

**Profile Merging and Code Versioning for  
Automated Profile Guided Optimization Systems**

by

**Rahul Saxena**

B.E., Mumbai University, 2003

A thesis submitted to the  
Faculty of the Graduate School of the  
University of Colorado in partial fulfillment  
of the requirements for the degree of  
Master of Science  
Department of Electrical and Computer Engineering  
2007

This thesis entitled:  
Profile Merging and Code Versioning for Automated Profile Guided Optimization  
Systems  
written by Rahul Saxena  
has been approved for the Department of Electrical and Computer Engineering

---

Professor Daniel A. Connors

---

Professor Manish Vachharajani

---

Professor Amer Diwan

---

Date \_\_\_\_\_

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Saxena, Rahul (M.S., Computer Engineering)

Profile Merging and Code Versioning for Automated Profile Guided Optimization Systems

Thesis directed by Professor Daniel A. Connors

Traditional Profile Guided Optimization schemes (PGO) require developers to manually select profiles which are representative of typical program executions, in order to optimize the final release binary. Most profile merge algorithms use a weighted averaging scheme to accumulate profile data into a composite (merged) profile. However, such simple additive schemes have been shown (Wang et. al) to degrade the effectiveness of profile information by diluting branch biases. The requirement of an accurate merge algorithm becomes all the more important in the context of Managed Run-time Environments (MRTE). Within such systems PGO schemes are required to be deployed on client machines and have to function automatically and transparently to user intervention. To address these criteria, we propose a detailed merge heuristic which resolves conflicts in branch by examining factors such as predictability of branch bias and branch weight. This algorithm is conservative on branch bias when bias information from different profiles are conflicting. The algorithm also takes into account the repetitiveness of execution runs to more accurately favor the behavior of typical execution scenarios. We hypothesize that our proposed merge algorithm can improve the quality of a composite profile by maximizing performance gain across input data sets while minimizing regressions for individual runs. We are currently in the process of implementing the proposed merge algorithm and ongoing work will focus on evaluating its performance benefit as compared to traditional schemes.

## Contents

### Chapter

<b>1</b>	Introduction	1
	1.1 Profile Guided Optimization for Managed Run Time Systems . . . . .	1
	1.2 Enabling Code Versioning . . . . .	2
<b>2</b>	Background – The Role of Profile Merging in Profile Guided Optimization	4
	2.1 Traditional Profile Guided Optimization Techniques . . . . .	4
	2.2 Profile Guided Optimization in Managed Run-time Systems . . . . .	6
	2.2.1 Requirements for Profile Guided Optimization for MRTE . . . . .	8
<b>3</b>	Motivation – An Intelligent Profile Merge Algorithm	10
	3.1 Need for Intelligent Profile Merging . . . . .	10
	3.2 Classifying Profile Data . . . . .	12
	3.2.1 Previous Work on Profile Comparison . . . . .	12
	3.2.2 Using Manhattan Distance to Calculate Profile Similarity . . . . .	14
<b>4</b>	Implementation of Profile Merging	15
	4.1 Profile Collection . . . . .	15
	4.1.1 An examination of Hardware Sampled Profiles . . . . .	15
	4.1.2 Software based Instrumentation . . . . .	15
	4.2 A Heuristic for Profile Merging . . . . .	15

4.2.1	Detecting Similarity between profiles . . . . .	17
4.2.2	Filtering Outliers . . . . .	17
4.2.3	Resolving Conflicting Biases . . . . .	19
<b>5</b>	<b>Employing Profile Similarity to Enable Code Versioning</b>	<b>20</b>
5.1	Motivation for Code Versioning . . . . .	20
5.2	. . . . .	20
<b>6</b>	<b>Summary and Conclusion</b>	<b>22</b>
	<b>Bibliography</b>	<b>23</b>

## Tables

### Table

## Figures

### Figure

2.1	Compiler based Profile Guided Optimization . . . . .	5
2.2	Managed Run-time based Profile Guided Optimization . . . . .	7
3.1	Profile degradation due to additive merging . . . . .	11
4.1	Similarity when comparing reference input 3 other reference inputs for 253.perlbnk in SPEC CPU2000 benchmark suite . . . . .	16
4.2	Similarity when comparing reference input 4 with other reference inputs for 403.gcc in SPEC CPU2006 benchmark suite . . . . .	16
4.3	Detecting Abberant Profiles . . . . .	18
5.1	Profile differences when comparing train input 1 with other inputs for 253.perlbnk in SPEC CPU2000 benchmark suite . . . . .	21
5.2	Profile differences when comparing reference input 4 with other reference inputs for 403.gcc in SPEC CPU2006 benchmark suite. . . . .	21

## Chapter 1

### Introduction

Traditional compiler based Profile Guided Optimization (PGO) techniques are characterized by a two-step process which involves (i) executing programs for gathering execution profile data, and (ii) employing a profile merge algorithm to generate a composite profile which is then use to recompile the final binary. Such systems rely on the application developer to select profiles which are typical of usage scenarios. A judicious selection of profiles is important because performance gains can only be realized for input sets where the execution profile is a subset of the composite profile. The merge algorithms employed by PGO use weighted averaging schemes to construct the composite profile. However, for scenarios where a large number of profiles are used during merging, the weighted merge (additive) heuristic has been known to dilute the branch biases and hence reduce the utility of the composite profile[10].

