A Programmable Hardware Path Profiler

Kapil Vaswani Matthew J. Thazhuthaveetil Y. N. Srikant

Department of Computer Science and Automation
Indian Institute of Science, Bangalore
{*kapil, mjt, srikant*}*@csa.iisc.ernet.in*

Abstract

For aggressive path-based program optimizations to be profitable in cost-sensitive environments, accurate path profiles must be available at low overheads. In this paper, we propose a low-overhead, non-intrusive hardware path profiling scheme that can be programmed to detect several types of paths including acyclic, intra-procedural paths, paths for a Whole Program Path and extended paths. The profiler consists of a path stack, which detects paths and generates a sequence of path descriptors using branch information from the processor pipeline, and a hot path table that collects a profile of hot paths for later use by a program optimizer. With assistance from the processor's event detection logic, our profiler can track a host of architectural metrics along paths, enabling context-sensitive performance monitoring and bottleneck analysis. We illustrate the utility of our scheme by associating paths with a power metric that estimates power consumption in the cache hierarchy caused by instructions along the path. Experiments using programs from the SPEC CPU2000 benchmark suite show that our path profiler, occupying 7KB of hardware real-estate, collects accurate path profiles (average overlap of 88% with a perfect profile) at negligible execution time overheads (0.6% on average).

1. Introduction

The design of several aggressive compiler and architectural optimizations is driven by the well-known principle of making the common case faster. Here, the common case generally refers to frequently executed or *hot* regions of code and repeating patterns in the program's control flow. Focusing on hot regions allows an optimizer to minimize the costs and side-effects of an optimization without sacrificing much of its benefits.

In its bid to identify hot regions, the optimizer is typically assisted by a *program profile*. Several types of program profiles have been proposed, each catering to the demands of different profile-guided optimizations. These profiles can be classified as *point* profiles or *path* profiles. Point profiles, such as basic block, edge [4] and call-graph profiles [1], record the execution frequency of specific points in the program. In contrast, path profiles capture control flow in more detail by tracking paths or sequences of instructions through the program. Point profiles find extensive use in several profile-driven compiler optimizations because of the ease with which they can be collected and analyzed. However, the task of profiling program paths, in general, is much more complex and imposes significant space and time overheads. To overcome this limitation, paths have been characterized in different ways [5, 26, 18], each characterization involving a trade-off between the amount of control flow information encoded in the profile and the overheads of profiling. For instance, the *acyclic, intra-procedural path* [5] is defined as a sequence of basic blocks within a procedure that does not include any loop back-edges. Importantly, it has been shown that acyclic, intra-procedural path profiles can be obtained with relatively low complexity and execution time overheads [5, 30]. These efficient profiling techniques have given rise to a new class of aggressive path-based optimizations [2, 11]. Moreover, the use of path profiles over point profiles has also proven to be beneficial in classical profile-driven optimizations [31].

However, existing path profiling techniques suffer from the following limitations.

- They incur significant space and time overheads due to large number of paths they track, precluding their use in cost-sensitive online systems.
- Overheads increase manifold if the scope of profiling is extended beyond acyclic, intra-procedural paths.
- They do not facilitate profiling execution in system calls or dynamically linked libraries.
- Precisely associating informative architectural events, such as the number of cache misses or branch mispredictions, with paths is a non-trivial task [1].

This paper proposes a programmable, non-intrusive hardware-based path detection and profiling scheme that

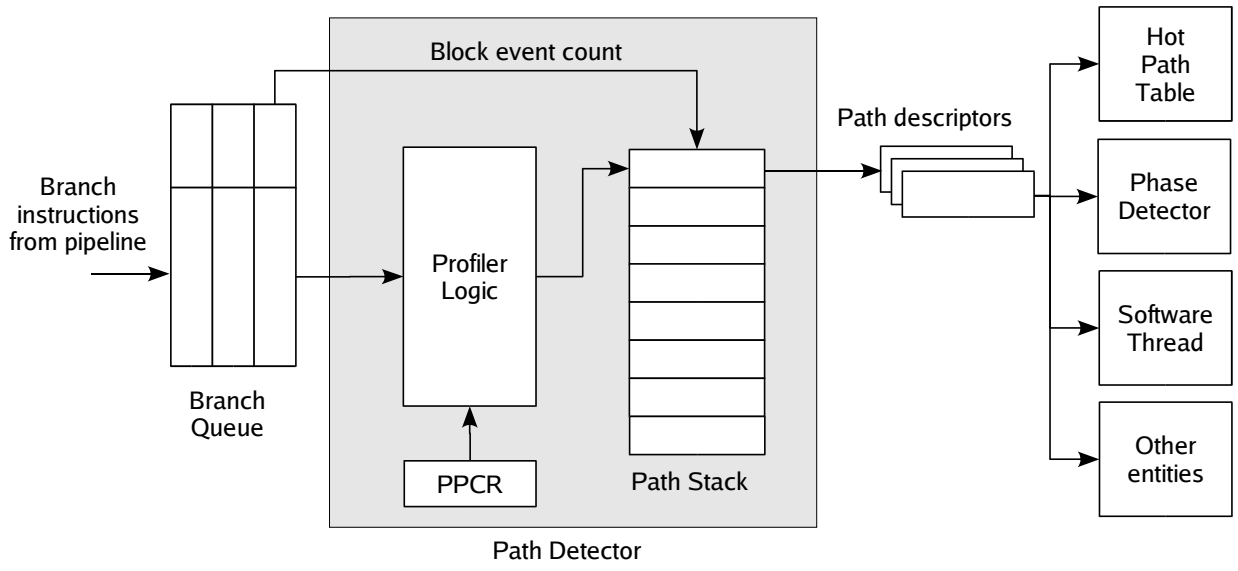

Figure 1. Components of the Hardware Path Profiling Infrastructure.

overcomes these limitations. Figure 1 illustrates the components of the proposed profiling hardware. At the heart of the hardware profiler is a **path detector** that uses a **path stack** to detect paths by monitoring the stream of retiring branch instructions emanating from the processor pipeline during program execution. The path detector can be programmed to detect various types of paths and track architectural events that occur along the paths. The detector generates a stream of *path descriptors*, which is available to all interested hardware/software entities.

The second component of the hardware profiling infrastructure is a **Hot Path Table** (HPT), a compact hardware structure that processes the stream of path descriptors generated by the path detector. The HPT is designed to collect accurate hot path profiles, irrespective of the type of path being profiled, the duration of profiling and the architectural metric associated with paths. The success of a moderately-sized HPT in capturing accurate path profiles is attributed to locality exhibited by program paths i.e. programs typically traverse a small fraction of the numerous feasible acyclic, intra-procedural paths. And as illustrated in Figure 2, a small set of hot paths within the set of traversed paths dominates program execution. These properties continue to hold when paths are associated with architectural events such as L2 cache misses or branch mis-predictions (Figure 2(b)).

