Automatic Program Segment Similarity Detection in Targeted Program Performance Improvement

Haiping Wu[1], Eunjung Park[1], Mihailo Kaplarevic[1], Yingping Zhang[2]
Murat Bolat[1], Xiaoming Li[1], Guang R. Gao[1]

[1]University of Delaware
Dept. of Electrical and Computer Engineering
Newark, DE 19716 USA
{hwu, epark, kaplar, murat, xli, ggao}@ece.udel.edu

[2]Intel Cooperation
Digital Enterprise Group
Chandler, AZ 85226 USA
ying.m.zhang@intel.com

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 set of optimization compiler options could be applied.

In this paper we propose a method for automatic detection of performance sensitive program segments based on program segment similarity. First we create a proxy segment template database trained over a set of random input programs. The compiler identifies program segments by correlating them to the pre-build proxy segment templates using the syntax structure and architecture-dependent behavior similarity. We argue that the identified program segments can be custom optimized to improve the overall program performance.

The method is evaluated 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.

1 Introduction

The highest default optimization level, such as -O3 in GCC, does not necessarily produce the best performance speedup for all application [2, 8, 10, 12, 14]. Further performance improvement could be achieved by carefully choosing optimization options customized to performance sensitive program segments [3, 6].

1-4244-0910-1/07/$20.00

The key challenge is how to automatically identify performance sensitive program segments to which customized set of optimization options could be applied. To address this challenge we propose a machine learning method for automated detection of performance sensitive program segments based on program segment similarity.

The strategy behind our method is to pre-build a database of proxy program segments trained over a randomly selected set of programs to find their optimized set of optimizations (OSO)[1]. A compiler divides the input program into segments and correlate them with the proxy program segments based on the similarity between the processed program segments and the proxy program segments. The processed program segments are then compiled using the OSOs of their corresponding proxy program segments.

The kernel of our method is the similarity analysis of the processed program segments and the pre-build proxy program segments. In this paper, we define two types of program segment similarity: syntax structure similarity, and architecture-dependent behavior similarity. Only the program segments that have both types of similarity to the pre-build proxy program segments are identified and custom optimized.

The paper is organized as follows: In Section 2, we present the method for creating proxy segment templates; Section 3 discusses identification of optimization sensitive 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.

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

2 Proxy Segment Template Creation

This section describes the creation of proxy segment template.

A proxy segment template consists of those program segments which have similar syntax structures and share a unique OSO.

2.1 Program Segment Data Structure

The proposed method completely relies on a static analysis of an input program. 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 shown in Figure 1.

Type	The type of statement structure
Number	The number of statements included in this structure
End	The last node of this statement structure
Operands	The number of operands in this statement itself
Operators	The number of operators in this statement itself
Prev	Pointer to the previous node
Next	Pointer to the next node

The major statement structure types are

L : Loop statement structure
A : Assignment statement structure
C : Condition statement structure
S : Switch statement structure
G : Compound assignment statement structure

Figure 1. Data Structure used for Representing Program Segments

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

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

2.2 Syntax Structure similarity Detection

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.

The algorithm for detecting syntax structure similar program segments initially creates a pool of similar program segment (SPS-$pool$). 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 a match is detected, a syntax structure similar segment is found and the similarity weight value of the matched element in the SPS-$pool$ increments. 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.

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 associated OSO.

2.3 Program Segment Template Formalization

In our method, a processed program segment does not directly correlate with a proxy segment template. It correlates with an intermediate representation using syntax structure similarity. This intermediate representation is called program segment template.

A program segment template consists of a "pure" program structure stream representation and a weight sequence of architecture-dependent behaviors. The difference be-

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

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

The n in the sequence represents the number of $compounded$ statements that appear in the template. A $compounded$ statement consists of a continuous assignment statements. 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 a range of architecture-dependent behavior value (discussed in Section 3) of the corresponding $compounded$ statement in a program segment represented by the template.

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

Definition 3 In a k–$element$ program segment template, the instance for which each weight element has the minimum value is called the *lower bound instance* of this template; 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-$element$ program segment template, any program segment represented by the template can have k $compounded$ statements. The $i-$th pair of values in the weight sequence of the template describes the minimum value and the maximum value of the architecture-dependent behaviors for the $i-$th $compounded$ statement.

