TR 02-006

Phase Locality Detection Using a Branch Trace Buffer for Efficient Profiling in Dynamic Optimization

Wei-chung Hsu, Howard Chen, Pen-chung Yew, and Dong-yuan Chen

February 07, 2002

Phase Locality Detection Using a Branch Trace Buffer for Efficient Profiling in Dynamic Optimization

Wei-Chung Hsu Howard Chen Pen-Chung Yew
Department of Computer Science
University of Minnesota

Dong-Yuan Chen
Microprocessor Research Labs
Intel

Abstract

Efficient profiling is a major challenge for dynamic optimization because the profiling overhead contributes to the total execution time. In order to identify program hot spots for runtime optimization, the profiler in a dynamic optimizer must detect new execution phases and subsequent phase changes. Current profiling approaches used in prototype dynamic optimizers are interpretation or instrumentation based. They have either very high overhead, or generate poor quality profiles. We use the branch trace buffer and the hardware performance monitoring features provided in the IA-64 architecture to detect new execution phases and phase changes. The branch trace buffer records the last few branch instructions executed before an event based interrupt is generated. Using the branch trace buffer, our profiler continuously samples execution paths leading to critical performance events, such as cache misses and pipeline stalls. A set of frequently executed traces and their respective performance characteristics within a time interval are considered as an execution phase. Since such phases tend to repeat over time, a dynamic optimizer can exploit the phase locality to drive optimization. We check for new phases and phase changes at the end of each time interval. Although a new phase or a changed phase can be a candidate for optimization, our phase detector delays the invocation of the optimizer until a relatively stable phase is detected. We report the effectiveness of various phase detection methods using Spec2000int as the benchmark. Our results indicate branch trace based phase detection can be suitable for dynamic binary optimizers.

1. Introduction

Static profile-directed optimization (PBO) [Chan91] improves program performance significantly, especially on modern processors [Ayer98, Holl96, Cohn98]. The performance of modern processors is often dominated by the frequency of cache misses and branch mispredictions, and by the effectiveness of instruction scheduling. Using execution profile information, the compiler may perform cache prefetching, static branch prediction for trace layout, and trace scheduling [Fish92]. Static profile-directed optimization has been adopted by many commercial compilers and has had a profound impact on latest processor architectures [Gwen99, Intel00a, Sun00].

Many forms of profiles can be used to guide program optimizations. Simple execution profiles[Grah82] can be used to identify time-consuming procedures. Edge-profiles and path-profiles [Ball96] are used to select hot execution paths and enable more effective scheduling and optimization [Kist01]. Cache miss profiles can direct effective cache prefetch insertion. Inter-procedure call graph profiles are critical for cross-module procedure inlining [Ayer98]. Memory conflict profiles allow more aggressive data speculation (i.e. advanced loads) [Conn97]. Instruction nullification profiles can help determining whether predication [Mahl92] should be reverted back to branches.

Profile-directed optimizations can be deployed in different ways. First, software vendors can use profiles to compile and optimize their applications. We call this *vendor site PBO*. Second, a distributed application binary can be re-optimized at user site using profiles gathered from users' actual workload.

This approach has been used by Spike [Cohn98] and Morph [Zhan97]. We call this *use- site PBO*. Third, a binary can be optimized during its execution, using profiles collected at runtime. We call this *runtime PBO*. Runtime PBO has been used in dynamic optimizers such as Dynamo [Bala00] and CPO [Kist01].

Runtime PBO has several advantages over user -site and vendor -site PBO. First, at the same user site, there is often a mix of different microarchitectures of the same ISA (Instruction Set Architecture). For example, a site may have a mix of Pentium, Pentium Pro, Pentium III and Pentium IV based machines. An application binary can be re -optimized specifically for the machine during each run. If the user -site PBO is used, the optimizer may need to generate several different versions of the binary code for different micro-architectures and store them on disk for future invocations. Second, recent applications extensively use dynamically linked libraries. With dynamically linked libraries, it is difficult for a non -dynamic optimizer to conduct cross -library optimization because the library code is not visible until run time. Third, for some long running applications, it makes more sense to optimize while the program is running since the program may not exit for the off -line optimization to be performed. Fourth, dynamic optimization [Bala00, Gsch00, Ebci01, Kist01] uses a profile collected from the current execution, which is usually more representative than profiles collected from other runs.

Unfortunately, most of the profiling techniques proposed and used today are for intended for static PBO. In static PBO, a profile is collected for one or more *entire* program executions before it is used for the program optimization. Hence, it has a summarized view of the program. However, for runtime PBO, optimizations are deployed as soon as optimization opportunities have been identified using only a view of execution up to that point. Since some phases may come up late in the execution, and some observed phases might change over time, profiling must be continuous so that optimization can be applied adaptively to the changes. Furthermore, profiling and optimization are carried out in the same run, and trace generation and optimization time is considered part of the total execution time. Hence, the way profiling is carried out and used for dynamic optimization must be quite different. In this paper, we propose to take advantage of the phase changes observed during a program execution to minimize the overhead of hot trace generation and dynamic optimization. An *execution phase* is considered as a set of execution paths with its respective performance characteristics. A phase change means some paths in a phase have changed their branch directions or some paths have changed their performance behavior. Profiling is done continuously, and only when phase changes are detected, is the dynamic optimizer invoked to handle new hot traces. Many programs exhibit phased execution behavior. Since execution phases tend to repeat, by exploiting such phase locality, trace generation and optimization in dynamic optimization can be efficiently performed.