#### 1.1 Profile Guided Optimization for Managed Run Time Systems

The process of manually selecting profiles and the performance of the additive merge algorithm in optimizing the final executable prove to be limitations for applying PGO to Managed Run Time Environments (MRTE). MRTE present a new paradigm where part of the program compilation process is carried out at execution time and under control of the run-time environment. Hence, for PGO schemes to be successfully applied to MRTE, they must work independently and transparently of any user intervention.

The proposed algorithm is designed to satisfy these requirements.

The algorithm can be broken down into three main steps:

- (1) Filtering outliers (aberrant execution runs) from the profile database.
- (2) Detecting similarity for identifying frequently executed runs.
- (3) Resolving conflicting branch biases to generate a merged profile.

The first two steps of our merge algorithm deal with problem of actually selecting representative profiles. Firstly, we eliminate outliers by removing profiles which have marginal execution counts. The rationale behind this is to eliminate profiles which are generated due erroneous or exceptional runs. Secondly, we employ Manhattan Distance for detecting similarity between profiles. This allows us to count the frequency of execution of unique profiles. We experimentally verify Manhattan distance to be an effective metric for this purpose. Thirdly, the actual merge algorithm consists of resolving conflicts for individual branch biases by considering metrics such as Branch Bias, Branch Weight and frequency of execution of profiles. The overall goal here is to improve the quality of the composite (merged) profile and maximize performance gain across input sets while minimizing regressions for individual runs.

## 1.2 Enabling Code Versioning

Code versioning has been used recently to propose systems where trace level or function level code specialization is used to provide code which has specialized performance characteristics. For example, specialized code versions can provide better execution performance under specific input conditions.

Traditional Profile Guided optimization systems can also be thought of code versioning systems where only one version of the code exists in the binary. But with the advent of VM based run-time systems Its possible to monitor run-time performance of

a system using light-weight instrumentation systems or hardware based PMU systems to study the dynamic execution performance and switch between multiple versions of code depending on required behavior.

## Chapter 2

### Background – The Role of Profile Merging in Profile Guided Optimization

Talk about background of PGO, Mike Smith's paper on overview of profile feedback.

#### 2.1 Traditional Profile Guided Optimization Techniques

A traditional compiler based Profile Guided Optimization (PGO) system follow a 3 step approach to compiling optimized code. The overall organization of this approach is shown in Figure . As shown in the figure, the first step of PGO involves compiling the executable code with profiling turned on. The GNU compiler sytem, gcc is capable of profiling branch edges and certain expression values [3]. Similar profiling features are available with the Intel [6] and Microsoft [8].

The second step of the PGO compilation process consits of executing the profiled binaries with training inputs. Its usually referred to as the training stage. This aim is to collect execution profiles for input data sets which are representative of typical execution behavior. This is the most critical aspect of PGO; the optimization decisions are directly influenced by the profile data collected at this stage.

The final step of in PGO based compilation is the recompilation of the executable while using the profile data which is gathered from the taining stage. Optimizations such as Basic Block Layout, Function Layout, Conditional Branch Optimization, Dead

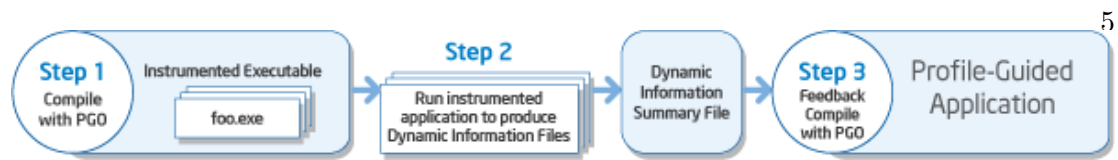


Figure 2.1: Steps in a traditional compiler based Profile Guided Optimization (PGO) system

Code Separation, EH Code Separation, Inlining, Virtual Call Specialization and Register Allocation.

The Microsoft compiler can additionally compile executables using Link Time Code Generation (LTCG). LTCG essentially consists of delaying code optimization until the link stage of compilation. The compiler initially compiles individual source files into an intermediate language of representation. Profile information can then be used in conjunction with LTCG to further improve the quality of optimized code. The code performance under LTCG is better because at that point the compiler has the most complete picture of the code flowgraph which can be possibly known at compile time.