2.4 Creation of Proxy Segment Templates

A proxy segment template represents a subset of all program segment instances represented by the corresponding program segment template. 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 associated to each proxy segment template.

The differences between proxy segment templates and program segment templates are:

- The weight sequence of a proxy segment template is a subrange of the weight sequence of a corresponding program segment template.
- 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 program segment template, 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 proposed clustering algorithm divides the instance space of a program segment template into a set of subspaces.

The set the clustering criterion so that the lower bound instance and the upper bound instance in a subspace have the same OSO.

We have adopted *Mean Value* method to cluster the instance space of a program segment template to create the proxy segment templates. The detail of our clustering method can be referenced in [6].

3 Identifying Optimization Sensitive Program Segments

By building a proxy segment template database, a compiler can automatically identify the optimization sensitive program segments in the processed code. The identification mechanism 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 sensitive program segments.

3.1 Architecture-Dependent Behaviors

In this paper we focus on the correlation between optimization sensitive 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 sensitive 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 sensitive 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 1 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.

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 1. Intel XScale Architecture Characteristics and Quantified Value

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 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 succeeding 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 Sensitive Program Segments Detection

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

To support our method, the compiler's front-end needs to be modified 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 template 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 template. The modified compiler then searches the proxy segment template database that corresponds to this template 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 sensitive program segments in two stages:

- Stage 1: Syntax Structure Matching and Weight Sequence Creation

- Step 2: Proxy Segment Template Matching

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 precise-match, and 2) a fuzzy-match.

In the precise-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 precise-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 precise-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 sensitive segments can be recognized in the input code, the amplitude of the performance improvement depends on the following three factors:

- A similar segment recognition procedure must be applied after all program transformations are done, 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 sensitive segments are targeted "hot-spots".

4 Experiment

Our method is applicable to any platform, and should provide improved performance for a randomly chosen input C program. In this section, we evaluate the method on the Intel embedded XScale PXA255 architecture. The creation of the proxy template database is architecture dependent but it is done only once.

To config the experimental platform, we first create a proxy segment template database using 20 randomly selected sample programs that feature large fraction of loop structures. The reason for selecting this type of cases is that the current program segment templates are created from the loop segments. Then we integrate this database into GCC 3.3 compiler, modify the front-end of GCC 3.3 to identify and record program segments that match the proxy segment template database, modify the back-end of GCC 3.3 to dynamic adjust the optimization options to the program segments. Finally, we use 30 programs from 4 benchmark packages (CommBench, DSP kernel suite, Mediabench and Mibench) to evaluate the similarity match rates, performance speedup and compilation overhead.

4.1 Proxy Segment Template Database

Table 2 shows the results after the creation of the program segment templates from the set of sample programs. Using the program segment template formalization algorithm, 17 different templates are created. After analyzing 11 most frequently recognized templates, we found that: 1) 11 program segment templates appeared 847 times in the set of sample programs; 2) the minimum number of recognized program segment templates in the sample programs is 1; and 3) the maximum number of recognized templates is 182. The $frequency$ field stores the number of appearances for each of the 11 program segment templates in the set of sample programs.

N_c: Number of used cases
N_p: Number of template found

N_c	N_p	Characteristics of 11 templates			
		All	Min	Max	Frequency
20	17	847	1	182	<528,135,108,29, 10,1,18,10,6,1,1>

Table 2. Program Segment Templates in the Experiment

Program Segment Template Index	Number of Creating 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 3. Characteristics of Pre-Build Proxy Segment Templates

Based on the 11 program segment templates, we create 25 proxy segment templates. Table 3 gives the features of the proxy segment templates derived from the corresponding program segment template by the applied mean value clustering algorithm. This table shows the number of proxy segment templates derived from each program segment template. The performance improvement field gives the minimal and the maximal performance speedup over the GCC -O3 among the proxy segment templates in each program segment template.

4.2 Template Match Rate Evaluation

Based on the pre-build proxy segment template database, we derive the match rates for the candidate program segments (matches to the program segment templates and matches to the proxy segment templates). Table 4 indicates the results of this analysis. The total of 3514 candidate program segments were detected, where 594 of them match program segment templates. The match rate is 16.9%. Among these matched segments, 441 and 546 segments are

Template Matching Strategy	Number of Candidate Segments	Program Segment Template	Proxy Segments Matched
Precise-match	3514	594	441
Fuzzy-match	3514	594	516

Table 4. Proxy Segment Template Matched Rate

proxy segment template matched when using the precise-match strategy and the fuzzy-match strategy, respectively. The proxy segment template match rates are 74% and 87%, respectively.