In this paper, we focus on runtime PBO and study the use of a hardware branch trace buffer feature [Intel00c] in identifying hot execution paths leading to performance bottlenecks. We exploit the phase locality existing in programs to drive dynamic optimization. When a phase change is detected, and when the confidence of a stable phase is established, the dynamic optimizer is invoked to improve the core set of traces in the stable phase. Our phase detection approach filters out unstable phases and phases that are not warranted for optimization. Combining hardware performance monitoring features and our software based phase change detection approach, we can reduce the profiling and optimization overhead.

The remainder of this paper is organized as follows. In section 2, we provide the background of profiling in dynamic optimization. In section 3, we describe the trace buffer and performance monitoring features in Itanium, as well as how we use these features to sample on frequently executed paths. Section 4 describes our phase change detection approach. Section 5 discusses various phase change detection methods and respective experiment results. The summary and conclusion are given in section 6.

2. Background

2.1 Profiling in Dynamic Optimizers

Profiling in previous dynamic optimization work has been performed using *interpretation* [Bala00] and *instrumentation* [Kist2000]. In Dynamo [Bala00], the original code is initially *interpreted*. After a branch target is interpreted a small number of times, it is considered hot and becomes a candidate to start a trace. A selected trace is optimized and stored in the code fragment cache. Subsequent execution of this trace will be redirected by the control system to the optimized code in the code fragment cache. Backpatching is used to link traces together so that a trace can branch directly to other traces without going through the control loop.

Profiling in Dynamo is conducted by having the *interpreter* maintain counters for backward branch and trace exit targets. The trace selection is very simple: the current execution path starting from the "hot" branch target is selected as the trace. Although this simple profiling and trace selection approach works reasonably well for several SPEC95int benchmarks, it does not work for more complex programs.

In order to minimize the time spent on *interpretation*, Dynamo uses a small threshold to determine if a code fragment is hot. Once a code fragment is selected and optimized, this code fragment no longer needs to be interpreted. Therefore, the small threshold allows frequently executed code to move into the code fragment cache sooner, minimizing interpretation overhead. From a profile gathering perspective, this approach is inherently weak. It does not distinguish important code from unimportant code (in programs where important code executes millions of times). Selecting unimportant traces will not only increase optimization overhead, but also take space in the code fragment cache. Its simple trace selection may form traces that are often exited before the complete the trace. These traces with early exit are more likely to decrease instruction locality in the code fragment cache. As shown in the Dynamo paper [Bala00] some programs suffer from the weak profiling approach. However; Dynamo avoids possible slowdowns by bailing-out to native execution when the number of optimization requests becomes high. Although "bail-out" avoids slow down, the need to bail-out also shows that the dynamic optimizer is not applicable to more complex programs.

In CPO [Kist2000], program *instrumentation* is used to collect profile information. To overcome the relatively high instrumentation overhead, dynamic optimizations in CPO must be substantial in order to break even or to achieve a speed up. One way to minimize the cost of instrumentation based profiling is to adopt *burst profiling* where the instrumented profiling code runs for a period of time and is later removed. The optimizer generated in the short burst is used to guide optimizations.

2.2 A Different Model for Dynamic Binary Optimization

In this paper, we focus on dynamic optimization which uses an input binary that is compatible with the host machine, the original code can run directly on the host machine rather than going through interpretation, as implemented in Dynamo. This significantly speeds up the execution of the un-optimized code. As a result, the dynamic optimizer does not need to optimize a large portion of the code, because the overhead of running un-optimized code is low. Our model for dynamic binary optimization does not interpret the original binary. This model includes four stages: profiling and phase detection, trace selection and formation, optimization, and binary patching. In the profiling and phase detection stage, execution paths are sampled while the input program is executing. A phase detector periodically checks the sampled execution paths and their performance characteristics to determine when a program's behavior is stable and the time consuming parts should be optimized. Once a set of traces is identified for optimization, the dynamic optimizer is invoked. The set of traces selected are optimized based on detected performance problems associated with the traces. After optimization, the optimized traces are stored in the code fragment cache. In the patching stage, the trace starting point is patched with a branch to redirect the execution to the improved trace in the code fragment cache.

In this dynamic optimization model, the original code does not need to be *interpreted*. Therefore, only the time critical traces with optimization opportunities are optimized. Furthermore, this model allows statistical sampling techniques to be used to identify program hot spots. DCPI [Ande97], Morph [Zhan97] and Spike [Cohn98] have demonstrated on Alpha machines that statistical sampling of a program counter can identify program hot spots with low profiling overhead. However, since Morph and Spike perform off-line optimizations, the time to process sampled PC information to identify traces is not a major concern. For dynamic optimization, the cost of trace analysis may be prohibitive since it will contribute to the execution time. Therefore, we use the branch trace buffer feature in IA-64 architecture to sample a set of branches in addition to a PC. This reduces the processing time to identify block boundaries for selecting and forming traces for optimization.

3. Trace Sampling and Hardware Performance Monitor Based Profiling

The Itanium processor defines and supports a set of performance monitoring features that can be used to characterize workload and to profile application execution [Intel00a, Intel00c]. The Itanium has four performance-counter registers that can be programmed to measure stall cycles in different categories as well as to count occurrences of over a hundred events. Furthermore, the Itanium supports Event Address Registers (EARs) for both Data and Instruction events as well as an eight-entry Branch Trace Buffer (BTB). The Instruction EAR can capture the addresses of instructions that trigger I-cache or ITLB misses. The Data EAR can capture the addresses of load instructions that cause D-cache or DTLB misses together with the target addresses of these loads. The Branch Trace Buffer is an 8-entry circular buffer that can capture information on the most recent branch instructions and their branch targets. These performance-monitoring features enable us to gain a detailed understanding of the dynamic execution behavior of the running application.