As previously mentioned, the path detector can track one of several architectural events that occur along paths. Events are tracked using *instruction event counters* associated with every instruction in the processor pipeline. In such cases, the HPT generated profile identifies paths that caused a significant fraction of those events, enabling context-sensitive bottleneck analysis and optimization. The generic nature of instruction event counters and our path-based event tracking mechanism can be further exploited by associating paths with more complex metrics. As an illustration, we associate paths with a power metric that provides an accurate estimate of the power consumption caused by instructions along each path in the cache hierarchy, a task hard to emulate using software profilers.

Our results indicate that our path profiler virtually eliminates the space and time overheads incurred by software path profilers. Moreover, the profiles collected in hardware, although lossy, are sufficiently accurate for program optimizers that can usually tolerate a small loss in profile accuracy. In offline environments, the hardware path profile can be serialized to a file at program completion for later use by a static program optimizer. Our profiler is all the more effective in online environments, where path profiles representative of the current program behavior can be obtained by enabling the profiler for short intervals of time. The availability of accurate profiles at low overheads helps the dynamic compiler generate optimized code efficiently and quickly, increasing the number of optimization opportunities exploited.

The rest of the paper is organized as follows. Section 2 presents prior work on path profiling techniques. In Section 3, we describe how different types of paths can be represented and detected in hardware. We propose extensions to the processor's event detection logic that enable the tracking of various architectural metrics along paths in Section 4. Section 5 discusses the design of the Hot Path Table, the hardware structure that collects hot path profiles. Section 6 evaluates the path profiling infrastructure. We conclude in Section 7 with suggestions for future work.

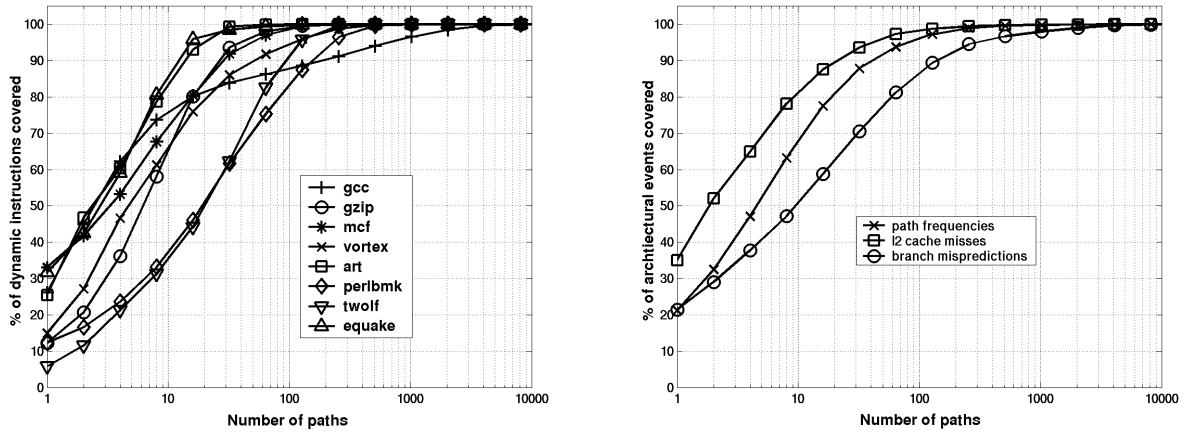

Figure 2. (a) Cumulative distribution of dynamic instructions along acyclic, intra-procedural paths. (b) Average cumulative distribution of path frequencies, L2 cache misses and branch mis-predictions along paths. The top 100 paths account for more than 80% of the dynamic instructions and a high percentage of L2 cache misses and branch mis-predictions in the SPEC CPU2000 programs studied.

2. Related Work

Although the notion of program paths has always generated significant academic interest, Ball and Larus [5] first demonstrated the feasibility of obtaining path profiles automatically and efficiently. The proposed instrumentation-based profiling scheme splits the dynamic instruction stream into acyclic, intra-procedural sequences, also referred to as Ball-Larus (BL) paths and tracks their occurrences. The profiler incurs average execution time overheads of 30-45%.

Subsequent research in path profiling has focused on alleviating two drawbacks of BL paths. First, such paths do not provide information about a program's control flow across procedure and loop boundaries, limiting their utility in inter-procedural and aggressive loop optimizations. Efforts to overcome this limitation include the Whole Program Path (WPP) [18] and extended path profiling [26]. A WPP is a complete sequence of the acyclic, intra-procedural paths traversed by a program. The sequence is compressed online by generating an equivalent context free grammar, which is compactly represented as a DAG. Despite high compression ratios, the WPP's size and profiling overheads hinder its widespread use. As a compromise between BL paths and the WPP, Tallam et al [26] propose the notion of *interesting* or *extended* paths – paths that extend beyond loop or procedure boundaries. Also proposed is a profiling scheme that reduces overheads by approximating the frequencies of extended paths from a profile of paths that are slightly longer than BL paths. Average execution time overheads of the profiling scheme are reported to be nearly 4 times the overheads of BL path profiling.

Other efforts [3, 14, 28] have focused on reducing the overheads of path profiling, a critical factor in cost-sensitive dynamic optimization systems. Although these schemes reduce profiling overheads without much loss in profile accuracy, they cause an increase in the time required to obtain representative profiles, which in turn delays the optimization process and results in fewer optimization opportunities being exploited. Moreover, it is not clear whether these schemes can be extended for profiling other varieties of paths or for profiling program binaries, as is required in binary translation/optimization systems.

The importance of efficient program profiling techniques in improving performance has also been recognized by computer architects. Most modern processors provide architectural support for performance monitoring, typically in the form of event counters [25, 24]. To meet the requirements of profile-guided optimizations, hardware profilers that construct approximate point profiles using information from the processor have also been proposed [8, 9, 19]. Our profiler subsumes existing hardware profiling schemes and has comparatively wider applicability because of its ability to collect various types of path profiles and the added ability to associate architectural metrics with program paths.

