Exploring the Potential of Performance Monitoring Hardware to Support Run-time Optimization

by

Alex Shye

B.S., University of Illinois, 2002

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

2005

This thesis entitled:
Exploring the Potential of Performance Monitoring Hardware to Support Run-time Optimization
written by Alex Shye
has been approved for the Department of Electrical and Computer Engineering

---

Daniel A. Connors

Prof. Manish Vachharajani

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.

Shye, Alex (M.S., Computer Engineering)

Exploring the Potential of Performance Monitoring Hardware to Support Run-time Optimization

Thesis directed by Prof. Daniel A. Connors

Run-time optimization defines the process of dynamically modifying an application's characteristics to promote desirable execution behavior. Since there is a wealth of information available at runtime which is unavailable to static compiler analysis, run-time optimization has substantially more potential to fully utilize processor resources. A critical component of run-time optimization systems is the run-time profiler which must accurately capture specific aspects of application execution behavior while maintaining a low overhead. Unfortunately, most existing profiling approaches cannot meet these constraints and therefore cannot feasibly be deployed in a run-time optimization system.

While modern microprocessors can collect run-time information through on-chip Hardware Performance Monitoring (HPM) support, it is not clear whether this technology can effectively guide a run-time optimization framework. To date the HPM information of various processor systems has almost solely been used in post-execution performance tools. This thesis evaluates the potential of performance monitoring hardware to support profiling for run-time optimization. The trade-offs in meeting the constraints imposed in a run-time environment are analyzed by evaluating various sampling rates and analysis techniques. Altogether, the thesis characterizes the amount of information available through PMU sampling as well as the extent in which compiler analysis can extend PMU information. Path profiling and code coverage analysis, important elements of run-time optimization, are evaluated to demonstrate the effectiveness of run-time profiling with hardware support.

Dedication

To my father, mother, and brother.

Acknowledgements

First and foremost, I would like to thank my advisor Dan Connors for his guidance in this work. With his deep insights, generous advice, and continuous encouragement, I have learned and accomplished more than I could have imagined when I first began graduate school. I look forward to continuing my research with him as I pursue a Ph.D degree.

I would like to acknowledge Andrew Pleszkun and Manish Vachharajani, members of my defense committee, for their valuable advice and suggestions.

I express my thanks to the entire DRACO research group. Matthew Iyer was instrumental to the development of ideas as well as the implementation of the work in this thesis. Alex Settle mentored me early on and was extremely helpful in teaching me research tools and skills. Vijay Janapa Reddi provided me with a lot of encouragement when I first started doing research. I thank Tipp Moseley, Joshua Kihm, Dave Hodgdon, Dan Fay, Jon Liu and Andy Janiszewski for their feedback throughout the research process and thank the whole DRACO research group as a whole for making the research lab fun and exciting.

I also thank Stephanie Tseeng. Her companionship and steady emotional support comforted me and kept me sane through graduate school.

Finally, I extend my deepest thanks to my father, mother, and brother. Their 25 years of love and support have shaped me into who I am today. I am grateful for everything they have done for me and dedicate the work in this thesis to them.

Contents

Chapter

1 Introduction 1

2 Background 5

2.1 Run-time Optimization Systems 5

2.2 Types of Profiles 7

2.2.1 Event Counts 7

2.2.2 Point Profiles 8

2.2.3 Fine-grained Profiles 9

2.3 Ideal Run-time Profiling Characteristics 10

2.4 Previous Profiling Approaches 11

2.4.1 Static Instrumentation 12

2.4.2 Dynamic Instrumentation 14

2.4.3 Most Recently Executed Tail (MRET) 15

2.4.4 Hardware-based Profiling 16

2.4.5 Summary of Hardware and Software Profiling Techniques 16

2.5 Hardware Performance Monitoring 17

2.5.1 Performance Monitoring Units 18

2.5.2 Implications of Hardware Sampling 20

2.5.3 Hybrid Profiling 21

3 PMU-based Profiling 23

3.1 Overview of PMU-based Profiling System 23

3.2 Program Annotation 24

3.3 Collection of Branch Vectors 25

3.4 Partial Path Creation 26

3.5 Compiler Analysis 26

3.5.1 Partial Path Extensions 27

3.5.2 Dominator Analysis 28

3.5.3 Path Profile Generation 31

3.6 Profile Information 37

3.7 Experimental Methodology 38

4 PMU-based Path Profiling 39

4.1 Generating an Estimated Path Profile 39

4.2 Effect of Sampling Period 40

4.3 Aggregating Data from Multiple Runs 41

4.4 Partial Path Characteristics 42

4.5 Actual Hot Paths 44

4.6 Accuracy Results 45

4.7 Summary 46

5 PMU-based Code Coverage 48

5.1 Code Coverage Methodology 48

5.2 Actual Code Coverage 49

5.3 PMU Code Coverage of a Single Run 49

5.4 PMU-based Code Coverage Stability 52

5.5 PMU Entropy Analysis 53

5.6 PMU Code Coverage with Multiple Runs 56

5.7 Summary 57

6 Future Work 58

7 Summary and Conclusion 60

Bibliography 62

Tables

Table

2.1 Existing approaches to profiling split into categories. 12

2.2 Itanium-2 PMU features and the filters available for each of the features. 19

3.1 Example of path matching of partial paths to region-based paths. 36

4.1 Average length of partial paths (number of instructions) initially, after partial path extensions, and after splitting on function boundaries and loopback edges. 43

4.2 Number of actual hot paths and their percent of total. 45

5.1 Number of instructions per benchmarks and actual code coverage. . . . 49

Figures

Figure

2.1 An example CFG with edge profile annotations. 9

3.1 Overview of compiler-aided, PMU-based profiling framework. 24

3.2 Example showing a branch vector, the corresponding partial path, and the extended partial path. 27

3.3 Partial path creation and path extension based on dominator analysis. . 28

3.4 (a) Dominator and (b) post dominator trees for control graph in Figure 3.3. Partial path blocks and blocks added from dominator analysis are shown. 30

3.5 Example of region formation. 34

4.1 Overhead of run-time collection of branch vectors and number of unique branch vectors for various sampling periods. 40

4.2 Number of unique paths found by aggregating data from runs with same input set. 42

4.3 Percentage of partial paths that cross function boundaries. 43

4.4 Accuracy versus sampling period. 46

5.1 Code coverage across different sampling periods (100K, 1M, 10M 100M) showing the effects of 1) using a single basic block per sample, 2) using branch vectors to create partial paths and 3) extending partial path information by using dominator analysis. 51

5.2 Code coverage for all types of coverage at a sampling period of 100K. . 51

5.3 Stability of percent of code coverage matched over 20 runs of each benchmark. Thick bars show the average coverage while the thin lines show the minimum and maximum percentages over the 20 runs. 53

5.4 Instruction execution distribution for **164.gzip** (a) actual and (b) PMU. 54

5.5 Entropy (Kullback-Leibler divergence) of actual/PMU coverage. 55

5.6 Coverage with execution thresholds. 55

5.7 Code coverage percentage from aggregating multiple runs using (a) regular sampling period and using (b) randomized sampling period of 100K. 57

Chapter 1

Introduction

Advances in programming languages and software design techniques create a trend towards increasingly complex software interactions during program execution. Object-oriented programming languages improve programmability but result in side effects such as delayed binding which makes static optimization difficult. Virtual machines (VMs) are growing in popularity but require an extra software layer in the form of a run-time interpreter or Just-in-time (JIT) compiler. In addition, a movement towards the usage of dynamically linked libraries (DLLs) adds extra indirection which traditional compilers cannot optimize and account for.

With this trend of increasing software complexity, it will be difficult for future computing systems to continue delivering performance improvements. Dynamic hardware optimization techniques such as out-of-order execution require hardware resources (i.e. instruction window and re-order buffer) which are difficult to scale. Object-oriented programming, VMs, and DLLs introduce complexities which limit the scope of traditional static compilers. In addition, aggressive profile guided optimizations are limited to performing optimizations specialized for specific input sets. As such, emerging run-time optimizations systems will play a crucial role in the performance improvements of next generation systems.

Run-time optimization systems have the unique ability to monitor application and

system behavior to automatically adapt and optimize programs. These systems have access to a wealth of run-time information which is not available to a traditional compiler. For example, run-time profile information can be used determine hot execution paths, identify run-time data values, and examine system behavior during execution. By analyzing profile information, the run-time system can know how to adapt system resources as well as drive aggressive profile-based optimizations (PBOs) such as superblock formation [26], code positioning [41], and function inlining [25].

While run-time optimization systems represent a great potential source of performance improvement, open issues remain regarding the run-time profiling mechanism for such systems. Specifically, it is unclear how the systems should determine what information to collect, how to collect it, and how to analyze it. However, it is clear that the success of a run-time optimizer depends greatly on a run-time profiling technology that incurs a low overhead while accurately capture a wide range of run-time program execution and system-level characteristics.

Unfortunately, most previous approaches at profiling are not suitable for use in a run-time optimization system. Software-based profiling techniques have typically been used in the traditional static compilation environment. These techniques are able to collect accurate profile information but incur high overheads. While a high overhead is manageable for static compilation, it is unacceptable for a run-time optimization system. At the other end, hardware-based profiling techniques are able to collect information at a low overhead, but are usually limited in a number of ways. First, they are limited in the information that they collect. Second, they are limited in the analysis that can be performed on the profile information. Complicated analysis is expensive to implement in hardware. Last, most proposed methods for hardware-based profiling do not exist in hardware eliminating the chance that they can be used in any current run-time optimization system implementations.

Modern microprocessors such as the Pentium-4, Itanium, and the Power PC 970

have begun to include support for Hardware Performance Monitoring (HPM) through special on-chip Performance Monitoring Units (PMUs). These PMUs allow for the configuration and sampling of hardware registers that can be used to provide run-time feedback information at the processor and system level. Although PMUs are generally used by software developers for performance tuning, they present a unique opportunity for leveraging existing hardware to create a hardware-software hybrid profiling system. In such a system, hardware is used for low overhead collect data collection and software is used to analyze the data to generate the profile. Hardware sampling ensures a low overhead while software allows for flexibility and deep analysis of the sampled data.

The work in this thesis utilizes the novel approach of using a compiler-aided, PMU-based, hybrid profiling system designed to support a run-time optimization system. In particular, the thesis uses the Itanium-2 PMU to periodically sample branch vectors. A branch vector is a set of correlated branch addresses which effectively represent a trace of program execution. The collected branch vectors are passed into a compiler infrastructure which is used to perform analysis on the branch vectors to create profile information. A compiler infrastructure is utilized because it contains a view of the entire program, as well as analysis routines, that are not available during run-time. The construction of such a system is demonstrated and then used to perform two case studies.

In the first case study, compiler analysis is used on the branch vectors to generate a PMU-based estimated path profile. A path profile is a list of paths in a program along with their associated execution counts. It can be used to determine hot paths of execution for PBOs such as superblock formation [26]. In the second case study, a different compiler analysis is utilized to perform code coverage analysis using the sampled branch vector information. Code coverage information may be useful for larger scale optimizations such as code placement [41] for improved instruction cache and Transition Look-aside Buffer (TLB) performance.

This thesis makes the following contributions:

(1) Characterize the information provided by sampling PMU branch vectors

(2) Characterize the amount a information a compiler infrastructure is able to add to the sampled PMU branch vectors

(3) Demonstrate the construction of a compiler-aided PMU-based profiling framework

The rest of this thesis is organized as follows. Chapter 2 provides background information on run-time optimization systems, profiling techniques, hardware performance monitoring and hybrid profiling systems. Chapter 3 describes the PMU-based profiling approach in this thesis as well as methods in which the compiler can provide valuable additional information. Chapter 4 presents experimental data related to the first case study of generating a path profile and Chapter 5 presents the PMU-based code coverage study. Chapter 6 outlines thoughts for future work and Chapter 7 concludes the thesis.

Chapter 2

Background

This chapter provides background information necessary for motivating and understanding the design decisions in the profiling framework used in this thesis. First, run-time optimizations systems are defined and then discussed to show the importance of the run-time profiler and how its requirements differ from standard offline profiling techniques. Next, ideal characteristics of a run-time profiler are discussed and then used to evaluate existing software and hardware approaches to profiling. Finally, hardware performance monitoring is discussed as well as existing hybrid profilers.

2.1 Run-time Optimization Systems

This thesis sets out to provide an efficient method of profiling for run-time optimization systems. In order to do this, it is important to first define what a run-time optimization system is as well as the requirements for the success of such a system.

Optimization systems may be divided into three major categories; static, dynamic and continuous. These term defined below:

> **Static Optimization System:** Static optimization implies that the optimization of a program is done in a one-time offline fashion. Examples of static optimization systems are classic static optimizing compilers and post-link optimization tools such as Ispike [35].

Dynamic Optimization System: Dynamic optimization is when a program is optimized during run-time. Typically, the dynamic optimizer monitors a specific application invocation, determines optimization opportunities typically in the form of hot spots or traces within the code, and then performs optimizations. The optimized traces are usually then placed into a code cache so that subsequent executions of the hot traces may run from the optimized forms from within the code cache. Examples of such systems are Dynamo [4], Mojo [14], DELI [17] and the ADORE dynamic optimizer [33].

Continuous Optimization System: A continuous optimization system can be viewed as a persistent combination of static and dynamic optimization. In a continuous optimizer, programs are continuously monitored and profiled. Optimizations are performed either offline, during idle processes or on idle processors, or online during execution. Upon completion of optimization, the next invocations of the programs utilize the new binary. If an instance of the program is still running, the newly optimized binary may be hot-swapped with the old binary. Examples are Digital Continuous Profiling Infrastructure [2] and Kistler's Continuous Program Optimization framework [31].

The term **run-time optimization system** refers to any system which effects a binary during execution or across multiple executions of an application. Therefore, dynamic optimization systems as well as continuous optimization systems can both fall under the definition of being run-time optimization systems.

Run-time optimization is an interesting domain of optimization because it has a number of benefits over traditional static optimization. There is a large amount of program information and behavior that is only known at run-time including knowledge of hot code, run-time values, as well as much of system state. System state will be particularly important in the upcoming chip multi-threaded and simultaneous multi-threaded chips which share on-chip hardware units. Traditional aggressive static optimizations attempt to optimize an application based on a test input set. If the test input does not match the behavior of a program run, performance may not be seen. In the worst case, performance may be decreased. Run-time optimization uses information from the current program run for optimization. This means that the run-time optimizer effectively

has perfect profile information. Also, run-time optimizers can be transparent and do not require the inconvenient extra compilation phases required by static optimizers.