3.1 Branch Trace Buffer

The Branch Trace Buffer can be configured to record retired branch instructions depending on the direction of the branch (taken/not-taken/all) and the outcomes of various branch-prediction structures. For example, one could program the buffer to capture only the most recent taken-branch instructions. Each retired branch instruction that is recorded in the Branch Trace Buffer may take up to two entries, one entry for the address of the branch instruction and the other entry for the address of the branch target. A mispredict bit is also encoded in the entry for the address of the branch instruction to indicate the outcome of branch prediction. Hence, the Branch Trace Buffer is capable of capturing a dynamic trace consisting of the most recent four *taken* branch instructions when a sample is taken.

We take advantage of this capability to form our hot traces. Each time a sample is taken, a trace can be formed by following the last four *taken* branches recorded in the Branch Trace Buffer. Note that the trace thus formed can have more than four basic blocks because there could be multiple *fall-through* branches. In our experiments, a trace sampled often includes more than 8 basic blocks. In addition, the dynamic optimizer could stitch partially overlapping traces together to form longer traces. Obviously, there could be many other ways of forming hot traces using the Branch Trace Buffer. However, the evaluation of different trace generation techniques is beyond the scope of this paper.

3.2 Itanium Performance Tool (IPT)

A profiling tool that utilizes the performance monitoring features, called Itanium Profiling Tool (or IPT), has been developed on the Itanium running 64-bit Linux in the Microprocessor Research Laboratory (MRL) of Intel. IPT requires a customized performance monitor device driver (PMU driver) that runs as part of Linux. IPT supports various modes of profiling on a running application, including the counting of all performance events supported by the Itanium processors, and the collection of samples on various events. The IPT program interacts with the PMU driver to configure the performance monitoring registers and to receive profiling or sampling data from the PMU driver and store them in a profiling file.

4. Phase Detection

We define a *phase* to be a set of repeated execution paths in a program with a set of common performance characteristics. In programs, these repeated program behaviors often include cache misses, branch mispredictions, and other performance critical events. These behaviors repeat in predictable ways because most programs execute a set of code in a loop. Additionally, these loops are nested and therefore repeat behavior at different frequencies. In our study, we examine coarse-grained phases, sets of execution paths that repeat millions of times or more. We focus on this level of granularity because we are looking to perform software optimizations that predict behavior for several million executions of a set of code fragments. In this context, we define a “phase change” to be a change in the set of likely executed code or its behavior.

Phases are important to our system because we seek to optimize the performance characteristics of hot code throughout the program. For example, after detecting the “hottest” or most commonly executing code, the dynamic optimizer forms a set of optimized traces. When the previous “hot code” is no longer executing frequently, the optimizer needs to find and optimize a new set of traces. And after determining that the behavior of some optimized traces has changed, the optimizer may need to replace some traces with newly optimized traces.

4.1 Phase Granularity

We use the concept of a nested loop to explain *phase granularity*. At any given time, some aspect of loops may change. However, since a loop generally only has a limited number of paths and data to work with, some aspects of behavior in the loop stay constant. Larger loops contain these smaller stable loops and could exhibit some degree of higher-order stable behavior. Thus, the program behavior can be seen as repeating many different sets of behavior at different loop levels.

In general, program structures such as loops can maintain different levels of granularity. At a fine-grained level, we exploit behaviors that are likely to repeat in the next execution of a loop. At a coarse level of granularity, we exploit behaviors that are likely to repeat thousands, millions, or billions of iterations. At a program level of granularity, we exploit behavior that loop repeats over an entire program execution.

To illustrate this point, consider a loop that contains two paths, A and B. One program executes one code path one million times in a row, then another path one million times in a row. Another program that executes path A followed by path B followed by path A and repeats this ABAB pattern one million times. Finally, a third program executes path A with 50% probability and path B with 50% probability. At a fine level of granularity, the first example could be seen as two stable loop patterns, the second example could be seen as a single loop pattern, and the third example could be seen as an unpredictable pattern. The distinction is that in the first two examples, the result of the next loop iteration can be accurately predicted. In the last example, the result cannot.

At a coarser level of granularity, the first example could still be seen as two stable loop patterns, and the second example could still be seen as a single repeating pattern. However, the third example could be identified as a single stable pattern because, at a coarse level of granularity, the result of the next one million loop iterations can be predicted to be 50% A and 50% B.

In this study, we focus on coarse level phase locality. This is because profiling and optimization in our model are implemented in software, we need to carefully control the overhead involved in profiling and optimization. Hardware techniques such as trace cache [Rote96], hot spot detector [Mert00], and frame constructor [Pate01] have been proposed to directly capture hot traces. When such hardware mechanisms become available, we may be able to exploit phase locality in finer granularity.

4.2 Building Traces in a Phase

We approximate a phase as a list of branch traces leading up to certain events, and a count of how many times each trace leads to a type of event. We allow a single trace to start at any given branch instruction. This is done because a bra nch can only be patched to execute a single path. Branch traces are formed from sampled information from the Intel Profiling Tool (IPT).

To deal with the run -time overhead costs of sorting and analyzing sampled data at runtime, we need an efficient way t o encode traces and minimize trace search and other basic operations. For example, a sequence of branch and target addresses can be compressed in a number of ways. Trace records can also be compressed by storing only the differences between branch addresse s. These optimizations are important if a high sampling rate is used. However, these issues are beyond the scope of this paper.