Profiles which are collected during the training runs vary between themselves depending on the input data. This is shown in ???. Because the effectiveness of profile guided optimization directly depends on the profiles, it is essential to ensure that ??? the collected profiles are representative of a wide range of input data behavior and usage scenarios. Infact it has been show in previous studies by [?] that PGO systems which employ unrepresentative profiles for optimizing the final binary can actually degrade program performance. Current commercial tools such as the Microsoft Compiler PGO tools [8] can combine multiple profiles from multiple input scenarios using a simple additive method or a weighted additive method.

\*compile once run many times scenario

## 2.2 Profile Guided Optimization in Managed Run-time Systems

Managed run-time systems (MRTE) are best exemplified by the Java Virtual Machine [4] or the Microsoft .NET Framework[9]. Such environments provide a virtual execution environment for the execution of programs. This virtual environment provides another layer of abstraction for executing code which is above the machine abstraction provided by the operating system. This abstraction allows programs to execute within the virtual machine independently of the underlying operating system facilities and architecture. But run-time abstraction also necessitates that program code be stored in some intermediate representation, from which it can be compiled into native machine code. Intermediate code can then be compiled for any target machine for which a Virtual Machine implementation exists.

The process of translating the intermediate representation can happen at run-time. This is usually referred to as Just in Time (JIT) compilation. However the translation can also take place at install time. In this scheme the program code is translated to native code well before execution and cached on non-volatile memory. Moreover, the virtual machine can choose to re-translate the intermediate representation at any time, as needed. It is essential to differentiate between these implementation schemes because PGO based recompilation techniques can be optimized to minimize recompilation in install time schemes.

Infact, translation caching schemes have also been proposed for traditionally JIT based systems and significant performance improvements are achieved by the persistence of translated code. Hence, recompilation strategies for PGO based MRTE can become a significant aspect of PGO implementations.

PGO when applied to MRTE presents several challenges which are not present in traditional PGO systems. A generic run-time execution system which employs PGO is shown in Figure . The challenges for deploying PGO on MRTE can be summarized as

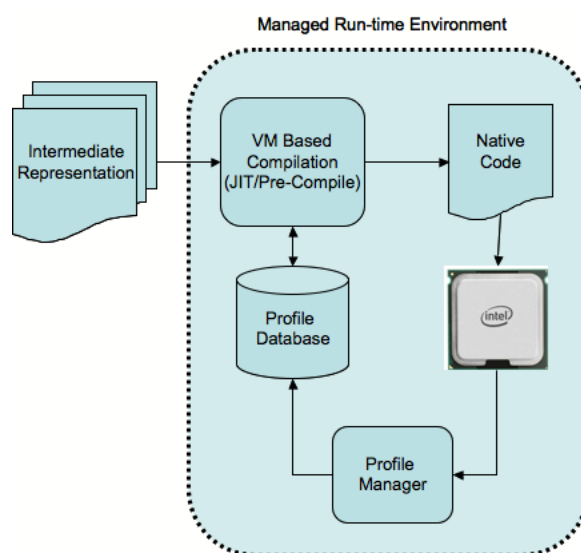


Figure 2.2: Steps in a managed run-time based Profile Guided Optimization (PGO) system

follows:

### 2.2.1 Requirements for Profile Guided Optimization for MRTE

**No user intervention:** MRTEs like the JRE and the .NET Framework are designed so that only the program execution behavior is visible to the user. Details about code compilation and translation details pertaining to organization of the VM are abstracted away from the user. This implies that any PGO mechanisms would have to function independently and be immune to user actions.

Implication of MRTE on PGO.

\* Decision making process in MRTE

**Profile Collection:** MRTEs can potentially collect execution profiles at every program execution. One of the aspects of collecting profiles is to determine a strategy for deciding where execution profiles should be collected during an instance of execution.

**Profile Classification:** The problem of storing the possible multiple number of execution profiles is partially answered by being able to classify and select execution profiles which are appropriate for program optimization.

**Regressive Behavior:** As is the case with traditional compiler optimization systems, there is always a possibility of causing performance regressions when a program is optimized to address one particular execution trait. MRTE must guard against introducing such regressions in program behavior.

**Optimization Decisions:** When does one decide to optimize the binary.

The challenges in applying PGO to MRTE can also be seen as a super-set of issues which arise when PGO need to be applied to automated compilation systems which follow the traditional, **compile once execute many times** scenario.

What are the drawbacks of PGO ? -Multiple Executions runs -Availability of Representative Inputs -Onus of Input set selection lies with Developer.