Locality in instruction streams is also responsible for the effectiveness of architectural enhancements such as global branch predictors [20], trace caches [23], trace predictors [13] and trace-based optimizers [21]. These mechanisms capture and exploit patterns in a program's control flow using approximate path representations like the recent global history, traces and frames. Unlike these schemes, our profiler is designed to detect and *collect* profiles for a larger class of program paths. Our hardware structures and policies are geared towards generating *accurate profiles*, irre-

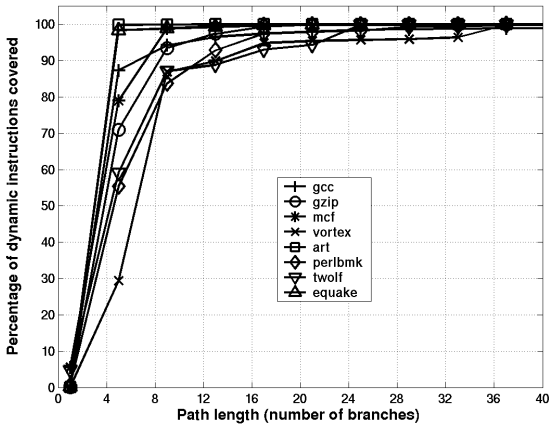

Figure 3. Distribution of dynamic instructions according to the length of BL paths they execute along. Paths with at most 16 branches account for over 90% of program execution in most programs.

spective of the duration of profiling.

3. Representing and Detecting Paths in Hardware

Paths have traditionally been represented by the sequence of indices/addresses of basic blocks that fall along the path. However, this representation is expensive to maintain in hardware. In our profiler, a path is uniquely represented by a *path descriptor* which consists of *(1)* the path's starting instruction address, *(2)* the length of the path in terms of the number of branch instructions along the path, and *(3)* a set of branch outcomes, one for each branch along the path[1]). This representation is compact and expressive enough to describe all types of paths.

The hardware version of the path descriptor does have a limitation; it can accommodate only a predetermined, fixed number of branches per path. Our path detection hardware overcomes this limitation by splitting paths if their length exceeds this threshold. Figure 3 shows the distribution of dynamic instructions according to the length of BL (Ball-Larus) paths they are executed along for programs from the SPEC CPU2000 benchmark suite. We observe that paths of length less than 16 branch instructions account for over 90% of dynamic instructions in most programs. For the rest of this study, we assume a path descriptor representation that allows paths to grow up to 32 branch instructions, large enough to accommodate a majority of BL paths and extended paths without splitting.

The hardware profiler uses a hardware *path stack* to detect paths. Each entry on the path stack consists of a path descriptor, an 8-bit path event counter, a path extension counter and other path specific information. The path profiler receives information pertaining to every retiring branch instruction from the processor pipeline via a *branch queue*, which serves to decouple the processor pipeline from the path profiler. Every branch read from the branch queue is decoded and classified as a call, a return, an indirect branch, a forward branch or a backward branch. The profiler then performs one or more of the following operations on the path stack depending on the type of branch being processed:

- *path_stack_push* : Pushes a new entry on the path stack with the starting address field of the path descriptor set to the target address of the branch being processed. All other fields of the new entry are reset to zero.
- *path_stack_pop* : Pops the current entry on top-of-stack and makes it available to all interested entities.
- *path_stack_update* : Updates the entry on top-of-stack with information about the branch being processed. The update involves incrementing the length of the path, updating the branch outcomes and incrementing the event counter. During update, if the length of the path on top-of-stack exceeds the predefined threshold value, the profiler logic pops the entry and pushes a new entry for a path beginning from the target of the current branch.
- *path_stack_update_count* : Increments the event counter associated with the path descriptor on top-of-stack without updating the path length or outcomes. This operation is used if it is desired that branches like calls and returns should not be explicitly recorded in the path.

The mapping between the branch type and set of operations to be performed is specified by the programmer in a *Path Profiler Control Register* (PPCR). However, the following restrictions on the mapping apply. For any branch, either one of *path_stack_update* or *path_stack_update_count* can be performed. Also, the order in which operations are performed is fixed, *viz* *path_stack_update/path_stack_update_count* followed by *path_stack_pop* and *path_stack_push*. These restrictions notwithstanding, the hardware path profiler can be programmed to detect several types of paths.

Detecting acyclic, intra-procedural paths: The branch type−profiler operations mapping shown in Table 1 enables the profiler to detect a variant of BL paths that terminate on backward branches. Path entries are pushed on calls and popped on returns. On a forward branch, the path descriptor on the top-of-stack is updated with information about the branch. The path entry on top-of-path stack is updated and terminated on all backward branches. Paths are also terminated on indirect branches since such branches can have several targets. These profiler operations ensure that

[1]This bit is set to 1 for unconditional branches.

Branch type	*update*	*update-count*	*pop*	*push*
call	×	√	×	√
return	×	√	√	×
forward	√	×	×	×
backward	√	×	√	√
indirect	√	×	√	√

Table 1. Branch type−profiler operation mapping for detecting BL paths

Branch type	*update*	*update-count*	*pop*	*push*
call	×	√	√	√
return	×	√	√	√
forward	√	×	×	×
backward	√	×	√	√
indirect	√	×	√	√

Table 2. Branch type-Profiler operation mapping for detecting paths for a Whole Program Path.

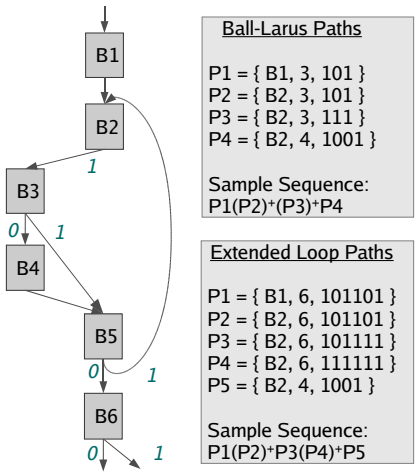

Figure 4. The control flow graph, path descriptors and a sequence of BL paths and extended loop paths detected for a sample execution of a procedure. Blocks *B1* and *B4* do not end with branches whereas *B2* ends with an unconditional jump to *B3*.

there exists exactly one entry on the path stack for every active procedure, and that the path stack grows and shrinks in the same way as stack frames on the program's runtime stack. Each entry on the path stack records the BL path that the corresponding procedure is currently traversing. Figure 4 shows a control flow graph, the set of BL path descriptors and a sequence of BL paths generated by the path stack for a sample procedure.

Detecting extended paths: In order to detect extended paths, the path stack supports a *path extension* counter with every entry on the path stack. The programmer is required to specify the following options via the PPCR: (1) the type of extended paths that should be profiled i.e. paths spanning across procedure boundaries or those that extend beyond loop boundaries, and (2) a maximum extension count that indicates the number of procedure calls or backward branches that the path is allowed to span across. The mapping shown in Table 1 is reused.

When programmed to detect extended paths, the path detector processes forward and indirect branches in the usual manner. However, the operations performed on backward branches, calls and returns are qualified by the value of the extension counter of the path stack entry on top-of-stack. If the profiler is programmed to detect paths that span across loop boundaries and a backward branch is encountered, the path on top-of-stack is allowed to expand if the corresponding extension counter value is less than the maximum extension count. In such a scenario, only the *path_stack_update* operation is performed and the extension counter is incremented. When the extension counter reaches the maximum count specified via the PPCR, the default set of operations are performed i.e the path is terminated. Figure 4 illustrates a set of extended loop paths detected and a sequence of paths recorded by the profiler for the given control flow graph.

