Number of pattern found

N_c	N_p	Statistics of 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: Statistics of Pattern Creation

Pattern Number	Number of Template	OSO Difference		Performance Improvement(%)	
		Min	Max	Min	Max
1	4	1	1	2	75
2	2	3	3	4	74
3	2	1	1	20	74
4	2	2	6	25	75
5	2	4	4	3	75
6	2	3	3	29	72
7	2	2	2	7	73
8	2	5	5	3	75
9	2	1	1	27	75
10	3	3	5	1	75
11	2	4	4	1	75

Table 2: Statistics of Template Clustering Algorithm on PXA255

7.2 Experimental Results

This section describes two sets of experimental results for PXA255 (Table 2, Table 3, Figure 8, Figure 9, Figure 10 and Figure 11) and for Pentium 4 (Table 4, Table 5, Figure 12, Figure 13, Figure 14, Figure 15 and Figure 16), respectively. Due to limited space, we only discuss the PXA255 results in this paper. However, the same reasoning can be applied to the Pentium 4 platform. The results given in Table 1 are architecture-independent and are shared by the two experimental platforms.

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

When the platform-independent pattern database is built, we randomly select two sets 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 loop segments. The set of patterns described in Section 4 are derived from these sample programs.

The first set of sample programs consists of 20 randomly chosen programs. Using the algorithms described in Section 5, 17 different patterns are created. We analysis 11 most frequent patterns. We find that: 1) 11 patterns appear 847 times in the set of sample programs; 2) the minimum number of patterns appearing in a program is 1; and 3) the maximum number of patterns appearing in a program is 182. The $frequency$ field gives the appearance

Template	Found template			Coverage(%)	
	Number	Min	Max	Min	Max
1	241	1	30	0.6	58.3
2	36	1	7	0.1	4.6
3	132	1	22	0.3	38.5
4	39	1	5	0.4	0.7
5	26	1	3	0.2	13.1
6	2	1	1	5.5	5.5
7	959	1	120	0.1	33.3
8	163	1	14	0.7	45
9	45	1	5	0.4	62.5
10	20	2	2	0.3	1.2
11	0	0	0	0	0
12	0	0	0	0	0
13	52	1	8	0.9	61.5
14	14	1	2	0.2	71.4
15	55	1	9	0.4	62.4
16	0	0	0	0	0
17	14	1	4	0.1	1.6
18	0	0	0	0	0
19	0	0	0	0	0
20	0	0	0	0	0
21	2	1	1	5.0	10.2
22	0	0	0	0	0
23	0	0	0	0	0
24	0	0	0	0	0

Number of tested cases : 42

Table 3: Statistics of Templates in the tested cases on PXA255

frequency 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.

The results in Table 1 are platform-independent.

Table 2 gives the features of the pre-build template database that is derived from the pattern database by pattern instances clustering. This table shows the number of templates derived from each pattern; the minimum difference of OSO for any two templates, that is, the number of optimization options and the maximum difference.

The performance improvement field in Table 2 gives the minimal and the maximal additional performance improvement over the GCC -O3 among the templates in each pattern. An interesting observation is that for

Pattern Number	Number of Template	OSO Difference		Performance Improvement(%)	
		Min	Max	Min	Max
1	4	9	9	10	34
2	2	5	5	17	76
3	2	4	4	14	79
4	2	6	11	10	81
5	1	0	0	23	23
6	2	12	12	8	68
7	1	0	0	30	30
8	2	6	6	4	75
9	1	0	0	57	57
10	1	0	0	11	11
11	1	0	0	17	17