Contributions of this paper

-A Profile Merge Heuristic which is suited for MRTE. -Identifying the drawbacks of additive profile merging

-Using manhattan distance as a measure of similarity. -Similarity can be used to optimize the storage of profiles

-A study of Versioning on code performance characteristics.

## Chapter 3

### Motivation – An Intelligent Profile Merge Algorithm

#### 3.1 Need for Intelligent Profile Merging

As shown in 2, the process of profile merging is essential to the process of code optimization using run-time profiles. Using unrepresentative profiles can cause performance degradation in the optimized binary.

Currently available compiler tool chains such as the ones provided by Microsoft, Intel and GNU systems employ an additive scheme for combining the various profiles collected from a series of training runs of a program to come up with a single representative profile. This profile is then fed back into the final compilation phase of PGO to optimize the final binary.

However, this additive scheme of combining profiles suffers from the shortcoming that accumulating profile data by algebraic addition can eventually degrade the information content of the resultant profile.

Figure demonstrates the effects of algebraic profile combination. Consider the edge-profile data for a branch hammock for profiles A and B. In profile A the execution of this branch is biased 900 to 100 towards basic-block A. Profile B shows the edge profile of the same branch under a different execution circumstance. Here the branch is again biased 900 to 100 but towards basic-block B. If the above two profiles are algebraically combined

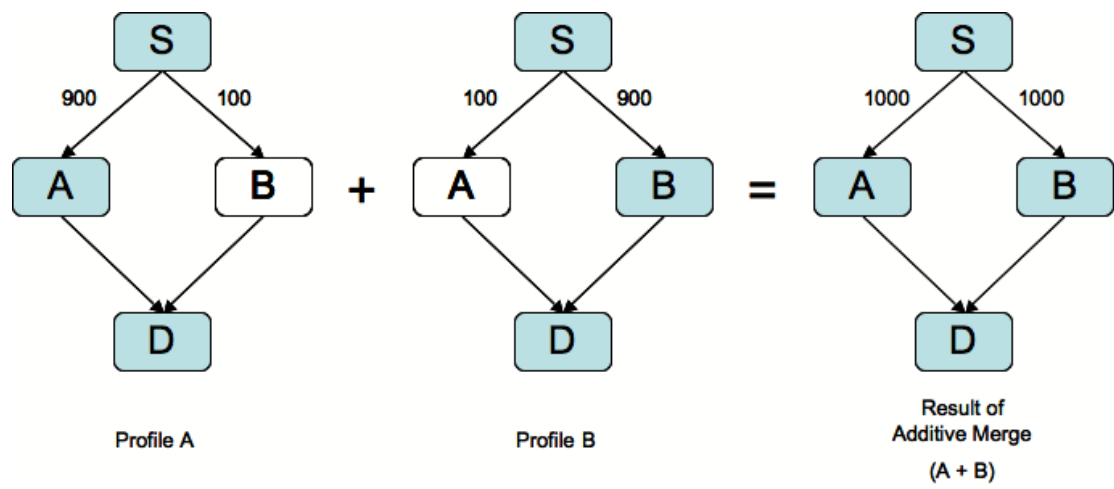


Figure 3.1: Profile degradation due to additive merging

to come up with a merged profile we see that for the branch under consideration the calculated edge profile is biased 1000 to 1000. The edge profile for this branch has changed from being highly biased (90%–10%) to completely unbiased.(50%–50%). This change in bias information can lead to performance degradation for the optimizing passes in the compiler because, optimization algorithms such as the ones used in code-layout or function call specialization cannot cannot optimize in favor of either profile.

Profile degradation can become specially prominent if an additive scheme is used for combining profiles in an automatic profile guided optimization systems, such as ones which are deployed for managed run-time systems.

For managed run-time systems to effectively utilize profile information two aspects related to profile data are addressed.

- (1) Effective classification of execution profiles.
- (2) Heuristic based algorithm for profile merging

## **3.2 Classifying Profile Data**

As shown in the following sections, the ability to classify profiles relative to each other plays a central role in the heuristic based merging algorithm which we use to arrive at a merged profile.

For the purpose of evaluating the

### **3.2.1 Previous Work on Profile Comparison**

Kistler and Franz [7] have previously proposed using vector based approach to quantifying profile data. This approach also proposes to represent profile data into an  $N$  Dimensional vector representation and further proposes work-arounds to circumvent degenerative cases which arise from using profile data.

Wang and Rubin[10] study user specific execution profiles of interactive applications on a Windows NT platform using FX!32[5]. They observe that it is possible to combine too many user specific profiles and over generalize profile data and cause a performance regression. They specifically attribute the performance regression to sub-optimal code layout due to extra procedures which are included during translation.

