Representing, Detecting and Profiling Paths in Hardware

Kapil Vaswani T. Matthew Jacob Y. N. Srikant

IISc-CSA-TR-2004-2
http://archive.csa.iisc.ernet.in/TR/2004/2/

Computer Science and Automation
Indian Institute of Science, India

July 2004

Representing, Detecting and Profiling Paths in Hardware

Kapil Vaswani* T. Matthew Jacob† Y. N. Srikant‡

Abstract

For aggressive path-based optimizations to be profitable in cost-senstive environments, accurate path profiles must be available at low overheads. In this paper, we propose a low-overhead, programmable hardware path profiling scheme that can be configured to (1) detect a variety of paths including acyclic, intraprocedural paths, extended paths and sub-paths for the Whole Program Path and (2) track one of the many architectural metrics along paths. The profiler consists of a path stack that detects paths using branch information from the processor pipeline and a hot path table that records the path profile during program execution. Our experiments using programs from the SPECCPU 2000 benchmark suite show that the path profiler occupying 7KB of hardware real-estate collects accurate path profiles at negligible overheads (0.6% on average). The availability of path information in hardware also opens up interesting possibilities for architectural optimizations. As an illustration of the potential benefits, we present an online phase detection scheme that detects changes in program behavior with an accuracy of 94%, substantially higher than all existing schemes.

Keywords: path profiling, phase detection

1 Introduction

It has been widely accepted[1] that understanding and exploiting the dynamic behavior of programs is the key to improving performance beyond what is possible using static techniques. A program's control flow is one such aspect of its dynamic behavior that has been extensively used to optimize program execution, typically by identifying frequently executed regions of code that an optimizer can focus on and exposing control flow patterns that can be used to predict the program's future behavior.

A program's control flow can be characterized in several ways. All existing control flow profiling techniques can be classified as either point profiles or path profiles. Point profiles record the execution frequencies at specific points in the program; basic block, edge[2] and call-graph profiles are instances of point profiles. Point profiles have been extensively used in driving profile-based compiler optimizations, essentially because they are easier to collect and analyze. On the other hand, path profiles capture control flow in more detail by explicitly tracking sequences of instructions during execution. However, this additional detail comes at the cost of high profiler complexity and space and time overheads.

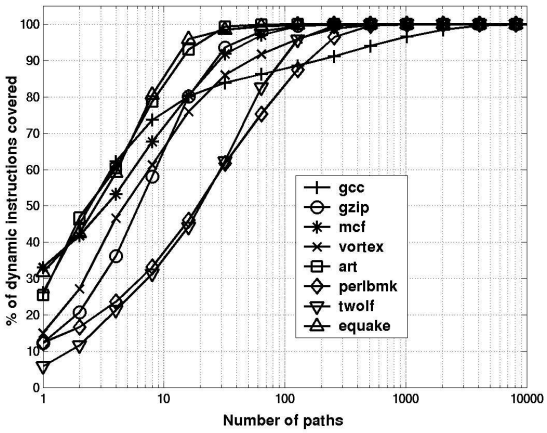

Figure 1: Distribution of dynamic instructions vs. the number of acyclic, intra-procedural paths that contribute to their execution. The top 100 paths account for more than 80% of the dynamic instructions in all the SPEC CPU 2000 programs studied.

Paths have themselves been characterized in different ways, each characterization involving a trade-off between the amount of information encoded in the profile and the overheads of profiling. The acyclic, intra-procedural path[3], defined as a sequence of basic blocks within a procedure that does not include loop back-edges, is a category of paths that is easy to comprehend and encodes enough control flow information required for most path-based optimizations[4, 5, 6]. Such paths exhibit interesting properties of locality and hotness[7]. Typically, programs traverse a

*Author for TR correspondence. kapil@csa.iisc.ernet.in
†mjt@csa.iisc.ernet.in
‡srikant@csa.iisc.ernet.in

small fraction of the numerous feasible acyclic, intra-procedural paths. And as illustrated in Figure 1, an even smaller set of hot paths within the set of traversed paths dominate program execution. The feasibility of existing path profiling schemes[3, 8] and the effectiveness of path-based compiler optimizations can be attributed to these properties.

However, existing path profiling techniques have several limitations.

- Since profilers must track a large number of paths, even efficient profiling techniques incur significant space and time overheads, precluding their use in cost-sensitive environments.
- Profiling overheads increase many fold if the scope of profiling is extended beyond acyclic, intra-procedural paths. The overheads must be tolerated if aggressive path-based optimizations that exploit correlation across procedure and loop boundaries are employed.
- Existing path profiling schemes do not facilitate profiling the fraction of code executed via system calls or calls to dynamic linked libraries.
- Precisely associating informative architectural events, such as the number of cache misses or branch mis-predictions, with program paths is a non-trivial task[9].

This paper proposes a unified, programmable hardware-based path detection and profiling infrastructure that overcomes the limitations listed above. Figure 2 illustrates various components of the proposed profiling hardware. At the heart of the hardware profiler is a *path stack* that detects paths by monitoring the stream of retiring branch instructions emanating from the processor pipeline. The path stack can be programmed to detect virtually any type of path by altering the set of actions performed for different types of branch instructions. The path stack generates a stream of paths, each path represented by a *path descriptor*, which is available to all hardware/software entities interested in the path sequence.

The second component of the profiling infrastructure is a *Hot Path Table* (HPT), which receives the stream of path descriptors and maintains a profile of paths that dominate program execution. The HPT eliminates space and time overheads incurred due to expensive hash-and-update operations performed by software path profilers[Reference] since these operations are now performed efficiently in hardware. The profile generated using the HPT is accurate but lossy since only a limited number of paths can be tracked in hardware. Since program optimizers are usually interested only in the set of hot paths, a small loss in accuracy can usually be tolerated. At program completion, the HPT generated profile can be serialized to a file for later use by a static program optimizer. The HPT 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.

Computer architects have also attempted to exploit the hotness and locality properties exhibited by path sequences[10, 11, 12]. However, the complextiy of path profiling has forced architects to use approximate path representations like traces or the recent branch history. We believe that the availability of information pertaining to different kinds of paths in hardware opens the doors to several architectural optimizations. As an example, we show that changes in program behavior can be accurately detected using the sequence of acyclic, intra-procedural paths generated by the hardware profiler.

The rest of the paper is organized as follows. Section 2 presents prior work on path profiling techniques and the use of paths in various compiler and architectural optimizations. In Section 3, we describe how different types of paths can be efficiently represented and detected in hardware. Section 5 discusses the design of the Hot Path Table, the hardware structure that collects hot path profiles. In Section 6, we describe a phase detection scheme that benefits from the availability of path information in hardware. Section 8 presents an evaluation of the path profiling infrastructure and assesses the impact of using path information to drive phase detection. We conclude in Section 9 with suggestions for future work.

