Analyis of Path Profiling Information Generated with Performance Monitoring Hardware

Alex Shye Matthew Iyer Tipp Moseley David Hodgdon Dan Fay
Vijay Janapa Reddi Daniel A. Connors
Department of Electrical and
Computer Engineering
University of Colorado at Boulder
{shye, iyer, moseleyt, hodgdon, fay, dconnors}@colorado.edu

Abstract

Even with the breakthroughs in semiconductor technology that will enable billion transistor designs, hardware-based architecture paradigms alone cannot substantially improve processor performance. The challenge in realizing the full potential of these future machines is to find ways to adapt program behavior to application needs and processor resources. As such, run-time optimization will have a distinct role in future high performance systems. However, as these systems are dependent on accurate, fine-grain profile information, traditional approaches to collecting profiles at run-time result in significant slowdowns during program execution.

A novel approach to low-overhead profiling is to exploit hardware Performance Monitoring Units (PMUs) *present in modern microprocessors. The Itanium-2 PMU can periodically sample the last few taken branches in an executing program and this information can be used to recreate partial paths of execution. With compiler-aided analysis, the partial paths can be correlated into full paths. As statistically hot paths are most likely to occur in PMU samples, even infrequent sampling can accurately identify these paths. While traditional path profiling techniques carry a high overhead, a PMU-based path profiler represents an effective lightweight profiling alternative. This paper characterizes the PMU-based path information and demonstrates the construction of such a PMU-based path profiler for a run-time system.*

1 Introduction

Current semiconductor breakthroughs have the potential to enable unparalleled advances in computer system performance. Such gains, however, are unlikely to achieve expected performance improvements by simply scaling hardware resources or deploying instruction-level parallel compiler techniques. Next generation systems will need the ability to adapt to system resources and program behavior. Run-time optimizing systems for these future processors will deploy sophisticated profile-guided optimizations to improve performance. However, in order to maximize the performance gain of these run-time optimizations, efficient profiling techniques are required that can accurately describe a program's runtime behavior.

Profiling provides valuable information to a whole class of optimizations: superblock formation [12], code positioning [20], and improved function inlining[11]. A common method of collecting profile information for these profile-directed optimizations is through program instrumentation. The program is first compiled with counters inserted to generate profile information. The profile-enabled program is then run to collect profile data. Finally, the profile is fed back to the compiler to re-compile the program and perform profile-directed optimizations. Although these profile-directed optimizations have been shown to improve program performance, they have not been widely deployed by software vendors for a number of reasons. One reason is that they require the inconvenient compile-run-recompile sequence described earlier. Furthermore, high overhead introduced by instrumentation makes the process inconvenient.

The ideal run-time profile collection system should have three distinct characteristics. First, it should provide accurate profile information for a dynamic optimizer to utilize. Second, the system ideally should gather all profile information in one stage. Finally, and most importantly, the run-time collection of information should occur with little to no overhead. Unfortunately, most approaches to profiling only meet one or two of these two goals. Instrumentation-based techniques provide an accurate profile while sacrificing the cost of overhead as well as convenience of compilation. Novel hardware-based techniques are emerging to collect run-time events using *hardware performance counters*. Although such structures efficiently capture run-time information, researchers have only begun to study the characteristics and benefit of the amount and type of information needed for driving run-time optimization [7, 16].

Traditionally, most profile-directed optimizations use point-based profiles which locate specific events to profile. The most popular of these is edge profiling in which branch directions are profiled. Edge profiling, as well as other types of point-based profiling, cannot correlate specific events with each other. Path profiling [4] has been recently shown to be a superior form of profiling [5]. It correlates branches by keeping track of path execution counts instead of simple branch counts. However, path profiling usually comes with a significant increase in overhead(31% for path profiling versus 16% for edge profiling [4]).

In this paper, we demonstrate a low-overhead PMU-based path profiling system and characterize the hardware collected statistics. This path profiling system is based on the Intel Itanium-2 PMU [13]. The Branch Trace Buffer(BTB) is used to periodically sample up to the last four taken branches and dump the sampled data to a file. Later, a compiler-aided offline analysis phase uses these samples to recreate paths through the program and generate a path profile. Accuracy is measured comparing the PMU-based path profile to a full path profile.