When the profiler is tracking paths that span across procedure boundaries and a call instruction is encountered, the path on top-of-stack is terminated only if the extension counter value is equal to the maximum. Otherwise, the path is allowed to expand and the extension counter incremented. Conversely, paths are allowed to expand on procedure returns if the extension counter value is non-zero. Each return also decrements the extension counter. This set of actions generates path descriptors that represent inter-procedural paths through the program. It is important to note that with our profiler, the time complexity of detecting extended paths is similar to the complexity of detecting acyclic, intra-procedural paths.

Detecting paths for a Whole Program Path: The basic element of a Whole Program Path (WPP) is a variant of the BL path that also terminates at call instructions. A WPP is formed by compressing the sequence of such paths. Our profiler, configured using the mapping shown in Table 2, simplifies the process of constructing the WPP by generating the path sequence at low overhead and complexity. While processing a call, the profiler performs a *path_stack_pop* and terminates the current path before pushing a new entry on the stack. Similarly, the profiler performs a *path_stack_pop* followed by a *path_stack_push* on every return. Path descriptors generated by the path stack are fed to a software WPP profiler running as a separate thread, which compresses the sequence online and constructs the WPP.

```
compute dcache_access_power;
num_dcache_ports = 2;
if (num_dcache_access) {
  if (num_dcache_access <= num_dcache_ports)
    total_dcache_power +=  dcache_access_power;
  else
    total_dcache_power +=
      (num_dcache_access/num_ports)
           * dcache_access_power;
}
```

(a) Cycle level power model

```
compute dcache_access_cost;
num_dcache_ports = 2;
if (num_dcache_access) {
  if (num_dcache_access == 1)
    apportion_cost = dcache_access_cost;
  else
    apportion_cost = dcache_access_cost / 2;
  distribute apportion_cost to num_dcache_access instructions;
}
```

(b) Apportioning logic

Figure 5. (a) A power model that computes power consumption in L1 dcache and (b) corresponding apportioning logic that assigns costs to instructions that simultaneously access L1 dcache. Here, both $dcache\_access\_power$ and $dcache\_access\_cost$ are computed a priori; these are constants for a specific process technology.

Path stack consistency: For the path profiler to work correctly, the path stack must be maintained in state consistent with the program's execution. This requires special handling of certain events such as exceptions and setjmp/longjmp operations. The handlers for such events must repair the path stack, typically by popping entries of the path stack or purging the stack. Further, path stack overflows can occur due to the fixed size of the path stack. Implementations have the choice of handling overflows either by ignoring older entries on the stack, or by allowing the stack to grow into a region of memory specially allocated by the OS for the program being profiled. We find that a 32-entry path stack eliminates overflows for most of the programs we studied; we assume a path stack of this length for the rest of this study.

4. Associating architectural metrics with paths

Existing software-based path profilers are designed to track the frequency with which each path is traversed. However, future analysis tools and optimizations are likely to be interested in path-wise profiles of other metrics such as cache misses, branch mis-predictions and pipeline stalls. Such profiles are important because the paths of interest to an optimizer could differ significantly depending on the metric associated with paths. For example, our studies using benchmarks from the SPEC CPU2000 suite have shown that path profiles for various architectural and power metrics have only small percentage of information in common with a path-frequency profile (25% for a branch misprediction profile, 32% for a L2 cache profile and 62% for a power profile) [27]. These results justify the need for flexible profiling schemes that can capture program behavior across different architectural metrics.

Extending existing profiling schemes to associate architectural metrics with paths is complicated due to the intra-procedural nature of paths and perturbation effects of the instrumented code [1]. However, our hardware path profiler can perform this task accurately and non-intrusively since precise information regarding all architectural events is directly available to the profiler. To track the occurrence of such events, each instruction in the processor pipeline is annotated with an *event counter* (an extension of per instruction tag in [24]). An instruction's event counter is incremented every time the instruction causes an architectural event of type *X* specified via the PPCR. When an instruction commits, the value in the instruction's event counter is used to update a *block event counter* maintained at the commit stage of the pipeline. The block event counter value is passed to the path profiler along with every committing branch, which in turn updates the event counter associated with the path on top-of-stack. When a path is popped off the path stack, its event counter value represents the number of events of type *X* that occurred along the path. The block event counter is itself reset after every branch instruction.

A limitation of this scheme is that architectural events caused by non-committing, speculative instructions are not accounted for since the profiler monitors committing instructions only. Apart from this anomaly, the profiler is capable of associating any architectural event with paths, thereby enabling precise path-based performance monitoring and bottleneck analysis.

Associating power consumption metrics with paths: Virtually all existing hardware profiling schemes are focused towards detecting and aggregating data pertaining to performance-related architectural events. While this was also the case with the initial design of our profiling scheme, we were subsequently interested in exploring possible reuse of the proposed hardware in tracking metrics related to power consumption. Our goal was to associate program paths with a count that provides an estimate of the power consumption caused by instructions along the path in a specific processor component/set of components. Such profiles enable power-aware compilers to identify and focus on regions of code that account for a significant fraction of the power consumption. Such profiles can also assist a programmer in analyzing the impact of traditional compiler/architectural optimizations on power consumption in

specific regions of the program.

Our power profiling scheme assumes the availability of an accurate power model for each processor component of interest, parameterized by the component configuration and one or more architectural events. The power model is used to assign a *relative cost* to each architectural event related to the component. The cost associated with an event indicates the power consumed by the occurrence of the event relative to the event with the lowest consumption. For instance, the relative costs of accesses to each level of the cache hierarchy are derived from a power model for caches parameterized by the cache configuration and number of accesses. Since an instruction cache access typically consumes the lowest power, the costs of other events are relative to the instruction cache access.

Once costs are assigned to all events, the event detection logic associated with each component is extended to apportion the event cost to instructions that cause the events. For simple analytical models, this translates to logic that increments the event counters of all event-causing instructions by an amount equal to the cost of the event. More complex cycle-level models can be implemented using logic that dynamically computes the apportioned cost based on online information. Figure 5 illustrates one such dynamic power model for the L1 data cache [6] and the corresponding apportioning logic implementation. Here, the actual power consumption and the apportioned cost are determined based on the number of simultaneous data cache accesses in each cycle and the number of ports available.