There are three main tasks any run-time optimization system must perform which can contribute to run-time overhead. First, is the collection of profile information for a program during it's execution. Next, the profile information is used to recognize optimization opportunities and carry out these optimizations. Third, the optimized code is deployed so that program execution runs from the optimized code. The simple goal is to apply run-time optimizations where the performance benefits outweigh the overhead costs. If performed during run-time, all of these tasks are potential causes of overhead. However, the only task which is completely necessary to perform during run-time is the profiling. The overhead of optimization and deployment of the optimized code may be removed either by performing them offline during idle process time, or on a separate processor. Therefore, in lowering the overhead of a run-time optimizer, it is critical to focus on efficient low-overhead techniques for the collection of profile information.

2.2 Types of Profiles

The main purpose for collecting profiles is to understand program behavior and to use them uncover optimization opportunities to drive PBOs. However, depending on the specific PBO being deployed, different types of profile information are often needed. Profiling information can generally be split up into three main categories; event counts, point profiles, and advanced fine-grained profiles. These are described in the following subsections.

2.2.1 Event Counts

The simplest type of profile is one which keeps counts of coarse-grained events during program execution. These coarse-grained events may be microarchitectural events

such as branch mispredicts and cache misses, or system level events such as page faults. The counts are typically sampled periodically and used to represent program behavior over time.

Profiles with event counts can provide very useful high level information. One such application is determining phase behavior of a program. Programs are known to execute in distinct phases [42] where different phases represent different execution behavior patterns. Run-time optimization systems often attempt to determine phase transitions to know when to collect information and apply optimizations. Event counters have been shown to be successful in capturing phase information [19]. Event counters have also been shown to be effective in characterizing and showing correlations between system level and microarchitectural events [23, 44, 47].

2.2.2 Point Profiles

A point profile is one which locates specific points in a program to profile. The points may include individual instructions, branches, or functions. The majority of past research in PBO actually utilizes point profiles – specifically **edge profiles** [6, 11, 12, 22]. An edge profile is a profile in which branch bias information is captured for each branch executed within a program. It is simple to implement because it only requires a mechanism for maintaining counts for each time a branch is taken or not taken.

One use of point profiles is to determine hot paths for optimizations such as trace scheduling [22] or superblock formation [26]. Hot edges are commonly placed together to estimate hot paths in a program. Also, point profiles may be used to derive code coverage information such as function coverage, branch coverage or statement coverage. Coverage information may be used to for a high level of code placement such as placement of code onto pages. Another form of point profile utilization is for generating a call graph. If function calls are counted, they can represent call graph information and be used to drive inter-procedural optimizations such as function inlining [25].

2.2.3 Fine-grained Profiles

Fine-grained profiles refer to more sophisticated types of profiles which generally require more advanced hardware features and often more robust software algorithms. An example of a more advanced fine-grained profile is the path profile [5]. A path profile is a profile which contains a set of paths enumerated from the original control flow graph as well as their associated execution counts. As far as hot trace layout optimizations are concerned, path profiles have been shown to be more effective than edge point profiles [7].

Figure 2.1, which shows a control flow graph (CFG) annotated with edge profile information, provides an example of the difference between an edge profile and a path profile. The edge counts in the CFG indicate that the hot path contains basic blocks $ABDFG$. However, suppose that all of the paths through blocks ACD continued down through blocks F and G. In this case, the count for path $ABDFG$ would be 50, and would not be the 70 suggested by an edge profile. While a edge profile can only provide estimated hot paths which may be misleading, an actual path profile provides accurate counts of paths.

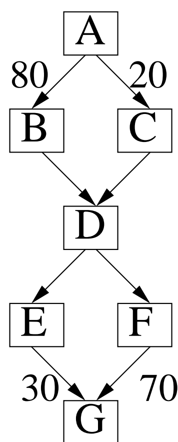


Figure 2.1: An example CFG with edge profile annotations.

A cache-miss profiles are another example of a more fine-grained profiling technique. In this case, more robust hardware features are required for collecting this type of information such as the Event Address Registers (EARs) on the Itanium-2 PMU [28]. PMUs are discussed in more detail in Section 2.5. The EARs can be used to sample instruction and data addresses for cache or TLB misses as well as their associated latencies. This information can be used to identify critical load instructions which frequently causes cache misses. This knowledge can then be used to drive memory optimizations. An example, is the ADORE dynamic optimization system [33] which uses the Itanium-2 PMU to collect a data cache miss profile to dynamically determine locations to insert memory prefetch instructions.

2.3 Ideal Run-time Profiling Characteristics

Now that run-time optimization systems and background on profiling has been discussed, it is time to discuss the characteristics which a profile mechanism for a run-time optimization system should have. The ideal characteristics for such a profiling system are listed and then detailed below. These characteristics are later used to evaluate previous profiling approaches.

- Low overhead
- High accuracy
- Broad applicability
- Transparency
- Low hardware cost and complexity

Low overhead is perhaps the most important of the characteristics. Low overhead is crucial because all overhead costs must be outweighed by performance gains.

Overhead is defined as the total amount of time necessary during program run-time for profiling.

High accuracy is defined as a profile's ability to reflect run-time execution. The accuracy of profile information is crucial to the success of PBOs. An accurate profile will uncover many optimizations opportunities for PBOs which can lead to drastic performance improvements. On the other hand, a poor profile can cause PBOs to perform poor optimizations which may lead to a decrease in performance.

Broad applicability refers to the ability to collect various granularities and types of profiles. This is important as run-time systems need to apply a wide range of optimizations at many different levels and granularities.

Transparency is the ability for the profiling system to collect information without perturbing the original application execution. This is important to different degrees depending on the type of profile needed. For example, if instrumentation code is inserted, it would not make a difference if a basic block profile is desired. However, if an instruction cache miss profile is needed, than the additional code fragments would perturb the cache and become an issue.

Low hardware cost and complexity is defined by the impact the profiling mechanism has on the hardware design cycle of a processor. There are two issues here. The first is the hardware real estate required. A larger design consumes more power and may take longer to design. The second issue is hardware complexity. The more complex the hardware, the longer the design as well as verification time.

2.4 Previous Profiling Approaches

Past profiling approaches can be split into three broad categories; 1) software, 2) hardware and 3) hybrid. As the name implies, software techniques employ a complete software solution for gathering profile information while hardware techniques are implemented entirely in hardware. Hybrid techniques typically use hardware to collect

Categories	Profiling Approaches
Software	Static Instrumentation Dynamic Instrumentation Most Recently Executed Tail (MRET)
Hardware	Conte's BTB Profile Buffer Hot Spot Detection Programmable Path Profiler
Hybrid	Digital Continuous Profiling Infrastructure (DCPI) Continuous Program Optimization (CPO) OProfile ProfileMe Vertical Profiling ADORE Dynamic Optimizer

Table 2.1: Existing approaches to profiling split into categories.

information and software to perform analysis. Table 2.1 shows the three categories of profiling techniques and the previous approaches which fit into each category. The hardware and software techniques are discussed in detail in this section. The hybrid techniques are discussed in the following section (Section 2.5) with the discussion on hardware performance monitoring.

2.4.1 Static Instrumentation

A common method of gathering profile information is through static software-based instrumentation. The program is initially compiled with the insertion of code fragments, or instrumentation code. The binary is then run to collect profile information. Static instrumentation is useful because it can be used to obtain highly accurate profiles. The reason for this is that any arbitrary instrumentation code may be placed at arbitrary locations in order to collect any information that is desired. An alternative to having a compiler insert instrumentation is to use a static instrumentation tools such as ATOM [21] which inserts instrumentation prior to program execution.

While software instrumentation can gather accurate profiles, a large disadvantage

is that it typically incurs a high profiling overhead. This occurs for two reasons. First, execution time is required for the insertion of instrumentation code. Second, during the profile run of the program, the instrumentation code must be run with the original code. The overhead due to simply running the instrumentation code itself can be quite large. For example, studies [6] show that instrumentation for a collecting an edge profile incurs about a 21% overhead. Note that this is just for a point profile. More advanced profiles such as path profiles would incur a higher overhead.

Because path profiles are a superior form of profile for hot trace layout optimizations, path profile instrumentation algorithms have been a very popular topic of research. The original path profiling algorithm was proposed by Ball and Larus [5]. In this algorithm, the CFG is divided into regions which are typically along function boundaries. Loop-back branches are removed, and replaced with a dummy edge from the head of the CFG to the loop header and another dummy edge from the loop exit to the exit of the CFG. The replacement of loopback edges with the dummy edges effectively creates a directed acyclic graph (DAG) from each region in which loop paths can be distinguished from full paths in the CFG. A single counter per region is then used in conjunction with a edge numbering algorithm to determine points of instrumentation that can be used to define distinct paths of execution. This added algorithmic complexity often causes path profile collection to require a larger overhead than simple edge profile collection. The Ball and Larus path profiler averaged a 31% slowdown, with slowdowns as high as 97% for gcc.

Targeted Path Profiling (TPP) [29] and Practical Path Profiling (PPP) [8], extensions of Ball and Larus path profiling, make an effort to decrease overhead in the context of staged dynamic optimization systems. Both are based on the idea that an edge profile from a previous stage of dynamic optimization can be carried into a path profiling stage, and used to locate obvious paths that do not need to be instrumented for path profiling. TPP lowers overhead to around 16% by ignoring these obvious paths.

PPP is an extension of TPP which ignores more paths and decreases overhead to an average of 5%. While these overhead numbers may seem low, remember that they do require information from an edge profile. This means that an extra stage is needed to compile for the edge profile and an additional extra stage is needed to run the edge profile enabled binary. Therefore, the overhead is much higher than the run-time overhead percentages suggest.

While static instrumentation is able to provide a highly accurate profile, the insertion of instrumentation code creates a high overhead and also results in non-transparency. The applicability is limited to information that can be collected by code insertion. Architectural behavior such as branch mispredicts and cache misses cannot be profiled by using instrumentation. While instrumentation also limits profiling to the application, more information can be collected if the kernel as well as the libraries are instrumented [23, 44, 47].

2.4.2 Dynamic Instrumentation

Instead of inserting instrumentation at statically, instrumentation can also be inserted during run-time using dynamic instrumentation. Dynamic instrumentation tools such as Pin [34], DynamoRIO [9] and Dyninst [10] may be used to inject arbitrary code fragments into arbitrary points in a running executable. While dynamic instrumentation removes the initial compilation phase for inserting instrumentation, it often results in higher run-time overhead due to the dynamic interpretation or compilation. PIN-instrumented binaries average slowdowns of 2.8x (up to 20x slowdowns) [34] for simple basic block counting of integer benchmarks.

Dynamic instrumentation is similar to static instrumentation in that it is able to provide highly accurate profile information, it has a high overhead, and is not transparent. It also has most of the same limitations as static instrumentation except that

since each instruction can be viewed and instrumented dynamically, instrumentation can extend to library code.

2.4.3 Most Recently Executed Tail (MRET)

Interpretation-based dynamic optimization systems such as Dynamo [4] and Mojo [14] have developed a speculative version of path profiling known as Most Recently Executed Tail (MRET) [18]. The goal of this techniques is to attempt to locate hot traces of code with minimal profiling.

MRET begins by marking special basic blocks as potential trace heads. In Dynamo [4], the targets of backwards branches as well as the targets of trace exits are marked as trace heads. Once a basic block is marked as a potential trace head, a count is associated with each execution. Once a trace head's execution count reaches a predetermined threshold, the next dynamic execution path from that trace head is recognized as a hot path, optimized, and placed in the code cache. MRET is built upon the premise that if a trace of execution is hot, choosing the next executed path will statistically be the correct hot path. This method has two main drawbacks. First, the next dynamic execution path may not be the hot trace, causing this method to be overly speculative. Also, MRET is not able to distinguish between traces once they are determined to be hot. A hot trace that only executes ten more times should be treated and marked differently from a trace that continues to execute millions of times.

It should be noted that the idea of MRET is not limited to software-based profiling techniques. However, since MRET is typically been used in interpretation-based systems, it is placed in the software category in Table 2.1. Compared to instrumentation-based path profilers, MRET results in a lower overhead because the only the trace heads must be monitored. In addition, the accuracy is decreased due to its speculative nature and it is not a transparent profiling mechanism.

2.4.4 Hardware-based Profiling

A number of hardware-based techniques have been proposed in attempts to reduce profiling overhead. Conte [15] proposes the sampling of a profile buffer which is coupled with the branch prediction hardware in processors. The branch information in the profile buffer is used to fill out an edge profile. Merten [36, 37] discusses using a branch behavior buffer in his hot spot detection scheme which is essentially a hardware table for storing branch addresses and counts for the branch directions. The information in the branch behavior buffer, which is effectively edge profile information, is used to locate hot spots of execution during run-time. More recently, Vaswani [45] proposed hardware support for collecting path profile information by using a path stack and a hardware table for storing paths and their execution counts.

The proposed hardware techniques mentioned above would all substantially decrease overhead as compared to the software-based techniques. In fact, the overhead can be completely removed if the profiling hardware is ensured to be off the critical path. Because all the profiling is performed in hardware, the program execution behaves as it normally would without profiling resulting in transparency. Hardware techniques are usually very limited in their applicability. For example, Conte's and Merten's, approaches are limited to collecting edge profiles. Hardware can only perform what it is designed for and nothing more. In addition, hardware cost and complexity becomes an issue.

2.4.5 Summary of Hardware and Software Profiling Techniques

Software and hardware techniques attack profiling from two opposite ends of the spectrum. Software profilers typically are high overhead, non-transparent, but very

flexible allowing for the collection of multiple types of profiles. On the other hand, hardware-based profiling techniques are usually low overhead and transparent but limited in functionality. They are especially limited in analysis of the collected information because complicated algorithms are expensive and not feasibly implemented in hardware. Hardware techniques also require hardware costs which impacts the design cycle of a processor.