The problem of profile combination has been studied by Fisher and Freudenberger. [2]They have looked at different ways to combine profiles into a single hybrid profile by employing **unscaled**, **scaled** and **polling** approaches. The unscaled approach uses arithmetic addition to add frequency counts from different input runs. The scaled approach works by normalizing the frequencies before addition, while the polling approach works by employing a majority voting scheme which is used to make decisions when building the merged profile.

In chapter 4 we will see that the proposed merge heuristic is in fact a combination of employing **scaled** and **polling** techniques for selecting combinations between profiles.

Recent work which is most similar in spirit to the work presented in this thesis has been put forth by Arnold and Rajan[1]. They describe an architecture for augmenting a Java Virtual Machine implementation with a profile repository which allows the VM to remember profile data and optimization decisions across program executions.

However the work presented in this thesis encompasses the general problem of examining profile similarity with a view of classifying program behavior. The work by Arnold and Rajan limits program inputs to either “short” or “long” running executions without delving into the nature of underlying execution profiles. With our work of examining profile data directly we aim to establish a greater understanding of program behavior.

### 3.2.2 Using Manhattan Distance to Calculate Profile Similarity

**Manhattan Disatance** or  $x - y$  distance can be defined as the absolute distance between two points in an Euclidean space with a Fixed Cartesian coordinate system. For example, the manhattan distance between points  $P_1 (x_1, y_1)$  and  $P_2 (x_2, y_2)$  in a two dimensional space is given by  $|x_1 - x_2| + |y_1 - y_2|$ .

The edge profile of the dynamic execution of a given binary (for a particular input set) can be described as a set of values which are defined by an edge and the count of dynamic execution traversals by that edge. Mathematically, an edge profile consisting of  $N$  edges forms an  $N$  dimensional vector where the dynamic execution count of each edge forms the magnitude of the vector in that dimension.

$$P_N = \sum_{i=1}^N |x_i - y_i|$$

where  $x_i$  and  $y_i$  represent edge probabilities which constitute profiles  $X$  and  $Y$ . An edge probability  $p_i$  is given by

$$p_i = \frac{N_i}{\sum_{i=1}^N N_i}$$

However to usefully compare profiles which are generated from separate input runs it is important to normalize the edge frequencies within a given profile. Normalized edge frequencies allow for comparing the relative edge frequencies irrespective of the absolute edge execution counts.

In the next chapter we present a sample Profile Merging algorithm which employs the notion of Manhattan distance for calculating the Similarity between profiles and hence allows us to develop further heuristics and facilitates a more efficient profile merging algorithm.

\*Show the lost opportunity in optimization by showing the final bias difference of intelligent merge and simple additive merge.

\*How can i use PMU profile convergence as motivational data ?

## Chapter 4

### Implementation of Profile Merging

#### 4.1 Profile Collection

##### 4.1.1 An examination of Hardware Sampled Profiles

##### 4.1.2 Software based Instrumentation

#### 4.2 A Heuristic for Profile Merging

As outlined in section 2.2 for PGO to be efficiently deployed on MRTE several strategies need to be developed with regards to generating profiles, merging profiles and utilizing profiles to optimize code.

In this section we examine a possible strategy for analyzing execution profiles in a MRTE. We further concentrate our analysis with the aim of utilizing profile data to optimize execution performance.

The basic steps in our proposed analysis of profile data are outlined below:

- (1) Detecting similarity for identifying frequently executed runs
- (2) Filtering outliers from profile database.
- (3) Resolving conflicting biases to generate a merged profile.
- (4) Generating Versions Based on Profile Similarity. (Next section)