Note that our implementation of the power models assumes a constant activity factor, a significant parameter in power models for components such as caches, buses and register files. Rounding off errors during the process of computing relative costs, approximations in the apportioning logic and the loss of information due to non-committing but power consuming instructions also introduce inaccuracies. However, our results suggest that although these factors cause discrepancies in the total power estimates, their impact on the quality of hot path profiles is minimal. We evaluate the effectiveness of our profiling scheme in collecting power specific profiles in Section 6.

5. Collecting Hot Path Profiles

Much of the execution time overhead incurred by existing path profiling techniques is attributed to the hash-and-update operation performed when a path terminates [14]. This overhead can be reduced if the hash table that stores the path profile is maintained in hardware. Such a hardware implementation must be capable of generating a profile accurate enough to drive path-based optimizations without a loss in their effectiveness. Additionally, the quality of the profile must be independent of the duration of profiling and the metric associated with paths.

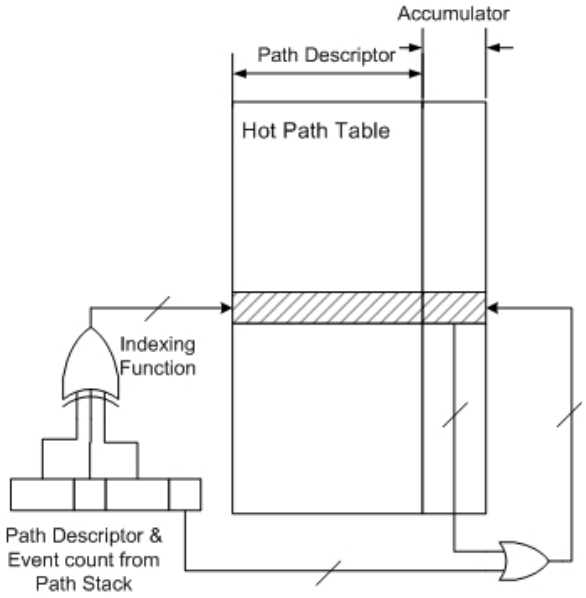

Figure 6. The Hot Path Table that collects hot path profiles. The HPT is indexed using bits from the incoming path's address, length and branch outcomes.

We evaluated several hardware profiler design configurations and next describe a simple, low overhead path collection scheme that meets these requirements. The path collection mechanism is based on a hardware structure called the *Hot Path Table* (HPT) illustrated in Figure 6. Each entry in the HPT consists of a path descriptor and a 32-bit accumulator. The HPT receives a sequence of path descriptors and associated counts from the path stack. An index into the HPT is computed from the fields of each incoming path descriptor. If a corresponding entry is found in HPT, the accumulator is incremented by the count associated with the incoming path. If the lookup fails, an entry from the indexed HPT entry set is selected for replacement and initialized with information about the incoming path descriptor.

The HPT design parameters that determine the effectiveness of the hot path collection scheme include the HPT size and associativity, the replacement policy and the indexing function. Our initial experiments indicate that LRU replacement, which is oblivious to the frequency of access of entries, does not capture and retain information about an adequate fraction of paths. The Least Frequently Used (LFU) policy serves the purpose of retaining frequently used entries. However, the execution time overheads of implementing LFU (*log n* for an n-way associative structure) have prevented its use in other cache structures. In the context of the HPT, a moderately expensive replacement policy does not have a significant impact on the overall execution time because *(1)* the HPT does not lie on the processor's critical path and *(2)* HPT updates are relatively infrequent (once

Processor core	Out-of-Order issue of up to 4 instructions per cycle, 128 entry reorder buffer, 64 entry LSQ, 16 entry IFQ
Functional units	4 integer ALUs, 2 integer MULT/DIV units, 4 floating-point units and 2 floating-point MULT/DIV units
Memory hierarchy	32KB direct mapped L1 instruction and data caches with 32 byte blocks (1 cycle latency) , 512KB 4-way set associative unified L2 cache (10 cycle latency), 100 cycle memory latency
Branch predictor	Combined: 12-bit (8K entry) gshare/(8K entry) bimodal predictor with 1K meta predictor, 3 cycle branch mis-prediction latency, 32 entry return address stack, 2K entry, 4-way associative BTB

Table 3. Baseline Processor Model

Parameter	Low	High
Number of HPT entries	128	2048
HPT associativity	2	32
HPT indexing scheme	Bits from path starting address	XOR(Bits from path starting address, path length, branch outcomes)
L2 cache size	256KB	2048KB
L2 cache associativity	1	8
Branch predictor	2K entry, 2-level predictor	Hybrid with 8K entry bimodal, 2K entry 2-level and 8K entry metatable

Table 4. Parameters considered during experiments based on Plackett-Burman design to determine an effective HPT configuration. The corresponding low and high values are also listed.

for every path) and HPT replacements even less frequent. Our experiments with different hotness criteria and HPT configurations reveal that the LFU replacement policy outperforms others without a significant increase in execution time overheads. Moreover, an implementation of LFU for the HPT does not incur additional space overheads since frequency information is available in counters associated with every HPT entry. For the rest of the paper, we assume an HPT implementation that uses LFU replacement. We discuss the impact of HPT size, associativity and indexing scheme on profile accuracy and profiler overheads in Section 6.

6. Experimental Evaluation

The success of a profiling technique can be measured by the accuracy of the profile, implementation costs and overheads of profiling. In this section, we evaluate our hardware path profiler on these counts.

- We explore the impact of various profiler design parameters on profile accuracy by comparing hardware generated profiles with a complete profile obtained using an infinitely large HPT. We compare profiles using the *overlap percentage* metric [3, 10], which indicates the percentage of information common to the profiles. Our results show that average profile accuracy of 88% is obtained using an HPT that occupies approximately 7KB of real estate.
- We assess the quality of hardware generated profiles in a real-world application by using the profiles to drive superblock formation in the *gcc* compiler. We find that performance benefits of superblock scheduling driven by HPT generated path profiles are similar to those obtained using a complete path profile.
- We evaluate the use of our profiler in gathering profiles where paths are associated with architectural metrics such as L2 cache misses, branch mis-predictions and a metric that estimates power consumption in the cache hierarchy.
- Using a cycle-accurate superscalar processor simulator, we find that the execution time overheads of our profiling scheme are low (0.6% on average), enabling the use of the profiler in cost-sensitive environments.

6.1 Experimental Methodology

We performed simulation experiments using 12 programs from the SPEC CPU2000 benchmark suite *gcc, gzip, mcf, parser, vortex, bzip2, twolf, perlbmk, art, equake, mesa* and *ammp*. We extended SimpleScalar [7], a processor simulator for the Alpha ISA, with an implementation of our hardware path profiler. The baseline micro-architectural model of the processor is shown in Table 3.