Because of their high overhead costs, software-based profiling systems are not viable for run-time optimization systems. This leaves hardware profiling as the other option. There are two problems with this. One is the limited flexibility and analysis possible with hardware. The other is that none of the hardware proposed in Section 2.4.4 exist in hardware. However, a different type of hardware unit (PMU) is making its way onto modern processors. These are being utilized in hybrid profiling techniques which effectively bridge the gap between hardware and software approaches. The emergence of PMUs and hybrid profiling systems are discussed in Section 2.5.

2.5 Hardware Performance Monitoring

In practice, chip designers have been hesitant to include hardware for performance monitoring on processors. A large reason for this that hardware real estate, as well as manpower for design and verification, would be needed for a unit of hardware that does not directly translate into processor performance. Time-to-market pressures in industry dictate that manpower be focused on designing and validating elements vital to functionality of the processor – performance monitoring hardware often does not fit into this picture.

Fortunately, modern microprocessors are beginning to be designed with support for HPM capabilities that can be used to provide feedback on architectural and system-level events. Software developers have found HPM support useful for performance tun-

ing of an application. For example, HPM can easily be used to indicate where an application is taking performance penalties such as high branch misprediction or cache miss rates. This information can be used to influence the software design if performance is critical. Developers of processor simulators also often use the information from HPM to validate their simulators. While HPM has mostly been motivated for performance tuning and validation, they are beginning to be used to drive hybrid profiling systems as well as run-time optimizers. This section describes capabilities and limitations of HPM as well as the hybrid profiling systems which leverage this technology.

2.5.1 Performance Monitoring Units

To provide feedback information on how an application and the operating system are performing, modern systems provide support for HPM through on-chip PMUs. Processors such as the Pentium-4 [43], Itanium [28], and Power PC 970 [27] all contain PMUs that allow for the configuration and collection of various hardware performance monitoring registers. However, because a standard for performance monitoring does not exist yet, there is a lot of variation between the features that are supported on PMUs across different processors.

PMU features can be roughly divided into two main categories; coarse-grained and fine-grained features. Coarse-grained features reflect high level architectural or system-level behavior and are fairly common across PMUs. All existing PMUs support a number of event counters which can be configured to count coarse-grained events such as branch mispredicts, cache misses, TLB misses, page faults or pipeline flushes. Some PMUs support more events than others. For example, the Pentium-4 supports 18 counters and 45 events while the Itanium-2 supports only 4 counters but 497 events. The fine-grained performance monitoring features vary greatly across PMUs. They collect more detailed information such as branch execution or cache miss behavior.

Feature	Description
Event Counters	A set of four counters that can be figured to count any of 497 events including cpu cycles, branch mispredicts, L1 instruction cache misses, etc.
BTB	The Branch Trace Buffer registers are a set of eight registers that act as a circular buffer. Each branch instruction inserts its instruction address and target address which effectively creating a circular buffer of the last four branches executed. *BTB Specific Filters:* Taken/Not Taken, Target Address Predicted Correct/Incorrect, Branch Predicted Correctly/Incorrectly, Branch Type (non-return indirect, returns, IP-relative branches, or all)
I-EAR	Instruction Event Address Registers, can be configured to sample instruction cache misses or I-TLB misses. Holds instruction cache line address and latency for a single sample.
D-DEAR	Data Event Address Registers may be configured to sample a data cache load miss, FP load, L1 D-TLB miss, or ALAT misses. Holds instruction address, data address, and latency for a single miss.
Filters	**Description**
Opcode Matching	Can be used to match specific opcodes or types of bundle instructions (ex. M, I, or B)
I-RR	Supports up to four separate instruction address range restrictions.
D-RR	Up to four data address range restrictions may be configured to filter for data specific counts or samples.

Table 2.2: Itanium-2 PMU features and the filters available for each of the features.

Table 2.2 shows an example of the capabilities provided with the Itanium-2 PMU [28] which is used for the experiments in this thesis. It shows each of the features of the PMU as well as the filters available for use with each of the features. First, it contains four simple event counter registers which can be configured to count any of the 497 events. It also supports a number of more fine-grained features such as the Branch Trace Buffer (BTB) which is a circular buffer allowing for the sampling of the last four branches executed. This set of branches is defined as a **branch vector** and can be used to effectively represent a trace of program execution. The BTB allows for a number of BTB specific filters which are shown in the table. In addition to the BTB,

it also supports instruction and data EARs for sampling information on cache, TLB or Advanced Load Address Table (ALAT) misses. Filters may be used with any of these PMU features. These filters include opcode matching and address range restrictions.

Because early PMUs were fairly crude with a few event counters and typically used for processor validation purposes, the specifications were usually not released for public use. More recently, PMUs have been designed with more convenient interfaces for outside use. Tools such as the perfmon interface [20, 24] (which is used in this thesis), the CHUD Tool package [3], and the PAPI interface [40] have been developed as tools and interfaces for easier access, configuration, and usage of PMUs.

2.5.2 Implications of Hardware Sampling

There are several barriers to address in using sampled hardware monitoring information for program analysis. First, there is a strong relation between sampling period and the overhead incurred by sampling period. Each sample requires the transfer of processor control to a system call which reads from the PMU registers. Aggressive sampling rates can interfere with execution time. Deficient sampling rates may miss important program execution related to the second issue of **sample aliasing**. This occurs when important execution or program phases are missed due to the sampling period. For example, if an important phase of execution happens to always begin and end between sampling periods, it will always be missed. A solution to this problem is to use randomized sampling where a random sampling period within a specified range is determined for each sampling period. Third, these systems are limited by the amount of information the PMU can provide. For example, information from branch vectors are limited number of branches that are collected per sample (e.g. 4 for the Itanium processor). If a PMU does not support branch vectors, it would be limited to simple program counter (PC) sampling which only corresponds to a single basic block.

2.5.3 Hybrid Profiling

With PMUs resident on modern processors, a new hybrid profiling systems have been developed that combine ideas from both hardware and software techniques. Hybrid profilers lower overhead by gathering much of their information from PMUs. Some information, such as library calls and system-level events, may not be available at the PMU level. Software instrumentation is sometimes used to gather information in these cases. Hybrid profilers are not completely transparent but can be close. The PMU is used to gather most of the information. However, occasionally, there is an interrupt which is used to gather the PMU information. Any software instrumentation, which is usually minimal in hybrid systems, also perturbs the original application execution. The accuracy of these systems may suffer a bit because they rely on sampled information. Instead of collecting an entire profile, pieces of the whole pictures are sampled and software analysis is used to piece them together. Software can perform analysis which is expensive in hardware and can also be used to transform one type or profile to another. By using both hardware and software, a broad profiling applicability can be attained.

2.5.3.1 Hybrid Profiling Systems

Digital Continuous Profiling Infrastructure (DCPI) [2] is a system-wide hybrid profiler which utilizes the PMU on Digital Alpha processors to monitor running applications and shared libraries. DCPI periodically uses an interrupt to stop program execution, note the current PC and event counters, and then analyze these samples for profile information. It uses PMU event counters for branch mispredict and cache miss information in an estimated cycle accounting algorithm. OProfile [39] is a similar system wide profiler for Linux inspired by DCPI.

ProfileMe [16] is another hybrid profiler which utilizes a different form of HPM.

Instead of sampling events, ProfileMe samples random individual instructions through the processor pipeline. Event are captured from each of these instructions through the pipeline and software algorithms extract useful profile information.

IBM's vertical profiling [23, 44, 47] infrastructure is yet another example of a hybrid profiler. Vertical profiling collects information across all system layers ranging from hardware to a running Java VM. A PMU is used for architectural profile information while instrumentation is used in libraries, the kernel, and the Java VM.

2.5.3.2 Hybrid Profilers and Run-time Optimization

Hybrid profilers utilizing PMUs are beginning to be deployed in actual run-time optimization systems. For example, the Continuous Program Optimization framework [31] is a continuous optimization system which uses PMU samples to drive recompilation during run-time. When the new binary is finished compiling, it is hot-swapped with the old binary.

The ADORE dynamic optimizer [33] is the first dynamic optimization system which exploits a PMU for profiling and monitoring. Previous dynamic optimizers have used interpretation or instrumentation-based techniques to run and monitor applications which incur a higher overhead than hardware sampling. ADORE uses the Itanium BTB registers for hot trace selection [13] and the Data EARs to capture a data cache miss profile in order to insert prefetches [32] for improving memory performance. The work in this thesis builds off of ideas in the ADORE optimizer performing further characterization of sampled BTB data; specifically pertaining to extrapolating PMU path profiling and PMU code coverage information with the sampled BTB registers.

Chapter 3

PMU-based Profiling

This chapter presents the hybrid PMU-based profiling system for extrapolating run-time code behavior. It begins with an overview of the entire framework and then describes the details of each part of the framework. After that, various compiler techniques are discussed which may be used to extend the PMU information.

3.1 Overview of PMU-based Profiling System

Figure 3.1 shows a high level overview of the entire compiler-aided, PMU-based profiling system. The system consists of two main phases; an online PMU collection phase, and an offline analysis phase. The online collection phase configures the PMU and periodically samples and stores branch vectors. An offline compiler-aided phase then performs analysis on the sampled branch vectors. Note that there are two consequences of performing the offline software analysis. First, the usage of software for analysis allows for deeper analysis. There is a lot of information available in a compiler infrastructure that is not readily available during run-time. Second, it helps to decrease overhead. The only part of profiling which is required during run-time is collection of run-time information. The analysis portion of the profiling scheme can theoretically be decoupled from the collection of the profile information and be performed offline, during idle processes, or even on a completely separate processor.

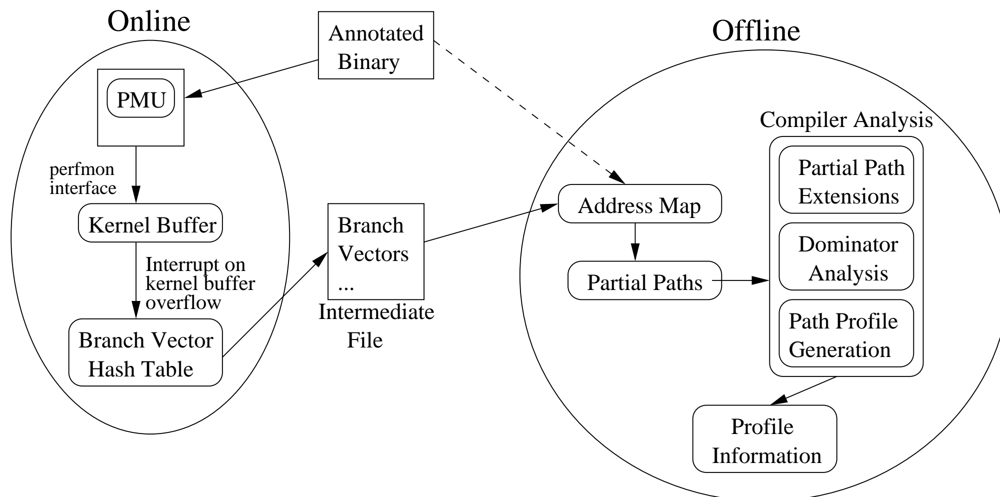


Figure 3.1: Overview of compiler-aided, PMU-based profiling framework.

3.2 Program Annotation

The BTB registers on the Itanium-2 PMU provide branch vectors which are simply a set of branch instruction addresses and branch target addresses. A problem arises in that there must be a method of mapping the addresses in the collected branch vectors back to instruction in the offline phase. In this case, the addresses must be able to be mapped back to the compiler low-level intermediate representation (IR) of the code. To manage this, programs must be compiled with an extra annotation phase. The annotation phase should be performed last during code generation to attach a specific label to each assembly instruction which is generated. The label provides information that can be used to map the instruction back to its representation in the compiler infrastructure. By annotating each instruction in a binary, an address map can be generated which is a direct map from program address to compiler IR instruction.

3.3 Collection of Branch Vectors

The online phase of PMU-based profiling consists of configuring, sampling, and storing branch vectors. While previous profiling systems like DCPI [2] use PC-sampling techniques to infer run-time code information, more robust PMUs support the collection of *branch vectors*. A branch vector is a set of correlated branch instruction addresses and branch targets that can be used to represent the a trace of program execution. A single PC can only provide information for a single instruction and at most can guarantee the execution of a basic block. By using branch vectors, the execution of multiple basic blocks can be inferred as well the specific path of execution along the basic blocks.

The experiments in this thesis utilize the Itanium-2 PMU which implements branch vector sampling through its BTB registers (described in Section 2.5.1). The BTB is configured to sample only taken branches since fall through branches can easily be followed with knowledge of the original CFG in a compiler infrastructure. By sampling branch vectors of four taken branches it is possible to gather longer paths per branch vector by inferring fall through branches which corresponds to the collection of more run-time information per sample.

A kernel buffer is set up which is able to store N total samples from the BTB. At each sampling period, the BTB registers are sampled and placed into a kernel buffer. When the kernel buffer fills up after N sampling periods, an interrupt occurs. On this interrupt, the N branch vectors in the kernel buffer are copied into user space data structures and stored in a specialized hash table. The hash function of the hash table is a combination of shifts and XOR operations on the branch vector addresses. The branch vector hash table collects and maintains counts for duplicate branch vectors. Upon the completion of program execution, the hash table contents of branch vectors and their respective occurrence counts are written to a file that is to be used in the offline analysis phase. In a system wide PMU-based profiling system, the hash table

contents could be written to a database of performance monitoring data. It should be noted that although this section describes the configuration and usage of the Itanium-2 PMU, the methods can be applied to any PMU which supports the sampling of similar branch vectors.

3.4 Partial Path Creation

The first step in the offline analysis phase is to associate each of the branch vectors back to the original CFG within the low-level IR of the program within a compiler framework. A branch vector mapped to low level IR instructions in the compiler framework is defined as a *partial path*. This mapping is enabled by used specially annotated binaries as discussed in Section 3.2. Partial paths contain the same basic blocks indicated by their corresponding branch vectors with one significant difference: the fall through branches are inferred and added to the partial path.

A low-level compiler IR instruction is similar to an assembly instruction as it is the lowest-level of the ISA-independent IR. This low-level IR is appropriate for PBO optimizations such as inlining [25] or superblock formation [26] and can also be raised to higher levels of IR or even source code if necessary. However, such work is not the focus of this thesis.

