Software Profiling for Hot Path Prediction: Less is More

Evelyn Duesterwald
duester@hpl.hp.com

Vasanth Bala
vas@hpl.hp.com

Hewlett-Packard Labs
1 Main Street
Cambridge, MA 02142

ABSTRACT

Recently, there has been a growing interest in exploiting profile information in adaptive systems such as just-in-time compilers, dynamic optimizers and, binary translators. In this paper, we show that sophisticated software profiling schemes that provide highly accurate information in an offline setting are ill-suited for these dynamic code generation systems. We experimentally demonstrate that hot path predictions must be made early in order to control the rising cost of missed opportunity that result from the prediction delay. We also show that existing sophisticated path profiling schemes, if used in an online setting, offer no prediction advantages over simpler schemes that exhibit much lower runtime overheads.

Based on these observation we developed a new low-overhead software profiling scheme for hot path prediction. Using an abstract metric we compare our scheme to path profile based prediction and show that our scheme achieves comparable prediction quality. In our second set of experiments we include runtime overhead and evaluate the performance of our scheme in a realistic application: Dynamo, a dynamic optimization system. The results show that our prediction scheme clearly outperforms path profile based prediction and thus confirm that *less* profiling as exhibited in our scheme will actually lead to *more* effective hot path prediction.

1. INTRODUCTION

Program profiles are collected to identify where in the code a program spends its time. This information may be fed to a profile-based optimization system [8], may be used in performance tuning or to aid in program understanding. Traditionally, program profiling is performed *offline*, that is, program profiles are collected in a separate preparatory run of the program and the information is then consumed afterwards, for example, during a re-compilation of the program.

Recently, there has been a growing interest in exploiting profile information in dynamic compilation systems such as just-in-time compilers [7], dynamic optimizers [3,4] and binary translators [17,11]. Profile information is used to focus the costly runtime operations on only the hot portions of the program. Unlike the traditional approach of profiling, these new dynamic systems require profile information to be collected and consumed *online*, that is, within the same run.

Online profiling is a fundamentally different challenge. An obvious difference is the much increased need for efficiency. Clearly, efficiency is a desirable feature in any offline profiling system. However, in a dynamic compilation system, low-overhead profiling is a core requirement for the system to be of any use. Another more subtle difference is that offline profiles are *summaries* of program behavior while online profiles are *predictions*. Online profiling is typically not concerned with establishing precise relative frequency variations between the profiled units. Instead, online profiling is a prediction problem, such as the prediction of hot paths (or hot branches, hot edges, hot call sites, etc.). We focus in this paper on hot path prediction but most of the discussed material applies equally well to the prediction of other hot program units such as branches, basic block or call sites.

Various hardware schemes have long established that effective predictions can be made based on small collected profile histories. Numerous techniques have been developed that rely on effective hardware branch prediction to improve instruction bandwidth [9,12,15] including techniques that perform a limited form of path profiling to trace branch correlation [19,14]. However, these hardware prediction schemes are usually not architecturally visible and thus not available to user software. Even if hardware prediction schemes are accessible by user software [13], they may not be of much use in systems like a just-in-time compiler. A just-in-time compiler needs profile information about the virtual branches of the input source program. However, the branches that are exposed to the hardware for profiling are the executing branches of the just-in-time compiler code, not of the input source program Thus, there may be mismatch between the information collected by the hardware and the information needed by the dynamic compilation system.

A more general solution to the path prediction problem can be provided by a software scheme. A software prediction scheme can collect frequency information through instrumentation [5], or emulation [10]. In cases where a mapping between the executing addresses and the profiling units of interest can be established, frequency information may also be collected by sampling [21] or via hardware counters [1].

A straightforward approach to implement an online prediction scheme in software is to adapt an existing offline path profiling scheme [5] by only profiling each path up to a certain prediction threshold. When the execution frequency of a path exceeds the prediction threshold, the path is predicted as hot.

Existing path profiling schemes provide accurate path frequency information in an offline setting. But their offline quality does not necessarily make them a superior technique for online path prediction. Clearly, when considering lower prediction threshold values the prediction accuracy will drop and the prediction will be increasingly speculative. But in spite of this increased degree of speculation, there are several factors that demand a low prediction threshold. First, the runtime overhead of collecting profile information makes very high threshold values prohibitively expensive. But more importantly, there is the more hidden cost of missed opportunity that results from the prediction delay. The longer the program execution is profiled, the later will predictions be made and, consequently, the lower will be the reuse (i.e., potential benefit) of the predictions. We show in our experiments that prediction delay is a significant factor in evaluating the quality of a prediction scheme. If predictions are delayed for too long, the overall benefits of the prediction can easily vanish.

Thus, in practice the prediction delay and profiling duration have to be kept short. However, with limited profiling, sophisticated path profiling schemes can no longer offer any prediction advantages over simpler schemes that exhibit much lower runtime overheads. Thus, while intuition may call for longer and more elaborate profiling, we will show in fact show that the opposite it true: *less* profiling actually leads to *more* effective predictions.

In this paper we present a new low-overhead profiling scheme for hot path prediction that exploits the above observations. Our scheme significantly reduces the runtime and space overhead of path profile based prediction while maintaining the same prediction quality. The key idea is to focus the profiling effort on only the potential starting points of hot paths. Once a path starting point has become hot a prediction is made by speculatively selecting the *Next Executing Tail (NET)* as the hot path. The NET scheme was developed and implemented as part of the Dynamo dynamic optimization system [6,7]. Dynamo accelerates native program binaries at runtime and heavily relies on effective hot path prediction.

We followed two approaches to evaluate the NET path prediction scheme. We first developed abstract metrics to assess the quality of a prediction scheme independently of its implementation overhead. The *abstract benefit* of a prediction is measured by the *hit rate* or reuse of the predicted paths. The *abstract cost* of a prediction is the amount of *noise*, i.e., the number of cold paths, inadvertently included in the prediction. Based on these metrics our first evaluation is independent of both (1) the specific way the path predictions are exploited in a compilation system (concrete benefits) and (2) the specifics of the implementation of the profiling scheme (concrete cost).