2 Related Work

Because of the sheer amount of control flow information they encode and the complexity of profiling them, program paths have generated significant academic interest and continue to remain objects of great intrigue. In one of the earliest works on path profiling, Ball et al[3] propose an instrumentation based scheme for profiling acyclic, intra-procedural paths, also referred to as Ball-Larus (BL) paths. Given a control flow graph of a procedure, the profiling algorithm determines a minimal set of edges and a set of values, one for each edge, such that each BL path produces

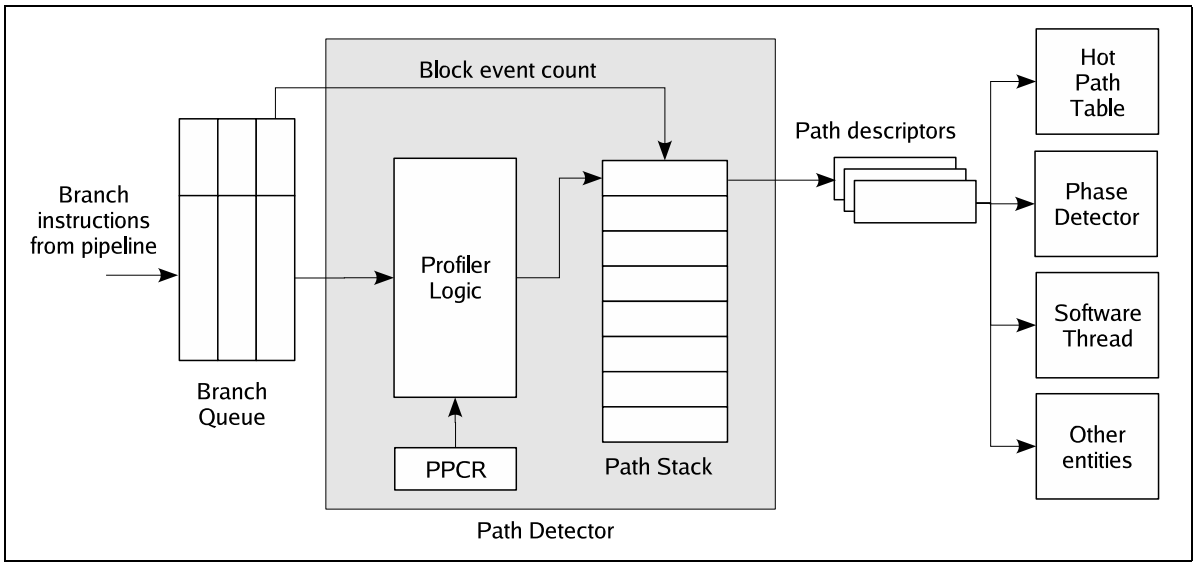

Figure 2: Components of the Hardware Path Profiling Infrastructure.

a unique sum of edge values along the path. These edges are instrumented with instructions that increment a global counter with the value associated with the edge. During program execution, the value of the global counter at loop back edges and procedure returns uniquely identifies the path that was just executed and is used to update a hash table that stores the profile. Apart from the space required for a hash table large enough to accommodate all paths, the profiling scheme incurs average runtime overheads of 30-45%.

Much of the subsequent research in path profiling has been focused on alleviating two drawbacks of BL paths. First, such paths do not provide any information about a program's control flow across procedure and loop boundaries, rendering them of limited use in several inter-procedural and aggressive loop optimizations. Efforts to overcome this limitation include the Whole Program Path (WPP)[13] and extended path profiling[14]. A WPP comprises of the sequence of acyclic, intra-procedural paths traversed by the program. Since explicitly storing such a sequence is prohibitively expensive, the authors propose an online compression scheme that exploits the presence of repeating patterns in the WPP and generates an equivalent context free grammar, which can be compactly represented as a DAG. Despite the high compression ratios, the WPP's size, profiling overheads and general lack of usability continue to hinder its widespread use. As a compromise between acyclic, intra-procedural paths and the WPP, Sriram et al propose the notion of interesting or extended paths i.e. paths that extend beyond a single loop or procedure boundary. Also proposed is a profiling scheme that reduces overheads by approximating the execution frequencies of extended paths from a profile of paths that are slightly longer than BL paths. Average execution time overheads are reported to be 86% or four times the overheads of acyclic, intra-procedural path profiling.

Recent efforts have also focused on reducing the overheads of path profiling, a factor critical to the success of path-based optimizations in cost-sensitive dynamic optimization systems. Arnold and Ryder[15] propose a generic software-based sampling scheme that reduces overheads by conditioning the execution of instrumented code on a global counter value. Targeted profiling[16] is an approach that identifies paths whose frequencies can be accurately deduced from an edge profile. Eliminating such paths from the path profiling phase leads to improved efficiency because of the significantly smaller number of paths that must be tracked. Structured profiling[17] tackles the complexity of path profiling by dividing functions hierarchically into smaller components, profiling each component in stages and finally composing a profile of the function from the individual profiles of its components. These schemes reduce profiling overheads considerably without a significant loss in accuracy of the path profile. On the flip side, these schemes cause an increase in the amount of time required to obtain a representative profile, delaying the optimization process as a result. Moreover, it is not clear whether these schemes can be extended for profiling other varieties of paths or for profiling program binaries as required in several

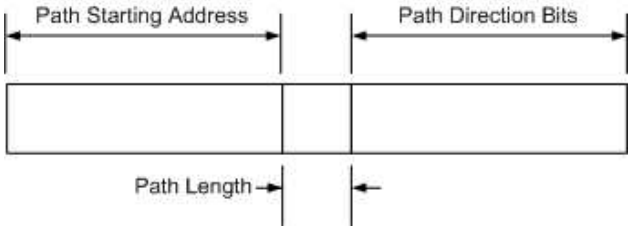

Figure 3: A Path Descriptor.

binary translation/optimization systems.

The importance of program profiling 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[18, 19]. Because data obtained from event counters is aggregate in nature and usually devoid of any contextual information, their use is mostly limited to identifying performance bottlenecks. To meet the requirements of profile-guided optimizations, profilers that construct approximate control flow profiles in hardware using information from the processor pipeline or from structures like the branch predictor have also been proposed[20, 21, 22, 23]. In comparison, the hardware path profiler has wider applicability and to a certain extent subsumes the existing profiling schemes because of the inherent nature of path profiles[24], the flexibility of detecting various types of path profiles at negligible overheads and the added ability to associate architectural metrics with program paths.

3 Representing and Detecting Paths in Hardware

While paths have traditionally been represented by the addresses of basic blocks that fall along the path, such a representation is too expensive to maintain in hardware. As shown in Figure 3, in our hardware profiler, a path is uniquely represented by a *path descriptor* which consists of *(a)* the path's starting address (the address of the first instruction on the path), *(b)* the length of the path (the number of branch instructions along the path) and *(c)* a set of direction bits (one for each branch along the path indicating whether the branch was taken or not). The path descriptor is compact and expressive enough to describe all types of paths known to the authors.