The main contributions are:

- a characterization of path information that can be inferred from the Itanium-2 PMU
- an algorithm for extrapolating a path profile from incomplete path information
- quantification of the effectiveness of this techniques

The rest of this paper is organized as follows. Section 2 discusses background and related work. Section 3 continues into a description of our path profiling technique. Experimental data and analysis are provided in Section 4. Section 6 plans out future work, Section 5 discusses related work, and we conclude the paper in Section 7.

2 Motivation

In this section, we first discuss the characteristics of the ideal profiler. Then we move into the advantages of path profiling over traditional point-based profiles. We then describe existing profiling techniques and compare them to the characteristics of our ideal profiler. Finally, we discuss performance monitoring and how we believe it can be used for providing path profiles.

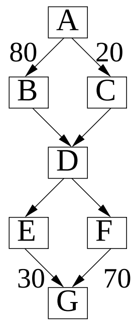


Figure 1. Edge Profiled CFG

2.1 The Ideal Profiler

The effectiveness of profile-directed optimizations hinge upon the quality of the profile information. Profiles which do not reflect run-time execution result in poorly optimized code. An accurate profile enables many optimization opportunities. Unfortunately, the quality of profile accuracy often comes with the trade-off of the increased overhead required for gathering this fine-grained run-time information.

The ideal run-time profiler should have three characteristics: 1) high accuracy, 2) single-stage profiling, and 3) low profiling overhead. Accuracy is crucial for enabling effective optimization. An accurate profile is one that correctly reflects a program's run-time execution behavior. In order for profiling to be feasible in a run-time system, it must be done within a single stage. Finally, and perhaps most importantly, profile collection overhead must be minimal.

2.2 Path Profiling

Most past research into profile guided optimizations utilize point-based profiles. Edge profiles [3], which capture branch bias information by instrumenting branches, are the most popular of the point-based profiling techniques. However, edge profiling is limited in its ability to accurately describe execution behavior. Figure 1, a control flow graph (CFG) annotated with edge profile information, provides an example of the limitations of edge profiling. An edge profile of the CFG would indicate that the hot path is through basic blocks ABDFG. Suppose that all of the paths through ACD continued down through blocks F and G. In this case, the count for path ABDFG is 50, not the 70 suggested by an edge profile. Path profiling (PP) paints a better picture of the program's execution by correlating branches together into paths. Instead of collecting data at branch instructions, path profiling collects counts for execution of specific paths through the code.

2.3 Static Instrumentation

Most path profiles use software-based instrumentation to determine paths. A program is initially compiled with counters strategically placed in the original code to keep track of dynamic path execution. The advantage of this technique is that high accuracy can be attained by inserting whatever counters are desired through instrumentation. The main disadvantage is that software-based instrumentation incurs a high overhead.

The original path profiling algorithm, proposed by Ball and Larus [4], divided the code into regions, typically along function boundaries, and created directed acyclic graphs (DAGs) in each region by removing loop-back branches. A single counter per region is used in conjunction with an edge numbering algorithm to determine paths. In terms of overhead, their path profiler averaged a 31% slowdown, with slowdowns as high as 97% for gcc.

Targeted Path Profiling (TPP) [14] and Practical Path Profiling (PPP) [6], extensions of Ball and Larus' PP, 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%.

Our approach to path profiling offers a few advantages over PP, TPP and PPP. First, we do not need extra stages for profiling. PP requires the 3-phase compile-run-recompile method. TPP and PPP add two extra stages for compiling and running to gather the edge profile. Sampling will be used to statistically ignore cold paths similar to ideas behind TPP and PPP. Additionally, by using hardware to do the actual data collection, run-time overhead will be reduced.

2.4 Dynamic Instrumentation

Instead of inserting instrumentation at compile time, which requires access to source code, instrumentation can also be inserted during runtime using binary instrumentation. PIN [21] is a binary instrumentation tool functionality similar to ATOM [9], but can dynamically inject instrumentation into a running executable. This makes it possible to attach PIN to an already running process to collect profile information. PIN-instrumented binaries, however, experience a high overhead, averaging slowdowns of 2.8x for integer benchmarks(up to 20x slowdown) [17] for simple basic block counting. Although dynamic instrumentation allows for high accuracy and removes the need for multiple profile-execute stages, it drastically increases overhead.