Table 4: Statistics of Template Clustering Algorithm on Pentium4

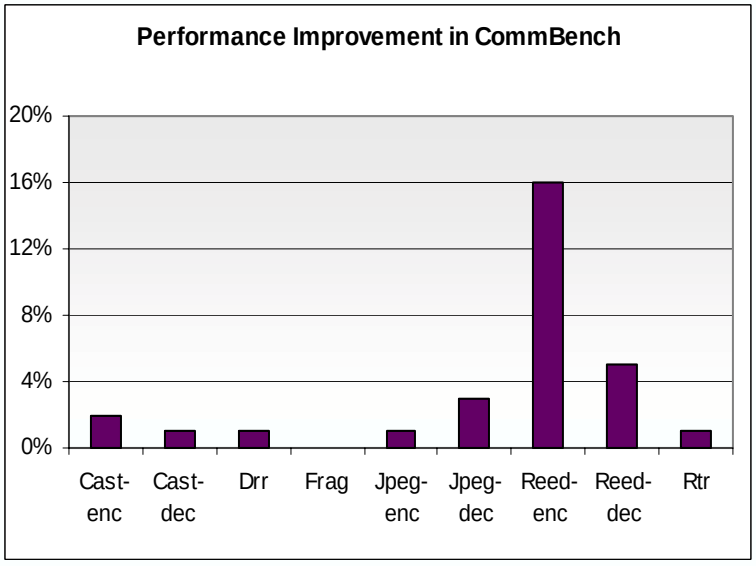

Figure 8: Performance Improvement in CommBench over GCC -$O3$ on PXA255

two similar program segments with different weight vector sequences, there is a big difference in the additional performance improvement .

In Table 2, only a few templates are driven from each pattern. We believe that this is caused by a set of small values used to set the upper bound of weight vector sequence of each pattern. Clearly, having more templates would bring better performance improvement, but this would not affect the fundamental evaluation of the real benefi ts of our methodology.

Table 3 shows the statistic data for templates found in the tested programs. Each row in the table gives the total number of times each template was found in the tested programs, the minimum and the maximum number

Template	Found template			Coverage(%)	
	Number	Min	Max	Min	Max
1	378	1	57	0.6	58.3
2	46	1	7	0.03	4.6
3	194	1	34	0.3	38.5
4	56	1	12	0.4	1
5	36	1	5	0.1	13.1
6	2	1	1	5.5	5.5
7	1269	1	145	0.1	33.3
8	337	1	60	0.3	45
9	62	1	7	0.02	62.5
10	23	1	2	0.3	1.2
11	0	0	0	0	0
12	58	1	8	0.03	61.5
13	14	1	2	0.2	71.4
14	55	1	9	0.4	62.5
15	16	1	4	0.1	1.6
16	0	0	0	0	0
17	0	0	0	0	0
18	3	1	1	0.01	10.2
19	0	0	0	0	0