While there can be an arbitrarily large number of branches along a path, only a fixed number of branches can be accommodated in a hardware path descriptor. This limitation forces the path detection hardware to split paths if their length exceeds a predetermined threshold. Figure 4 shows the distribution of dynamic instructions according to the length of BL 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 the dynamic instructions. For the rest of this study, we assume a path descriptor representation that allows paths to grow up to 32 branch instructions, a limit that is sufficiently large to accommodate a majority of BL paths as well as extended paths without splitting.

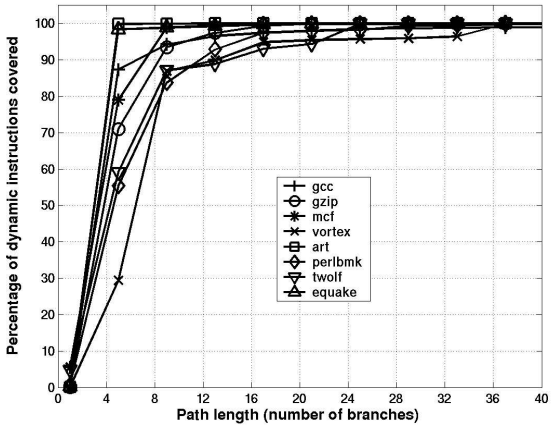

Figure 4: Distribution of dynamic instructions according to the length of BL paths they execute along.

The hardware profiler uses a hardware *path stack* to detect paths. Each entry on the path stack consists of a path descriptor, a 8-bit path event counter and other path specific information. The state of the path stack (e.g. pointer to the topmost entry on the stack) is maintained in a set of internal registers associated with the stack. 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 the 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

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

Table 1: Branch type-Profiler operation mapping for detecting BL paths

are reset to zero.

- *path-stack-pop* : Pops the current entry on top-of-stack and provides the same to interested entities.
- *path-stack-update* : Updates the path descriptor on top-of-stack with information about the branch being processed. The update involves incrementing the length of the path, updating the direction bits with the direction of the current branch and incrementing the event counter. During update, if the length of the path on top-of-stack exceeds its maximum, 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 direction bits. This operation is used when 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, hardware path profiler can be programmed to detect several types of paths.

Detecting acyclic, intra-procedural paths: The mapping of branch type and profiler actions shown in Table 1 enables the profiler to detect a variant of BL paths that terminate on all 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

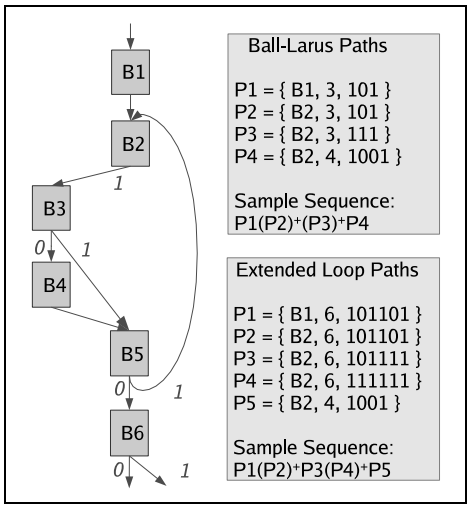

Figure 5: The control flow graph, path descriptors and a sequence of BL paths and extended loop paths detected for a sample execution of a procedure. Blocks *B1* and *B4* do not end with branches whereas *B2* ends with an unconditional jump to *B3*.

can have several targets. These profiler operations ensure that there always exists one and only one entry on the path stack for every active procedure, and that the path stack grows and shrinks in the same way as activation records on the program's runtime stack. Each entry on the path stack records the BL path that the corresponding procedure is currently traversing. In the presence of loops, the sequence of branches leading to the loop entry along with the first iteration of the loop together form a path. Each loop iteration other than the last last iteration is recorded as a separate path. Finally, the last iteration of the loop along with the sequence of branches following the loop exit also form a path. If the program uses setjmp/longjmp operations or exception handling mechanisms, the OS or the exception handler must repair the path stack by popping an adequate number of entries from it. Figure 5 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.

The path stack can overflow while recording BL paths if the number of active procedures exceeds the size of the path stack. Path stack overflows can be handled in two ways. On a stack overflow, entries from the bottom of the path stack can be removed to create space for the new entries. When a corresponding underflow occurs, a new entry on the stack is created and marked incomplete. When an incomplete path entry is popped, the corresponding

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

Table 2: Branch type-Profiler operation mapping for detecting subpaths that form a Whole Program Path

path is ignored. This strategy leads to a small loss of information if overflows/underflows occur frequently. If this loss in precision of the path profile is not acceptable, the path stack can be allowed to grow into a region of memory specially allocated by the OS for the program being profiled. The oldest path stack entries then are pushed into memory on overflows and underflows generate memory read operations to retrieve path stack entries.

Constructing a Whole Program Path: The basic element of a whole program path (WPP) is a variation of the BL path that also terminates at call instructions. A WPP is formed by compressing the sequence of such paths. The mapping shown in Table 2 configures the hardware path profiler to generate such a sequence. While processing a call, the profiler performs a path-stack-pop operation 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 generates a WPP.

Detecting extended paths: In order to detect extended paths, the path stack supports a *path extension* counter, typically 1-3 bits in size, for every entry on the path stack. The programmer is required to specify the following options via the PPCR: (1) whether paths must span across procedure or loop boundaries and (2) a maximum extension count that indicates the number of procedure returns or backward branches that the path is allowed to span across. In the course of detecting paths, if a new path is pushed onto the path stack, its extension counter is reset to zero. When a backward branch or a procedure return in 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 this scenario, only a path-stack-update operation is performed and the extension counter value is incremented. However, if the path has already expanded beyond a specified number of loop/procedure boundaries (known via the maximum extension count field in the PPCR), the default operations for the boundary branch are performed and the path is terminated. Similarly, a path is allowed to expand on a procedure return and the extension counter is decremented if the counter value is positive. It is important to note that the complexity of detecting extended paths in hardware is the same as the complexity of detecting acyclic, intra-procedural paths. Figure 5 illustrates set of extended loop paths detected and a sequence of paths generated for an imaginary control flow graph; here we are interested in paths that span across one loop boundary.

4 Associating architectural metrics with paths

Motivation: 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. To motivate the use of such profiles, we conducted an experiment that quantifies the similarity between these profiles and a path frequency profile. If these profiles are found to differ sufficiently, we can conclude that the paths of relevance to an optimizer (typically the hot paths) vary with the architectural metric associated with paths and cannot necessarily be inferred from a path frequency profile. We generated path-wise profiles for various architectural and power metrics benchmarks from the SPEC CPU2000 suite run using the reference inputs. We use the *overlap percentage* (see Section 8) to compare these profiles with a path frequency profile.