Using the abstract metrics we show that NET prediction achieves the same prediction quality as path profile based prediction at practically relevant threshold values. Importantly, NET prediction uses 60% less counter space and significantly reduces the runtime profiling overhead.

We also provide a second concrete evaluation of the NET prediction scheme by demonstrating its performance in a realistic application using the Dynamo dynamic optimization system [.3,4]. We implemented both NET and path profile based prediction in Dynamo. Our experiments demonstrate that NET prediction is considerably more effective in practice. Due to the high profiling overhead, running Dynamo with path profile based prediction was ineffective and could not reproduce the speedups we achieved when using the NET scheme.

This paper demonstrates that it is possible to use a software scheme to deliver effective hot path predictions with very low profiling overhead. Our experiments also indicate that it is imperative to recognize program hot spots early. The missed opportunity cost that otherwise results may render the prediction useless. An important implication of these results is that dynamic optimization systems may in fact not benefit much from sophisticated hardware mechanisms that allow for prolonged monitoring of the program execution. An efficient and easy to implement software solution like our NET scheme appears to be sufficient for the needs of these dynamic systems.

The next section provides the background in path profiling. Section 3 defines the problem of hot path prediction and Section 4 presents the details of our NET prediction scheme. Our experimental evaluation based on abstract metrics is presented in Section 5, and Section 6 presents the experiments with Dynamo. Section 7 discusses issues with phase changes. Related work is discussed in Section 8 and the paper concludes in Section 9.

2. Computing Path Profiles

Path profiling views the execution trace of a program as a sequence of finite program paths. By considering paths of limited size the number of possible program paths is bounded but it may be exponential in the size of the program. A path profile determines a frequency distribution over the set of program paths and assigns each path p a frequency $freq(p)$ that describes how many times p was executed. The execution flow represented by a path profile is given as:

$$Flow = \{ freq(p) \mid p \text{ a path} \}.$$

Several path profiling algorithms and corresponding definitions of program paths have been developed. Ball and Larus [5] defined *intraprocedural acyclic forward paths*, where a forward path ends at a backward branch or procedure return. Their path profiling algorithm requires a preparatory static analysis of the program to establish a minimal path encoding. Using a spanning tree algorithm a minimal low-cost set of edges is selected for instrumentation. The instrumented edges produce the necessary information at runtime to establish the unique path number of each executing path. The algorithm only considers intraprocedural paths.

Bit tracing is another approach to computing path profiles. A path is identified by the following *path signature*:

<start_address>.<history>,<indirect_branch_target_list>

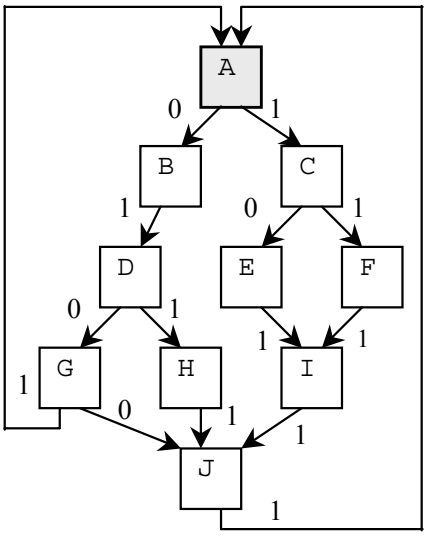

Figure 1: Multiple paths through a loop.

consisting of a start address followed by a history of branch outcomes and a list of indirect branch targets. The history associates a 1-bit value with the outcome of every branch on the path indicating whether the branch was taken or not. For each indirect branch the target address is appended to the indirect branch target list. Figure 1 shows examples of program paths and corresponding path signatures. Path signatures are constructed as the program executes by shifting a 1 or 0 value into the current signature register. Upon reaching the end of a path the current path signature is used as an index into a path table to update the execution count for the corresponding path.

Bit tracing can produce less compact path representations than Ball and Larus' algorithm. However, its advantage is that path signatures can be constructed on the fly and no preparatory static analysis phase is needed.

Young and Smith [20] presented a different program path definition as *k-bounded general paths*. A *k*-bounded general path is an intraprocedural program path whose length (i.e., number of branches) is bounded by *k*. Unlike a Ball and Larus path, general paths are not limited to forward paths and may include backward edges. General paths are computed at runtime using a *k*-size FIFO queue to store the most recently executed *k* branches. Young and Smith use a lazy algorithm that allows for fast updates of program path counters each time a new branch is entered into the FIFO queue.

3. Hot Path Prediction

The goal of hot path prediction is to predict what will be one of the most frequently executing paths based on a limited amount of execution history. Since the hot execution flow in a program must involve cycles, hot path prediction is primarily concerned with the prediction of hot paths through loops. Paths that start at targets of backward branches, as in Ball and Larus' definition, naturally capture loop iterations. Thus, we use Ball and Larus' paths as a basis for our path definition but extend it to the interprocedural case:

An interprocedural forward path starts at the target of a backward taken branch and extends up the next backward taken branch. The path may extend across procedure call or return statements unless the call or return is a backward branch. If a path includes a (forward) procedure call it will terminate at the corresponding return branch, if not earlier. Note that this path definition captures recursive loops without unfolding the recursion. Also, a path may include any number of indirect (forward) branches.

Recall that *freq(p)* denotes the total execution frequency for path *p*. For a set *P* of paths we define the *flow* of *P* as:

$$freq(P) = \{ freq(p) \mid p \in P \}.$$

A path *p* is a *hot path* if *freq*(*p*) is greater than some *hot threshold h*. The set of hot paths with respect to *h* is defined as:

$$HotPath_h = \{ p \mid freq(p) > h \}.$$

The *hot flow* represented by $HotPath_h$ is $freq(HotPath_h)$.