4.3 Phase Changes

Using our metrics, a phase could be considered changed for a number of reasons:

1) One or more new traces are detected
2) The performance characteristics of the set of traces have changed
3) One trace has become dominant over another trace

In our model, a new phase is detected when a set of new traces have been established. As for the phase change, there are two typ es of changes to consider. First, the execution path may change its branch bias. For example, the execution takes path A-B-C-D frequently, but later it takes path A-B-C-E. In the second type, the performance behavior of an existing path has changed. For ex ample, path A -B-C-D originally leads to frequent data cache misses, but later incurs few misses.

To detect phase changes, we use a time -interval based approach. An interval is defined to be the time it takes to collect a constant number of samples. At the end of each periodic interval, we count how many unique branch traces exist by forming a hash table with chaining when collisions occur. We then use this information to update a similar global table which contains all unique sampled paths.

Since our in terval size is kept constant and different programs require an indeterminate number of samples to capture their behavior, we apply a system that tries to verify that we have correctly captured a working set. In each interval, a prediction of the next phas e is formed. This prediction is compared against the real outcome of next interval. If the prediction is good, the captured set of samples is frozen and tested against future intervals. If the predictions continue to perform well in subsequent intervals, the set of sampled traces are optimized and patched into the program. Otherwise, the program tries to capture a new set of samples and repeats the process.

Example:

Interval	Hot traces captured
1	{abcd, efdg, hhhh}
2	{abcd, efdg, hhhh}

3	{abef, efdg}
4	{ijkl, mmmm (cache miss) }
5	{ijkl, mmmm (no cache miss)}

In the above example, interval 1 has collected three hot traces, *abcd*, *efdg* and *hhhh*, where each letter represents a unique branch address. For example, *hhhh* represents a single repeating basic block in a loop. From interval 1 to interval 2, there are no detected changes. From interval 2 to interval 3, a change is detected, since a trace starting at branch address *a* has now changed its path direction. From interval 3 to interval 4, a new phase is detected since a set of new traces has been captured. From interval 4 to interval 5, there is also a phase change because the loop *mmmm* has changed its data cache behavior.

After a set of sampled traces is found and patched in, the program continues to test the se t of traces in the current interval. If they result in a poor prediction, the program begins forming a new set of sampled traces. If a new set of sampled traces is found and the previous set of traces continues to perform poorly, the new traces are patc hed in, replacing formerly patched branches but keeping any non -conflicting previously patched branches. In this study, we do not limit the number of traces to be selected. However, our experiment results in section 5 show that a very large set of traces is rarely necessary.

4.4 Traces Selection

Dynamically optimized traces are executed by patching original code to execute dynamically generated code. The limitation of this approach is that each location in the original code can only be redirected to one dynamically generated trace, and therefore we need a way to select one trace to optimize over the other. In our experiments, we have studied the following four different models for trace selection:

1) First Interval – In the first interval, a number of s amples are taken and the hottest branch path is selected for each unique path starting branch address. The target of the branch instruction is never changed for the rest of execution. This is similar to a profiling model that collects profiles only in the first few seconds of program execution. For simple programs with one main loop, the first few seconds of profiling is often sufficient to capture all important traces.

2) First Hot – Similar to the first interval approach, once a branch instruction is ta rgeted towards a specific trace, it is not changed. However, new branch paths can be selected in each later interval, after a constant number of samples. At the end of each interval, the most commonly taken branch paths are selected if they have not been previously selected. This model adds new execution paths from new phases. However, it does not consider phase changes. Once a trace is selected, even if later execution shows the branch direction has changed, the already selected trace will remain in the code cache. This approach is similar to Dynamo's where phase changes are not considered.

3) Dynamic Interval – Similar to the First Hot, the most commonly taken branch paths from every sample interval are selected for optimization. However, branches may be redirected to new branch paths in each sampling interval. This model includes phase changes, modifying selected traces in the code cache with new paths collected in a new interval.

4) Full History – Like in the Dynamic Interval, branch paths are selec ted in every interval and they could be selected in later intervals. However, all samples since the beginning of program execution, not just the samples in the previous interval, are kept and used to select which branch path to take. This approach may be impractical to implement due to the resulting amount of sample information collected. However, it is used as a reference point to understand the usefulness of the entire history of traces.

5. Experiments and Discussions

We gathered results by sampling t he performance registers on an IA -64 beta machine running TurboLinux64. The benchmark used in this study is the Spec2000int [Henn00] suite. They were compiled by the Intel C/C++ compiler, and optimized at O2 level. Unmodified Spec2000int binaries were run by the IPT tool that set the performance registers to throw an interrupt at specific events. For example, every one million clock cycles or ten thousand I or D -cache misses, the performance registers throw an interrupt and the kernel records the current branch trace registers into a reserved page of memory. A user daemon writes the data in this page of memory to disk for offline processing.

5.1 Frequent Path Sampling

Table 1 shows the basic execution path distribution information using branch trace bu ffer sampling. The first row shows the total number of paths sampled from the execution of a complete run using the reference input set. The sampling rate was 1 million cycles per sample. The second row shows the number of paths starting with unique branch addresses, and the third row shows the total number of unique paths (may have the same starting branch address). The fourth row indicates the number of unique paths that account for 50% of all the samples, and the fifth row indicates the number of paths that account for 90% of all the samples.