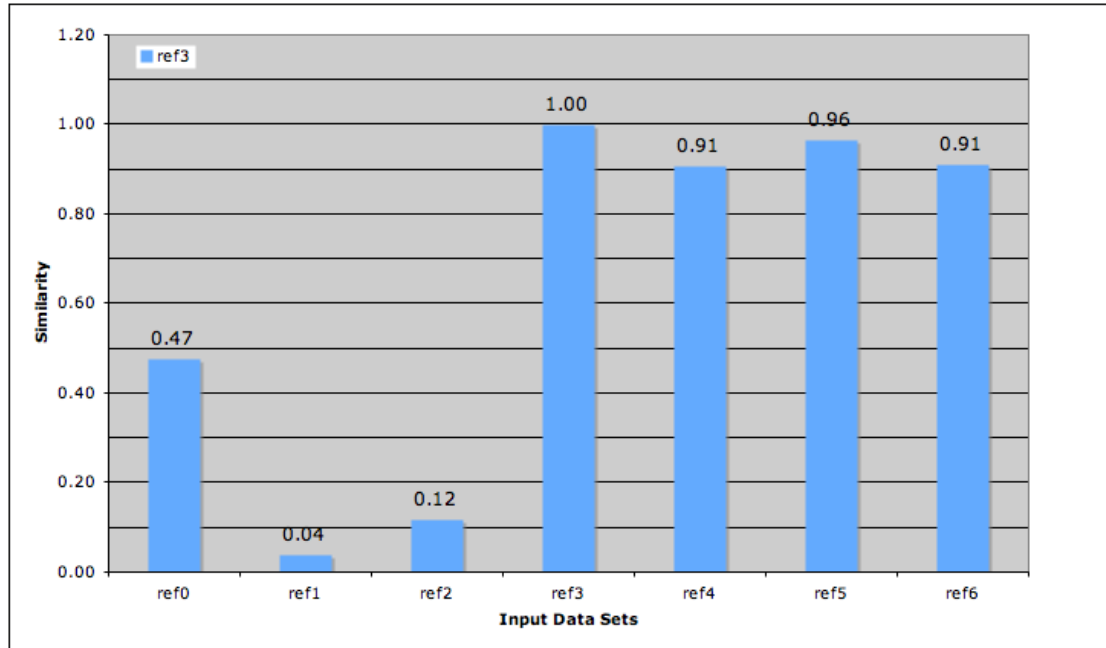


Figure 4.1: Similarity when comparing reference input 3 other reference inputs for 253.perlbnk in SPEC CPU2000 benchmark suite

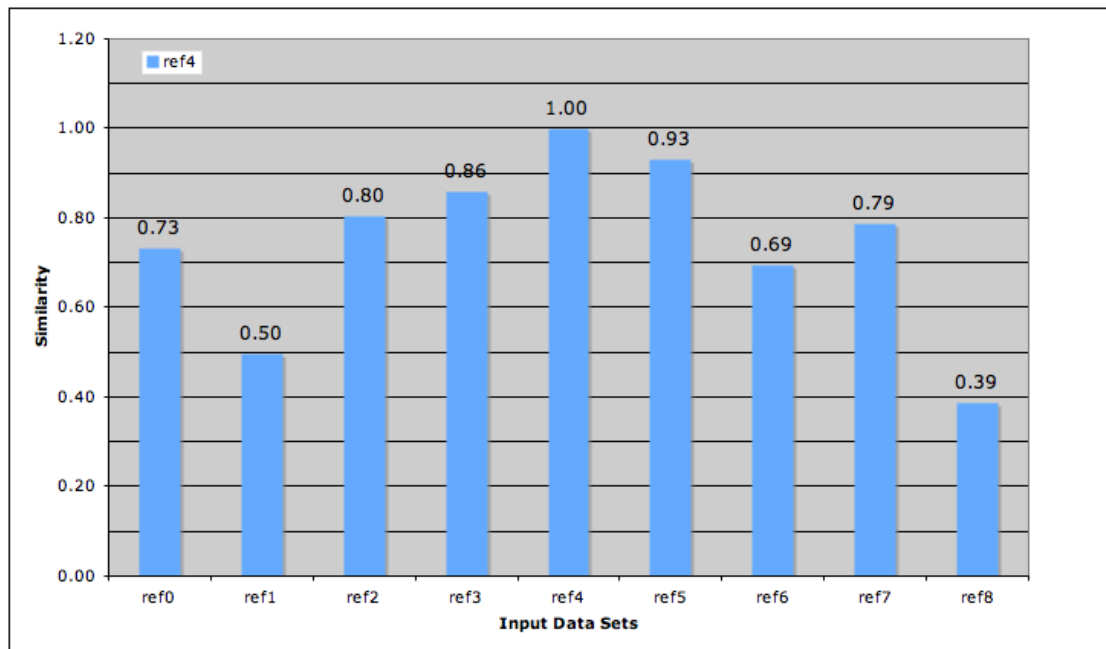


Figure 4.2: Similarity when comparing reference input 4 with other reference inputs for 403.gcc in SPEC CPU2006 benchmark suite

### 4.2.1 Detecting Similarity between profiles

As described in ??, we employ manhattan distance to calculate the similarity among all the profiles which are under consideration. Using this method multiple profiles which are similar can be collapsed and classified as a single representative profile. The representative profile can then be stored as part of a profile database and used for further similarity analysis and eventually in profile merging when required.

### 4.2.2 Filtering Outliers

The similarity between profiles is expressed in-terms of the comparison between the relative frequencies of edges there is no notion of profile relevance when computing profile similarity. Two profiles may be dissimilar to each other and hence form two distinct classes of profiles which should be separately considered when merging profiles to generate the final binary.

However, if the profile **A** has a relatively low instruction execution count as compared to profile **B** then the performance impact associated with optimizing the program binary using profile **A** can be negative to the overall performance of the program with the longer execution time.