Number of tested cases : 49

Table 5: Statistics of Templates in the tested cases on Pentium4

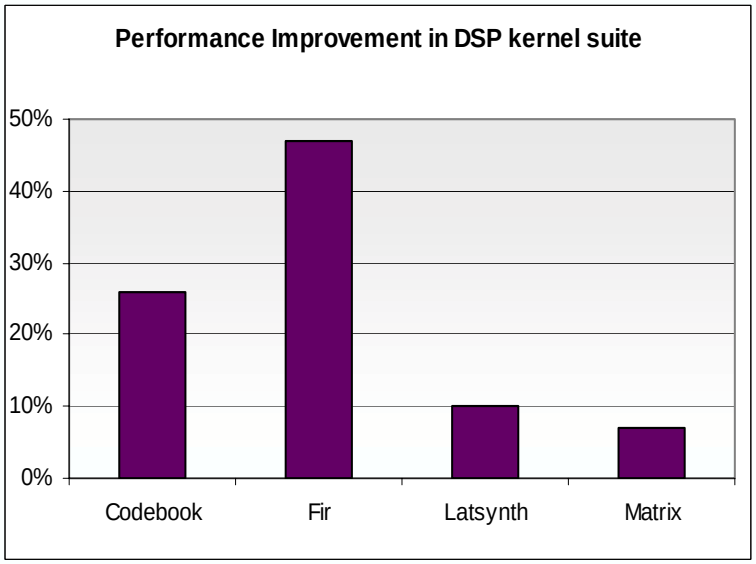

Figure 9: Performance Improvement in DSP Kernel Suite over GCC -$O3$ on PXA255

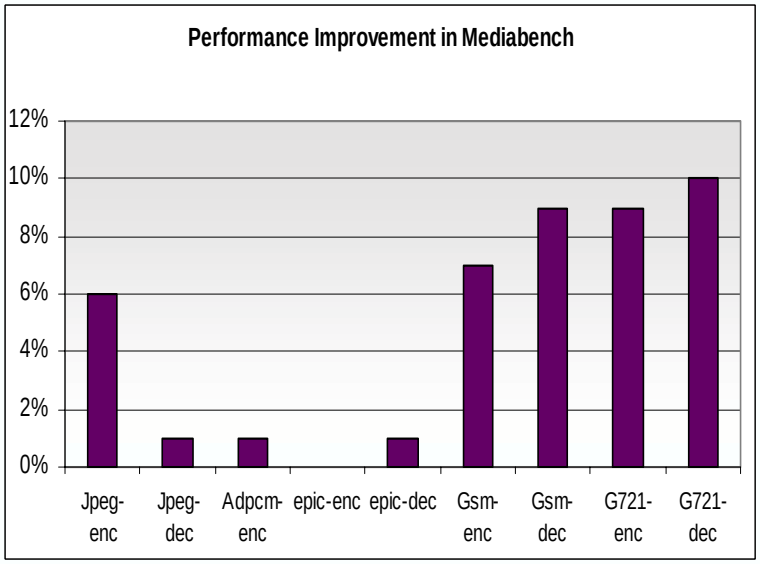

Figure 10: Performance Improvement in Mediabench over GCC -*O*3 on PXA255

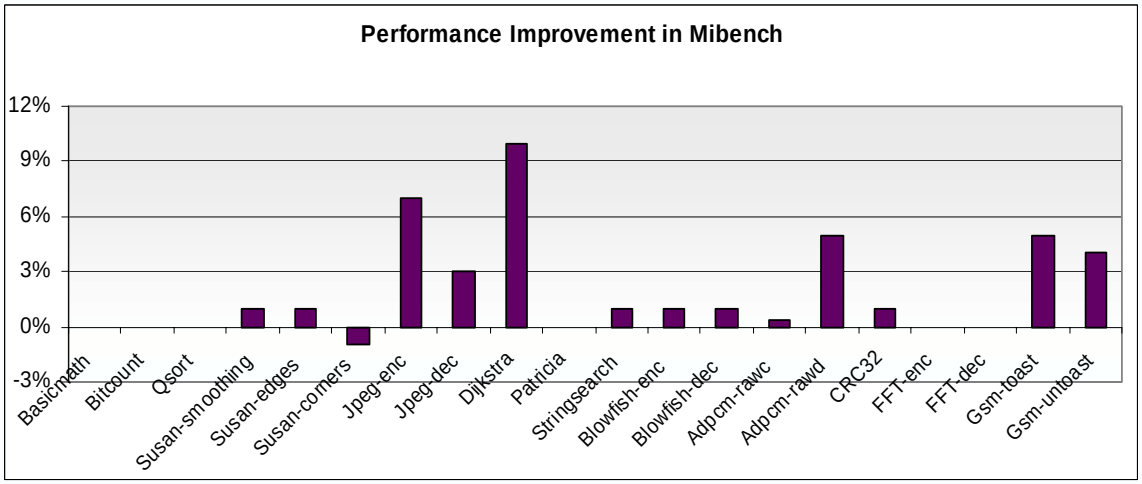

Figure 11: Performance Improvement in Mibench over GCC -*O*3 on PXA255

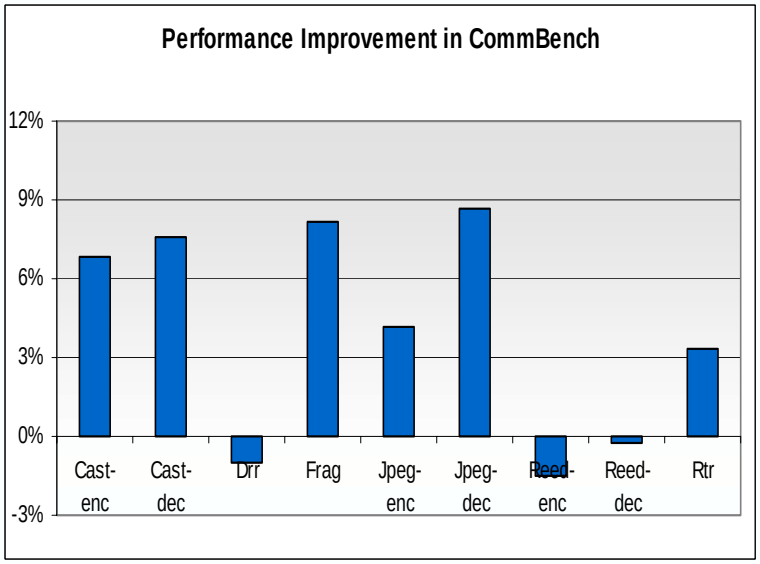

Figure 12: Performance Improvement in CommBench over GCC -$O3$ on Pentium4

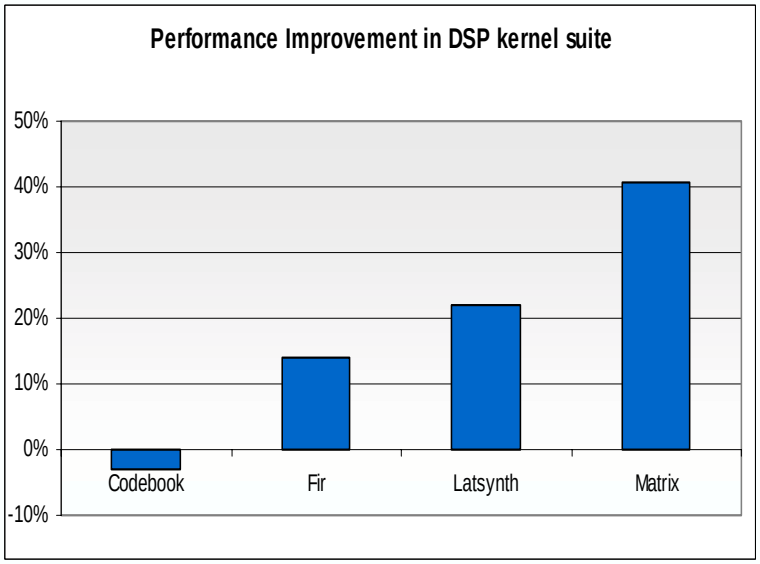

Figure 13: Performance Improvement in DSP Kernel Suite over GCC -$O3$ on Pentium4

of the template occurrences in the tested programs (for those programs where the template appears at least once). The table also gives the minimum and the maximum source line percentage for each template, for the programs where it appears. Clearly, each template represents a small fraction of source codes for most tested programs.

Figure 8, 9, 10 and 11 quantify an additional performance improvement over -O3 among the tested programs in each benchmark. The average improvement is 6.0% with the maximum being 47%. 33 programs from the total of 42 on PXA255 have showed improved performance over that of -03.

From Figure 8, 9, 10, 11, 12, 13, 14, 15 and 16, we find that there are some programs that show no improve-