Figure 3 shows the overlap percentages between various path-wise profiles and a path frequency profile. We find that path profiles for various architectural and power metrics have only small percentage of information in common with the path-frequency profile (25% for a branch mis-prediction profile, 32% for a L2 cache profile and 62% for a power profile). These results suggest that a path-based optimizer that targets bottlenecks such as cache misses, branch mis-predictions or power consumption can be misled into making false assumptions about program hotspots from a path frequency profile. Further,

	branch misprediction profile	L2 cache miss profile	static power profile
gcc	15.90	38.65	61.55
gzip	37.81	2.37	69.34
mcf	42.45	67.42	77.41
vortex	17.29	37.33	79.13
art	15.82	18.56	28.92
equake	20.99	37.42	57.73
bzip2	36.43	38.04	75.61
mesa	25.77	18.24	56.75
ammp	18.00	45.49	46.91
perlbmk	25.87	20.19	71.70
Average	**25.63**	**32.37**	**62.50**

Table 3: Overlap percentages between various path-wise profiles and a path-frequency profile. A high percentage indicates more information in common between the profile.

these results justify the need for efficient and flexible profiling schemes that can capture program behavior across different architectural metrics.

Processor extensions: 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 [9]. 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 [19]). 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 6 illustrates one such dynamic power model for the L1 data cache [25] and the corresponding apportioning logic implementation. Here, the actual power consumption and the apportioned cost are determined based on the number of simultaneous data

```
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 6: (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.**

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 8.

5 Collecting Hot Path Profiles

Previous studies[16] have shown that a significant fraction of the overheads incurred by existing path profiling techniques can be attributed to the hash-and-update operation that must be performed when a path terminates. The overheads of this operation can be significantly reduced if the hash table that stores the path profile maintained in hardware. However, a hardware based path collection scheme must satisfy the following requirements.

- The path profile generated by the collection scheme must be accurate enough to effectively drive profile-guided optimizations. In other words, the distribution of paths in the hardware profile must closely resemble the distribution in the actual profile.
- The hardware collection scheme must provide accurate profiles at low overheads irrespective of duration of profiling. While paths have traditionally been used exclusively by offline compilers, the availability of path profiles at low overheads will make them attractive for use in dynamic environments of the future.
- Virtually all existing path profile based optimizations focus on hot paths, where the hotness of a path is determined by its execution count. However, future optimizations or performance monitoring schemes might want to associate paths with other architectural metrics such as cache misses, branch mis-predictions or pipeline stalls and redefine the hotness criteria accordingly. The profiling scheme must be designed to cater to such requirements without any loss in the quality of the profile collected.

We have evaluated several profiler design configurations and next describe a simple, low overhead path collection scheme that meets the requirements listed above. The path collection mechanism is based on a hardware structure called the *Hot Path Table* (HPT) illustrated in Figure 7. 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. When a path descriptor is received, a lookup on the HPT is performed using an index computed using fields of the incoming path descriptor. If an entry is found in HPT, the corresponding accumulator is incremented by the count associated with the incoming path. However, if the lookup fails, an entry from the indexed HPT entry set is selected for replacement based on a replacement policy. The vacated entry is updated with information from the incoming path descriptor and its accumulator is initialized to the count associated with the incoming path.

There are several HPT design parameters that determine the effectiveness of the hot path collection

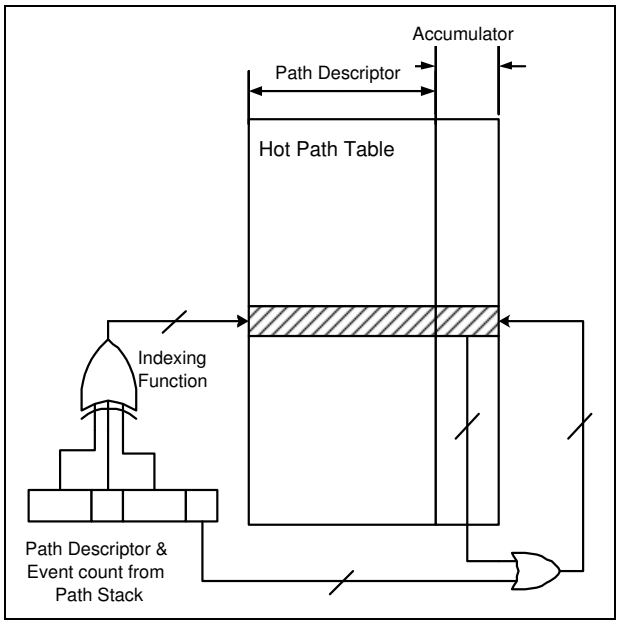

Figure 7: The Hot Path Table that collects hot path profiles.

scheme. These include the HPT size and associativity, the replacement policy and the indexing function. Our initial experiments indicate that an LRU like replacement policy, which is oblivious to the frequency of access of entries, does not succeed in capturing and retaining information about a significant fraction of paths over the entire duration of a program's execution. The Least Frequently Used (LFU) policy serves the purpose of retaining frequently used entries but the execution time overheads of implementing LFU (log(n) for an n-way associative structure) have prevented its use in other cache structures. However, in the context of the HPT, a moderately expensive replacement policy is unlikely to 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 lookups are relatively infrequent (once 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 all others and that the latency of implementing LFU can indeed be tolerated. Moreover, an implementation of LFU for the HPT does not incur additional space overheads since frequency information is available in the counters associated with every HPT entry. For the rest of the paper, we assume an HPT implementation that uses LFU replacement. A detailed study on the impact of HPT size, associativity and the indexing scheme on the profile accuracy and profiler overheads is presented in Section 8.