3.5 Compiler Analysis

Once all the branch vectors are mapped back to partial paths, the PMU information is in a form that the compiler can easily analyze and manipulate. At this point, analysis and tools within the compiler infrastructure can be leveraged to extend the amount of information provided per hardware sample. The addition of compiler-aid is crucial because much of the information and analysis available at the compiler level is either not available at run-time or would incur a high overhead and would not be

feasible to perform at that time. There are three main compiler extensions utilized; 1) partial path extensions, 2) dominator analysis and 3) path profile generation.

3.5.1 Partial Path Extensions

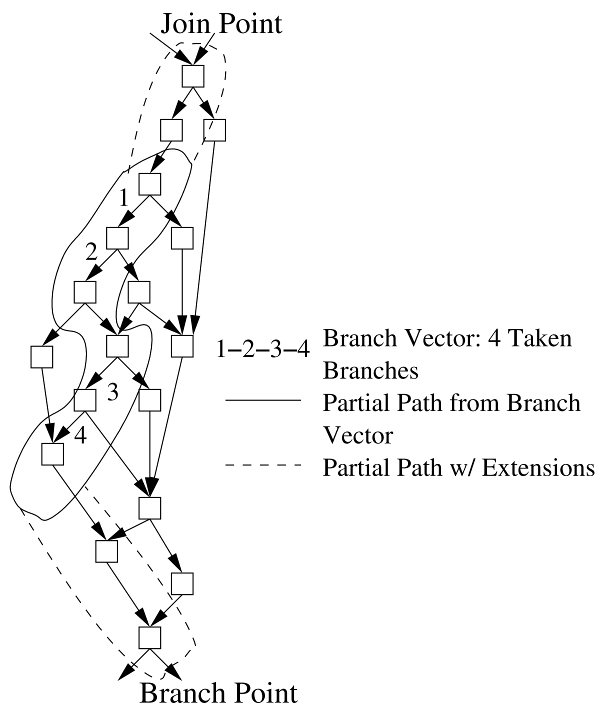


Figure 3.2: Example showing a branch vector, the corresponding partial path, and the extended partial path.

Compiler information can be used to extend the length of partial paths under certain conditions. For example, if the head of a partial path only has one source flow coming into it, then the partial path can be extended up the source flow and include the corresponding basic block. This process can be repeated until a point of uncertainty is reached; in this case, a join point. The tail of the partial path may also be extended if there is only a single destination flow. The extensions may continue until a branch point is reached.

Figure 3.2 shows an example of partial path extensions. Suppose the branches numbered 1 through 4 show the four taken branches in a particular sampled branch

vector. The blocks circled by the solid line indicate the partial path corresponding to the branch vector. Here, a partial path of six basic blocks can be formed by using 4 taken branches and inferring fall through targets. The dotted lines show the basic blocks that can be added to by extending partial paths until a point of uncertainty. In this example, a partial path extensions can be used to increase the partial path length from six to ten basic blocks.

3.5.2 Dominator Analysis

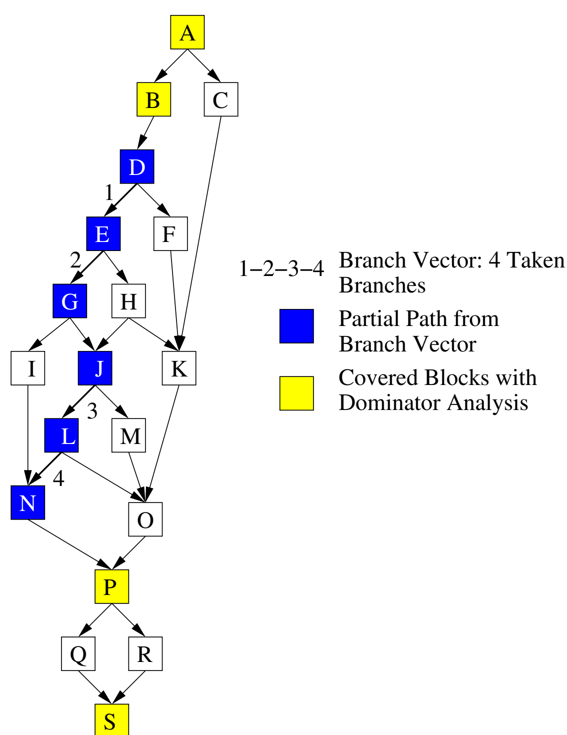


Figure 3.3: Partial path creation and path extension based on dominator analysis.

A number of compiler optimizations and transformations rely on dominator analysis [1] to determine guaranteed execution relationships of blocks in a control flow graph. There are two commonly analyzed dominator relationships: dominance and post dominance. Terms commonly used in dominator analysis are defined below:

Dominance: Basic block u dominates basic block v if every path

from the *entry* of the CFG to basic block v contains basic block u.

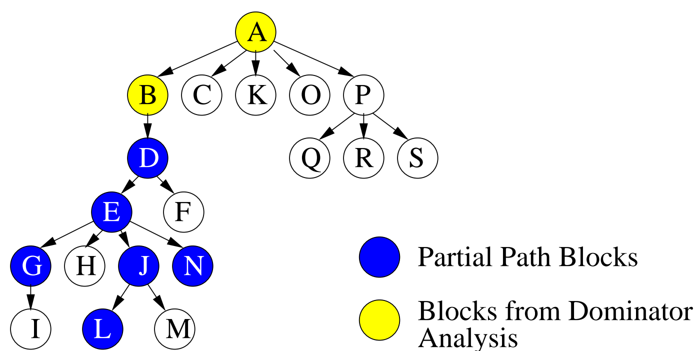
Immediate Dominator: Basic block u is the immediate dominator of basic block v if u dominates v and is the closer in the CFG to v than any other basic block which dominates v.

Post Dominance: Basic block u post-dominates basic block v if every path from v to the *exit* of the CFG contains basic block u.

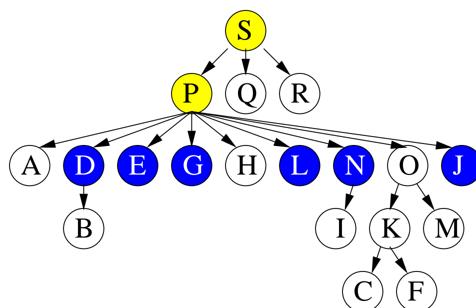
Immediate Post Dominator: Basic block u is the immediate post dominator of basic block v if u post dominates v and is closer in the CFG to v than any other basic block which post dominates v.

Tree representations of the information summarize the dominance and post-dominance relationships. A **dominator tree** is a directed graph created by starting at the *entry* block of a graph. Each block is connected to its immediate dominator to form a tree where the basic block for every node in the tree dominates all of its child nodes. Likewise, a **post-dominator tree** is simply a tree starting at the *exit* block of a graph where each basic block node is connected to its immediate post dominator. Each node in the post-dominator tree post dominates all of its child nodes. Examples of dominator and post-dominator trees for the control flow graph of Figure 3.3 are shown in Figure 3.4(a) and Figure 3.4(b).

The hierarchy of the trees indicate transitive relationship of dominance. For example, node A is at the top of the Figure 3.4(a) as it dominates every node in the graph. Specifically this means that prior to executing any other block in the graph, node A is guaranteed to execute. Further, from Figure 3.4(a) it can be inferred that when any block in the list $[D, E, F, G, H, I, J, L, M, N]$ execute, both A and B must have also been executed. The post-dominator tree (Figure 3.4(b)) indicates the relationship between executing a block in the CFG and executing another block before reaching the exit block of the graph. For example, execution of block C will guarantee the execution of all nodes above it in the tree, namely that corresponding blocks S, P, O, K execute before exiting the graph.



(a) Dominator Tree



(b) Post Dominator Tree

Figure 3.4: (a) Dominator and (b) post dominator trees for control graph in Figure 3.3. Partial path blocks and blocks added from dominator analysis are shown.

By leveraging compiler support for dominator analysis, PMU-based branch vector information can be extended. The partial path (blocks D, E, G, J, L, N) created from branch vector information in Figure 3.3 illustrates an example of this. By assessing the dominator tree in Figure 3.4(a) with the partial path, it is asserted that blocks A and B dominate the partial path and therefore are guaranteed to execute. Using the post dominator tree shown in Figure 3.4(b), it is likewise known that blocks P and S both post-dominate the partial path and are guaranteed to execute. It is clear that by sampling branch vectors and performing dominator analysis, more information can be extracted out of each sample. Overall, in this example, dominator analysis helps guarantee the execution of four additional program blocks for PMU-generated information, collectively indicating the execution of ten basic blocks.

Partial path extensions and dominator analysis are related in they they both provide additional basic blocks which are guaranteed to execute. Basic blocks added by partial path extensions are a subset of the blocks which are known to execute through dominator analysis. The difference is that partial path extensions show blocks that are guaranteed to execute and are constrained to be connected to the original partial path. Therefore, partial path extensions and dominator analysis should be used in different contexts. If specific path information is of interest, partial path extensions should be utilized. If any kind of code coverage analysis the subject of interest, then dominator analysis should be used to maximize the information added by the compiler.

3.5.3 Path Profile Generation

The last compiler algorithm used is this thesis is path profile generation. A path profile consists of a set of program paths and their associated counts or weights. The goal here is to use the compiler infrastructure to analyze partial paths and map them back to pre-determined paths to create a path profile. The original Ball-Larus path profiling scheme [5] used pre-determined paths which started at function and loop heads and ended at loopback edges and function returns.

Due to the nature of partial paths, there are a number of issues to deal with when trying to match partial paths back to pre-determined paths in the path profile. One issue is that partial paths have the ability to span loopback edges as well as function boundaries. They represent exact traces of program execution while typically paths in a path profile represent paths through the CFG of a given function or region. The only way to map partial paths back to the paths in the compiler view of the CFG is to split partial paths along function boundaries and loopback edges.

Another issue is that of **path ambiguity**. After the partial paths are split, they are still not guaranteed to match up with the pre-determined paths. Branch vectors

are randomly sampled meaning that they can start and end anywhere in the CFG and therefore are not constrained to the beginnings and endings of the pre-determined paths. To overcome this, an algorithm is necessary for matching and crediting the actual paths in the path profile by using partial paths.

The last issue, which is generic to all path profiling techniques, is that the number of paths in a given CFG grows exponentially. There may exist resource complications when dealing with large CFGs. The OpenIMPACT Research Compiler [38], as well as most other compiler infrastructures, handle CFGs at the function granularity. Problems can arise in trying to enumerate all paths in large functions. In fact, there are cases where the number of paths in a function exceeds the limit of using a 64-bit path ID. This problem is managed by limiting CFGs to regions which are guaranteed avoid resource complications.

Path profile generation consists of the following steps. First, functions are split into manageable regions. Partial paths are split along these region boundaries. Afterwards, a path matching and path crediting algorithm is used for filling out the weights of paths in the path profile by using partial paths. These steps are discussed in more detail in the following subsections.

3.5.3.1 Region Formation

The first step in path profile generation is to create regions to guarantee CFGs which are manageable in size. Before describing the region formation algorithm used for these experiments, a few relevant terms must first be defined. The terminology used in these definitions is consistent with Ball's study [7].

> *Let **Region** $R(V,E)$ be a sub-graph of the directed graph $G(V,E)$ with a unique entry vertex R_{ENTRY}, a set of body vertices R_{BODY}, and a set of exit vertices R_{EXIT} all of which are reachable from R_{ENTRY}. An*

edge $e = v \rightarrow w$ connects source vertex v (denoted by $src(e)$) to target vertex w (denoted by $tgt(e)$).

*A **Partial Path** in R is represented as a sequence of edges $E(p) = [e_1, e_2, ..., e_n]$, where $src(e_i) \in R_{BODY}$.*

*A **Region-based Path** in R is a partial path with the added constraints $src(e_1) = R_{ENTRY}$ and $tgt(e_n) \in R_{EXIT}$. The set of paths from R_{ENTRY} to R_{EXIT} in which edge e appears is denoted by $P(e)$.*

*A **Threshold** T exists such that any region R may not contain more than T number of region-based paths.*

Thus, according to the definitions above, the following rules govern the formation of a region R:

(1) R can only be entered via its single entry vertex

(2) R_{BODY} is constrained by loop boundaries. It may not contain basic blocks outside of a loop body as well as basic blocks which are inside a loop body

(3) The total number of paths from R_{ENTRY} to all R_{EXIT} vertices must be less than a threshold T

(4) R_{BODY} should contain as many basic blocks as possible.

To form regions within these limitations, a greedy region formation algorithm is implemented. The algorithm begins with the single region R containing just one vertex: the *entry* of the CFG. In a breadth-first manner R is expanded along the target edges of its exit vertices. When the algorithm can no longer expand R, it marks all vertices R_{EXIT} as the entry vertices for new regions. The algorithm is repeated for these vertices and so on until all the basic blocks within the CFG have been added to regions or the size of the CFG reaches a pre-determined threshold and contains an unmanageable number of paths.

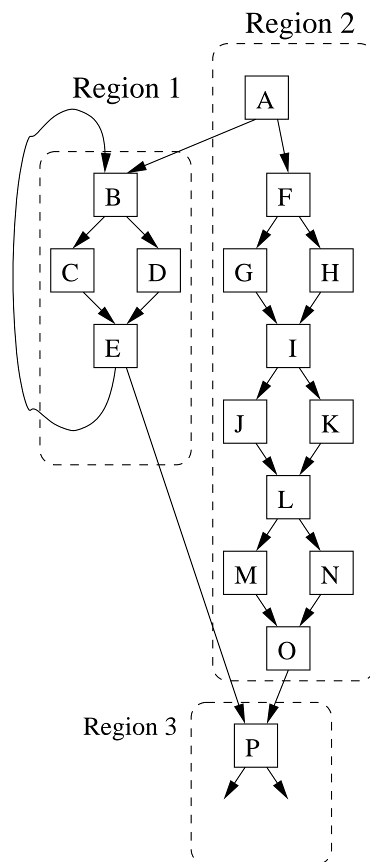


Figure 3.5: Example of region formation.

Figure 3.5 provides a simple example of region formation. The algorithm begins with basic block A. Basic block B cannot be added into the region because it belongs to a loop. But the algorithm is able to include basic blocks F, G, H, I, J, K, L, M, N, and O. Basic block P cannot be included because it breaks the first rule that a region may only have one entry. This region is shown as Region 2 in the figure. The loop including basic blocks B, C, D and E forms another region shown as Region 1 and naturally, basic block P begins a new region. When a CFG has been converted to its composite regions, the full paths in each region are enumerated and stored in data structures which are later used for the path matching and crediting (discussed in Section 3.5.3.3).