The goal of hot path prediction is to determine a set of paths *P* that best predicts $HotPath_h$. We determine how well a set of paths captures $HotPath_h$ by computing its *hit rate*. The hit rate is determined by crediting each path *p* in *P* that is also contained in $HotPath_h$ the appropriate portion of *freq*(*p*) that still remains after the prediction has been made. Thus, the hit rate takes into account the missed opportunity cost that results from the prediction delay. Hit rate is the online analog to the coverage metric that has been used in evaluating offline path profiles [6].

Assume a path *p* is predicted after it has executed τ times. τ is called the *prediction delay* and the flow captured by this prediction is: $freq(p) - \tau$. We define the flow captured by a set *P* of paths as:

$$Hits(P) = freq(P \cap HotPath_h) - |(P \cap HotPath_h)| \times \tau .$$

The hit rate for *P* now results as:

$$HitRate(P) = (Hits(P) / freq(HotPath_h)) \times 100.$$

The missed opportunity cost that results from the prediction delay is given as:

$$MOC(P) = |(P \cap HotPath_h)| \times \tau$$

As our experiments in Section 5 show, longer profiling intervals (i.e., a longer prediction delay τ) will generally not lead to better path predictions due to the sharp increase in missed opportunity cost.

If hit rate was the only measure of prediction quality making optimal path predictions would be trivial: simply predict every path when it first executes. Thus, we need an additional criterion to penalize for the amount of misprediction. *Noise* measures the percentage of cold flow that was inadvertently included in *P*:

$$Noise(P) = freq(P - HotPath_h) - |(P - HotPath_h)| \times \tau .$$

The corresponding noise rates for *P* now results as:

$$NoiseRate(P) = (Noise(P) / freq(HotPath_h)) \times 100.$$

The ideal path prediction scheme maximizes the hit rate while minimizing noise. Note that the two goals are conflicting. To minimize noise, predictions should be delayed so as to rule out as many cold paths as possible. Delaying predictions raises missed opportunity cost and thus reduces the hit rate.

4. Online Prediction Schemes

We can derive a prediction scheme from a given offline path profiling scheme in a straightforward way. Simply apply the path profiling scheme up to a *prediction delay* τ. As soon as the execution frequency of a path exceeds τ, the path is predicted as hot. We refer to such a prediction scheme as *path profile based prediction*. Path profile based prediction relates to path profiling as hardware branch prediction schemes relate to offline branch profiling.

Obviously, the length of the prediction delay is critical for the prediction to be effective. The shorter the prediction delay the more noise will be included in the prediction. However, prolonging the prediction delay will increase the missed opportunity cost and thus lower the benefits of the prediction. Missed opportunity cost is not the only expense to consider when prolonging the prediction delay. There is also the runtime overhead of profiling each path execution up to τ times. The runtime overhead consists of the amount of counter space needed and the cost of profiling operations.

Path profile based prediction incurs a very high runtime overhead. The amount of counter space needed is equal to the number of dynamic paths, which can be exponential in the size of the program. Runtime profiling operations include a counter update after every path execution and further profiling operations to trace the execution of branches. In the case of bit tracing, every branch execution requires the shifting of a bit into the current history register. If Ball and Larus' spanning tree algorithm [5] is used, the number of branches that require profiling operations can be minimized but still remains in the order of the total number of branches.

4.1 NET Hot Path Prediction

The runtime overhead of path profile based prediction combined with missed opportunity cost make long prediction delays impractical. However, the shorter the prediction delay the more speculative the prediction will be. The question arises as to whether we need the full support for path profiling in order to make what ultimately will be an inherently speculative prediction.

To answer this question we developed a new path prediction scheme: *Next Executing Tail (NET*) prediction[1]. The objective of NET prediction is to significantly reduce profiling overhead while still performing as well as path profile based prediction when considering prediction delays that are practically relevant. Lower priority was given to how well the scheme compares to path profile based prediction if infinitely long prediction delays are allowed, i.e., in an offline setting.

In NET, a path is divided into *a path head*, i.e., the path starting point, and the *path tail*, which is the remainder of the path following the starting point. For example, in path "ABDG" in Figure 1, block "A" is the path head and "BDG" is the path tail. NET reduces profiling cost by using speculation to predict path tails while maintaining full profiling support to predict hot path heads. The rationale behind this scheme is that a hot path head indicates that the program is currently executing in a hot region and the next executing path is likely to be part of that region.

[1] The scheme was called *Most Recently Executing Tail (MRET)* prediction in an earlier publication [4].

Execution counts are maintained only for executed path heads, that is, only for targets of backward taken branches. No further profiling is performed. For example, in Figure 1 only one profiling count is maintained for the entire loop at the single path head at the start of block A. Once the counter at block A has exceeded its threshold, the next executing path is predicted. Assume the loop has one or two dominant paths. In such a case, NET is statistically likely to predict the correct hot path. On the other hand, if there are no dominant paths through the loop and execution is divided fairly evenly among all five paths, NET may select any path. However, there is not a better prediction to be made in such a case, even under a path profile based prediction scheme.

4.2 Implementing NET

An important advantage of the NET prediction scheme is the ease of engineering it. Since profiling is limited to the potential trace heads, NET requires even less profiling than block or branch profiling schemes. If the execution frequency at a path head exceeds the prediction threshold dictated by the predication delay τ, the path head is considered hot and the next executing path can be collected using incremental instrumentation. With incremental instrumentation the profiler collects the next path by subsequently collecting each non-branching sequence in that path. For instance, during each step, the profiler can place a breakpoint at the end of the next non-branching code sequence. Executing the code sequence will raise the breakpoint and the profiler handles the breakpoint by removing it and preparing the next step, i.e., placing the next breakpoint. This process continues until the end of the path has been encountered and the complete path has been collected. Alternatively, if profiling is implemented inside an emulator, such as in a binary translator, the NET path can directly be collected during emulation.

To measure the overhead of NET prediction we consider the amount of counter space needed and the number of dynamic instrumentation points. The NET prediction scheme requires maintaining execution counts at every target of backwards branches. The number of targets of backwards branches is only a fraction of the number of branches and bound by $|B|$, where B is the set of basic blocks in the program's control flow graph. Path profiling in comparison requires one count per path which may be $2^{|B|}$ in the worst case.