Table 1. Unique Path Distribution (1M cpu cycles/sample)

	Gzip	Bzip2	Gap	Mcf	Parser	Twolf	Vpr	Vortex	Gcc
Total #	8867	14367	77251	123941	136994	91248	32767	34561	35018
Unique start	119	176	767	112	1111	251	129	1178	2773
Unique path	312	861	2280	316	3675	943	426	2779	4791
# path 50%	5	7	11	1	17	18	11	32	16
# path 90%	33	133	297	6	317	107	57	735	1852

As shown in Table 1, a relatively small number of frequently executed paths represent most of the samples. These hot execution paths are clearly the optimization candidates. It is worth noting that there is one single path in MCF that accounts for 50% of all the samples. It is the inner while loop in procedure "refresh_potential". Due to intensive data cache misses occur in the while loop, this single execution path takes nearly 50% of the total execution time. It may seem suspicious that 16 paths would account for 50% of the sampled paths for Gcc since it is well know that Gcc has a rather flat execution profile. It turns out that in this Spec2000version Gcc spends much time in memcpy and memset. Two traces from these two library routines account for almost 30% of the execution time. Overall, a small number of traces can cover nearly half of the execution time. Thi s is desirable for dynamic optimization because we try to minimize the number of traces selected for optimization.

Figure 1 shows the distribution of the top 5 execution paths leading to D -cache misses for the benchmark programs. To optimize for the data cache miss events, we do not need many paths. A small number of execution paths can cover a majority of the data cache miss events. Figure 2 shows the distribution of the top 10 time-consuming paths in each benchmark. While some programs need only a small set of traces to reach a high coverage, some other programs such as Crafty, Perl and Eon need more traces to reasonably cover the execution time.

Figure 1. Distribution of Top 5 execution paths leading to D-cache misses

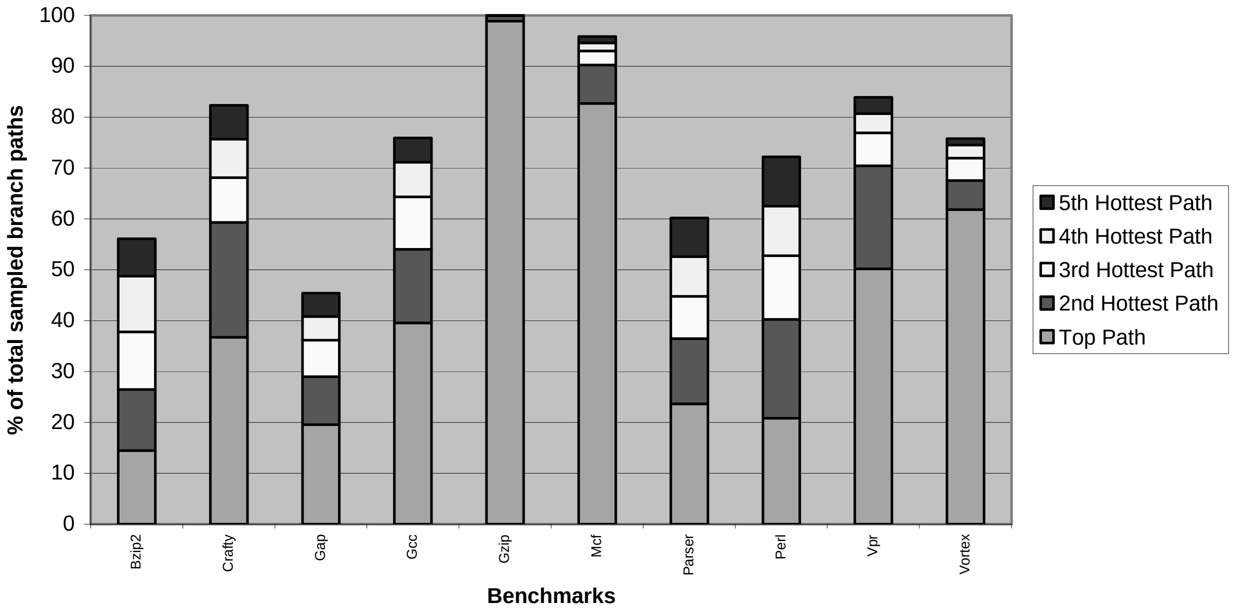

Figure 2. Distribution of Top 10 Execution Paths (1M cycles/sample)