3.5.3.2 Region-based Paths

The reason for creating regions is to limit the number of paths in a given CFG to a manageable number of paths. In order to accomplish this, all paths must be confined to region-based paths. There are a couple of implications of constraining paths to region boundaries that must be discussed.

First, using region-based paths means that paths are limited at function boundaries and loop boundaries. This means that original partial path correlations between region boundaries are lost once they are split. If regions are not created well, than it is possible that the hot region-based paths may be as effective as full Ball-Larus [5] paths when used to drive code layout PBOs. However, because a high percentage of program execution exists within loops, it makes sense to build regions around loop boundaries.

Splitting paths on region boundaries creates slightly different paths than Ball-Larus paths used in most previous path profiling studies. For example, in the example CFG shown in Figure 3.5, the path $ABCEP$ is a Ball-Larus path but is not a path found with region-based paths as it spans multiple regions. On the other hand, path $AFGIJLMO$ is a region-based path but would not be considered a path using the Ball-Larus algorithm. Again, because programs spend most their execution in loops, it may be acceptable to limit paths to region-based paths. The effects of different algorithms for discovering paths should be explored and is left for future work.

3.5.3.3 Region-based Path Profile Generation

Once regions are formed and region-based paths are discovered and enumerated, the next task is to use partial path information to add counts to the paths to create the path profile. Each partial path can be extrapolated to a **matching set** of region-based

paths as described below:

> *Given a single partial path p consisting of edges $[e_1, e_2...e_n]$, the minimum* ***matching set*** *M of region-based paths which contain p is the intersection of the set of all region-based paths $P(e)$ for each edge of p.*
>
> $$M_p = \bigcap_{i=1}^{k} P(e_i)$$

The first step in the path profile generation algorithm is to generate the matching set for each of the partial paths. Depending on the characteristics of the partial path and the corresponding CFG, the matching set M may contain an indeterminate number of region-based paths. After generating the matching set for a partial path, the next step is to increment the count of each of the paths in the set of matching region-based paths. The distribution of the partial path counts across the matching set is done with the following assumptions:

(1) There is an equal probability of execution of each path within M

(2) Random sampling of actual program flow

(3) Adequate samples are collected to clearly distinguish frequently executed paths

Partial Path	**Count**	**Matches**	**Inc**	**Total**
IJLMO	100	*AFG***IJLMO**	+50	50
		*AFH***IJLMO**	+50	50
BCE	50	**BCE**	+50	50
AFGIJ	300	**AFGIJ***LMO*	+150	200
		AFGIJ*LNO*	+150	150
LMO	200	*AFGIJ***LMO**	+50	**250**
		*AFHIJ***LMO**	+50	100
		*AFGIK***LMO**	+50	50
		*AFHIK***LMO**	+50	50

Table 3.1: Example of path matching of partial paths to region-based paths.

Given these assumptions, the distribution of counts across the matching set becomes simple; the count is equally distributed across every matching path. This algorithm may falsely attach weights of a partial path to all of its matched paths but given enough sampled partial paths placed randomly across the CFG, the hot paths within a region can still be distinguished.

For example, consider partial paths and their occurrence counts from the CFG in Figure 3.5 and Table 3.1: $IJLMO(100)$, $BCE(50)$, $AFGIJ(300)$, and $LMO(200)$. $IJLMO$ matches both **AFG**$IJLMO$ and **AFH**$IJLMO$ and assigns a weight of 50 to both. The intra-loop partial-path BCE yields a unique match and needs not distribute its weight. The third path, $AFGIJ$, matches both $AFGIJ$**LMO** and $AFGIJ$**LNO** and distributes a weight of 150 to each. Finally, LMO matches the four paths **AFGIJ**LMO, **AFHIJ**LMO, **AFGIK**LMO, and **AFHIK**LMO and assigns each a weight of 50. The hot path $AFGIJLMO$ becomes prominent due to continued random sampling of paths within the region.

Once all partial paths have been matched and weighted as described above, the sampled path profile is sorted by path weight and the paths whose total program flow exceed an arbitrarily-defined hot threshold are extracted as hot paths. These hot paths are ready for immediate use in path-profile based compiler optimization.

3.6 Profile Information

After the compiler analysis of the partial paths, profile information can be generated. But, what profile information should be generated? The type of profile which is necessary is dependent on the type of optimization which is to be performed. For example, path profile information is great for superblock formation [26] but terrible information for function inlining [25]. One of the advantages to using a hybrid profiling technique is that the hardware samples can be used in different ways to produce different profiles by simply varying the software analysis techniques employed. This

thesis presents two case studies illustrating this. Compiler analysis is performed on the sampled branch vectors to produce a PMU-based path profile and perform PMU-based code coverage analysis. The implementation and results for these studies are presents in Chapter 4 and Chapter 5.

3.7 Experimental Methodology

The experiments in this paper are performed using a set of the *SPEC CPU 2000* benchmarks compiled with the OpenIMPACT [38] Research Compiler on an Itanium-2 with Redhat Advanced Workstation 3.0 and the 2.6.10 kernel. The applications are compiled with the base OpenIMPACT configuration which include many classical optimizations and but do not include more aggressive profile-directed optimizations. The PMU collection tool used during the run-time monitoring phase is developed using the perfmon interface and libpfm-3.1 library [20, 24] with the kernel buffer size set to hold 64 complete sets of BTB entries. The tool configures the PMU to sample only taken branches every sampling period. Fixed sampling periods are generally used. Randomized sampling periods are only used when mentioned in the experimental data. During each sampling period, the PMU tool collects branch vectors, and inserts them into a specialized branch vector hash table. The off-line compiler-aided analysis phase is accomplished by feeding the branch vectors back into an OpenIMPACT module which performs the code coverage analysis.

The actual path profiles and actual code coverage for the benchmarks used for comparison are generated using Pin Tools [34]. One Pin Tool generates the region-based paths and their counts, and another Pin Tool finds all instructions executed in a program as well as the number of times each was executed for comparison in the code coverage study.

Chapter 4

PMU-based Path Profiling

This chapter presents the first case study in this thesis which utilizes sampled branch vectors to generate path profile information. For trace layout optimizations such as trace scheduling [22] or superblock formation [26], a path profile is the most sought over type of profile. As discussed in Section 2.2.3, it can be used to provide more accurate path information than point-based profiles such as edge profiles. The rest of this chapter describes the method of producing an estimated path profile from sampled branch vectors and then presents experimental results for this path profiling approach.

4.1 Generating an Estimated Path Profile

An estimated path profile is generated by using the hybrid profiling framework discussed in Chapter 3 with the compiler analysis section tailored to creating a path profile. The branch vectors are first mapped to partial paths using the address map. Next, the compiler analysis attempts to lengthen each partial path as much as possible by performing partial path extension on each of the partial paths. After that, the path profile generation algorithm discussed in Section 3.5.3 is applied by creating regions, splitting partial paths on region boundaries and using the path matching and crediting algorithm. The following sections of this chapter discuss the experimental results of this path profiling approach.

4.2 Effect of Sampling Period

There exists a strong tradeoff between sampling period and overhead. On one hand, it is desirable to collect as much information as possible which calls for a small sampling period. However, if the sampling period is too low, it can cause significant run-time overhead. On the other hand, if the sampling period is increased by too much, there may be a considerable loss in information.

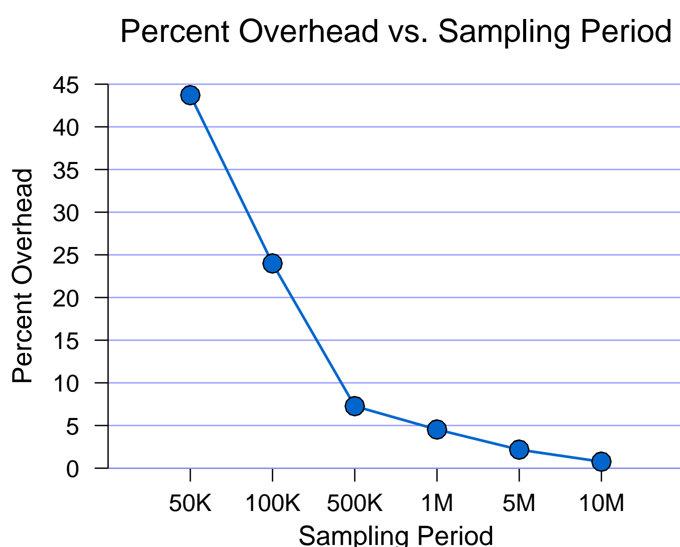


Figure 4.1: Overhead of run-time collection of branch vectors and number of unique branch vectors for various sampling periods.

Figure 4.1 shows the effect of sampling period on the overhead associated with the run-time collection of branch vectors. The sampling period is varied from 50K to 10M clock cycles. This graph shows that the percent overhead decreases as the sampling period is increased with a significant point in the curve occurring at a sampling period of around 500K to 1M clock cycles. At this point, the overhead increases dramatically with an overhead of around 40-50% at a sampling period of 50K. Increasing the sampling period above the point shows a leveling with an overhead of less than 1% at a sampling period of 10M or above.

There are three main causes which contribute to the overhead incurred by collecting and storing the PMU information. The first cause is that an interrupt occurs every sampling period which copies the BTB registers into a kernel buffer. As the sampling period increases, this causes the overhead to increase because interrupts are more frequent. The second cause is upon kernel buffer overflow, the sampled data must be copied between the kernel buffer and user space. The third major cause of overhead relates to the processing time of storing PMU samples from the user buffer into the specialized branch vector hash table. The hashing mechanism uses a hash formed by operating a sequence of 16-bit shifts and XOR operations on the 64-bit addresses which compose the branch vector.

It should be noted that the infrastructure implemented for the experiments are designed in a research environment to allow for flexibility in exploring various designs. Therefore, the overhead numbers shown here can be viewed as an upper bounds. If implemented in a production system, it is possible to remove much of the overhead by optimizing the collection system. For example, the extra user buffer could be removed as well as the hash table if the sampled data was simply dumped to a file or piped to a different process or processor for manipulation and storage. However, with the PMU collection system described here, the percent overhead is still relatively low. Even at the lower sampling periods(40-50% overhead), the overhead is acceptable if compared to interpretation or instrumentation systems.

4.3 Aggregating Data from Multiple Runs

By decoupling the PMU sample collection and the analysis phases, a unique opportunity exists of analyzing aggregate data from multiple runs of a program. This can be implemented by simply concatenating output files together before using the offline analysis tool. By aggregating multiple runs, it is possible to collect run-time information from one run that was missed in previous runs. Figure 4.2 shows the effects

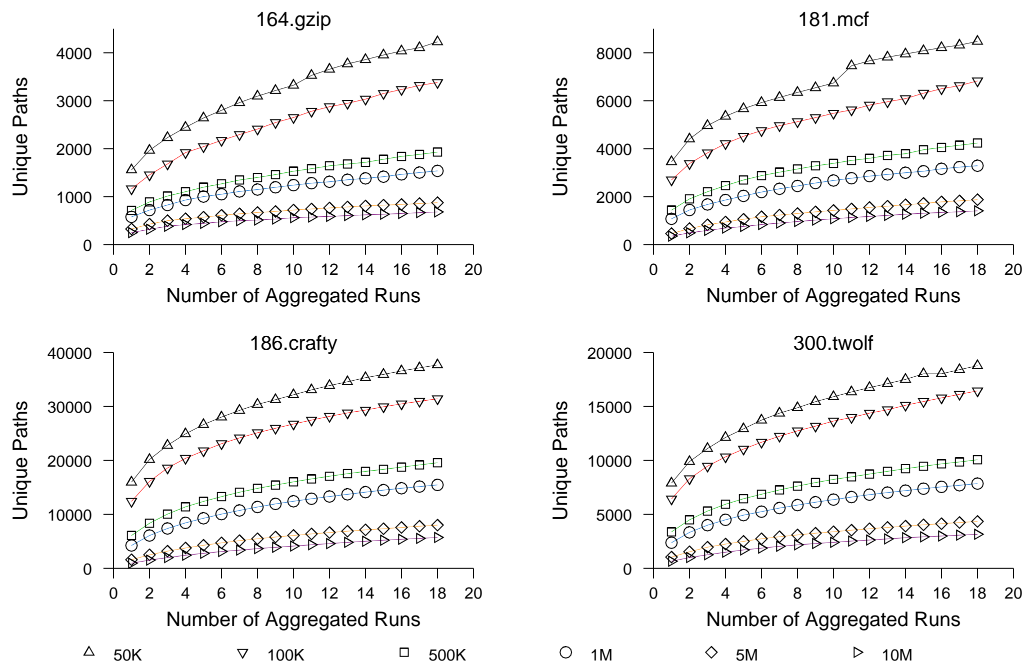


Figure 4.2: Number of unique paths found by aggregating data from runs with same input set.

of aggregating the PMU BTB samples from up to 20 different runs of a few benchmarks with the same input. The graph shows four benchmarks but all of the benchmarks experimented with exhibit similar characteristics. It can be seen that additional runs clearly does increase the number of unique paths or branch vectors discovered. The greatest increase occurs from combining up to 10 runs. There is a slight leveling off after 10 runs but still a steady increase in unique paths found. This shows that there is promise in using PMU samples from multiple runs to drive the offline analysis phase of PMU-based profiling.

4.4 Partial Path Characteristics

An important note is that because of the nature of PMU sampling, partial paths are capable of spanning across loopback edges, function boundaries and may even extend

between the user program and shared libraries. For the experiments in this thesis, addresses that are outside of the user program are ignored and thrown away. However, in a different context, this extra PMU information could be leveraged to profile across these function and library boundaries. Unless the partial path analysis phases are able to support partial paths that span these boundaries, the partial paths must be split. Due to limitations in the base compiler infrastructure, the partial paths are split along function boundaries as well as loopback edges.