6 Path-based Phase Detection

While it is generally known that programs exhibit phased and/or periodic behavior, recent studies[26, 27, 28, 29] have empirically analyzed this behavior, proposed hardware support for online detection and prediction of phase changes, and demonstrated the use of phase information in dynamically reconfiguring processor policies. Based on the premise a phase can be characterized by the manner in which code is executed during the phase, various statistical measures that summarize the contents of the instruction stream during a program interval have been used for detecting phase changes; working set signatures[26], conditional branch counts[27] and basic block vectors (BBV)[28] are examples of such measures.

BBV-based techniques have proven to be relatively accurate in detecting phase changes as they encode fine-grained information about a program's execution in a given interval. Sherwood et al[29] propose a phase detection scheme that approximates BBVs in hardware using an array of accumulators. For every retiring branch instruction, an index into the accumulator array is computed and used to increment the accumulator by an amount equal to the number of instructions executed since the last branch. The values in the accumulators at the end of each program interval represent a distribution of instructions executed during that interval. A significant shift in the distribution between the current and the previous interval indicates a phase change. The scheme achieves an accuracy of around 80% in detecting phase changes and is able to detect stable intervals with a similar accuracy.

However, like basic block profiles, BBVs do not always represent control flow accurately. While tracking BBVs, control flow changes in successive program intervals may go unnoticed if their basic block vectors have a similar distribution of instructions. This would result in fewer phase changes being detected and fewer optimization opportunities. Moreover, the scheme is prone to errors introduced by mapping a possibly large number of basic blocks to a small set of counters in the accumulator array. These limitations of BBVs can be be overcome if a program's control flow is represented by the distribution of paths instead of the distribution of basic blocks. We observe that the distribution of paths is very sensitive to changes in control flow; even changes in program behavior result in a substantially different distribution of executed paths. Moreover, the number of paths that dominate execution

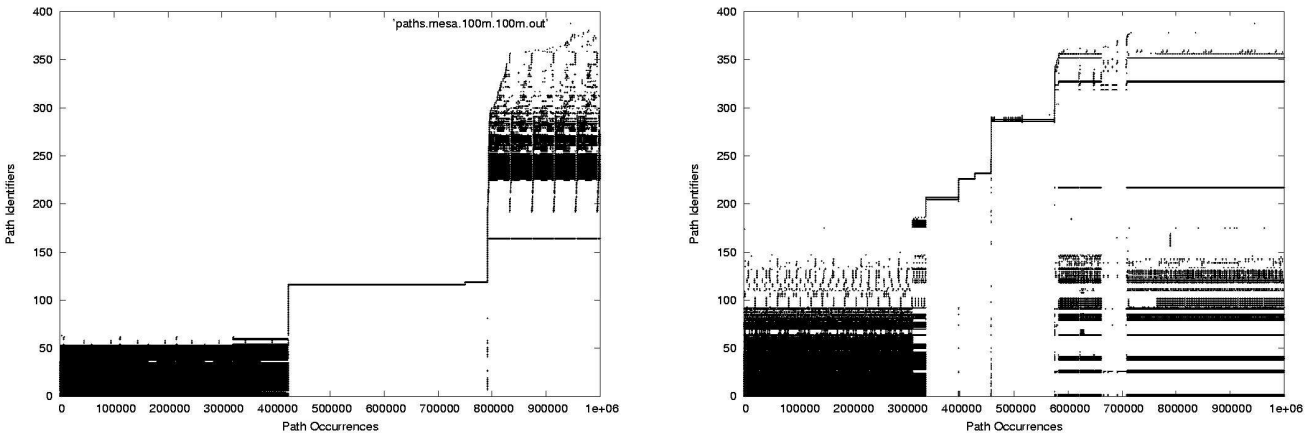

Figure 8: Variation in the distribution of paths caused by a phase change for instruction windows in *mesa* and *perlbmk* respectively

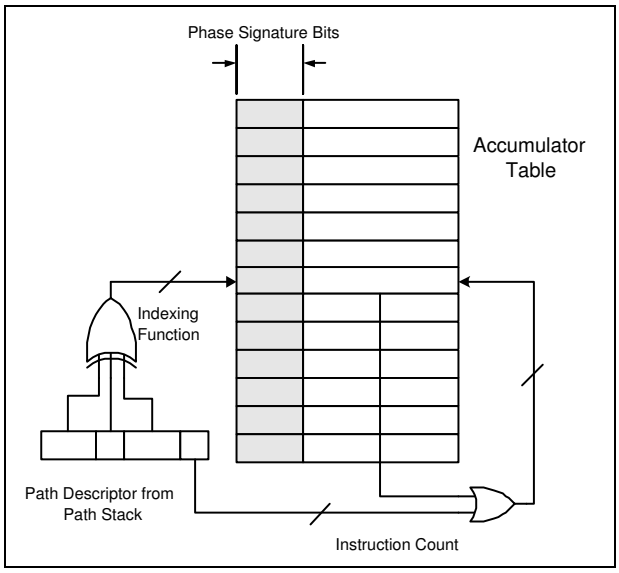

Figure 9: Path-based phase detection hardware.

during a phase tends to be small. Figure 8 provides visual proof for these observations. The figures plot the occurrence of paths in an instruction window during which a phase change occurs for two SPEC CPU2000 benchmarks, *mesa* and *perlbmk*. Such significant shifts in the distribution of paths due to a phase change are observed for virtually all benchmarks.

We therefore propose a modified version of the BBV-based phase detection scheme that uses path descriptors generated by the path stack to detect changes in program behavior. Figure 9 illustrates the main components of the path based detector. The path stack is configured to detect acyclic, intra-procedural paths and to keep track of the number of instructions executed along each path. The path-based phase detector receives a stream of path descriptors generated by the path stack along their instruction counts. The detector employs a hardware configuration that consists of an array of accumulators. The accumulators are reset to zero at the beginning of each program interval and updated on the occurrence of every path. An index into the array is computed using fields from the path descriptor, and the indexed accumulator is incremented by the number of instructions along the path being processed. At the end of a program interval, the accumulator values form an approximate path-specific distribution of instructions executed during the interval. In a manner similar to scheme of Sherwood et al, signatures for each interval are formed using higher order bits from the accumulators are classified as belonging to the same phase if the Manhattan Distance between them is lower than a threshold. A detailed evaluation of this scheme is presented in Section 8.

7 Simulation Methodology

We performed simulation experiments using 12 programs from the SPECCPU 2000 benchmark suite [30] *gcc, gzip, mcf, parser, vortex, bzip2, twolf, perlbmk, art, equake, mesa* and *ammp*. We extended SimpleScalar [31], 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 4.

Due to simulation time constraints, overlap percentages in Section 8.1 are reported from simulations

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 4: 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 address, path length, branch outcomes)
L2 cache size	256KB	2048KB
L2 cache associativity	1	8
Branch predictor	2K entry, 2-level predictor	Combined: 8K entry bimodal, 2K entry 2-level, 8K entry metatable

Table 5: Parameters considered during experiments based on Plackett-Burman design to determine an effective HPT configuration. The corresponding low and high values are also listed.

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 8.2 are estimated using out-of-order processor simulations for complete runs of the programs with the MinneSPEC inputs [32]. We evaluate our phase detection scheme (Section 8.3) using phase information generated by the SimPoint tool [28] as the baseline. Results are reported for simulations of 15 billion instructions using the reference inputs, as in [29].