Dynamic optimization systems such as Dynamo [2] implement Most Recently Executed Tail(MRET), a speculative version of path profiling for trace creation. Once a trace head's execution count reaches a predetermined threshold, it is considered hot and the next dynamic execution path from that trace head becomes a trace. This method has three drawbacks. First, the next dynamic execution path may not be the hot trace, causing this method to be overly aggressive. Also, it 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. The final drawback suffered by these dynamic optimization systems is the high overhead due to interpretation.

2.5 Hardware Profiling Techniques

A number of hardware profiling techniques have been proposed for collecting run-time profile information. Conte [8] uses branch handling hardware coupled with the branch predictor to obtain branch information. Merten's [18] work discusses using a branch behavior buffer for collecting branch profile data.

These techniques incur a low overhead and can effectively gather data during one program run but suffer in accuracy because they are designed to collect edge profiles. Our technique incurs a higher overhead compared to specialized hardware but offers two advantages. First, because the analysis is done in software instead of hardware, more analysis can be performed. Second, our method utilizes existing performance monitoring hardware.

2.6 Performance Monitoring

Modern microprocessors such as the Intel Pentium 4, Intel Itanium, and IBM PowerPC 970 provide a rich set of performance counters. The work in this paper uses the Intel Itanium-2 PMU [13]. It includes a set of counters which can be configured to count any four of almost 500 events. It also allows for sampling of Event Address Registers to capture recent data or instruction cache or TLB misses. This paper takes advantage of the PMU's ability to sample BTB registers to obtain partial paths.

3 PMU Path Profiling

Clearly, there is not a solution for run-time profiling yet that meets the criteria of the ideal profiler. We propose a new two-phase path profiling technique to meet these criteria. The first phase consists of sampling a PMU to collect run-time path information. This path information is then run through an off-line analysis phase to create a full path profile. This technique is able to achieve low overhead by sampling hardware but is able to create accurate path information in the off-line phase. In addition, the profile can be gathered during run-time without any previous instrumentation phases or interpretation.

Using the Itanium-2 PMU as an example, section 3.1 describes the run-time sampling phase. The remaining sections describe the off-line analysis. Sections 3.2 and 3.3 describe how we prepare the PMU data for extrapolation, and section 3.4 shows how we generate an estimated path profile.

3.1 Collection of BTB (Branch) Traces

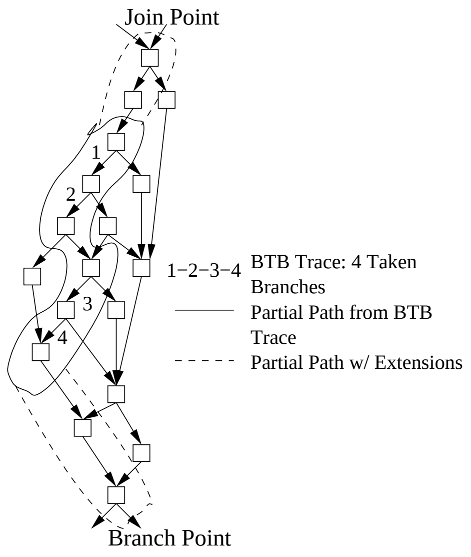
The Itanium-2 PMU BTB contains eight registers which are treated as a circular buffer. Each executed branch instruction usually requires two of the BTB registers; one for the branch instruction address and another for the branch target address. Because of this, the BTB registers effectively act as a four branch circular buffer. In the Itanium-2 PMU, the user is able to conduct BTB samples through a set of user-defined filters. In this work, we configure the BTB to sample only taken branches. This improves path collection for two reasons. First, sampling all branches is redundant because the off-line tool uses the CFG to interpolate fall-through paths between taken branches. Second, code optimizations emphasize fall-through paths over taken paths for increased code locality. Therefore, sampling only taken branches provides more path information and a more accurate path profile.

Upon conclusion of each sampling period, the PMU is queried for the contents of the BTB. The BTB registers form a ***BTB trace***. The trace is stored in a hash table that collects and aggregates duplicate traces. At the end of program execution, the BTB traces and their respective occurrence counts are written to a file to be used by the off-line analysis tool.