One way to eliminate such profiles is to employ a filtering mechanism which determines whether the total edge execution count of profile is comparable in magnitude to the average execution count for profiles in the database.

Figure 4.3 shows a distribution of the number of branches retired during the execution of the train and reference input data sets of the *400.perlbench* binary in the SPEC CPU2006 benchmark suite. The average branch retired count across the different input set is  $\tilde{4}50,000$ . If we consider the high threshold for aberrant execution counts to be 5% we can safely eliminate *input 5*.

Its is intuitive that the benefit of optimizing code based on profile created by

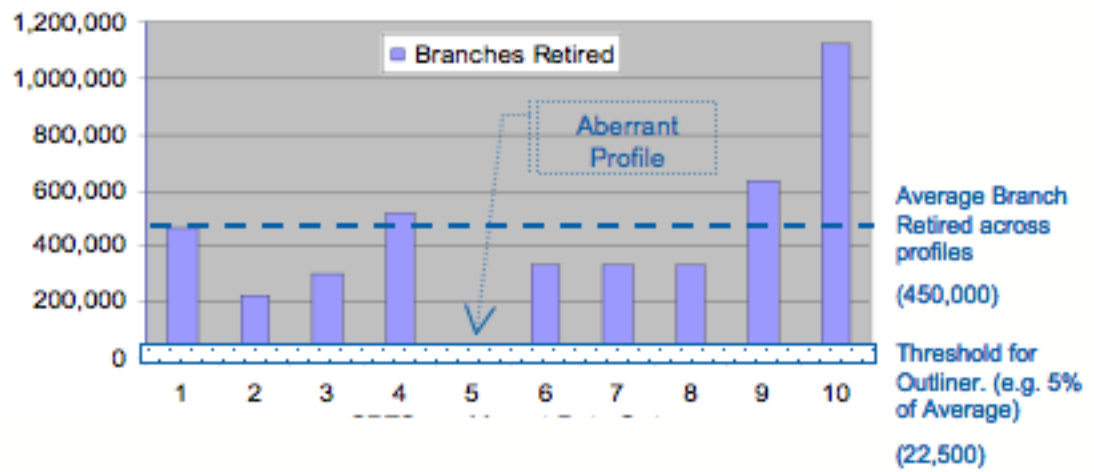


Figure 4.3: Detecting Abberant Profiles

execution of input data 5 will be minimal because of the low execution count. However, one of the challenges ?? of employing PGO on MRTE systems is also to detect shifts in code execution behavior where the instruction execution count of a binary varies significantly and possibly repetitively between execution instances.

### 4.2.3 Resolving Conflicting Biases

Describe implementation of profile based optimization for GCC Implementation

We chose edge-profiling, but path-profiling or even value profiling can be chosen to allow a comparison between two profiles.

## Chapter 5

### Employing Profile Similarity to Enable Code Versioning

#### 5.1 Motivation for Code Versioning

\*Code versioning can improve performance on input specific execution scenarios (software vendors can use profile data feedback to Distribute optimized code)

\*Add graph which shows possible function versioning based on edge profiles

In figures 5.2 5.1 we examine the cause of the differences between profiles of the train and reference input data sets of gcc and perl based benchmarks in the SPEC CPU benchmark suite.

The key observation for the gcc benchmark is that, the difference between profiles created by running difference input sets vary from each other mainly because of differences in edge profiles of the executed code and only to a minimal extent due to the input set exercising new execution paths which are not common between executions.

#### 5.2

\*Why are function level studies more relevant than trace level ones.

Experiments 0. Core2 PMU results ? 1. show per edge bias difference between intelligent merge and add merge