Benchmark	**Initial**	**Ext**	**Func**	**Loop**
164.gzip	28.9	34.8	22.8	20.4
175.vpr	41.8	50.5	30.6	19.6
177.mesa	44.6	53.8	35.3	33.0
179.art	29.5	34.7	32.1	22.9
181.mcf	32.0	38.8	33.7	25.5
183.equake	65.8	75.1	66.8	54.5
188.ammp	31.3	39.5	36.4	28.5
197.parser	28.7	35.2	14.7	12.7
256.bzip2	38.8	45.8	33.4	22.7
300.twolf	37.8	46.5	32.5	25.4
Average	37.9	45.5	33.8	26.5

Table 4.1: Average length of partial paths (number of instructions) initially, after partial path extensions, and after splitting on function boundaries and loopback edges.

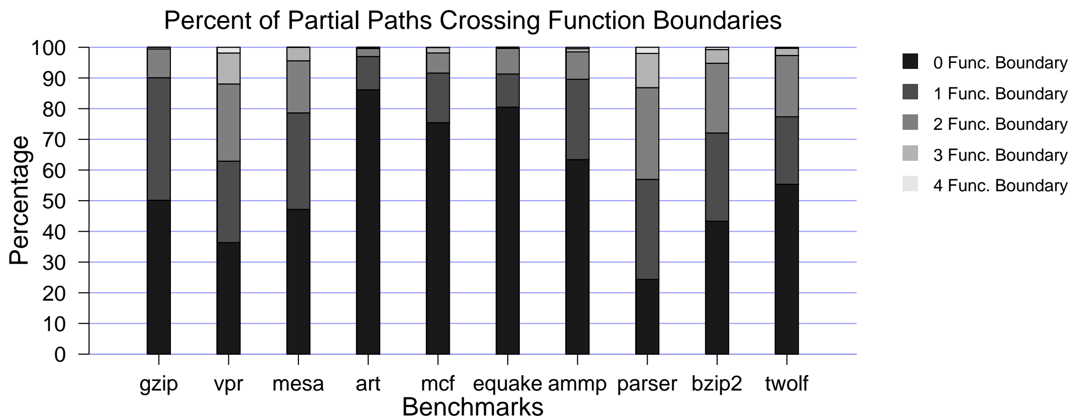


Figure 4.3: Percentage of partial paths that cross function boundaries.

Table 4.1 shows the average length of partial paths initially (Initial), after partial path extensions (Ext), after splitting on function boundaries (Func) and after splitting on loopback edges (Loop). Path lengths are measured in number of low level IR instructions. On average, the system locates paths about 38 instructions long. Partial path extensions can significantly improve partial paths increasing their average length by about 20%. However, after splitting along function boundaries and loopback edges, the average path length drops about 40% from the extended length to about 27 instructions.

When partial paths are split along function boundaries, not only are the length of partial paths substantially decreased on average, but correlation is lost between functions. This correlation could be used to drive inter-procedural optimizations such as function inlining [25]. Figure 4.3 shows the inter-procedural characteristics of the initial partial paths in Table 4.1. It shows the number of partial paths per benchmark that span between zero and four function boundaries. The zero column shows the number of partial paths which are intra-procedural. All the other partial paths span at least one function boundary. A surprisingly large number partial paths not only cross one function boundary, but span across multiple function boundaries. In particular, a large majority of partial paths in **197.parser** cross at least one function boundary. This indicates that by splitting on function boundaries, there is a great deal of information that is lost within partial paths. These function correlations could be stored to drive inter-procedural optimizations. This exploration is reserved for future work.

4.5 Actual Hot Paths

Table 4.2 shows the number of actual hot paths in each benchmark as determined be the path profiling Pin Tool. Hot paths are determined by identifying flows above a hot threshold regarding the percent of total execution flow that the hot paths account for. Flow is defined as the path's count divided by the sum of all counts of paths. The hot threshold is set to 0.125% as used in previous path profiling studies [7, 8]. The data

Benchmark	# Hot Paths	% Total Flow
164.gzip	30	98.96%
175.vpr	29	96.05%
177.mesa	8	97.99%
179.art	50	99.60%
181.mcf	52	97.84%
183.equake	24	98.53%
188.ammp	33	96.73%
197.parser	168	82.41%
256.bzip2	78	96.52%
300.twolf	80	92.55%

Table 4.2: Number of actual hot paths and their percent of total.

indicates that using a hot threshold of 0.125% indeed provides a relatively low number of hot paths corresponding to a very high percentage of total program flow. 197.parser is the exception with a larger number of hot paths contributing to a lower amount of total program flow. In the rest of the benchmarks 80 or fewer paths account for over 90% of the total flow within the run of the program.

4.6 Accuracy Results

The estimated PMU path profile is compared to the full path profile using a method similar to Wall's weight matching scheme [46] which defines accuracy as:

$$Accuracy\ of\ P_{estimated} = \frac{\sum_{p \in (H_{est.} \cap H_{actual})} F_{actual}(p)}{\sum_{p \in H_{act.}} F_{actual}(p)}$$

In this equation $F(p)$ is the flow of a path. H_{actual} is the set of paths in the full path profile which are above a set threshold. $H_{estimated}$ is then the created by selecting the hottest paths in the path profile equal to the number of paths in H_{actual}.

Figure 4.4 shows accuracy results with respect to various sampling periods ranging from 50K to 500M clock cycles. In general, at low sampling periods, this path profiling technique achieves fairly high accuracy. As sampling period increases, the accuracy

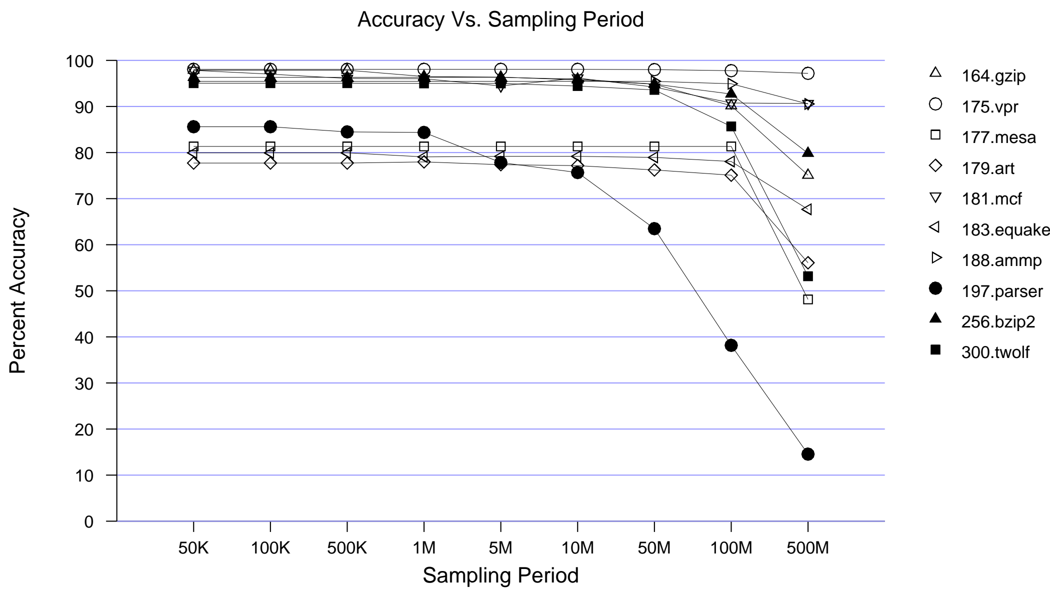


Figure 4.4: Accuracy versus sampling period.

remains relatively constant at around 88% until around a sampling period of 10M cycles. This indicates that each of these sampling periods contains enough samples to locate the important hot paths. However, once the sampling period is increased to a critical point (around 5-10M), accuracy suffers because an insufficient numbers are samples are collected and the hot path ratios can be estimated. However, if the sampling period is set to 10M, 88% accuracy can be obtained at approximately 1% run-time overhead.

4.7 Summary

Path profile information is critical to path-based PBOs. While traditional methods of gathering a path profile involve high overhead software techniques, the results of this path profiling case study indicate that PMU information looks promising in detecting hot paths for optimization. Partial path extensions are able to extend branch vector by around 20% in length and that path profile generation is able to detect hot paths well; if the sampling period is configured to be around 10M clock cycles, it is possible

to obtain around an 88% accuracy with respect to hot path detection with only a 1% run-time overhead.

Chapter 5

PMU-based Code Coverage

This chapter presents the second case study which applies compiler-aided analysis to branch vectors to perform code coverage analysis. While the first case study focuses on the determination of hot paths within a program, this code coverage study produces profile information at a coarser granularity. Specifically, it generates a list of the instructions that are known to execute based on branch vector information as well as an estimated count of how often the instruction was executed. This coarse-grained information can be useful for larger scale code placement optimizations such as code placement for improving instruction cache or instruction TLB performance.

5.1 Code Coverage Methodology

The PMU-based code coverage tool is very similar to the overview of PMU-based monitoring shown in Figure 3.1. The only significant note is that the compiler-aided analysis used is dominator analysis. Dominator analysis is performed for each basic block in each partial path to indicate additional basic blocks that are guaranteed to execute. Each time an instruction is encountered, either through a partial path or through dominator analysis, a count is incremented. In this manner, an code coverage statistics as well as an estimated probability distribution function can be generated.

5.2 Actual Code Coverage

Benchmark	# Ops	# Covered Ops
164.gzip	6,466	3,063 (47%)
175.vpr	23,573	12,229 (52%)
177.mesa	89,006	7,390 (8%)
179.art	2,201	1,515 (69%)
181.mcf	1,973	1,401 (71%)
183.equake	3,033	2,265 (75%)
188.ammp	19,562	5,835 (30%)
197.parser	17,541	11,271 (64%)
256.bzip2	5,095	3,138 (62%)
300.twolf	40,490	15,705 (39%)

Table 5.1: Number of instructions per benchmarks and actual code coverage.

Before exploring code coverage data, it is important to get an idea of the size of the benchmarks as well as their run-time instruction footprints. Table 5.1 shows the size of each benchmark in number of low-level IR instructions as well as the number of these instructions that are actually covered during run-time. Code size varies greatly in this set of benchmarks ranging from **181.mcf** with 1,973 instructions to **177.mesa** with 89,006 instructions. The number and percentage of covered instructions also ranges greatly from 8% coverage for **177.mesa** to 75% coverage for **183.equake**.

5.3 PMU Code Coverage of a Single Run

For the rest of the code coverage experiments, code coverage percentage is defined as the percentage of actual covered instructions (shown in Table 5.1) that is discovered with compiler analysis of branch vectors. Therefore, a code coverage percentage of 100% would mean that PMU-based code coverage has covered all the instructions that have actually been executed. Code coverage analysis is broken down into four main categories; Single BB, Single BB w/ Dominator Analysis, Branch Vectors, and Branch

Vectors w/ Dominator Analysis.

> **Single BB:** In this case, the first basic block of each branch vector is used for marking covered instructions. This is used to simulate the effect of gathering a PC each sampling as a PC can only be related back to a basic block.

> **Single BB with Dominator Analysis:** Again, only the first basic block indicated by a branch vector is used to simulate PC-sampling. Dominator analysis if performed on the single basic block to mark other basic blocks as covered.

> **Branch Vectors:** Here, branch vectors are use for code coverage analysis. The branch vectors are mapped to partial paths every instruction within the partial path is marked as a covered instruction.

> **Branch Vectors with Dominator Analysis:** Branch vectors are mapped to partial paths and then dominator analysis is performed on every path to extend the amount of coverage information per partial path.

Figure 5.1 shows the code coverage percentage of for multiple sampling periods (100K, 1M, 10M and 100M clock cycles) for the Single BB, Branch Vectors and Branch Vectors with Dominator Analysis categories. To get an idea of the effect of *Single BB with Dominator Analysis*, Figure 5.2 shows code coverage for all four coverage types at a sampling period of 100K. The data in Figure 5.1 shows that the *Single BB* leaves much to be desired in terms of code coverage. Even at a low sampling period of 100K cycles, the highest code coverage percentage is barely over 50% for **181.mcf**, **164.gzip**, **175.vpr** and **300.twolf** perform particularly poorly with only around 21-22% of the code coverage. The percentage only decreases as the sampling period increases for each benchmark.

By sampling branch vectors and mapping them back to partial paths, there are substantial increases in the percentage of covered code that can be discovered using PMU-based code coverage. At the lowest sampling period, the percentage increases

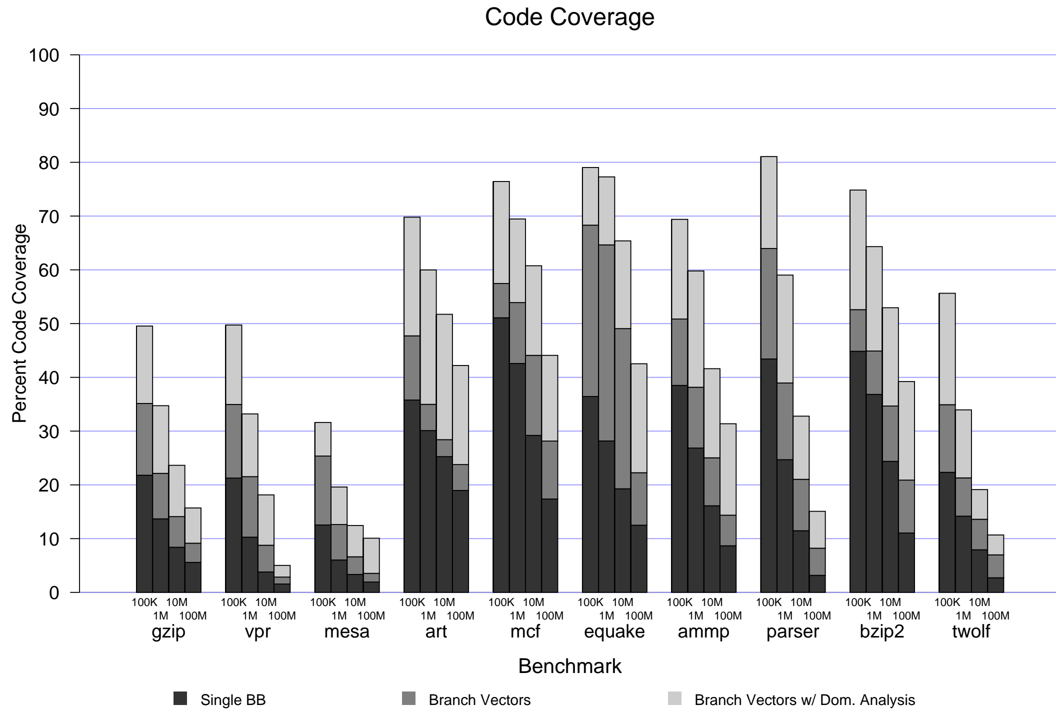


Figure 5.1: Code coverage across different sampling periods (100K, 1M, 10M 100M) showing the effects of 1) using a single basic block per sample, 2) using branch vectors to create partial paths and 3) extending partial path information by using dominator analysis.