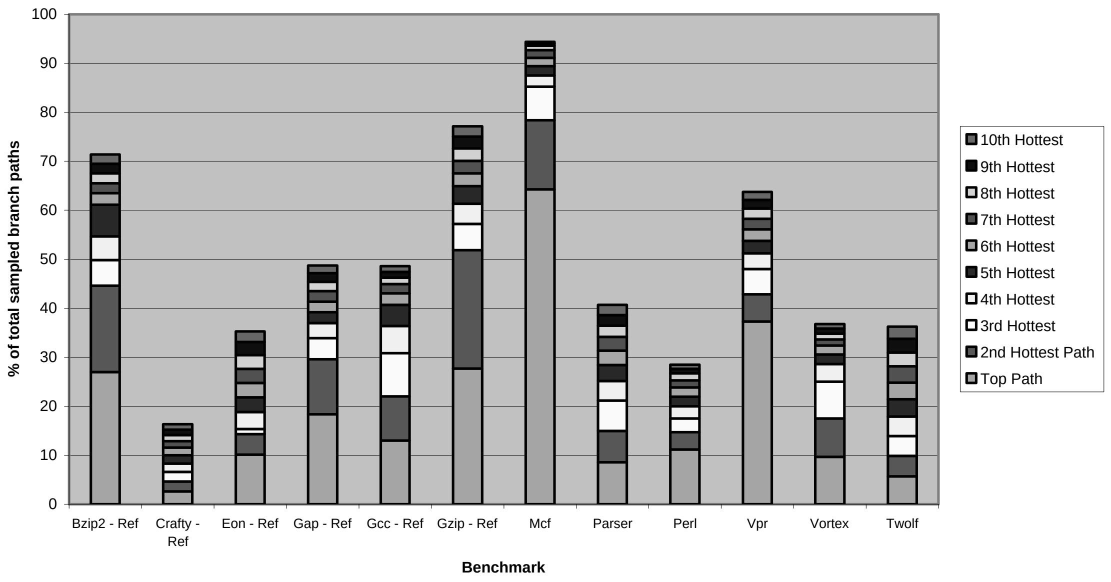

Table 1, and Figure 2 have shown that a small number of execution paths we sampled by using the branch trace buffer may cover a large fraction of the execution time. Such execution paths are good candidates for a static compiler to study. We gain confidence in our dynamic optimization model in which we attempt to optimize only a relatively small fraction of the code and leave the majority of the code running in native mode on the host machine. Our approach will be effective if we can capture a small set of traces covering a large fraction of the execution time.

5.2 Impact of Sampling Rate

In Table 1, the sampling rate is 1M cpu cycles per sample. Table 2 compares two programs with very different sampling rates.

Table 2. Impact of Fast Sampling Rate

	Gzip (1M/sample)	Gzip (10K/sample)	Gcc (1M/sample)	Gcc (10K/sample)
Total #	8867	942276	35018	355825
Unique start	119	312	2773	6366
Unique path	312	2001	4791	15743
# path 50%	5	5	16	16
# path 90%	33	33	1852	1893

When a much faster sample rate was used (1 million cycles per sample vs. 10K cycles per samp le), there were more unique paths sampled from Gzip and Gcc. For example, the number of unique paths sampled at the slow rate was 312 for Gzip and 4791 for Gcc. The numbers increased to 2001 for Gzip and 15743 for Gcc when the sampling rate increased by 1 00 times. However, the faster sampling rate generally resulted in the same "hot" execution paths for the two programs. For example, the number of paths accounting for 50% of samples is 5 for Gzip and 16 for Gcc, for both sampling rates. It is worth noting that the actual sampled paths are the same in both cases. Since we are only interested in the very hot paths in dynamic optimization, we will not use different sampling rates in the remaining part of this section.

5.3 Impact of Trace Selection in a Phase

Table 3. First Interval vs. First Hot at 1000 samples per interval

Program	Bzip	Parser	CC1	Twolf	Crafty	Vpr	Gap	Gzip	Vortex	Mcf
1st Interval	40.30%	74.30%	57.30%	59.00%	49.50%	77.40%	26.50%	39.10%	30.20%	18.00%
1st Hot	55.80%	74.30%	63.30%	61.10%	52.20%	77.40%	58.40%	40.60%	60.90%	92.00%

In Table 3, we compare the path coverage between two trace selection methods used in phase detection. The numbers shown in Table 3 are coverage of total sampled paths. For example, if we select the hottest paths fro m the first interval of 1000 samples, and keep this set of paths in the code fragment cache throughout the entire execution, this set of paths may cover 74% of execution time for Parser, and 18% of time for MCF. Table 3 shows that although the first 1000 samples are able to capture a relatively high percentage of program execution in some cases (such as Parser and Vpr), the first 1000 samples capture a poor percentage in others, such as MCF. In MCF, the first few seconds of program execution do not encounter the most common execution branch paths in the program. There have been proposals suggesting a dynamic optimizer can profile for the first few seconds of program execution, and use the profile collected during that time to guide optimization for the ent ire run. Table 3 shows evidence that this scheme will not work effectively for some programs. The First Hot model works better than First Interval on Bzip, CC1, Gap, Vortex and MCF. For programs where the First Hot outperforms First Interval, there must b e some important execution paths that do not show up in the first few seconds of execution. This set of results confirms that detecting new phases is important in the profiling for dynamic optimization.

Table 4. First Hot vs. Dynamic Interval

Interval size: 100 samples		Bzip	Parser	CC1	Twolf	Crafty	Vpr
	1st Hot	71.87%	82.11%	63.50%	56.90%	54.82%	75.69%
	Dyn Int	73.10%	84.03%	72.90%	62.90%	58.55%	75.79%

	Full Hist	74.20%	83.20%	73.00%	63.08%	61.79%	76.08%
Interval size: 1000	1st Hot	65.84%	74.85%	63.30%	61.10%	56.08%	79.83%
	Dyn Int	66.07%	74.85%	65.50%	61.60%	56.79%	79.83%
	Full Hist	66.07%	74.85%	67.53%	62.02%	57.76%	79.83%
Interval size: 100		Gap	Gzip	Vortex	Mcf	Avg.	
	1st Hot	60.30%	23.00%	66.20%	94.57%	64.90%	
	Dyn Int	74.80%	81.90%	71.20%	94.45%	74.96%	
	Full Hist	76.02%	82.75%	71.69%	94.72%	75.65%	
Interval size: 1000	1st Hot	58.40%	40.60%	60.80%	93.34%	65.41%	
	Dyn Int	69.10%	70.30%	61.20%	93.33%	69.86%	
	Full Hist	66.26%	75.55%	65.91%	93.35%	70.91%	

Table 4 compares the First Hot and the Dynamic Interval schemes. Out of all the programs, Gzip benefits the most from the ability to retarget branches to other branch paths (the dynamic interval covers 81.9% and 70% for interval size 100 samples and 1000 samples, respectively, while the first hot can cover only 23% and 40% respectively). As Gzip builds a larger dictionary table to match expressions against, the program spends more time searching the dictionary table and less time adding to it. This gradual phase change is captured by the dynamic interval and full history schemes, leading to better results. Other than Gzip, CC1, Twolf, Gap and Vortex also benefit from the ability to adapt traces to phase changes. This set of results shows evidence of changing branch bias behavior.