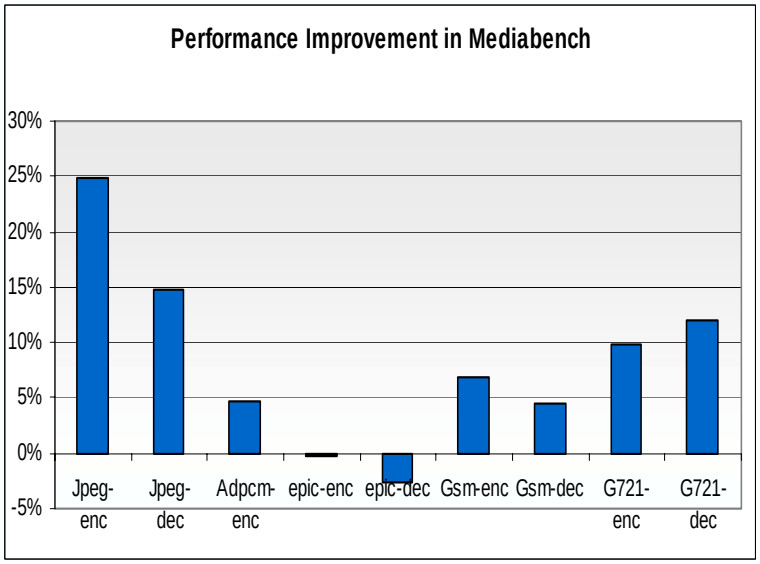

Figure 14: Performance Improvement in Mediabench over GCC -$O3$ on Pentium4

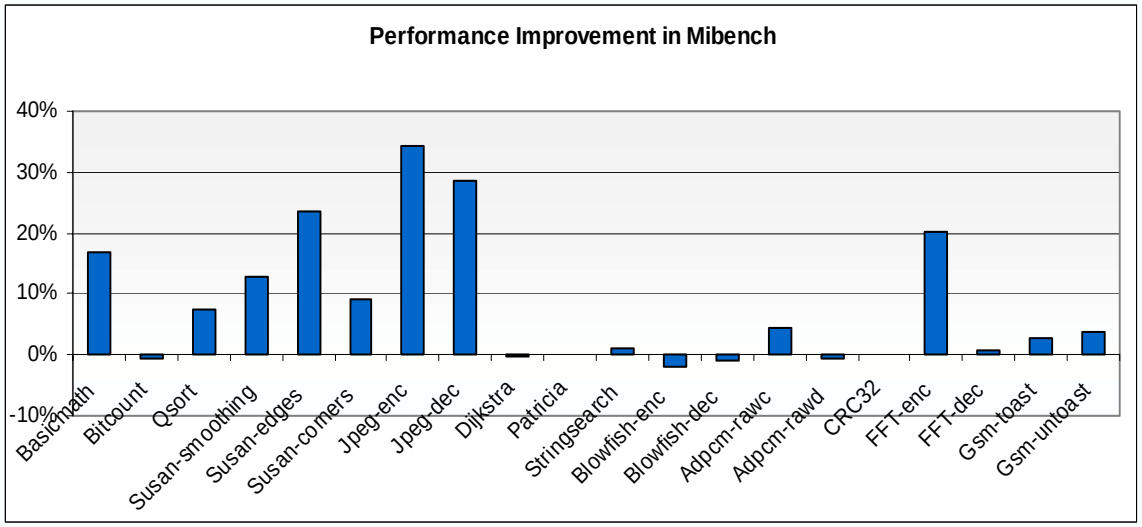

Figure 15: Performance Improvement in Mibench over GCC -$O3$ on Pentium4

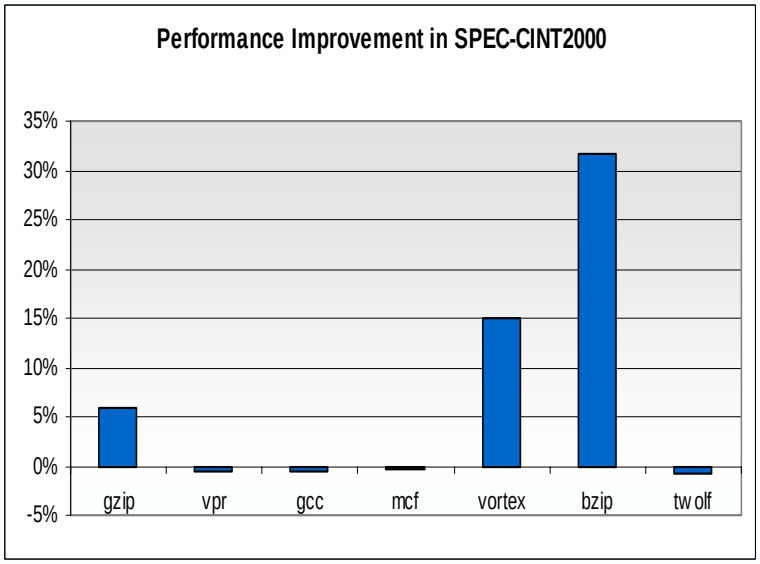

Figure 16: Performance Improvement in SPEC-CINT2000 over GCC -$O3$ on Pentium4

ment. This is because these programs have no templates recognized in our current compiler implementation.