3.2 Partial Path Creation

Forming partial paths The first profiling step interpolates a set of BTB traces to form the corresponding set of partial paths. A ***partial path*** is simply a BTB trace mapped onto the original CFG including two enhancements. First, it infers the fall-through edges between taken branches. Second, it extends the path above the first edge and below the last edge until a point of uncertainty is reached.

Figure 2 shows an example of partial path creation. The numbers 1 through 4 indicate the four taken branches from the BTB Trace. These four branches are related back to the CFG while following fall-through edges between the branches to produce an initial partial path shown by the solid line. The partial path can then be extended upward and downward to a point of uncertainty. For example, the



Join Point
1
2
3
4
1−2−3−4
BTB Trace: 4 Taken Branches
Partial Path from BTB Trace
Partial Path w/ Extensions
Branch Point

Figure 2. Extending Partial Paths

top of the initial partial path can be extended up until a join point in the CFG and the bottom can be extended down until a branch point. These extensions help to provide longer partial paths while ensuring that they are unique. Partial path extensions increase the probability that the partial path will span multiple regions. Region-based path discovery, the next stage of the off-line phase, is discussed in the following subsections.

Splitting partial paths After partial paths creation, they are split along function boundaries and loop backedges. We must perform this splitting because these partial paths are later compared to region-based paths. Our current implementation of regions constrains them to function and loop boundaries.

Complications Due to the nature of the PMU, several complications can arise in obtaining BTB traces. First, the branch instruction addresses entered into the PMU are inexact. From what we can gather, this is because processor performance was prioritized over accuracy of the PMU. Second, branch addresses are occasionally entered without a target. If the branch addresses were exact, this would not be a problem. However, it means that a bit of guesswork may be required to build a partial path from a list of BTB entries. In the rare situations where we cannot safely guess the path of execution, the BTB trace is discarded.

3.3 Region-based Path Discovery

Unlike instrumented path profiling, sampled path profiling introduces the problem of ***path ambiguity***. Under the random sampling assumption, samples do not have guaranteed starting points, and due to the inexact nature of current performance monitoring hardware, sampled paths contain

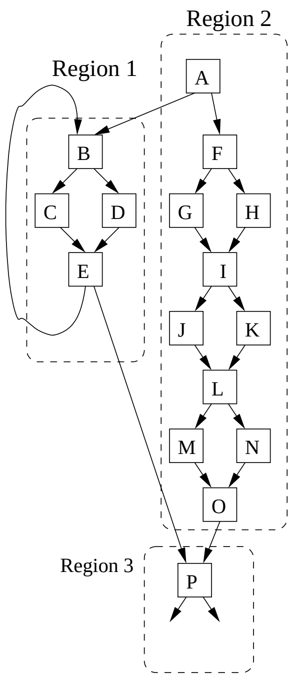


Figure 3. Example of region formation

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 1. Example of path matching of partial paths to region-based paths

an inconsistent number of edges. To mitigate these challenges we propose ***region-based paths***.

First, we define relevant terms used in the description of our algorithm. Our terminology is consistent with Ball's study [5].

Definition 3.1 *Let $R(V, E)$ 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 full 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 $P(e)$.

Under the paradigm of region-based path discovery, a CFG is divided into regions of basic blocks such that given a partial path p we can accurately derive the set of full paths M that contain all the edges of p. Thus, the following rules govern the formation of a region R:

1. R can only be entered via its entry vertex
2. R_{BODY} must not contain basic blocks both inside and outside a loop
3. The total number of paths from R_{ENTRY} to all R_{EXIT} vertices must be less than some limit N.
4. R_{BODY} should contain as many basic blocks as possible.

To form regions within these limitations, we use a greedy algorithm. This method begins with the single region R containing just one vertex $entry$ 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 program within the CFG have been added to regions.

Figure 3 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 would break 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 memory. Each full path within a set of CFGs can be accessed by a unique identifier. During the stage of path enumeration, we also implement the function $P(e)$ by using a hash table.

3.4 Path Profile Generation

At this point in our off-line analysis we have built a set of partial paths from BTB traces and divided the entire CFG of the application into regions. We extrapolate each partial path into its matching set of full paths as described below.