In addition, we compare the performance of each trace selection model in two interval sizes -- 100 and 1000 samples per interval. The advantage of a small interval size is that traces can be added to the working set sooner and therefore the program may benefit from optimi zed traces earlier. Therefore, the coverage is usually higher for the smaller interval size. However, a small interval size may detect more phase changes, requiring more optimizations and higher optimization overhead. If optimization overhead is a major co ncern, we should favor using larger interval size since it detects phase changes at a larger granularity and thus deploys optimization more conservatively.

Table 4 has shown that our interval based phase detection approach, using dynamic interval trace selection can collect a set of traces for the dynamic optimizer to cover 60% to 90% of the execution time. When the interval size is smaller, the traces are selected and optimized sooner and the program execution has more time to benefit from optimization. However, small interval sizes is more sensitive to short term behavior changes and triggers more phase changes and incurring higher optimization overhead.

5.4 Execution Paths Leading to Data Cache Misses

Table 5. Execution Paths Leading to D-Cache Misses with the Dynamic Interval trace selection

		Mcf	Parser	Gap	Vpr	Vortex
Interval size: 100	Coverage	95%	88%	78%	94%	94%
	Dyn Opt call	7	4	19	1	3
Interval size: 1000	Coverage	88%	86%	69%	84%	88%
	Dyn Opt call	2	2	5	1	1

Table 5 compares coverage of D -Cache Misses for selected programs with significant D -Cache activity and the number of "working set" changes. A significant percentage of D -Cache misses have stable branch paths leading up to them. For example, when the interval size is 1000 samples, our phase detection approach will detect a new phase for MCF (this is when "refresh_potential" is first invoked), and make a call to the dynamic optimizer to select new traces for cache optimization. Although the dynamic optimizer only needs to be invoked twic e, the selected traces for cache optimization covered

88% of sampled data cache miss events. Table 5 also shows that using a small interval size may result in detecting more phase changes with the result that the dynamic optimizer is called more often.

5.5 Impact of Interval Size

Table 6. Coverage and Re-optimization at different Interval Sizes

Interval size		cc1	parser	gap	Vpr	vortex
Interval size : 100 samples	L1 I-cache	63%	66%	74%	77%	59%
	Re-opt	91	8	93	2	225
	L1 D-cache	82%	88%	78%	94%	94%
	Re-opt	2	4	19	1	3
	CPU	63%	84%	75%	76%	71%
	Re-opt	62	5	35	2	11
Interval size: 1000 samples	L1 I-cache	59%	51%	61%	58%	57%
	Re-opt	4	1	8	1	20
	L1 D-cache	--	86%	69%	84%	88%
	Re-opt	--	2	5	1	1
	CPU	63%	74%	63%	77%	61%
	Re-opt	2	1	6	1	1

Table 6 shows the number of times a new set of branch paths is selected to replace an existing set for 100 and 1000 samples per interval. This is also the time to invoke the dynamic optimizer to improve the set of traces selected. A smaller interval size leads to more branch path set replacements and optimizer invocations. Execution paths leading to D -cache misses usually have more stable behavior. They tend to have higher coverage (from 69% to 94%), and require fewer invocations of the dynamic optimizer to optimize a new set of t races. Execution paths leading to I -cache misses have less stable behavior. Their coverage is low, especially for a program with a large instruction footprint, such as Gcc and Vortex. This may be an indication that dynamic instruction prefetching optimizat ion is difficult because of the relatively unstable phase behavior.

6. Summary and Conclusion

Efficient profiling for dynamic optimization is a major challenge. Unlike the static PBO, profiling in dynamic optimizations must deal with new execution phases and phase changes. Furthermore, since profiling and optimization contribute to the execution time of the running program, their overhead must be managed carefully. Current profiling approaches used in prototype dynamic optimizers have either very high ove rhead, or generate poor quality profiles. We propose a new dynamic optimization model with an execution phase based profiling approach. This approach uses statistical sampling of the branch trace buffer and performance monitoring events supported in the It anium processor to minimize the cost of phase detection and trace identification.

Using Spec2000int as the benchmark, we have shown 1) a small number of hot traces cover about half of the executed paths; 2) the number of traces covering 90% of execution p aths is also relatively small; 3) it is critical to detect new execution phases since in some programs important execution phases start rather late in the execution; 4) it is important to detect phase changes and adapt the traces selected for optimization since several programs exhibit phase change behavior.

We have also shown that our interval based phase detection approach can capture traces with a reasonably high coverage (on average, 70%) of the execution. In particular, our approach focuses on stable phases so that transient phases and relatively infrequent traces will be filtered out without invoking the optimizer.

Small intervals allow the profiler to identify phases sooner so that hot traces can be optimized and store in the code cache earlier. This would improve the coverage of the traces. However, we may need to pay the price of a higher processing and optimization overhead.