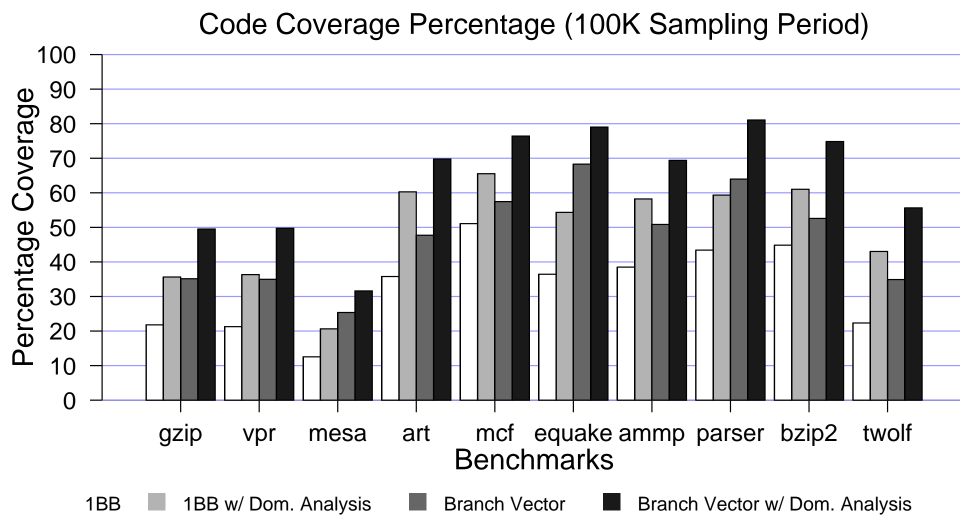


Figure 5.2: Code coverage for all types of coverage at a sampling period of 100K.

by an average of 14%. However, the usage of branch vectors has a range of effect of code coverage based on application. **183.equake** has the greatest improvement due to using branch vectors at around 30% improvement in coverage percentage. By observing Table 4.1, this can be expected as **183.equake** contains by far the largest number of instructions per partial path.

Combining the techniques of using branch vectors with compiler-aided dominator analysis, it is possible to considerably extend the amount of coverage information provided by each sample. By observing the data in Figure 5.1, it can be seen that the number of instructions or percentage of code coverage is typically more than doubled by using dominator analysis to extend hardware-collected branch information. Dominator analysis significantly extends the amount of code coverage information. As shown in Figure 5.1, for the lower sampling periods it consistently provides an additional 15-25% to the code coverage percentage. The exception is **177.mesa** which improves less that 10% for all sampling periods. Figure 5.2 indicates that dominator analysis also improves code coverage significantly if used with PC samples instead of branch vectors. On average the effect of dominator analysis on PC samples is about equivalent to the effect of sampling branch vectors. However, the combination of branch vectors and dominator analysis still provides the most code coverage information.

5.4 PMU-based Code Coverage Stability

A side effect to hardware sampling is that the branch vectors collected from a run of a program are non-deterministic. This means that different runs of the same program with the exact same input and sampling period will yield different results based on run-time system behavior. Figure 5.3 shows the variation in code coverage percentages over 20 runs of each benchmark. The bars show the average percentage while the thin lines show the maximum and minimum percentages. The variation in code coverage percentage is typically within 5-10% indicating that although each run is

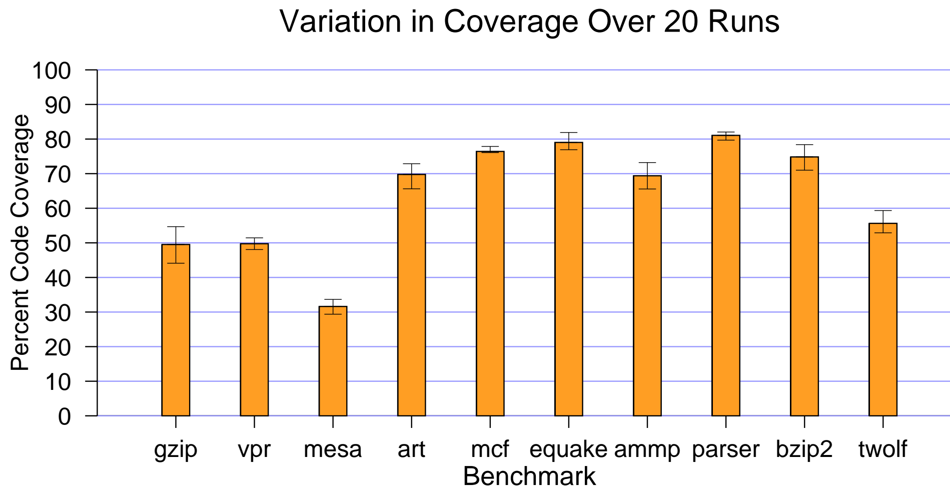


Figure 5.3: Stability of percent of code coverage matched over 20 runs of each benchmark. Thick bars show the average coverage while the thin lines show the minimum and maximum percentages over the 20 runs.

non-deterministic, code coverage across different runs is fairly stable in regards to the amount of code covered.

5.5 PMU Entropy Analysis

Figure 5.4 shows the probability distribution graphs for the code execution of application **164.gzip** determined by both complete coverage (a) and PMU-based coverage (b). Although sampling in PMU-base coverage may miss program behavior, the instruction execution distributions appear similar. However, more detailed analysis can assess the overall ability of PMU code coverage data to accurately characterize the actual coverage. The relative entropy (Kullback-Leibler divergence [30]) defines the distance between two probability distribution functions. Let a discrete distribution have probability function p_k, and let a second discrete distribution have probability function q_k. Then the relative entropy of p with respect to q is defined by:

$$d = \sum_{k=0}^{n} p_k log_2(\frac{p_k}{q_k})$$

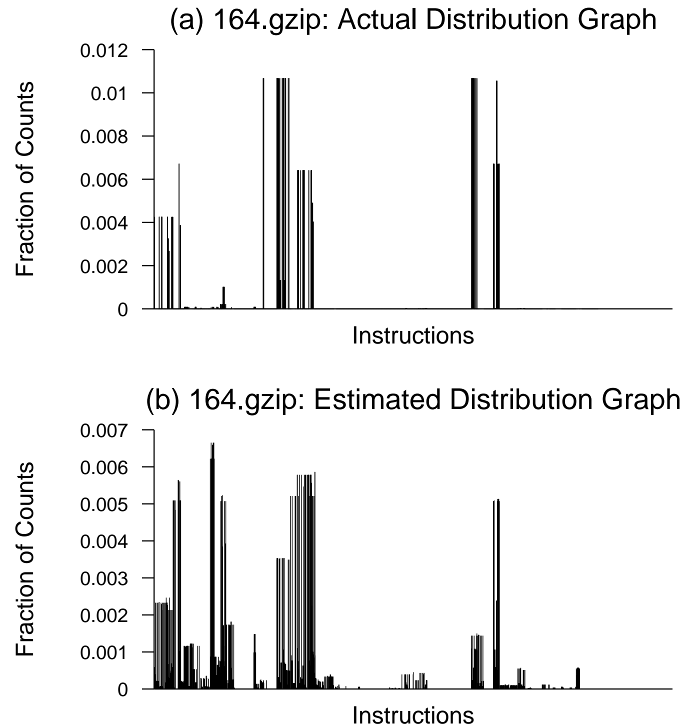


Figure 5.4: Instruction execution distribution for **164.gzip** (a) actual and (b) PMU.

Figure 5.5 presents the relative entropy numbers between actual and PMU code coverage. Three sampling rates are examined: 100K, 1M, and 10M. Results indicate that the average divergence between the actual and PMU distributions is about five. While the number is relative, it can be used to quantify the deviation of system parameters on gathering complete code coverage results. For instance, for applications **175.vpr**, **181.mcf**, and **256.bzip2** the divergence over the sampling rates increase by 3-4, indicating that sampling will have a direct role in the coverage accuracy. While the divergence of **164.gzip**, which has a smaller code execution footprint, is not effected by sampling rate. These results indicate that code coverage testing should be using variable sampling rates to maximize the trade-off between code coverage results and testing overhead.

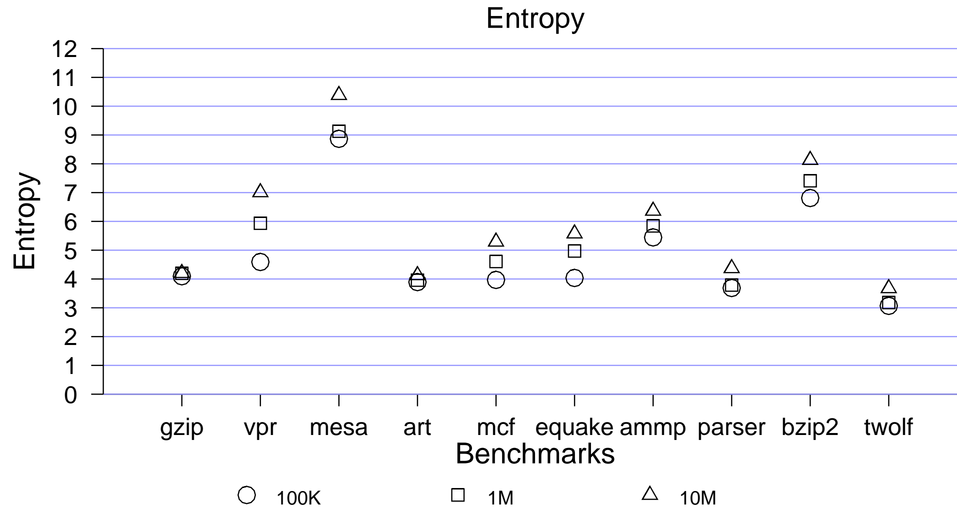


Figure 5.5: Entropy (Kullback-Leibler divergence) of actual/PMU coverage.

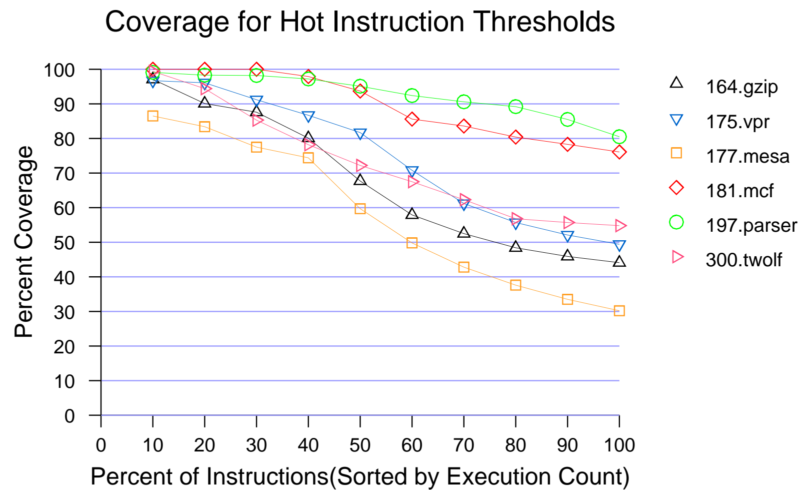


Figure 5.6: Coverage with execution thresholds.

Since sampling program execution can miss program behavior, it is expected that the coverage be biased to the most frequently executing section of program code. Figure 5.6 examines the code coverage (evaluated at 100K sampling rate) in relation to the execution frequency of code region. The leftmost data points (near 10%) on the x-axis show the percent of the top 10% of the most frequently executed instructions that

are discovered using PMU-based code coverage. All the benchmarks in the graph follow the same trend. For the top 10-30% of the most frequent instructions, PMU-based code coverage does very well usually discovering between 80-100% of all the instructions. However, at around 40-50%, there is generally a significant drop and then some leveling out of the coverage percentage. More specifically, for the lowest executing code portion (90-100% on the x-axis) of **177.mesa**, only 30% of the code will be detected and reported as executed. Nevertheless the results show that even infrequent code regions can be detected with hardware-based code coverage.

5.6 PMU Code Coverage with Multiple Runs

One opportunity to improve the quality of the PMU-code coverage approach is to aggregate data from multiple execution runs. This is simply a matter of collecting the PMU monitoring tool's output from multiple runs and performing off-line analysis. Figure 5.7(a) shows an example of aggregating up to 20 separate runs each at a regular sampling period of 100K clock cycles. There are two general trends for the benchmarks shown in Figure 5.7(a). The first is that some benchmarks such as **164.gzip**, **197.parser**, and **300.twolf** significantly improve their code coverage percentage by over 10% (**164.gzip** actually improves over 20%). In these cases, the aggregation of multiple runs seems very promising. In other cases, such as **177.mesa** and **181.mcf** do not result in substantial improvement.