There are 2 cases on PXA255 and 11 cases on Pentium 4 which show negative improvement. We have found that these negative results come from the potential interference between the GCC -O3 options and our template OSOs for certain program segments. We are continuously working on addressing this issue.

In summary, based on the experimental results, we can evaluate the real benefits of our methodology. In spite the small number of patterns and templates, for large randomly chosen programs, the average of 6.0% and 8.1% performance improvement can be achieved on a embedded system and a general-purpose system, respectively. These results confirm the observation 5 in Section 1.

8 Conclusion and Future Work

In this paper, we propose an automated methodology for mining compiler's inherent optimization abilities – The $Program$-SOS Methodology. The $Program$-SOS Methodology employs static segment-based analysis over an arbitrary program to derive customized compiler optimizations. The goal is to find the best optimized set of options customized for each recognized program segment, and use these sets in the compiling phase to improve the final code performance.

Behind the automated methodology, there is a set of techniques that 1) locate and recognize program segments; 2) formalize similar program segments into pattern; 3) cluster pattern instances into a template 3) identify template matching program segments and implement program segment based optimization over those segments.

We have analyzed these tasks and developed the following set of tools: 1) program transformation and manipulation tool, 2) pattern formalization tool, 3) pattern instance analysis and template abstraction tool, and 4) template identification tool.

There are two implementations of $Program$-SOS Methodology available for 1) an embedded system (Intel XScale-PXA255 platform), and 2) a general-purpose architecture (Intel Pentium 4 platform under Linux).

The experimental results show that the $Program$-SOS Methodology improves performance by 6.0% and 8.1% on average on the embedded system and on the general-purpose system respectively.

Although program segments, the subjects of our analysis, are constrained to loop sections only, theoretically they could be arbitrary program structures. For example, structures such as FIR (Finite impulse Response) in DSP (Digital Signal Processing) applications and FFT (Fast Fourier Transform) in scientific computations, can be directly formalized into patterns and the $Program$-SOS Methodology can be easily extended to cover variety of new program features.

We believe our methodology is only the first step toward reaching the final goal of finding the optimal or near-optimal sequence of compiler inherent optimizations for an arbitrary application. The first phase of the future work will be focused on incorporating profiling-directed feedbacks into $Program$-SOS Methodology, and on testing 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). In particular, we wish to acknowledge Rishi Khan for a through check of the final manuscript and many improvement he has suggested. We appreciate the help of Joseph Manzano for a thorough review of the final manuscript.

References

[1] 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.

[2] E. Boros, P. L. Hammer, T. Ibaraki, A. Kogan,E. Mayoraz and I. B. Muchnik. An implementation of logical analysis of data. In *Knowledge and Data Engineering*, volume 12, pages 292–306, 2000.

[3] E. Granston and A. Holler. Automatic recommendation of compiler options. In *Proceedings 4th Feedback Directed Optimization Workshop*, 2001.

[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] 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.

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

[7] 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.

[8] 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.

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

[10] P. Hammer, A. Kogan, B. Simeone and S. Szedmak. Pareto-optimal patterns in logical analysis of data. In *RUTCOR Research Report RRR 7-2001, Rutgers University*, 2001.

[11] P. Nagpurkar, C. Krintz, M. Hind, P. F. Sweeney and V. T. Rajan. Online phase detection algorithms. In *Proceedings of the CGO'06*, 2006.

[12] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. August. Compiler optimization-space exploration. In *Proceedings of the '03 International Symposium on Code Generation and Optimization*, pages 204–215, 2003.

[13] 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.

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

[15] 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.

[16] Z. Pan and R. Eigenmann. Fast, automatic, procedure-level performance tuning. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 173–181, 2006.