We have experimented with this phase detection profiling approach on the Itanium processor, using its branch trace buffer and performance monitoring features. On machines without support for branch traces, we may use traditional PC sampling [Zhan97, Ande97] or exploit other branch hardware features [Cont94] to reduce the cost of trace identification. By exploiting existing p hase locality in programs, our phase detection based profiling can efficiently capture hot traces for optimization. With this profiling approach, dynamic optimization may become more viable in the future.

7. References

[Henn00] Henning, John L., “SPEC CPU2000: Measuring CPU Performance in the New Millennium,” IEEE Computer, Vol. 33, No. 7, July 2000.
[Ande97] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger and William E. Weihl. “Continuous profiling: where have all the cycles gone?” ACM Transaction on Computer Systems, vol. 15, no. 4 , Nov. 1997, pp. 357-390.
[Ayer98] Ayers Andrew, Stuart De Jong, John Peyton, and Richard Schooler "Scal able Cross-Module Optimization", In Proceedings of the ACM SIGPLAN '98 conference on Programming language design and implementation, PLDI'98, June, 1998
[Bala00] Vasanth Bala, Evelyn Duesterwald, Sanjeev Banerjia. “Dynamo: A Transparent Dynamic Optimization System”, In Proceedings of the ACM SIGPLAN '2000 conference on Programming language design and implementation, PLDI'2000, June, 2000
[Ball96] Ball, T., and Larus, J. R. “Efficient Path Profiling,” In Proceedings of the 29th Annual International Symposium on Microarchitecture (Micro-29), Paris, 1996
[Chan91] P. Chang, S. Mahlke and W. Hwu, "Using Profile Information to Assist Classic Compiler Code Optimizations," Software Practice and Experience, Dec., 1991
[Cohn98] Robert S. Cohn, David W. Goodwin, P. Geo ffrey Lowney, “Optimizing Alpha Executables on Windows NT with Spike”, Digital Technical Journal, Vo l 9 No 4, June, 1998
[Cont94] Thomas Conte, Burzin Patel, J Cox. “Using Branch Handling Hardware to Support Profile -Driven Optimization”, In Proceedings o f the 27th Annual International Symposium on Microarchitecture (Micro-27), 1994
[Conn97] Daniel Conners, “Memory Profiling for Directing Data Speculative Optimization and Scheduling”, Master Thesis, EE Department, University of Illinois, Urbana Champion, 1997
[Ebci01] K. Ebcioglu and E. Altman, M. Gschwind, S. Sathaye, " Dynamic Binary Translation and Optimization” IEEE Transaction on Computers, vol. 50, no. 6, June 2001.
[Fish92] Fisher J. A., and S. Freudenberger, “Predicting Conditional Branch Direct ions From Previous Runs of a Program,” Proceedings of the 5 th International Conference on Architectural Support for Programming Languages and Operating Systems, Oct., 1992
[Grah82] Graham, S. L., Pb.B. Kessler, M.K. McKusick, “gprof: A Call Graph Execution Profiler”, Proceedings of the SIGPLAN ’82 Symposium on Compiler Construction, 1982.
[Gsch00] M. Gschwind, E. Altman, S. Sathaye, P. Ledak, and D. Appenzeller, “Dynamic and Transparent Binary Translation”, IEEE Computer, vol. 33, no. 3, March, 2000.
[Gwen99] Linley Gwennap, “Intel’s Itanium and IA-64: Technology and Market Forecast,” Micro Design Resources, Technical Library Report, 1999, http://www.mdronline.com/tech_lib/IA64/index.html
[Holl96] A. M. Holler, “Optimization for a Superscalar Out -of-Order Machine,” In Proceedings of the 29th Annual International Symposium on Microarchitecture (Micro-29), December, 1996

[Intel00a] Intel, "Intel IA -64" Architecture Software Developer's Manual, Volume 1: IA -64 Application Architecture
[Intel00b] Intel, "Intel I A-64" Architecture Software Developer's Manual, Volume 2: IA -64 System Architecture
[Intel00c] Intel, "Intel IA -64" Architecture Software Developer's Manual, Volume 4: Itanium Processor Programmer's Guide
[Kist01] Thomas Kisteler, Michael Franz. "Continuo us Program Optimization: Design and Evaluation", IEEE Transaction on Computers, vol. 50, no. 6, June 2001.
[Mahl92] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective Compiler Support for Predicated Execution Using the Hyperblock", In Proceedings of the 25th Annual International Symposium on Microarchitectures. (Micro-25), 1992
[Mert00] Matthew Merten, Andrew Trick, Erik M. Nystrom, Ronald D. Barnes, Wen -Mei Hwu, "A Hardware Mechanism for Dynamic Extraction and Relayout of P rogram Hot Spots", In Proceedings, International Symposium on Computer Architecture, ISCA-27, 2000
[Pate01] Sanjay Patel, S. S. Lumetta, "replay: A Hardware Framework for Dynamic Optimization", IEEE Transaction on Computers, vol. 50, no. 6, June 2001.
[Rote96] E. Rotenberg, S. Bennett, and J. Smith, "Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching," In Proceedings of the 29th Annual International Symposium on Microarchitectures. (Micro-29), 1996
[Sun00] Sun Microsystems, "MAJC -5200: A High Performance Microprocessor for Multimedia Computing", white paper, http://www.sun.com/microelectronics/MAJC/5200wp.html.
[Zhan97] Xiaolan Zhang, Zheng Wang, Nicholas Gloy, J. Bradley Chen and Michael D. Smith "System support for automatic profi ling and optimization", In Proceedings of the sixteenth ACM symposium on Operating systems principles, 1997, pp. 15-26.