5. Experimental Evaluation

This section evaluates the performance of the NET prediction scheme using the metrics from Section 3. We experimentally compare the hit and noise rates for NET prediction with the corresponding rates for path profile based prediction. We first focus on assessing the quality of the predictions and ignore the implementation and overhead characteristics of each scheme. Clearly, for the overall effectiveness of a prediction scheme, implementation characteristics and runtime overhead are crucial. We consider the actual runtime overhead of NET prediction and path profile based prediction in the second set of experiments when evaluating the two schemes in a realistic application in Section 6.

Table 1 shows our benchmark set that includes the SpecInt95 benchmarks and one C++ benchmark called *deltablue*, which is an incremental constraint solver [16]. For each program the total number of paths and the total flow is shown in Table 1.

Our experiments were run with the objective of predicting the *HotPath* set based on a hot threshold of 0.1%. The execution frequency of each path in this set is at least 0.1% of the total flow. Table 1 shows the number of paths in the 0.1% *HotPath* sets and the percentage of the total flow captured by them (%Flow). The captured flow is determines as: (*freq(HotPath) / Flow)× 100.*

Table 1 shows that the amount of flow captured by the *HotPath* set can vary significantly. In compress, the 0.1% *Hot Flow* set captures almost the entire execution flow (99.6%). In gcc, where the total number of paths is much higher and hot paths not as dominant, the 0.1% *HotPath* set captures less than 50% of the total flow.

Table 1. Benchmark set

		Flow	0.1 % *HotPath*	
Benchmark	#Paths	(M)	#Paths	% Flow
compress	230	3061	45	99.6
Gcc	36,738	2191	137	47.5
Go	29,629	1214	172	55.5
Ijpeg	62,125	635	74	93.3
Li	1,391	3985	111	93.8
m88ksim	1,426	2014	107	92.5
Perl	2,776	1514	146	88.5
Vortex	5,825	3016	95	85.8
deltablue	505	1799	28	93.9

5.1 Hit Rate and Noise

To collect the hit and noise rates we ran path profile based prediction and NET prediction with various prediction delays ranging from 10 to 1,000,000. For each run we divided the total flow into *profiled flow* and *predicted flo*w. Profiled flow is the amount of flow consumed by the prediction delay. Profiled flow contains cold flow and the portion of the hot flow that was missed during the prediction delay. The predicted flow contains the captured hot flow (hit rate) and noise. Clearly, with a prediction delay of 0 we obtain 0% profiled flow (100% predicted flow) and with an infinitely long prediction delay we obtained 100% profiled flow (0% predicted flow).

We measured the hit and noise rates that result given a certain allowance of profiled flow. The corresponding hit rate/profiled flow data is shown in Figure 2 and the analogous noise rate/profiled flow data in Figure 3.

Figure 2 compares the hit rate achieved by path profile based prediction (a-b) with the hit rate of NET prediction (c-d). The graphs (a) and (c) on the left depict the hit rate over the entire range of prediction delays. To provide a more detailed view of a more practically relevant range, the graphs (b) and (d) on the right zoom into the upper left 10% corner shown shaded in the figures on the left.

There is virtually no difference in the coverage data for path profile based and NET prediction. Comparing Figures 2 (b) and (d) shows that at 10% profiled flow both path profile based and NET prediction reach a hit rate of about 97.5 on average. Figures 2 (a) and (c) show that the hit rate decreases quickly with the amount of flow that is being profiled. Thus, prolonging the profiling interval will lead to increasingly less effective predictions due to the rising cost of missed opportunity. This trend is particularly pronounced in programs with dominant hot paths like compress, which has the fastest descending hit rate. Here, delaying predictions is costly since opportunity losses rise sharply. Other programs like go and gcc exercise a large number of cold paths. In these cases delaying predictions is not as lossy since comparatively less hot flow will be missed and the hit rate therefore descends much slower.

Figure 3 depicts the analogous data for noise. The top two graphs show noise rates for path profile based prediction and the bottom two graphs show the noise rates for NET prediction. The noise rate decreases more rapidly than the hit rate.

Comparing figures (a) and (c) on the left indicates that with longer prediction delays (i.e., 20%-70% profiled flow) NET prediction produces more noise than path profile based prediction. With path profile based prediction noise is reduced to less than 10% when profiling about 35% percent of the execution. In comparison, NET prediction needs to profile about 45% of the execution in order to reduce the noise rate to less than 10%. These results reflect that path profile based prediction gains accuracy advantages over the more speculative NET scheme when considering long prediction delays. However, Figure 3 indicates that any accuracy advantages of path profile based prediction arise only for prediction delays that are irrelevant for practical purposes. Profiling 20% or more of the total execution assumes prediction delays of 50,000-100,000 and results in significant missed opportunity cost as illustrated by Figure 2.

The graphs (b) and (d) on the right show the noise rate for the practically more relevant prediction delays. With shorter prediction delays NET prediction performs as well as path profile based prediction if not slightly better on average. When profiling 10% of the execution, NET prediction yields about 56% noise, whereas path profile based prediction results in about 65% noise.