Due to simulation time constraints, overlap percentages in Section 6.2 are reported from simulations of 15 billion instructions for alpha binaries precompiled at *peak* settings, running on their reference inputs. We validated our simulation methodology using complete runs of as many of the programs as possible, finding an average deviation of 6% from the reported values. We extended the *gcc* compiler (version 3.4) with a path-based superblock formation pass. Execution times for complete runs of superblock scheduled binaries optimized at level -O2 were obtained using the hardware cycle counter on an Alpha AXP 21264 processor under the OSF V4.0 operating system. Profiling overheads reported in Section 6.3 are estimated using out-of-order processor simulations for complete runs of the programs with the MinneSPEC inputs [17].

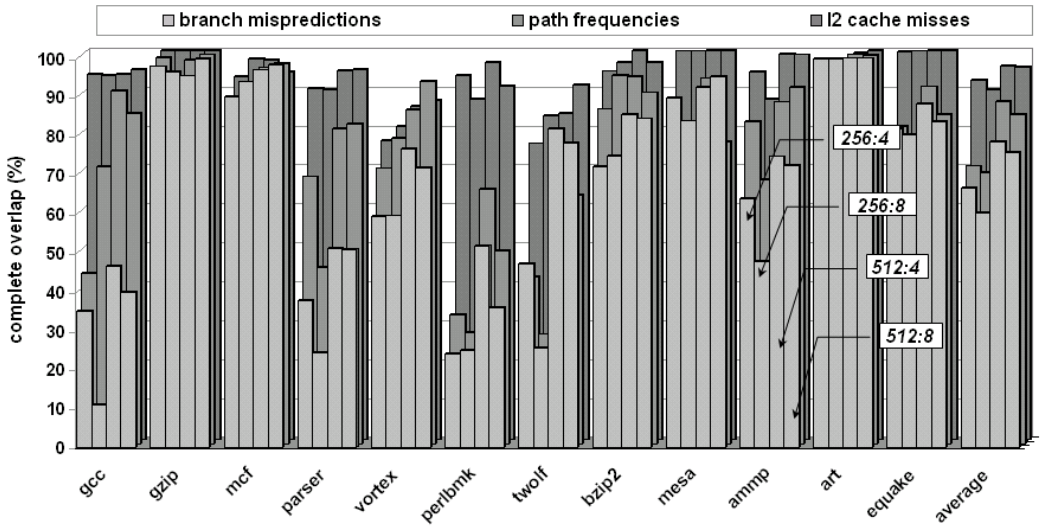

Figure 7. Complete overlap percentages for various profiler configurations when profiling path frequencies, L2 cache misses and branch mis-predictions.

6.2 Quality of Hardware Path Profiles

We use two metrics, the *complete overlap percentage* and the *dynamic overlap percentage* to quantify the accuracy of hardware generated profiles. The complete overlap percentage is obtained when profiling is enabled for the entire duration of program execution. To assess profile accuracy under conditions where the profiler is activated for short durations during program execution, we define the dynamic overlap percentage as the average of overlap percentages computed for path profiles over non-overlapping 100 million instruction execution windows.

We performed a simulation study using Plackett-Burman experimental design [22, 29] to identify a path profiler configuration that realizes our design goals. Table 4 lists the input parameters we used in this design with their low and high values. We also conducted Plackett-Burman experimental studies where metrics other than path frequency – number of L2 cache misses and number of branch mispredictions – are associated with paths. Our results show that the HPT size, HPT associativity and the HPT indexing scheme are the three most important profiler parameters [27]. On the other hand, the cache and branch predictor related parameters are of significantly less importance. This leads us to conclude that the hardware profile accuracy is unaffected by the underlying architecture. With these less important parameters eliminated, we next performed a full factorial study with HPT size, associativity and the indexing scheme as parameters. Our experiments reveal that an HPT indexing function that XORs bits from the path's starting address with path length and branch outcomes outperforms other functions we evaluated [27]. We use this indexing function in the rest of the study.

Figure 7 shows the complete overlap percentages for four HPT configurations. When profiling path frequencies, the best average complete overlap percentage of 88% is obtained using a 512 entry, 4-way set associative HPT. The overlap percentage with this configuration is 96% when the metric associated with paths is L2 cache misses. We attribute the higher coverage to the observation that L2 cache misses are concentrated along a smaller number of paths and exhibit more pronounced locality than paths themselves. On the other hand, the average complete overlap percentage dips to 78.4% when branch mis-predictions are profiled, a reflection of the fact that branch mis-predictions are usually distributed over a larger number of paths. Certain programs (*gcc, perl, twolf*) have lower overlap percentages since the working set of paths in these programs is large (Figure 2), making it harder for a hardware structure of limited size to capture a high fraction of execution. We also find that a further increase in HPT size leads to additional improvements in overlap percentages, although the gains taper off for HPTs with over 2048 entries [27].

The impact of HPT configuration on dynamic overlap percentages is summarized in Figure 8. Notice that the dynamic overlap percentages are higher than their static counterparts. This is to be expected since the dynamic overlap percentage is computed using path profiles collected over short time durations. For the 512 entry, 4-way set associative HPT, we obtain average dynamic overlap percentages of 95.5%, 98% and 91.2% when profiling path frequencies, L2 cache misses and branch mis-predictions respectively.

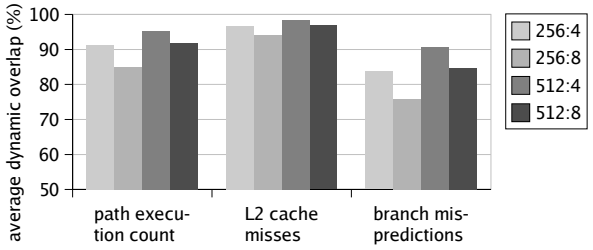

Figure 8. Average dynamic overlap percentages for various profiler configurations when profiling path frequencies, L2 cache misses and branch mis-predictions.

Although the overlap percentage metric quantifies profile accuracy, it says little about the performance impact of using a less than complete profile in a real world application. We obtain a direct assessment of the quality of hardware generated paths profiles by using them to drive superblock scheduling [12] in the gcc compiler. Our implementation of the path-based superblock formation pass identifies hot traces using path profiles followed by tail duplication, superblock enlargement and loop unrolling using heuristics similar to those used in [12]. The path-based superblock scheduler improves execution time by an average 9.3%, with a maximum of 14% over the baseline (-O2 optimized, traditional local scheduling).

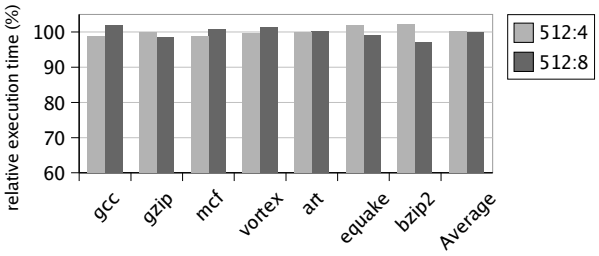

Figure 9. Execution times of binaries superblock-scheduled using path profiles from two HPT configurations normalized against the execution time of a binary scheduled using a complete path profile.