8 Experimental Evaluation

The success of any profiling technique is measured by the accuracy of the profile, implementation costs and overheads of profiling. In this section, we evaluate the hardware path profiler on these counts. We explore the impact of various profiler design parameters on the accuracy of the profile, which is estimated using the overlap percentage[33, 15]. We assess the quality of the hardware generated path profile in a real-world application by using the profile to drive superblock formation in the gcc compiler. Next, we determine the execution time overheads of our profiling scheme using a cycle-accurate superscalar simulator. Finally, we evaluate the effectiveness of the path-based phase detection scheme.

Our results show that average profile accuracy of 88% is obtained using a hot path table that occupies approximately 8KB of real estate. Experiments with superblock scheduling confirm the high quality of the hardware generated path profile. We find that superblock scheduling using the hardware path profile is substituted for a complete profile. Moreover, execution time overheads of profiling are low (0.6% on average) enabling the use of the profiler in cost-sensitive environments. Our results also indicate that a path-based phase detection technique outperforms BBV based schemes, improving the accuracy of phase detection by over 10%.

8.1 Quality of Hardware Generated Path Profiles

The hardware path profiler must generate accurate profiles irrespective of the duration of profiling and the metric associated with paths. We performed a simulation study using Plackett-Burman experimental design to identify a path profiler configuration that realizes these goals. Table 5 lists the input parameters we used in this design with their low and high values. For these experiments, we used two output met-

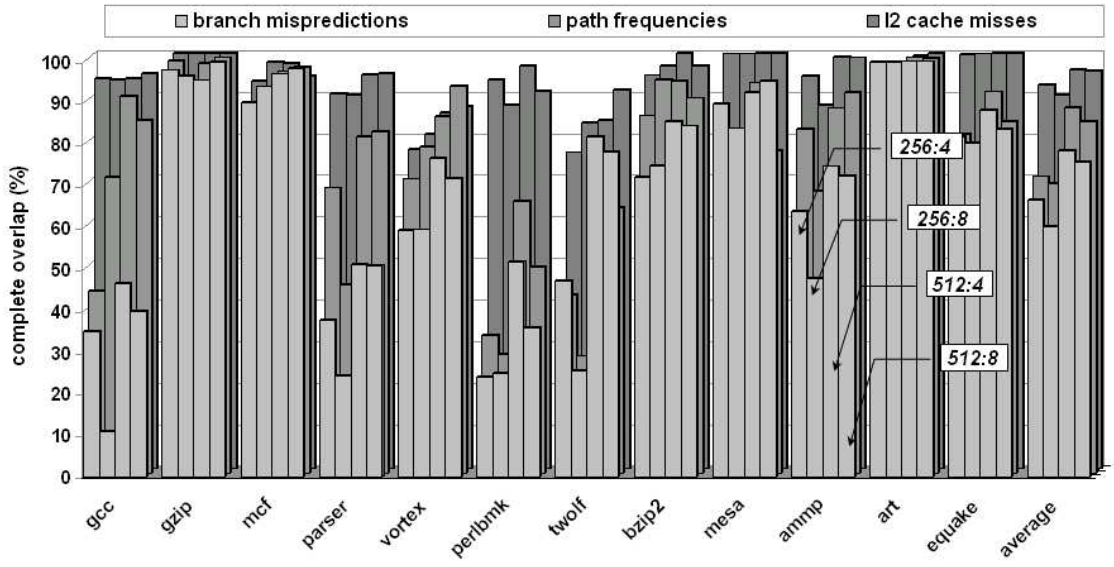

Figure 11: Complete overlap percentages for various profiler configurations when profiling path execution counts, L2 cache misses and branch mis-predictions.

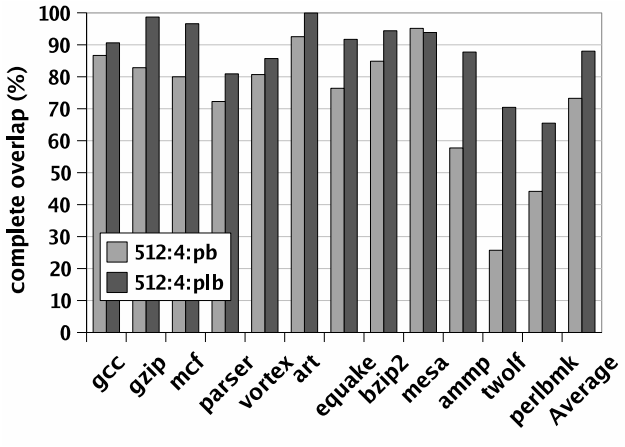

Figure 10: A comparion of complete overlap percentages obtained using two indexing schemes

rics, the *complete overlap percentage* and the *dynamic overlap percentage*. The complete overlap percentage is the overlap percentage of the hardware path profile obtained after profiling has been enabled for the entire duration of program's 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 of over non-overlapping 100 million instruction execution windows. We also conducted Plackett-Burman experimental studies where metrics other than path execution counts - number of level 2 cache misses and number of branch mis-predictions are associated with paths.

From the results of experiments using the Plackett-Burman design (refer appendix A for complete results), we find that the HPT size, HPT associativity and the HPT indexing scheme are the three most important profiler parameters. 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 parameters eliminated, we next performed a full factorial study with HPT size, associativity and the indexing scheme as parameters. Our initial experiments reveal that an HPT indexing function that XORs bits from the path's starting address with path length and the direction bits outperforms all other functions we evaluated. Figure 10 compares the complete overlap percentages obtained using two indexing schemes: one that uses bits from the path address, branch outcomes and length *(plb)* vs one that uses bits only from the path address and branch outcomes *(pb)*. We find that the plb scheme outperforms pb for almost all benchmarks and over 15% on the average. We use the *plb* indexing function in the rest of the study.

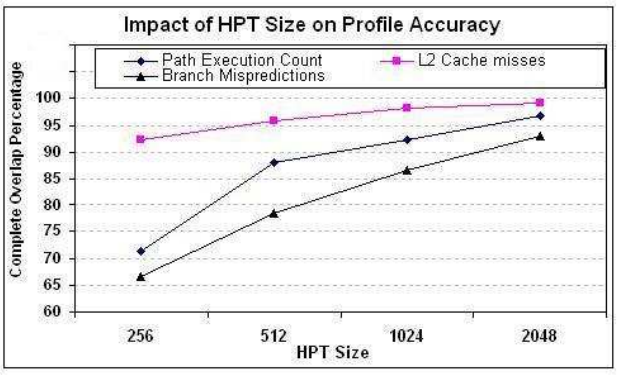

Figure 12: Variation in complete overlap percentages for various metrics with the change in HPT size. All HPT's are 4-way set associative.

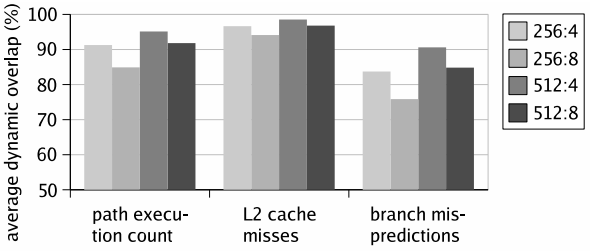

Figure 13: Average dynamic overlap percentages for various profiler configurations when profiling path execution counts, L2 cache misses and branch mis-predictions.

Figure 11 shows the variations in complete overlap percentages for four HPT configurations. An average complete overlap percentage of 88% is obtained using a 512 entry, 4-way set associative HPT when profiling path execution counts. The overlap percentage increases to 96% when the metric associated is L2 cache misses. We attribute the higher coverage to the observation that level 2 cache misses are concentrated along a smaller number of paths and exhibit more pronounced locality and hotness properties than paths themselves. On the other hand, the average complete overlap percentage dips to 78.4% when branch mispredictions are profiled, a reflection of the fact that branch mis-predictions are usually distributed over a larger number of paths. As shown in Figure 12, further increase in HPT size leads to additional improvements in overlap percentages. An HPT with 2048 entries captures virtually all paths, L2 cache misses and branch mispredictions.

The impact of HPT configuration on dynamic overlap percentages is shown in Figure 13. Notice that the dynamic overlap percentages are higher than their

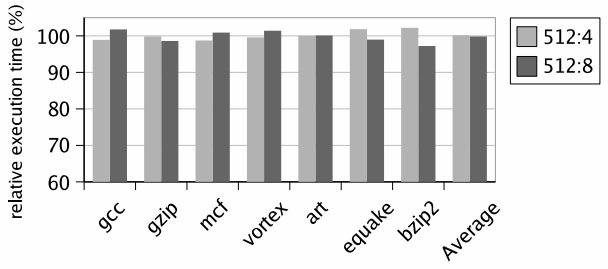

Figure 14: Relative execution times of binaries superblock-scheduled using path profiles from two HPT configurations.