Definition 3.2 *Given a single partial path p, the minimum matching set M of full paths which contain p is the intersection of all $P(e)$ for each edge of p. That is, for partial path p consisting of edges $[e_1, e_2...e_n]$,*

$$M_p = \bigcap_{i=1}^{k} P(e_i)$$

Depending on the number of edges within p and the characteristics of the CFG, the matching set M may contain an indeterminate number of paths. The next step of our algorithm assigns weights to each path of M based on the following assumptions:

1. Equal probability of execution of each path within M
2. Random sampling of actual program flow
3. Adequate samples to clearly distinguish frequently executed paths

Given these assumptions equal weights to every matching full path can be assigned; however, enough samples must be available in order to distinguish hot paths from cold. For example, consider partial paths and their occurrence counts from the CFG in Figure 3 and Table 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. These hot paths are ready for immediate use in path-profile based compiler optimization.

4 Experiment

4.1 Methodology

For our experiments, we explore this path profiling technique on applications that have not yet been highly optimized. The reason for this is to show that this technique can be used as a stand-alone profiler. Studying highly optimized code implies that a previous profile run has been used. We are interested in the effects of optimizations on the quality of profiles but leave this exploration up to future work.

The experiments use the *SPEC 2000* benchmarks compiled with the base configuration of the OpenIMPACT Research Compiler [19]. The benchmarks are compiled with classical optimization but without more aggressive profile-guided optimizations. The PMU collection tool developed using the perfmon kernel interface and libpfm library [10] initiates and samples the Itanium-2 PMU BTB registers on individual SPEC programs. The PMU samples are analyzed off-line with an OpenIMPACT module to generate the path profile.

4.2 Effect of PMU Sampling Period

Figure 4 shows the effect of sampling rate on run-time overhead as well as the number of unique paths discovered by the PMU for a few benchmarks. The sampling rate is varied from 50K to 10M clock cycles. Naturally, a lower sampling rate decreases the overhead but provides a lower number of unique paths while a high sampling rate increases the overhead but provides more unique paths.

PMU sampling overhead remains relatively low, less than 10%, from 10M all the way down to around 500K. When the sampling rate is decreased further, the percentage overhead increases quickly up to $\sim$50% for a sampling period of 50K. The number of unique paths discovered by the PMU rises steadily for each decrement in sampling rate.

4.3 Partial Paths

Once branch samples are collected, they are analyzed and related back to the IMPACT low-level intermediate representation(IR) to create partial paths. The nature of PMU sampling allows these partial paths to span function boundaries, stretch across loopback edges, and even extend from the user program into shared libraries. However, the system ignores paths in libraries and splits partial paths at function boundaries as well as loopback edges in this initial infrastructure.

Table 2 shows the average length of partial paths initially, after partial path extensions, and after splitting on function boundaries and loopback edges. Path lengths are measured in number of low level IR instructions. On average, the system locates paths $\sim$38 instructions long. Partial path extensions increase the average length by about 20%. After splitting, the average path length drops about 40% from the extended length to $\sim$27 instructions.

Table 3 characterizes partial paths in relation to the number of procedure boundaries crossed by the partial path. A surprisingly large number partial paths not only cross one function boundary, but span across multiple boundaries. In particular, a large majority of partial paths in 186.crafty and 197.parser cross at least one function boundary. This indicates that had our infrastructure been able to account