Figure 9 shows the execution times of optimized program binaries that use path profiles generated by different HPT configurations. Program execution time is reported relative to that of a binary executable generated using a complete path profile. Observe that the difference between execution times is 0.12% on the average with a maximum of 2.2%. For certain benchmarks, superblock scheduling using hardware-generated path profiles performs marginally better than scheduling using complete profiles; this behavior is attributed to secondary cache and branch prediction effects. On the whole, a minor change in average execution time indicates that hardware path profiles

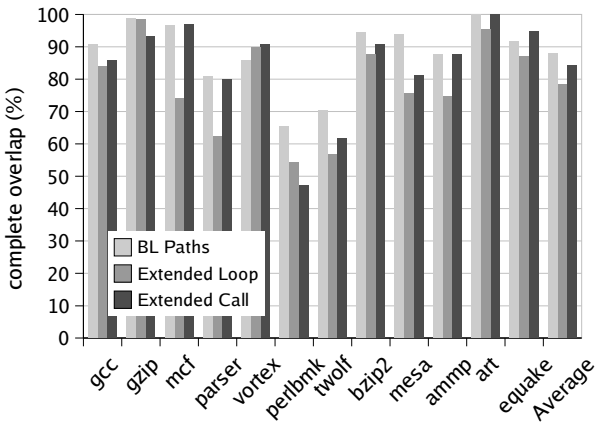

Figure 10. Accuracy of complete extended path profiles collected using the hardware path profiler. Paths extend across one backward branch/procedure call.

are comparable in quality to and can be used instead of complete path profiles.

Quality of extended path profiles: Next, we evaluate the effectiveness of the hardware profiler in collecting extended path profiles. Figure 10 shows the complete overlap percentages for extended paths that span across one backward branch and those that extend beyond one procedure call for a 512-entry, 4-way associative HPT. The average overlap percentages for such paths are 78.35% and 84.10% respectively. The reduction in overlap percentages when compared to a BL path profile is due to the increase in the number of unique paths traversed. It remains to be seen whether the reduced profile accuracy can be tolerated by real-world applications.

Quality of path-wise power profiles: Unlike other architectural metrics, the quality of path-wise power profiles cannot be assessed in isolation of the power models used to estimate power consumption in various components. In this section, we first determine whether the choice of a power model has an influence on the nature of path profiles. Our evaluation uses power consumption in the cache hierarchy as the metric associated with paths, primarily because a significant fraction of the overall power consumption is attributed to the caches [6]. Of the several candidate power models for caches [15, 16, 6], we chose two, an analytical power model from Kadayif et al [15] and a cycle level power model used by the Wattch power simulator [6]. The analytical model was used to compute the relative costs of hits and misses at each level of cache. Apportioning logic similar to Figure 5 was designed for the Wattch power model and integrated into a cycle-accurate processor simulator.

A comparison of the path-wise power profiles obtained

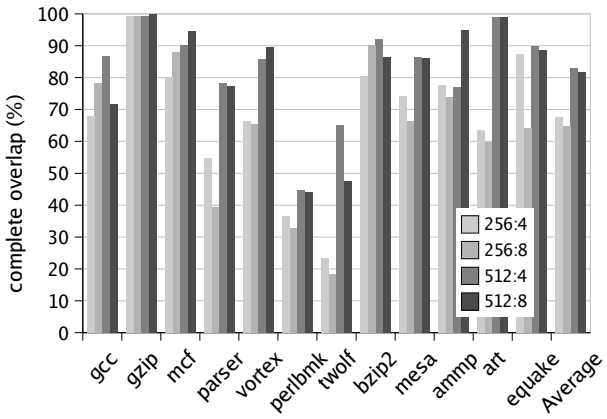

Figure 11. Accuracy of complete path-wise power profiles for various HPT configurations, obtained using an analytical model for power consumption in the cache hierarchy.

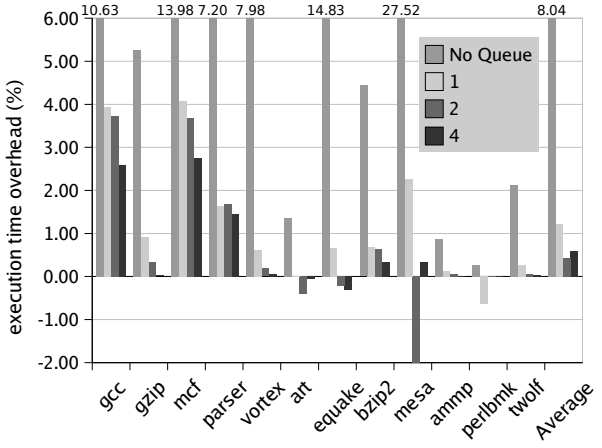

Figure 12. Execution time overheads incurred due to the hardware profiler while profiling BL paths for various branch queue sizes.

using the two power models shows a strong similarity in the relative ordering of paths in the profiles despite differences in the absolute value of the associated power metric. This observation suggests that for caches, the analytical model identifies hot paths as well as the accurate cycle-level power model. Figure 11 illustrates the impact of various HPT configurations on the accuracy of path-wise power profiles generated using the analytical power model. An average complete overlap percentage of 82.9% is obtained using a 512 entry, 4-way associative HPT. Further investigations into the relative quality of these profiles and their impact on power-aware compiler optimizations are left for future work.

6.3 Profiling Overheads

Using the hardware path profiler during program execution can lead to degraded performance if the profiler does not service branch instructions faster than their rate of retirement. To study this, we incorporated our path profiler into a cycle-accurate superscalar processor pipeline simulator. Profiler operations are assigned latencies proportional to the amount of work involved in carrying out those operations: the *path_stack_push*, *path_stack_update* and *path_stack_update_count* operations are assessed a latency of one cycle whereas the latency of a *path_stack_pop* is one cycle plus the latency of updating the HPT. Since the HPT uses a LFU based replacement policy, an HPT miss incurs a cost of *log(n)* cycles, where n is the associativity of the HPT. The latency of processing a branch is the sum of latencies of the profiler operations performed while processing the branch. If a retiring branch finds the branch queue full, the commit stage stalls and no further instructions are committed until the stalling branch can be accommodated in the branch queue.