4.3 Performance Speedup Evaluation

Among 30 tested cases originating from 4 benchmarks, there are 9 cases that show performance speedups over -O3 when either strategy is applied (the precise-match strategy and the fuzzy-match strategy). Figure 2 gives the performance speedups and the compilation overheads over the default -O3 option among 9 programs using the precise-match strategy and the fuzzy-match strategy, respectively. The average performance speedup is 1.13% and 2.31% for the precise-match strategy and the fuzzy-match strategy, respectively. Correspondingly, the average compilation overhead is 7.9% and 5.07% for the precise-match strategy and the fuzzy-match strategy, respectively. Consider the other 21 cases which show no performance speedup, they still consume extra compilation time over -O3. The average compilation overhead for all the 30 testing cases is 6.21% and 4.34% for the precise-match strategy and the fuzzy-match strategy, respectively (do not show in the figure).

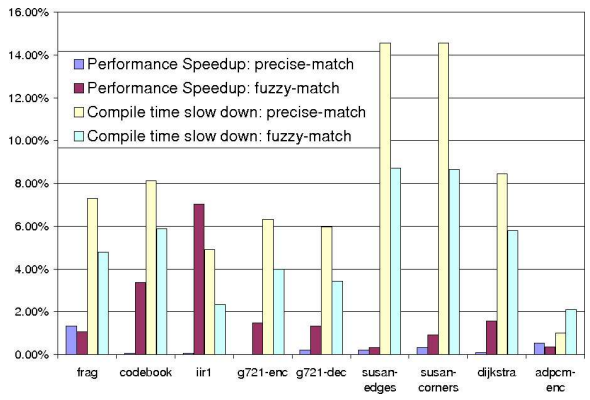

Figure 2. Performance Speedup & Compilation Overhead over GCC -*O3*

4.4 Analysis and Estimation

Based on the experimental results, the fuzzy-match strategy shows better performance speedup than the precise-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.

On the other hand, 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 (not enough proxy templates are created), we have to ignore certain optimization opportunities in the GCC compiler. We are continuously working on resolving these problems and hope to improve the results by enlarging the proxy segment template database.

5 Related Work

Wu et al. [3] proposed a technique for an automated search for optimization options. It creates a database for each application domain by turning a set of kernel programs from this domain. The optimized sets of optimization options are used to compile the other domain-specific matching programs. Cavazos and O'Boyle [4] presented another method for an automated search for optimization options. It focuses on method (function) level instead of the whole program. The matching strategy is based on the similarity of pure method features. What makes our method different is that we focus on a fine-grain level program analysis through program segments. We narrow down the similarity of program segments to the syntax structure and further more we correlate the segment features to the underlying architecture behaviors.

Annavaram [11] and Lau et al. [7] 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, 9] 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 a 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 program segment.

6 Conclusion and Future Work

We propose an automated method for performance sensitive program segments detection based on similarity be-

tween identified program segments and the records stored in a pre-build proxy segment template database.

We present a mechanism for a compiler automatically to capture the performance-sensitive program segments in arbitrary input programs by the use of syntax structure and architecture-dependent behavior similarity analysis.

We evaluate the applicability and performance of our method 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 a practical proxy segment template database.
- Develop a more accurate proxy segment template clustering algorithm.
- Revise compiler so that it can fully support our method.
- 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). Thanks to the anonymous reviewers for their helpful comments on draft of this paper.

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] Hai 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] John Cavazos and M.F.P. O'Boyle. Method-specific Dynamic Compilation using Logistic Regression. In *OOPSLA'06*, Portland OR, US, Oct. 2006.

[5] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M.F.P. O'Boyle, J. Thomson, M. Toussaint and C.K.I. Williams. Using Machine Learning to Focus Iterative Optimization. In *CGO'06*, New York NY, US, March. 2006.

[6] Hai P. Wu, E. Park, Murat Bolat, Mihailo Kaplarevic, Ying P. Zhang, Xiao M. Li and Guang R. Gao . An Automatic Methodology for Program Segment-based Compiler Optimization Search, *Technical Memo071*, CAPSL, Unive. of Delaware, Nov. 2006.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[14] 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.