Table 2. Number of paths and unique path heads

Benchmark	**#Paths**	**#Unique Path Heads**
compress	230	143
gcc	36,738	8,873
go	29,629	1,813
ijpeg	62,125	669
li	1,391	710
m88ksim	1,426	651
perl	2,776	1,053
vortex	5,825	3,414
deltablue	505	268

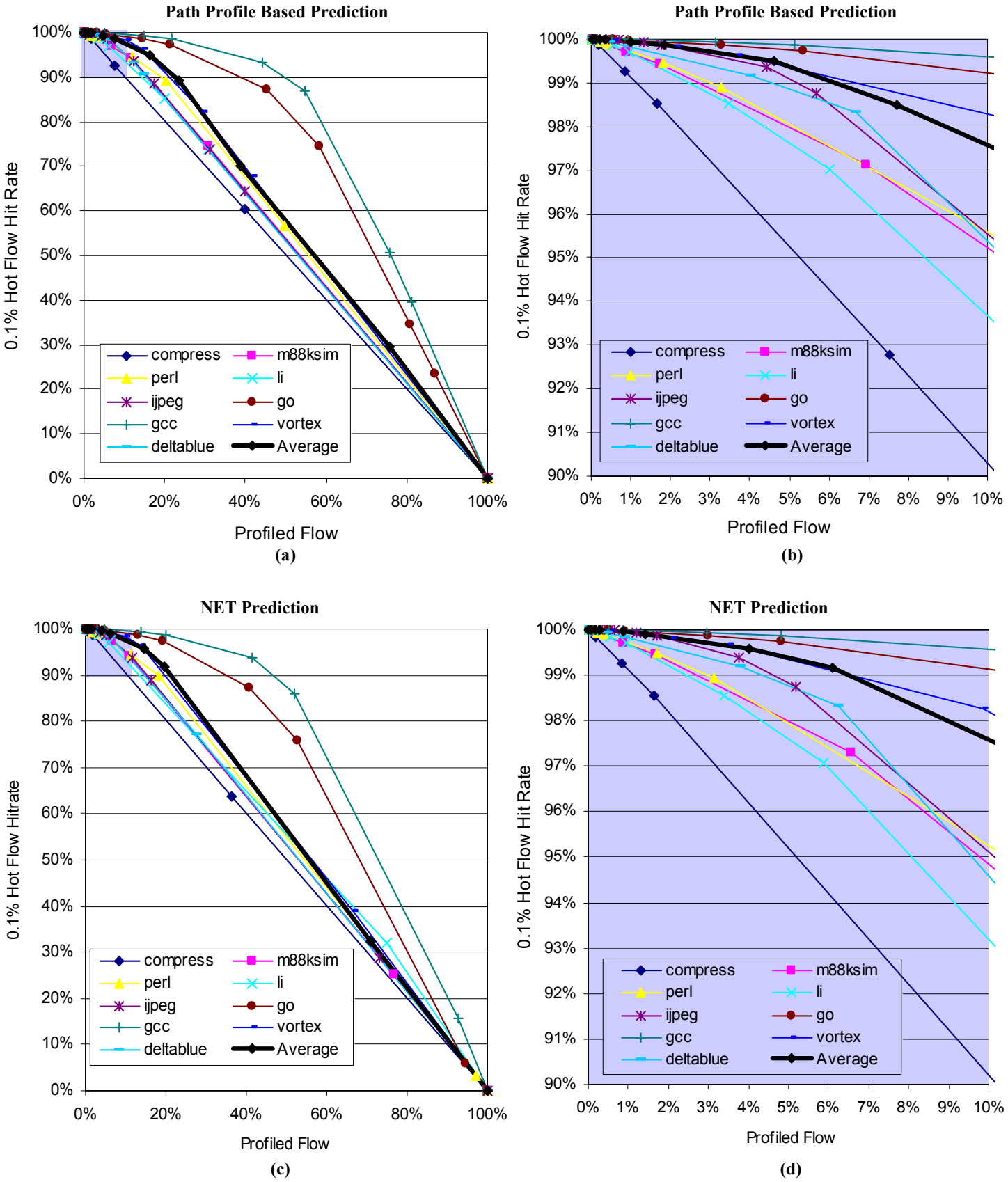

Figure 2: Hit rates for path profile based prediction (a-b) and NET prediction (c-d). The X-axis shows the percentage of the total flow that is profiled. The Y-axis shows the hit rate as the percentage of the 0.1 % Hot Path Set. The graphs (b) and (d) on the right zoom into the upper left 10% square shown shaded in graphs (a) and (c), respectively.