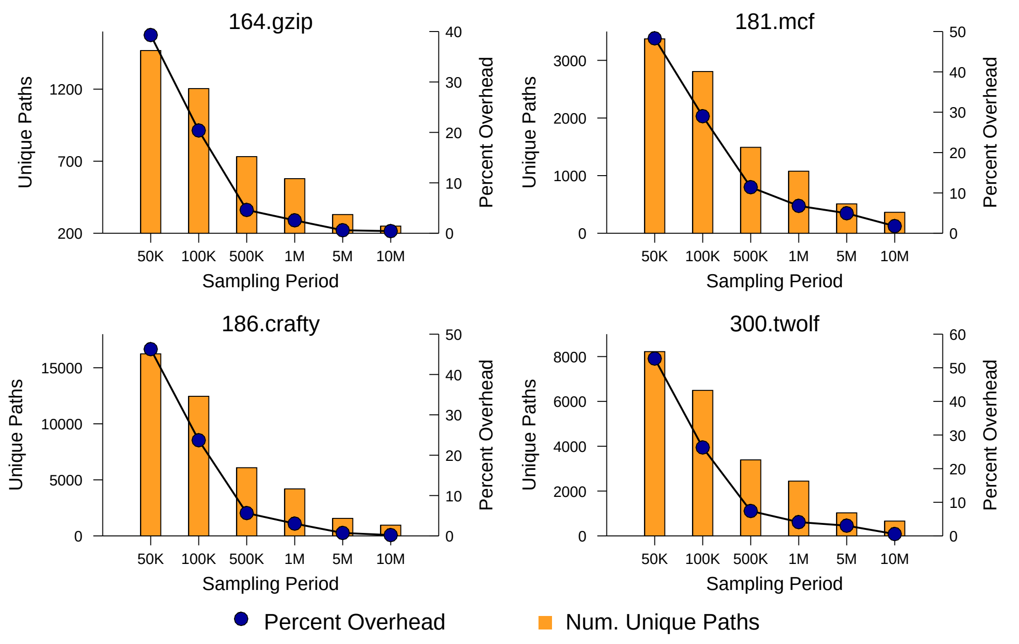


Figure 4. Overhead and number of unique paths for various sampling periods

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
186.crafty	36.0	45.2	31.7	31.1
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	37.7	45.4	33.6	26.9

Table 2. Average length of partial paths in instruction count initially, after partial path extensions, and after splitting on function boundaries and loopback edges

	Func. Boundaries Spanned				
Benchmark	**0**	**1**	**2**	**3**	**4**
164.gzip	187	149	35	2	0
175.vpr	234	171	162	65	12
177.mesa	75	50	27	7	0
179.art	230	29	7	1	0
181.mcf	1106	237	96	26	1
183.equake	231	31	24	1	0
186.crafty	1367	2281	968	154	13
188.ammp	298	123	42	5	2
197.parser	488	654	599	224	40
256.bzip2	532	353	279	55	9
300.twolf	1293	514	467	55	7

Table 3. Breakdown of partial paths spanning function boundaries(500K Sampling Period)

for paths spanning across function boundaries and loopback edges, it would have been able to utilize better path information. Future work is planned to integrate function and loopback correlation into our infrastructure.

4.4 Aggregating Data from Multiple Runs

The profiling infrastructure enables aggregate profile information to be collected from multiple runs of a program. By gathering PMU information over separate runs, analysis of lost paths due to statistical sampling can be measured. In Figure 5 shows the effects of aggregating the PMU BTB samples from multiple runs of a few benchmarks with the same input. Additional runs increase the number of unique PMU paths. The greatest increase occurs from combining up to 10 runs. There is a slight leveling off after 10 runs. It is possible that the paths collected from multiple runs will fill in missed important partial paths in other runs.

4.5 Accuracy Results

To measure accuracy of the PMU-generated paths, we generate a full path profile with a PIN tool. The PMU path profile is compared to the full path profile using a method

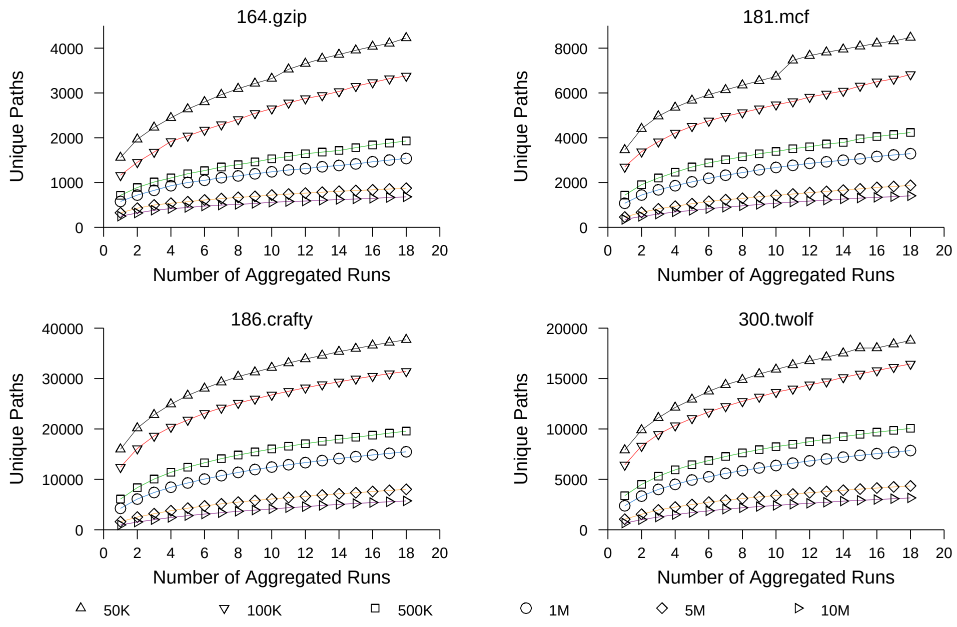


Figure 5. Number of unique paths found by aggregating data from runs with same input set