As mentioned in Section 2.5.2, one of the issues facing periodic sampling is sampling aliasing. It is possible to miss important sections of code that periodically execute in the time between samples. Randomized sampling periods may be used in order to account for sampling aliasing. Figure 5.7(b) shows 20 aggregated runs using randomized sampling. The behavior of benchmarks such as **175.vpr**, **197.parser** and **300.twolf** are not significantly altered. However, **177.mesa** and **181.mcf** see substantial improvements. In these cases, the random sampling is able to uncover sections of code

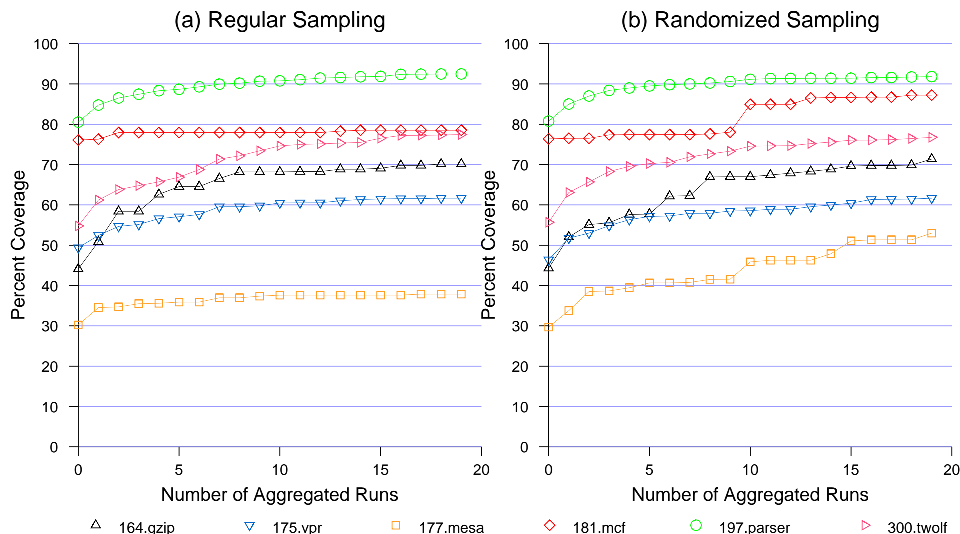


Figure 5.7: Code coverage percentage from aggregating multiple runs using (a) regular sampling period and using (b) randomized sampling period of 100K.

that regular sampling does not discover. From these improvements, it can be seen that randomized sampling allows for certain runs to uncover more sections of code that from what can be seen by using a fixed sampling period.

5.7 Summary

This chapter characterized the amount of information provided by sampling branch vectors with respect to code coverage. It is shown that a code coverage can be improved over 50% on average by utilizing branch vectors as opposed to PC samples. It is also demonstrated that dominator analysis is able to improve the code coverage about another 50% over code coverage with branch vectors. In addition, it is shown that code coverage is fairly stable across different runs of a program (with around 5-10% variation) and that randomized sampling periods may be used to uncover additional sections of code.

Chapter 6

Future Work

This thesis motivates and provides initial results in applying compiler analysis for path profiling and code coverage analysis. While the results of this approach seem promising, there is still much to investigate and explore. Future work includes:

- **Function Correlation:** As mentioned in Section 4.4, sample branch vectors provide a great deal of function correlation. Although this information is ignored here, it should be explored in providing profile data for inter-procedural optimizations.
- **Region Formation:** Region Formation is proposed as a means of keeping the number of total paths in a CFG manageable. However, as discussed in Section 3.5.3.2, the paths formed are different from traditional Ball-Larus [5] paths used in previous path profiling studies. The effect of using a different set of paths is set aside for possible future work. Also, different region formation algorithms could be explored to guarantee that most hot paths executed can be found in a single region.
- **Improved PMU Path Profiling Algorithms:** The path matching and path crediting algorithm described in Section 3.5.3.3 is overly simplistic and uses assumptions that are not necessarily true. Although it is still able to detect hot

paths, there is room for improvement with regards to accuracy.

- **Aggregating Multiple Runs:** The idea of aggregating multiple runs is motivated in this paper, but further investigation is necessary. The ability to use data from multiple previous runs of a program could be crucial for run-time systems; especially for continuous optimization systems in which profile data and optimizations are maintained system-wide across many runs of many programs.

- **Effect of Compiler Optimizations:** The work presented in this thesis uses benchmarks which are compiled with classical optimizations but without more aggressive profile-guided optimizations. As many aggressive optimizations such as superblock formation [26] attempt to straight-line hot code, it is expected that each sampled branch vector may provide further information facilitating more accurate analysis of run-time code behavior.

Chapter 7

Summary and Conclusion

This thesis motivates and presents initial results for a utilizing a hardware-software hybrid profiling system with the intent of driving code layout optimizations in a run-time optimization system. The hybrid profiling scheme utilizes an online monitoring phase and an offline analysis phase. Run-time overhead is reduced by sampling hardware as well as by decoupling the run-time data collection from the run-time data analysis. Software analysis of run-time data allows the profiler to be more flexible in the type of analysis performed and the type of profile generated.

Hybrid profiling frameworks, such as the one in this thesis, have been studied in past research. However, this work provides two important contributions to a hybrid profiling system. First, instead of sampling the PC like previous implementations, PMU branch vectors are sampled. A branch vector is a series of branch addresses which can be utilized to represent a trace of program execution. Branch vector sampling is supported on modern microprocessors and allows for the collection of more information when compared to simple PC-sampling for two reasons. First, a PC sample indicates the execution of a single basic block while a branch vector indicates the execution of multiple basic blocks. Second, the basic blocks provided by the branch vector are correlated and provide path information.

The second contribution is the use of a compiler framework for the software

analysis of the PMU samples. A compiler infrastructure has access to information that is not readily available at run-time. For example, the compiler infrastructure can place sampled information in the context of the complete program control flow. A compiler also has the ability to perform deeper program analysis and uncover information which is not immediately apparent from the PMU samples. Three specific forms of compiler analysis are described in this thesis; 1) partial path extensions, 2) dominator analysis, and 3) path profile generation.

Finally, the profiling framework demonstrated by doing two case studies. In the first, the path information inherent in branch vectors is used to generate an estimated path profile. In the second, the branch vectors are used for code coverage analysis. These studies illustrate the the flexibility of the hybrid framework showing how different profile information can be obtained by using the same hardware samples and utilizing different software analysis routines.

Overall, this compiler-aided, PMU-based profiling technique shows promise. The usage of branch vectors significantly improve the information provided by each hardware sample. Branch vectors provide a path of around 38 instructions (Section 4.4) and is able to improve code coverage over 50% when compared to code coverage with PC-sampling (Section 5.3). The compiler analysis is also able to significantly improve the amount of information per sample. Partial path extensions increase path length by around 20% (Section 4.4) while dominator analysis can be used to improve code coverage another 50% when compared to code coverage using branch vectors (Section 5.3). When this profiling infrastructure to generate path profiles, an accuracy of 88% can be obtained at 1% overhead using a sampling period of 10M clock cycles (Section 4.6). These results indicate that a hybrid compiler-aided, PMU-based profiling technique using branch vectors should be able to provide high fidelity profile information to a run-time optimization system while incurring little overhead.

Bibliography

[1] A. Aho, R. Sethi, and J. Ullman. Compilers: Principles, Techniques, and Tools. Addison-Wesley, 1986.

[2] J. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? In Proc. of the 16th ACM Symposium of Operating Systems Principles, pages 1–14, October 1997.

[3] Apple Computer, Inc. http://developer.apple.com/tools/performance/.

[4] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation, pages 1–12, June 2000.

[5] T. Ball and J. R. Larus. Efficient path profiling. In Proceedings of 29th Annual Int'l Symposium on Microarchitecture, pages 46–57, December 1996.

[6] T. Ball and J.R. Larus. Optimally profiling and tracing programs. In ACM Transactions on Programming Languages and Systems, July 1994.

[7] T. Ball, P. Mataga, and M. Sagiv. Edge profiling versus path profiling: The showdown. In Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 134–148, January 1998.

[8] Mike D. Bond and Kathryn S. McKinley. Practical path profiling for dynamic optimizer. In Proceedings of the 3rd International Symposium on Code Generation and Optimization(CGO-2005), March 2005.

[9] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In 1st Annual International Symposium on Code Generation and Optimization (CGO-03), March 2003.

[10] Bryan R. Buck and Jeffrey K. Hollingsworth. An API for Runtime Code Patching. In Journal of High Performance Computing Applications 14 (4), Winter 2000.

[11] P. P. Chang and W. W. Hwu. Trace selection for compiling large C application programs to microcode. In Proceedings of the 21st International Workshop on Microprogramming and Microarchitecture, pages 188–198, November 1988.

[12] P. P. Chang, S. A. Mahlke, and W. W. Hwu. Using profile information to assist classic code optimizations. Software Practice and Experience, 21(12):1301–1321, December 1991.

[13] Howard Chen, Wei-Chung Hsu, Jiwei Lu, Pen-Chung Yew, and Dong-Yuan Chen. Dynamic trace selection using hpm sampling. In Proceedings of the International Symposium on Code Generation and nOptimization(CGO 2003), March 2003.

[14] Wen-Ke Chen, Sorin Lerner, Ronnie Chaiken, and David M. Gillies. Mojo: A dynamic optimization system. In 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3), December 2000.

[15] T. M. Conte, B. A. Patel, K. N. Menezes, and J. S. Cox. Hardware-based profiling: An effective technique for profile-driven optimization. International Journal of Parallel Programming, 24(2):187–206, April 1996.

[16] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos. Profileme: Hardware support for instruction-level profiling on out-of-order processors. In Proceedings of the 30th Annual International Symposium on Microarchitecture, pages 292–302, December 1997.

[17] Giuseppe Desoli, Nikolay Mateev, Evelyn Duesterwald, Paolo Faraboschi, and Joseph A. Fisher. DELI: A new run-time control point. In 35th Annual International Symposium on Microarchitecture, December 2003.

[18] Evelyn Duesterwald and Vasanth Bala. Software profiling for hot path prediction: Less is more. In Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems, November 2000.

[19] Evelyn Duesterwald, Calin Cascaval, and Sandhya Dwarkadas. Characterizing and predicting program behavior and its variability. In Proceedings of the 12th International Symposium on Parallel Architectures and Compilation Techniques(PACT03), September 2003.

[20] Stephane Eranian. The perfmon2 interface specification. Technical Report HPL-2004-200R1, Hewlett-Packard Laboratory, February 2005.

[21] A. Eustace and A. Srivastava. ATOM: A flexible interface for building high performance analysis tools. In Proceedings of the 1995 USENIX Conference, January 1995.

[22] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. IEEE Transactions on Computers, C-30(7):478–490, July 1981.

[23] Matthias Hauswirth, Peter F. Sweeney, Amer Diwan, and Michael Hind. Vertical profiling: Understanding the behavior of object-oriented applications. In Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, October 2004.

[24] Hewlett-Packard Development Company. perfmon project http://www.hpl.hp.com/research/linux/perfmon/.

[25] W. W. Hwu and P. P. Chang. Inline function expansion for compiling realistic C programs. In Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation, pages 246–257, June 1989.

[26] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The Superblock: An effective technique for VLIW and superscalar compilation. The Journal of Supercomputing, 7(1):229–248, January 1993.

[27] IBM. PowerPC 740/PowerPC 750 RISC Microprocessor User's Manual, 1999.

[28] Intel Corporation. Intel Itanium 2 processor reference manual: For software development and optimization. May 2004.

[29] Rahul Joshi, Michael D. Bond, and Craig Zilles. Targeted path profiling: Lower overhead path profiling for staged dynamic optimization systems. In Proceedings of the International Symposium on Code Generation and Optimization(CGO-2004), March 2004.

[30] Michael Kearns, Yishay Mansour, Dana Ron, Ronitt Rubinfeld, Robert E. Schapire, and Linda Sellie. On the learnability of discrete distributions. In STOC '94: Proceedings of the twenty-sixth annual ACM symposium on Theory of computing, pages 273–282. ACM Press, 1994.

[31] T. Kistler and M. Franz. Continuous program optimization. In IEEE Transactions on Computers vol. 50 n. 6, June 2001.

[32] Jiwei Lu, Howard Chen, Rao Fu, Wei-Chung Hsu, Bobby Othmer, Pen-Chung Yew, and Dong-Yuan Chen. The performance of runtime data cache prefetching in a dynamic optimization sytem. In Proceedings of the 36th International Symposium on Microarchitecture(MICRO-36), December 2003.

[33] Jiwei Lu, Howard Chen, Pen-Chung Yew, and Wei-Chung Hsu. Design and implementation of a lightweight dynamic optimization system. In Journal of Instruction-Level Parallelism 6(2004), pages 1–24, April 2004.

[34] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In ACM SIGPLAN Conference on Programming Language Design and Implementation, Chicago, IL, June 2005.

[35] Chi-Keung Luk, Robert Muth, Harish Patil, Robern Cohn, and Geoff Lowney. Ispike: A post-link optimizer for the intel itanium architecture. In 2st Annual International Symposium on Code Generation and Optimization (CGO-04), March 2004.

[36] M. C. Merten, A. R. Trick, C. N. George, J. C. Gyllenhaal, and W. W. Hwu. A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. In Proceedings of the 1999 International Symposium on Computer Architecture, May 1999.

[37] M. C. Merten, A. R. Trick, E. M. Nystrom, R. D. Barnes, J. C. Gyllenhaal, and W. W. Hwu. A hardware mechanism for dynamic extraction and relayout of program hot spots. In Proc. 2000 Int'l Symp. on Computer Architecture, pages 136–147, June 2000.

[38] OpenIMPACT Research Compiler. http://www.gelato.uiuc.edu/.

[39] OProfile System Profiler for Linux. http://oprofile.sourceforge.net.

[40] PAPI: Performance Application Programming Interface. http://icl.cs.utk.edu/papi/.

[41] K. Pettis and R. C. Hansen. Profile guided code positioning. In Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation, pages 16–27, June 1990.

[42] Timothy Sherwood, Suleyman Sair, and Brad Calder. Phase tracking and prediction. In Proceedings of the 17th annual international conference on Supercomputing. ACM Press, 2003.

[43] Brinkley Sprunt. Pentium 4 performance-monitoring features. In IEEE Micro 22(4), pages 72–82, 2002.

[44] Peter F. Sweeney, Matthias Hauswirth, Brendon Cahoon, Perry Cheng, Amer Diwan, David Grove, and Michael Hind. Using hardware performance monitors to understand the behavior of java applications. In Proceedings of the 3rd Virtual Machine Research and Technology Symposium(VM2004), 2004.

[45] K. Vaswani, M. Thazuthaveetil, and Y. Srikant. A programmable hardware path profiler. In Proceedings of the 2005 International Symposium on Code Generation and Optimization, March 2005.

[46] D. W. Wall. Predicting program behavior using real and estimated profiles. In Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation, pages 59–70, June 1991.

[47] Robert W. Wisniewski, Peter F. Sweeney, Kartik Sudeep, Matthias Hauswirth, Evelyn Duesterwald, Calin Cascaval, and Reza Azimi. Performance and environment monitoring for whole-system characterization and optimization. In Proceedings of the 1st Watson Conference on Interaction between Architecture, Circuits, and Compilers, 2004.