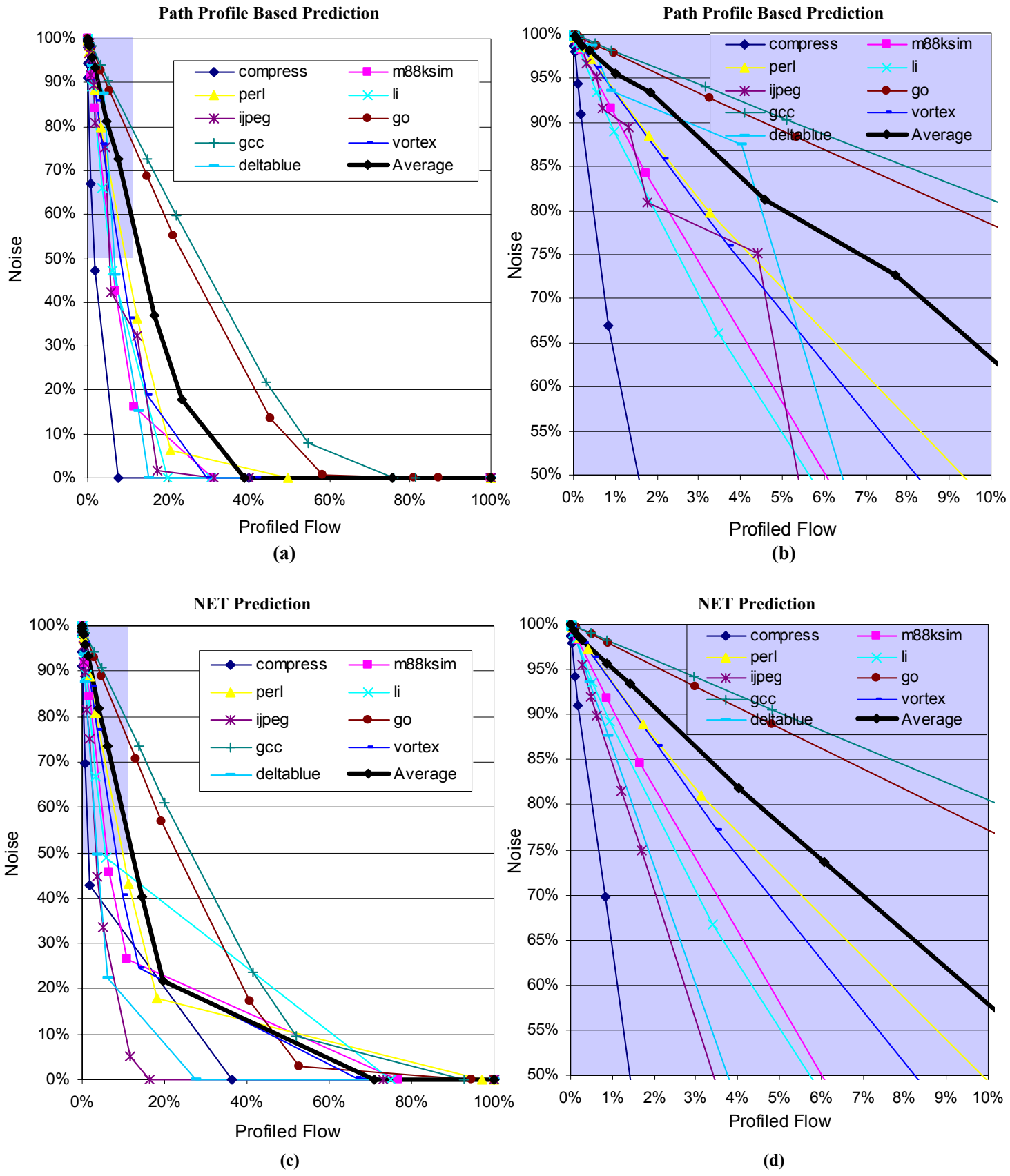

Figure 3: Noise rates for path profile based prediction (a-b) and NET prediction (c-d). The X-axis shows the percentage of the total flow that is profiled. The Y-axis shows the noise rates as the percentage of the 0.1 % Hot Path Set. The graphs (b) and (d) on the right zoom into the upper left rectangle shown shaded in graphs (a) and (c), respectively.

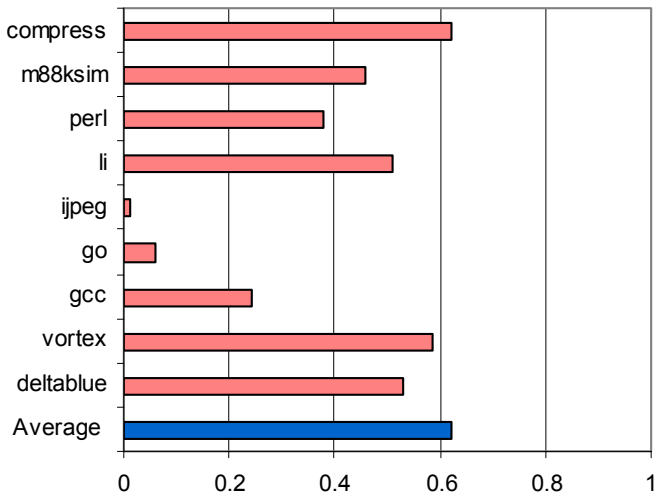

Figure 4: The amount of counter space used in NET prediction normalized to the counter space used in path profile based prediction.

5.2 Space Consumption

We also compared NET and path profile based prediction with respect to their space utilization. Specifically, we measured the amount of counter space needed by the two schemes. Recall that NET prediction requires only a single counter for each target of a backward taken branch whereas path profile based prediction requires a counter for each dynamic path. Table 2 shows the number of backwards taken branch targets (i.e., the number of unique path heads) along with the number of dynamic paths. Figure 4 depicts the corresponding reduction in the amount of counter space of NET prediction over path profile based prediction. The average bar shows that NET uses only about 60% of the counter space used in path profile based prediction.

6. Application: Path Prediction in Dynamo

The NET scheme was developed as part of the Dynamo project at HP Laboratories [3,4]. Dynamo is a dynamic optimization system that is capable of accelerating the performance of a native program binary at runtime. Dynamo operates by identifying and extracting the dynamically hot paths in the executing program binary. Dynamo achieves a performance boost by optimizing and laying out the programs' hot paths in a software code cache.

Initially, Dynamo observes the program behavior through emulation. During emulation profiling information is collected to identify hot paths for optimization. Copies of the hot paths are processed using lightweight optimization techniques and emitted into the code cache. Subsequent execution of these paths causes the cached optimized versions of the paths to be executed. Over time, the optimized version of the program's working set materializes inside the software code cache resulting in a performance boost. With Dynamo the performance of many SPECint95 binaries compiled with static optimization can further be accelerated [4]. Dynamo is written entirely in user level software and runs on a PA-RISC machine under the HPUX operating system.

Dynamo relies heavily on effective hot path prediction. A performance boost results only if the predicted paths are indeed hot so that their optimized version will be re-used sufficiently often to amortize the overhead of optimization. Driven by the need for the highest possible prediction accuracy we initially implemented a path profile based prediction scheme in Dynamo. However, path profile based prediction turned out to be ill-suited for Dynamo's purposes due to the significant runtime and space overhead. With path profile based prediction Dynamo could only achieve speedups in rare cases. A redesign of the path prediction logic resulted in the development of the NET scheme.