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. Number of actual hot paths and percent of total flow they account for

similar to Wall's weight matching scheme [22]. In this paper, accuracy is defined as

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

In this equation $F(p)$ is the flow of a path. This is defined as the path's count divided by the count of all the paths added together. This represents the percentage of the all counts that path p accounts for. H_{actual} is the set of paths in the full path profile which are above a set threshold. We use 0.125% as this threshold is used previous path profiling studies [5, 6]. $H_{estimated}$ is then the created by selecting the hottest paths in our path profile equal to the number of paths in H_{actual}.

Table 4 shows the number of hot paths in each benchmark which have flows above the hot threshold as well as the percent of total execution flow that the hot paths account for. The percentages indicate 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.

Figure 6 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 with one set of benchmarks in the high 90%s and another set in the 80%s. As sampling period increases, the accuracy remains relatively constant for a while. This is because each of these sampling periods contains enough samples for our technique to locate the important hot paths. However, once the sampling period is increased to a certain point(around 5M), accuracy suffers because we are not able to collect enough samples to recreate the hot paths. By observing this behavior, a sweet spot can be seen around a sampling rate of 5M-10M. If the sampling period is set to 10M, 88% accuracy can be obtained at $\sim$1% run-time overhead.

4.6 Incorrectly Identified Hot Paths

Although the method of assigning equal weights to multiply-matched partial paths allows for accurate identification of the hottest paths in an individual region, we sacrifice the ability to correctly identify the hot paths among all regions in a routine. To illustrate this flaw, we first define

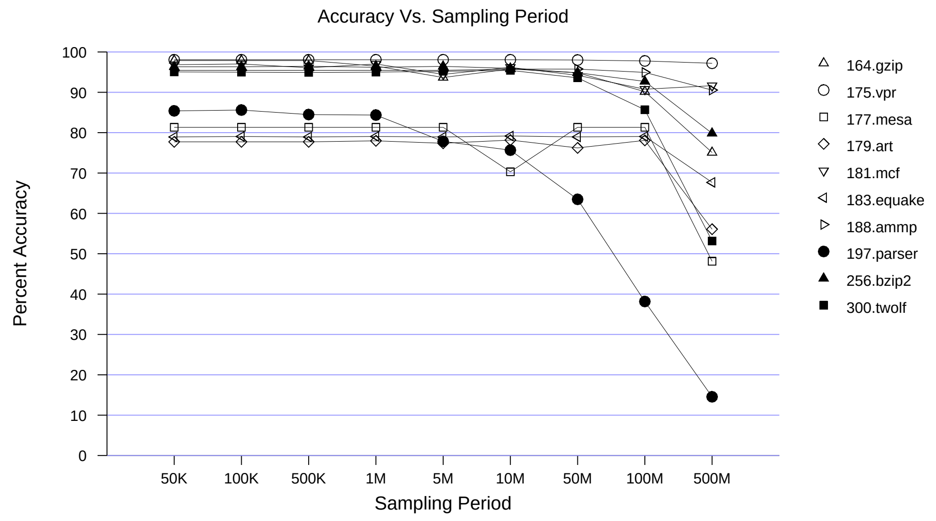


Figure 6. Accuracy vs. Sampling Period

the **matching ratio** for a region as the average size of the matching set M for all partial paths in the region.

Now consider an extremely hot region R whose paths represent a large percentage of total program flow. Assume that the matching ratio for R is significantly greater than one. In other words, attempts at matching partial paths within R are unable to uniquely identify full paths.

In this scenario, a substantial number of paths within the region have been falsely matched. Since R represents a large percentage of total program flow, all of the falsely-matched paths of R are likely to be incorrectly identified as hot paths. This phenomenon occurs frequently in large regions of code due to high matching ratios within them. Amplifying the over-matching problem is that applications tend to frequently execute code within large regions.