static counterparts. This is expected since the dynamic overlap percentage is computed using path profiles collected for short durations. The number of paths exercised in a short interval is likely to be smaller and the path distribution sharper, so that the profile can be easily accommodated in the HPT. For the 512 entry, 4-way set associative HPT, we obtain an average dynamic overlap percentages of 95.5%, 98% and 91.2% when profiling path execution counts, level 2 cache misses and branch mis-predictions respectively.

Although overlap percentages are reasonable quantifiers of a profile's accuracy, they say little about the performance impact of using a less than complete profile in real world applications. We therefore obtained a direct assessment of the quality of hardware generated paths profiles by using them to drive superblock scheduling in the gcc compiler. Our superblock scheduling implementation follows the traditional compile-profile-recompile cycle except that the initial compilation pass does not involve any instrumentation. Instead, while generating an unoptimized binary, the compiler emits information about the program's control flow layout necessary for mapping path descriptors in the hardware path profile to paths through the program's control flow graphs. Next, the unoptimized binary is profiled during trial runs of the program using train input sets during which a complete path profile along with path profiles for various HPT configurations are obtained. Finally, the profiles are fed back to the compiler which performs superblock formation and enlargement and generates optimized binaries. The optimized binaries are run using the SPEC reference input sets and their execution times are compared.

Figure 14 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 generated using

$$Energy = E_{bus} + E_{cell} + E_{pad} + E_{main}$$

$$E_{bus} = E_{add\_bus} + E_{data\_bus}$$

$$E_{cell} = \ * (word\_line\_size) * (bit\_line\_size + 4.8) * (N_{hit} + N_{miss})$$

$$E_{pad} = E_{add\_pad} + E_{data\_pad}$$

$$E_{main} = Em * 8L * N_{miss} * (1 + dirty\_r)$$

$$E_{add\_bus} = 0.5e\text{-}12 * pr1 * V^2 * (N_{hit} + N_{miss}) * W_{add}$$

$$E_{data\_bus} = 0.5e\text{-}12 * pr2 * V^2 * (N_{hit} + N_{miss}) * 32$$

$$E_{add\_pad} = 20e\text{-}12 * pr3 * V^2 * N_{miss} * W_{add}$$

$$E_{data\_pad} = 20e\text{-}12 * pr4 * V^2 * (1 + dirty\_r) * N_{miss} * 64$$

$$word\_line\_size = m * (8L + T + St)$$

$$bit\_line\_size = C / (m * L)$$

= technology parameter

Em = per-access offchip energy cost

C = cache size

L = cache line size

m = set associativity

T = tag size in bits

St = number of status bits

N_{hit} = number of hits

N_{miss} = number of misses

W_{add} = width of address bus

pr1-4 = activity rates

Figure 16: Analytical model for power consumption in caches. E_{cell} represents the cache energy and E_{bus} represent the address and data bus energy consumption. The values $pr1$ through $pr4$ are the activity factors assumed to be 0.25 for this study.

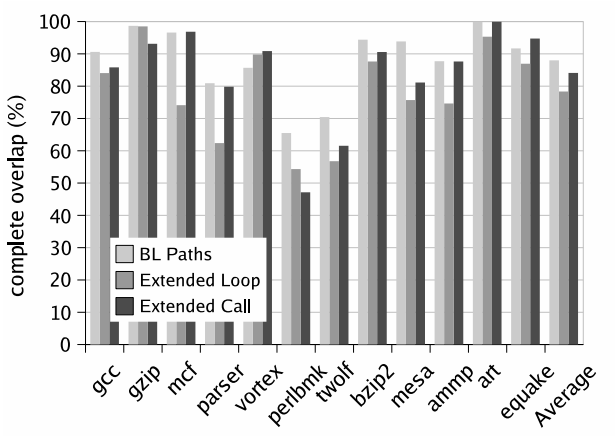

Figure 15: Accuracy of complete extended path profiles collected using the hardware path profiler.

a complete path profile. We observe that the difference between execution times is between ±2% for virtually all programs and 0.12% on average. This minor change in execution time proves that hardware path profiles are comparable in quality to and can be used instead of complete path profiles.

Next, we evaluate the effectiveness of the hardware profiler in collecting extended path profiles. Figure 15 illustrates the complete overlap percentages for extended paths that span across one backward branch and those that extend beyond one procedure call. The figures reveal that the hardware profiler can be used to collect extended path profiles with an accuracy of around 78%. 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 [25]. Of the several candidate power models for caches [34, 35, 25], we chose two, an analytical power model from Kadayif et al [34] (Figure 16) and a cycle level power model used by the Wattch power simulator [25]. The analytical model was used to compute the relative costs of hits and misses at each level of cache. Apportioning logic similar to Figure 6 was designed for the Wattch power model and integrated into a cycle-accurate processor simulator.

A comparison of the path-wise power profiles obtained 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 17 illustrates the impact of various HPT configurations on the accuracy of path-wise power profiles generated using the analytical power model. An aver-

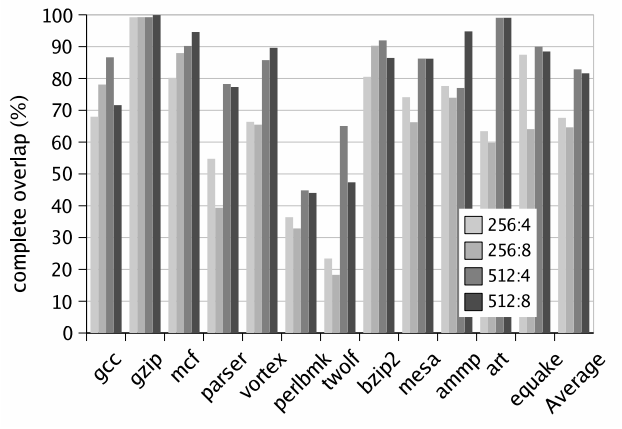

Figure 17: Accuracy of complete path-wise power profiles for various HPT configurations, obtained using an analytical model for power consumption in the cache hierarchy.

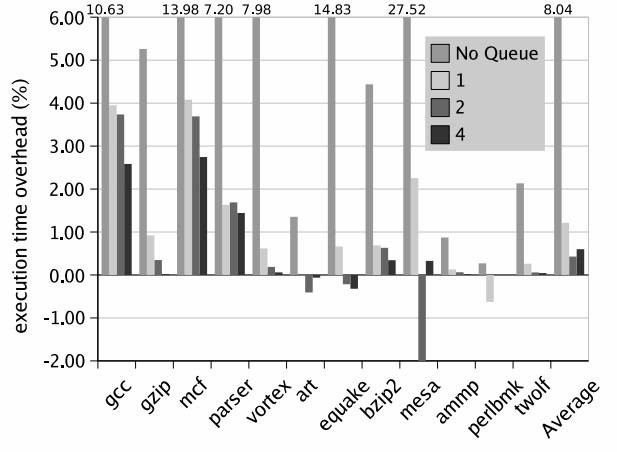

Figure 18: Execution time overheads incurred due to the hardware profiler while profiling BL paths for various Branch Queue sizes.

age 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.

8.2 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 the 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. A path-stack-push operation incurs 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 (as specified in the PPCR) performed while processing the branch. When detecting and collecting an acyclic, intra-procedural path profile, the latency of processing backward and indirect branches is five cycles and a call instruction is processed in two cycles. 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 by benchmarks from the SPEC CPU2000 suite while collecting a BL path profile on a 4-wide superscalar processor are shown in Figure 18. The overheads are computed for a 512-entry, 4 way set associative HPT that incurs a two cycle latency on every miss. For each benchmark, the leftmost bar represents the profiling overhead in the absence of a branch queue. The next two bars represent the profiling overheads with a branch queue of size 2 and 4 entries respectively. 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 represents a sharp drop from the typical 30-45% overheads incurred by traditional instrumentation based path profiling schemes. Moreover, the profiling overheads remain unchanged even when the profiler is configured to collect otherwise expensive extended paths.