Figure 5 depicts Dynamo's performance with path profile based prediction and with NET prediction. Note that Dynamo cannot produce speedups in programs with excessively high numbers of

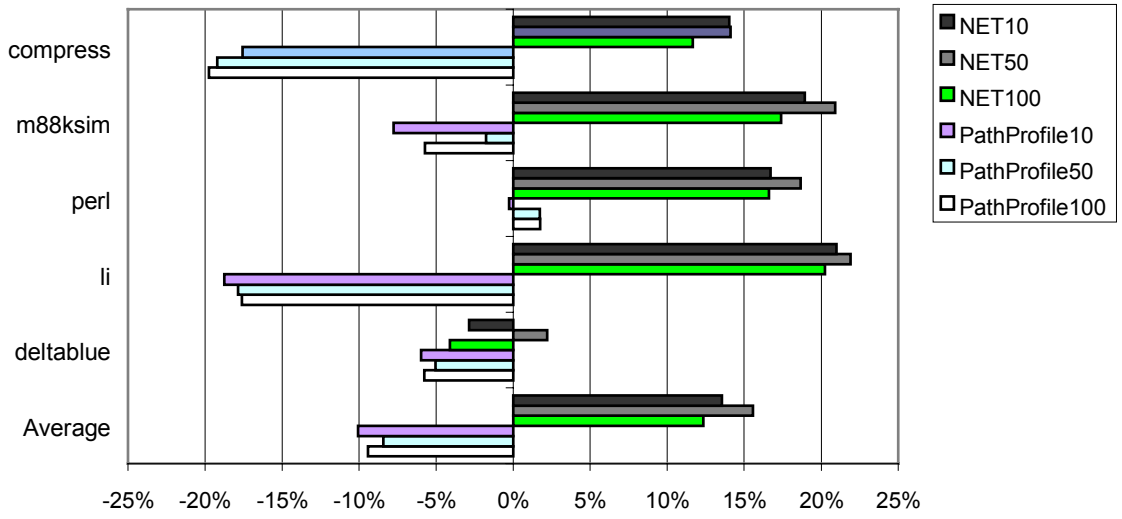

Figure 5: Dynamo speedup over native execution with path profile based and NET hot path prediction schemes. Each scheme is run with prediction delays of 10, 50 and 100.

dynamic paths and no dominant reuse, such as go and gcc. When run with these programs Dynamo gives up and bails out to native execution [3]. Figure 5 considers the programs from our benchmark set that are processed by Dynamo without bail-out.

We ran each prediction scheme with prediction delays of 10, 50 and 100. A prediction delay of 50 was for both schemes the most beneficial choice in balancing the amount of noise that results at lower thresholds and the rising profiling overhead and missed opportunity cost of longer prediction delays. Speedups progressively declined with prediction delays longer than 100. With a prediction delay of 50, less than 1% of the total execution flow is profiled.

Figure 5 demonstrates the performance advantages of NET prediction over path profile based prediction. Path profile based prediction could only produce speedups in perl and deltablue, whereas NET prediction lead to speedups in all these programs, averaging over 15%. The NET prediction scheme does not only offer performance advantages. Its simple design and ease of engineering make it an ideal candidate for a runtime system such as Dynamo.

6.1 Sensitivity to Phase Changes

So far we made the implicit assumption that hot paths are predicted for the entire run of a program. The metrics of hit rate and noise were developed with respect to accumulated profile information. Accumulated profile data is not time-sensitive and cannot provide information about phase changes that may occur during the run of the program. A path may be a hot path in a particular phase of the execution but may not have a high accumulated execution frequency.

Phase changes are implicitly recognized by path prediction schemes in the form of new paths' frequency counts exceeding their prediction delay. We can adjust a prediction scheme to more or less sensitivity to phase changes by changing the prediction delay. The longer the prediction delay the less sensitive the prediction scheme is to phase changes since short lived phases cannot be recognized. However, higher sensitivity to phase changes also introduces an increased noise problem: noise that results from paths that were formerly hot but have turned cold in the current phase. This kind of phase-induced noise cannot simply be controlled by prolonging the prediction delay. The prediction delay must be kept short to recognize phase transition in the first place. Thus, additional mechanisms for controlling phase-induced noise are needed, such as garbage collection or other path retiring schemes [13].

Dynamo addresses phase-induced noise by using a heuristic flushing scheme. Dynamo monitors the path prediction activity in order to identify sudden and sharp increases in the prediction rate. Such increases provide a good indication that a new phase is about to be entered. After detecting a phase transition, Dynamo triggers a cache flush and thereby removes all phase-induced noise from the cache. By scheduling cache flushes approximately at the time of a phase change we can keep the amount of useful paths that are inadvertently removed by the flush to a minimum.

We plan to extend our path metrics to model path removal from the prediction set. With a path removal model we obtain an abstract measure to evaluate how well a prediction scheme reacts to phase changes and how well it handles phase-induced noise.

7. Related Work

Hot path prediction has been used in the Boa binary translation system [17]. In Boa, hot groups are formed by collating individual paths based on collected branch frequencies. Profile information is collected during Boa's interpretation phase. When a hot group entry has been found, a path is selected by following the most likely successors according to the collected branch profile information. Unlike our NET scheme, Boa's prediction scheme requires every branch to be profiled. Furthermore, constructing paths from isolated branch frequencies ignores branch correlation, which may lead to paths that, as a whole, never execute.

Several hardware schemes have been developed that perform a limited amount of path profiling by capturing branch correlation through branch histories [9,12]. The trace cache uses hardware to build traces from predicted branch sequences [15]. These schemes, developed to improve instruction fetch bandwidth are generally not accessible by user software and can therefore not be exploited in a dynamic compiler. An exception is the profiling hardware described in [13] that was specifically developed to support runtime optimization. The hardware monitors branch execution and identifies hot spots from collected branch profiles. A hot spot is defined as a collection of frequently executing blocks. The hardware also includes mechanisms to detect when execution strays from previously recorded hot spot information, which can be used for path retiring. It has not yet been tested how well the hardware supplied information can be absorbed by a dynamic optimizer. Since the hardware's definition of a hot spot might differ form the one used in the dynamic optimization software, an additional software layer may be necessary to transform the hardware provided hot spot information into a form that can be utilized by the software. However, such a software layer may not always exist. For instance, when the hardware's notion of a branch does not match software's notion of a "virtual' branch as for example in a just-in-time compiler, the hardware supplied information may not be of much use.

Offline profiling techniques benefit from hardware support in the form of hardware counters [1] and in continuous profiling environments through PC sampling [2]. A software approach to reducing profiling cost has been pursued in ephemeral instrumentation. The idea is to enable intermittent profiling by providing mechanisms for rapid insertion and deletion of instrumentation code [18].