The execution time overheads incurred while collecting a BL path profile using a 512-entry, 4-way set associative HPT for various branch queue sizes are shown in Figure 12. We observe that a 4-entry branch queue sufficiently buffers the profiler from the pipeline and reduces average execution time overheads from 8.04% to 0.6%. This also represents a sharp drop from the typical 30-45% overheads incurred by the Ball-Larus path profiling scheme [5]. Moreover, profiling overheads remain unchanged even when the profiler is configured to collect otherwise expensive extended paths. Our experiments ignore the overheads of transferring the HPT generated profile (approximately 6KB in size) to user memory. Even conservative estimates of this overhead (a few thousands of cycles) are negligible when compared to the execution time of the program.

7. Conclusions and Future Work

This paper proposes and evaluates a hardware path detection and profiling scheme that is capable of generating high quality hot path profiles with minimal space and time overheads. The profiler derives its flexibility from a generic path representation and a programmable interface that allows various types of paths to be profiled and several architectural metrics to be tracked along paths using the same hardware. These characteristics enable the use of the profiler in a host of static and dynamic optimizations systems. We believe that the availability of semantically rich and detailed path information in hardware opens the doors to several architectural optimizations. As an example, we have proposed a phase detection scheme [27] that accurately detects phase changes using the sequence of acyclic, intra-procedural paths generated by our hardware profiler. Possible avenues for future work include the use of hot path information in improving the performance of trace caches

and precomputing memory reference addresses for cache prefetching.

Acknowledgements

We would like to thank P. J. Joseph, Anand Vardhan and Bharath Kumar for providing valuable suggestions and comments throughout this work. This work is funded in part by the Infosys Doctoral Fellowship.

References

[1] G. Ammons, T. Ball, and J. R. Larus. Exploiting Hardware Performance Counters with Context Sensitive Profiling. In *Proceedings of the SIGPLAN Conference on Programming Languages Design and Implementation*, pages 85–96, 1997.

[2] G. Ammons and J. R. Larus. Improving Data-flow Analysis with Path Profiles. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 72–84, 1998.

[3] M. Arnold and B. Ryder. A Framework for Reducing the cost of Instrumented Code. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 168–179, June 2001.

[4] T. Ball and J. R. Larus. Optimally Profiling and Tracing Programs. *ACM Transactions on Programming Languages and Systems*, pages 16(4):1319–1360, July 1994.

[5] T. Ball and J. R. Larus. Efficient Path Profiling. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 46–57, 1996.

[6] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *Proceedings of the Annual International Symposium on Computer Architecture*, pages 83–94, 2000.

[7] D. Burger, T. M. Austin, and S. Bennett. Evaluating Future Microprocessors: The SimpleScalar Toolset. Technical Report CS-TR-96-1308, University of Wisconsin-Madison, 1996.

[8] T. M. Conte, B. Patel, K. N. Menezes, and J. S. Cox. Hardware-Based Profiling: An Effective Technique for Profile-Driven Optimization. *International Journal of Parallel Programming*, pages 187–206, Vol 24, No. 2, April 1996.

[9] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos. ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors. In *Proceedings of the Annual International Symposium on Microarchitecture*, pages 292–302, December 1997.

[10] P. T. Fellar. Value Profiling for Instructions and Memory Locations. CS98-581, University of California, San Diego, April 1998.

[11] R. Gupta, D. A. Berson, and J. Z. Fang. Path Profile Guided Partial Redundancy Elimination Using Speculation. In *Proceedings of the 1998 International Conference on Computer Languages*, page 230, 1998.

[12] W. W. Hwu and S. A. Mahlke. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *The Journal of Supercomputing*, pages 7(1–2):229–248, May 1993.

[13] Q. Jacobson, E. Rotenberg, and J. E. Smith. Path-Based Next Trace Prediction. In *Proceedings of the International Symposium on Microarchitecture*, pages 14–23, 1997.

[14] R. Joshi, M. D. Bond, and C. B. Zilles. Targeted Path Profiling: Lower Overhead Path Profiling for Staged Dynamic Optimization Systems. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 239–250, March 2004.

[15] I. Kadayif, T. Chinoda, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and A. Sivasubramaniam. vEC: Virtual Energy Counters. In *Proceedings of Workshop on Program Analysis for Software Tools and Engineering*, pages 28–31, 2001.

[16] M. B. Kamble and K. Ghose. Analytical Energy Dissipation Models for Low Power Caches. In *Proceedings of the International Symposium on Low Power Electronics and Design*, 1997.

[17] A. KleinOsowski and D. J. Lilja. MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research. *Computer Architecture Letters*, 2002.

[18] J. R. Larus. Whole Program Paths. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 259–269, 1999.

[19] M. C. Merten, A. R. Trick, C. N. George, J. C. Gyllenhaal, and W. mei W. Hwu. A Hardware-Driven Profiling Scheme for Identifying Program Hot Spots to Support Runtime Optimization. In *Proceedings of the 26th International Symposium on Computer Architecture*, pages 136–147, June 1999.

[20] R. Nair. Dynamic path-based branch correlation. In *Proceedings of the Annual International Symposium on Microarchitecture*, pages 15–23, 1995.

[21] S. J. Patel and S. S. Lumetta. rePLay: A Hardware Framework for Dynamic Optimization. *IEEE Transactions on Computers*, June 2001.

[22] R. Plackett and J. Burman. The Design of Optimal Multifactorial Experiments. *Biometrika*, Vol 33, Issue 4:305–325, June 1956.

[23] E. Rotenberg, S. Bennett, and J. Smith. Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching. In *Proceedings of the 28th International Symposium on Microarchitecture*, 1996.

[24] B. Sprunt. Pentium 4 Performance Monitoring Features. *IEEE Micro*, pages 22(4):72–82, July-August 2002.

[25] Sun Microsystems Inc. *UltraSPARC User's Manual*, 1997.

[26] S. Tallam, X. Zhang, and R. Gupta. Extending Path Profiling across Loop Backedges and Procedure Boundaries. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 251–262, March 2004.

[27] K. Vaswani, M. J. Thazhuthaveetil, and Y. N. Srikant. Representing, Detecting and Profiling Paths in Hardware. Technical Report IISc-CSA-TR-2004-7, Indian Institute of Science, May 2004.

[28] T. Yasue, T. Suganuma, H. Komatsu, and T. Nakatani. Structural Path Profiling: An Efficient Online Path Profiling Framework for Just-In-Time Compilers. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sept-Oct 2003.

[29] J. Yi, D. Lilja, , and D. Hawkins. A Statistically Rigorous Approach for Improving Simulation Methodology. In *Proceedings of the International Symposium on High Performance Computer Architecture*, February 2003.

[30] C. Young. *Path-based Compilation*. PhD thesis, Harvard University, Jan 1998.

[31] C. Young and M. D. Smith. Better Global Scheduling Using Path Profiles. In *Proceedings of the 30th International Symposium on Microarchitecture*, November 1998.