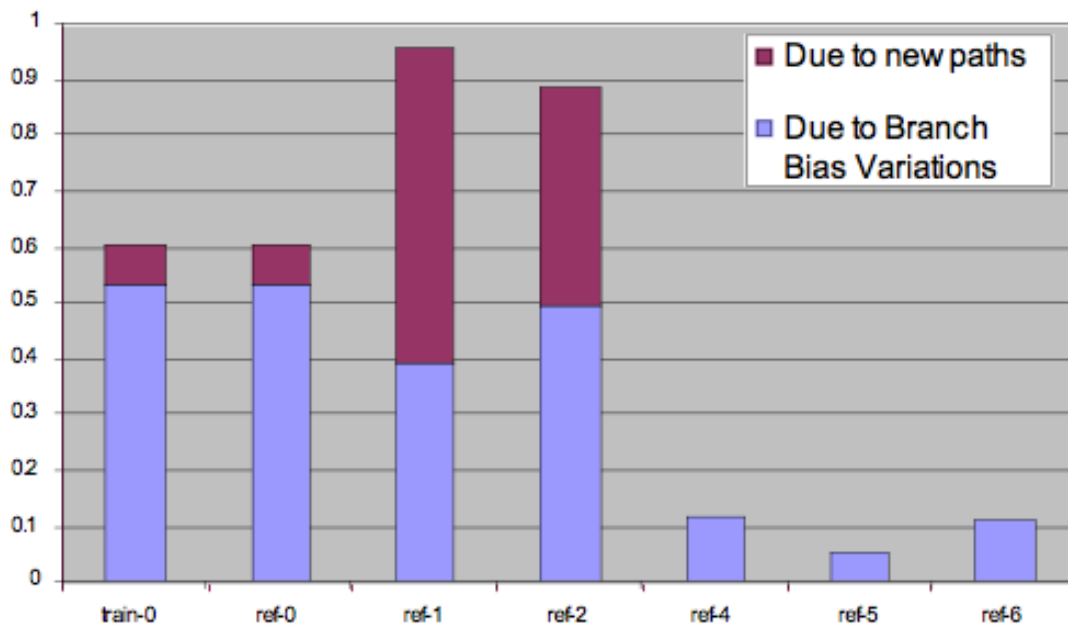


Figure 5.1: Profile differences when comparing train input 1 with other inputs for 253.perlbnk in SPEC CPU2000 benchmark suite

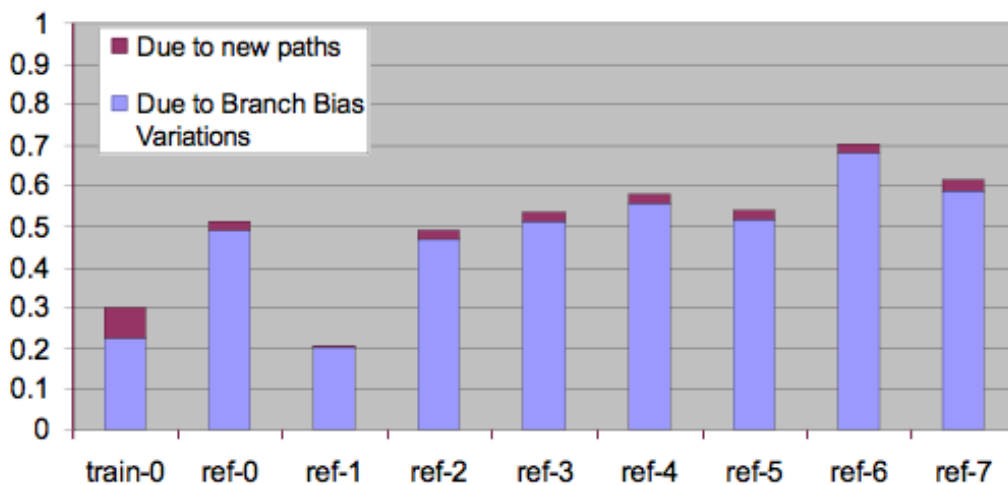


Figure 5.2: Profile differences when comparing reference input 4 with other reference inputs for 403.gcc in SPEC CPU2006 benchmark suite.

## Chapter 6

### Summary and Conclusion

Future Work:

IF each distinct profile is a vector, what is the profile of a vector which is completely opposite in bias to the merged profile ?

Is it possible to migrate profile information between recompilation runs ? –na  
this is done by BMAT.

## Bibliography

- [1] Matthew Arnold, Adam Welc, and V. T. Rajan. Improving virtual machine performance using a cross-run profile repository. In OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications, pages 297–311, New York, NY, USA, 2005. ACM Press.
- [2] J. A. Fisher and S. M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In Proceedings of 5th International Conference on Architectural Support for Programming Languages and Operating Systems, pages 85–95, October 1992.
- [3] GNU Project. Web site: <http://gcc.gnu.org/>, February 2007.
- [4] James Gosling, Bill Joy, and Guy Steele. The Java Language Specification. Addison-Wesley, Reading, MA, August 1996.
- [5] R. J. Hookway and M. A. Herdeg. Digital FX!32: Combining emulation and binary translation. Digital Technical Journal, 9(1), August 1997.
- [6] Intel Corporation. Web site: <http://www.intel.com/software/products/compilers/index.htm>, January 2007.
- [7] T. Kistler. Computing the similarity of profiling data. In Proceedings of the Workshop on Feedback-Directed and Dynamic Optimization, pages 53–66, 1997.
- [8] Microsoft Corporation. Web site: <http://msdn2.microsoft.com/en-us/vstudio/default.aspx>, January 2007.
- [9] Microsoft Corporation. Web site: <http://msdn.microsoft.com/net/>, January 2007.
- [10] Zheng Wang and Norman Rubin. Evaluating the importance of user-specific profiling. In Proceedings of the 2nd USENIX Windows NT Symposium, pages 21–30, August 1998.