Our evaluation of the NET prediction scheme has shown that producing good path predictions does not require a sophisticated path profiling scheme. A similar result regarding path profiling has also been obtained for the offline case. A study comparing edge and path profiles reports that collecting edge profiles provides sufficient information to compute a large percentage of the hot portion of the corresponding path profile [6].

8. Conclusion

This paper presented a new software profiling scheme for predicting hot program paths. Our NET scheme delivers prediction quality that is comparable to a path profile based scheme at only a fraction of the cost. Besides its performance advantages the simple design and ease of engineering make the NET prediction scheme an ideal candidate for use in dynamic compilations systems. We developed an abstract metric to

measure the quality of our prediction scheme independently of a particular implementation. In addition we also evaluated the NET prediction performance in a realistic dynamic optimization system. Both evaluations demonstrate the efficiency and effectiveness of the NET prediction scheme.

Our evaluation of hit rate and noise in path prediction sets shows that it is imperative for hot path predictions to be made early and based on only small amounts of profiling history. Intuition may suggest that if profiling is free, longer profiling intervals will always lead to better hot path predictions. However, this projection does not account for missed opportunity cost. Missed opportunity cost rises continuously with longer prediction delays and thereby progressively lowers the effectiveness of the hot path prediction. These observations suggest that dynamic compilation systems may in fact not benefit much from sophisticated hardware for the purpose of online profiling; a low overhead software solution such as the NET scheme may well be sufficient for the needs of these systems.

In the future, we plan to study the phase change problem further. Unlike accumulated offline profiling schemes, an online prediction scheme naturally reacts to phase changes. However, it is not clear at what granularity sensitivity to phase changes is most beneficial. We are working in extending our hit rate and noise metrics to model predictions in the presence phased program behavior.

Acknowledgements

We would like to thank Mike Smith for numerous discussions and for his helpful comments on this paper.

References

[1] Ammons, G., Ball, T., and Larus, J.R. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proc. of the 1997 Conf. on Programming Language Design and Implementation*, June 1997.

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

[3] Bala, V., Duesterwald, E., and Banerjia, S. Transparent dynamic optimization: The design and implementation of Dynamo. *Hewlett Packard Laboratories Technical Report HPL-1999-78. June 1999.*

[4] Bala, V., Duesterwald, E., and Banerjia, S. Dynamo: A transparent runtime optimization system. In *Proc. of the 2000 Conf. on Programming Language Design and Implementation*. Vancouver, B.C., June 2000.

[5] Ball, T. and Larus, J.R. Efficient path profiling. In *Proc. of the 29th Int. Symp. on Microarchitecture,* Paris. 1996.

[6] Ball, T., Mataga, P. and Sagiv, M. Edge profiling versus path profiling: The showdown. In *Proc. of the 25th Symp. on Principles of Programming Languages*, San Diego, CA, January 1998.

[7] Burke, M., Choi, J.-D., Fink, S., Grove, D., Hind, M., Sarkar, V., Serrano, M..J., Sreedhar, V.C., Srinivasa, H.. The Jalapeno Dynamic Optimizing Compiler for Java. In *Proc. of the 1999 ACM Java Grande Conference*, San Francisco, CA. June 1999.

[8] Chang, P., Mahlke, S.A., and Hwu, W.M. Using profile information to assist classic code optimization. *Software - Practice and Experience*, Vol. 21, No. 12, December 1991.

[9] Calder, B. and Grunwald, D. Fast and accurate instruction fetch and branch prediction. In *Proc. of the 21st Int. Symp. on Computer Architecture*. April 1994.

[10] Cmelik, R.F. and Keppel, D. Shade: a fast instruction set simulator for execution profiling. *Technical Report UWCSE-93-06-06, Dept. Comp. Science and Engineering, Univ. Washington*. 1993.

[11] Ebcioglu, K., Altman E., Sathaye, S., and Gschwind, M. Execution-based scheduling for VLIW architectures. In *Proc. of Europar'99*, Lecture Notes in Computer Science 1685, Springer-Verlag 1999.

[12] McFarling, S., and Hennesy, J. Reducing the cost of branches. In *Proc. of the 13th Int. Symp. on Computer Architecture*. 1986.

[13] Merten, C.M., Trick, A., George, C.N., Gyllenhaal, J.C., and Hwu, W.-M.W. A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. In *Proc. of the 26th Int. Symp. on Computer Architecture.* Atlanta, Georgia. 1999,

[14] Pan, S, So, K., and Rahmeh, J. Improving the accuracy of dynamic branch prediction using branch correlation. In *Proc. of the 5th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*. 1992.

[15] Rotenberg, E., Bennett, S., and Smith, J.E. Trace cache: a low latency approach to high bandwidth instruction fetching. In *Proc. of the 29th Int. Symp. on Microarchitecture,* Paris. 1996.

[16] Sannella, M., Maloney, J., Freeman-Benson, B., and Borning, A. Multi-way versus one-way constraints in user interfaces: experiences with the DeltaBlue algorithm. *Software – Practice and Experience 23*, 5 (May). 529-566. 1993.

[17] Sathaye, S., Ledak, P., LeBlanc, J., Kosonocky, S., Gschwind, M., Fritts, J., Filan, Z., Bright, A., Appenzeller, D., Altman, E., and Agricola, C. BOA: Targeting multi-gigahertz with binary translation. In *Proc. of the 1999 Workshop on Binary Translation*, Newport Beach, CA., October 1999.

[18] Smith, M. Private communication, March 2000.

[19] Yeh, T. and Patt, Y. A comparison of dynamic branch predictors that use two levels of branch history. In *Proc. of the 20th Int. Symp. on Computer Architecture.* 1993.

[20] Young, C. and Smith, M. Static correlated branch prediction. *ACM Transactions on Programming Languages and Systems*, Vol. 21, No. 5, September 1999.

[21] Zhang, X. et al. System support for automatic profiling and optimization. In *Proc. of the 16th ACM Symposium on Operating Systems Principles*, St. Malo, France. Oct. 1997.