We are looking at several different ways to detect and correct for the over-matching problem such as cold-edge elimination and obvious path elimination [14, 6], redistribution of weights based on heuristics, and adaptive weight distribution.

5 Related Work

The profiling approach for trace selection in the ADORE dynamic optimization system [7, 16] is very similar to our technique for profiling. ADORE uses the Itanium-2 PMU for collecting profile information aimed at improving data cache performance Our work can be viewed as an extension from their approach with a few differences. The goal of the ADORE optimizer is to use the PMU to detect a small amount of hot traces for optimization. Our work differs in that we use the PMU to gather and build a path profile. While ADORE is only interested in a few traces to optimize during run-time, we are interested in as gathering as much information as we can from the PMU, correlating these PMU samples and characterizing the nature of the PMU information with respect to gathering a path profile.

Other sampling ideas stem from continuous profiling and optimization systems [1, 15]. These systems sample performance monitors for profile information to drive feedback driven optimizations. They typically utilize simple event counters or PC sampling techniques. Our work could be an important addition into continuous optimization systems provided future processors are designed with more robust hardware monitoring units.

6 Future Work

As this paper presents the initial results and findings of PMU-generated path profiling, there a number of areas of work to make PMUs a viable profiling mechanism for runtime optimization. Future work involves the following:

- *Region Formation* - Regions represent a potential representation for effectively limiting the number of matches of PMU generated data. Judging from the characteristics of partial paths, it is important that further work investigate regions that span hot function boundaries and loop iterations.

- *Noise Elimination* - Path crediting may introduce noise into generated path profile weights creating false hot-path profiles. It is important to qualify and assess this occurrence in PMU generated profiles and explore algorithms for reducing path accreditation noise.

- *Effects of Optimization* - Aggressive optimizations previously performed will impact the characteristics of

PMU-based path profile information. The details of the impact of a program's optimization on PMU profiles will be investigated.

- *Aggregating Multiple Runs* - Analysis of multiple profiles of different input sets per application may reveal opportunities to deploy persistent optimizations in a run-time system.

7 Conclusion

In this paper, the initial rationale and results of using a PMU-based path profiling system in a run-time optimization system are presented. We demonstrate the construction of a PMU-based profiling system in existing hardware and point out challenges that need to be overcome for the success of this profiling technique. Overall, PMU-based path profiling shows promise as a run-time profiling technology with advantages of the speed of hardware sampling and the ability to accurately detect hot paths. With an overhead of only $\sim$1%, PMU path profiling can obtain an 88% accurate path profile when compared to a detailed software instrumentation-based path profile.

8 Acknowledgements

The authors would like to thank Wei Hsu, Howard Chen and Stephane Eranian for their assistance with the Itanium-2 PMU, and Hariharan Iyer and Manish Vachharajani for their helpful advice The authors would also like to thank the Intel Pin Team for their support and the anonymous reviewers for their insights and comments. This work has been supported by the Intel Corporation.

References

[1] J. Anderson and et al. Continuous profiling: Where have all the cycles gone? In *Proc. of the 16th ACM Symposium of Operating Systems Principles*, pages 1–14, October 1997.

[2] 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.

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

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

[5] 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.

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

[7] H. Chen, W.-C. Hsu, J. Lu, P.-C. Yew, and D.-Y. Chen. Dynamic trace selection using performance monitoring hardware sampling. In *Proceedings of the International Symposium on Code Generation and nOptimization(CGO 2003)*, March 2003.

[8] 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.

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

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

[11] 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.

[12] W. W. Hwu and et al. The Superblock: An effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, 7(1):229–248, January 1993.

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

[14] R. Joshi, M. D. Bond, and C. 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.

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

[16] J. Lu, H. Chen, P.-C. Yew, and W.-C. Hsu. Design and implementation of a lightweight dynamic optimization system. In *Journal of Instruction-Level Parallelism 6(2004)*, pages 1–24, April 2004.

[17] C.-K. Luk and et al. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Chicago, IL, June 2005.

[18] 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.

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

[20] 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.

[21] PIN Dynamic Instrumentation Tool. http://rogue.colorado.edu/pin/.

[22] 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.