8.3 Path-based Phase Detection

In the absence of a concrete definition of a program phase and an ideal technique that detects absolute phases in a program, phase detection techniques have been evaluated using metrics that assess their suitability in driving typical phase-based optimizations[36]. For this study, we evaluate the path-based phase detector using the following metrics: sensitivity, false positives, stability and average phase length. Figure 19 illustrates the variations in sensitivity and false positives for various Manhattan Distance thresholds. As expected, the sensitivity of the phase detector and the percentage of false positives increase with decreasing thresholds. An optimum working point is reached at a threshold of 6 million instructions where the de-

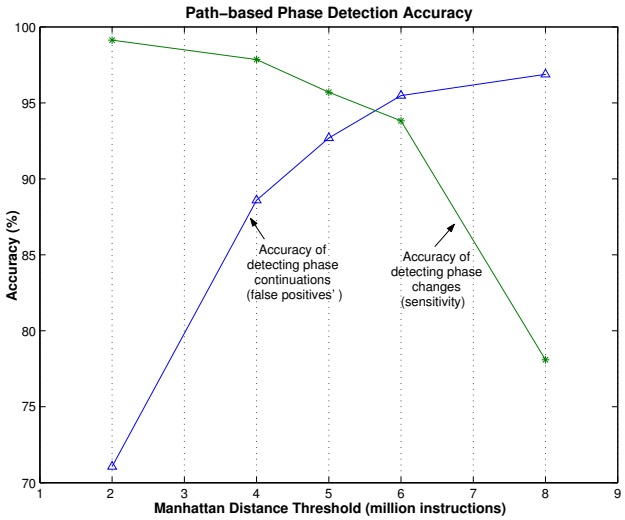

Figure 19: Variation in sensitivity and false positives of the path-based phase detector for different Manhattan Distance thresholds.

tector recognizes phase changes with an accuracy of 94% and detects consecutive intervals belonging to the same phase with an accuracy of 95%. This represents an improvement of 14% over the BBV-based scheme. Moreover, the operating point is attained at a much higher threshold when compared to the BBV based scheme, which suggests that changes in program behavior cause a significant change in the distribution of paths. Our results also show that the path-based detector achieves an average stability of approximately 88% at the optimum working point, significantly higher than 62% achieved using the BBV based technique. Moreover, the average length of phases detected using the path-based scheme is 5.5 intervals, which is comparable to that achieved using the BBV based scheme.

Another desirable attribute of the phase detector is the relative stability of program performance within a phase. We use the variance in CPI (standard deviation/mean) within phases as a metric to evaluate the path-based phase detection scheme on this count. We find that the average percentage variance in CPI for phases detected using the path-based scheme is 0.23 at the optimum working point, significantly lower than 0.66 measured for the BBV based scheme. From these results, we infer that the modified path-based phase detection scheme achieves higher accuracy and enhanced phase quality when compared to the BBV based scheme, making it more suitable for phase-based optimizations.

9 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. The potential of using path information in driving architectural optimizations is exemplified by the improved accuracy of detecting phase changes obtained using our path-based phase detection scheme. We are currently exploring the use of acyclic, intra-procedural path sequences in understanding and improving the predictability of branches. 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.

References

[1] Michael D. Smith. Overcoming the Challenges of Feedback-directed Optimization. In *ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo 2000)*, pages 1–11, 2000.

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

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

[4] Glenn Ammons and James R. Larus. Improving Data-flow Analysis with Path Profiles. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 72–84, 1998.

[5] C. Young and M. D. Smith. Better Global Scheduling Using Path Profiles. In *Proceedings of the 30th Annual IEEE/ACM Intl. Symp. on Microarchitecture*, November 1998.

[6] Rajiv Gupta, David A. Berson, and Jesse Z. Fang. Path Profile Guided Partial Redundancy Elimination Using Speculation. In *Proceedings of the 1998 International Conference on Computer Languages*, page 230, 1998.

[7] Thomas Ball and James R. Larus. Using Paths to

Measure, Explain, and Enhance Program Behavior. In *Transactions of IEEE Computer*, pages 33(7): 57–65, 2000.

[8] C. Young. Path-based Compilation. In *PhD Thesis, Harvard University*, Jan 1998.

[9] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with context sensitive profiling. In *Proceedings of the ACM SIGPLAN 97 Conference on Programming Languages Design and Implementation*, pages 85–96, 1997.

[10] Ravi Nair. Dynamic path-based branch correlation. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 15–23, 1995.

[11] Eric Rotenberg, Steve Bennett, and Jim Smith. Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching. In *Proceedings of the 1996International Symposium on Microarchitecture*, 1996.

[12] Quinn Jacobson, Eric Rotenberg, and James E. Smith. Path-Based Next Trace Prediction. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 14–23, 1997.

[13] James R. Larus. Whole Program Paths. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI '99)*, pages 259–269, 1999.

[14] Sriram Tallam, Xiangyu Zhang, and Rajiv Gupta. Extending Path Profiling across Loop Backedges and Procedure Boundaries. In *International Symposium on Code Generation and Optimization (CGO 2004)*, pages 251–262, 20-24 March 2004.

[15] Matthew Arnold and Barbara Ryder. A Framework for Reducing the cost of Instrumented Code. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Languages Design and Implementation (PLDI)*, pages 168–179, 20-22 June 2001.

[16] Rahul Joshi, Michael D. Bond, and Craig B. Zilles. Targeted Path Profiling: Lower Overhead Path Profiling for Staged Dynamic Optimization Systems. In *International Symposium on Code Generation and Optimization (CGO 2004)*, pages 239–250, 20-24 March 2004.

[17] Toshiaki Yasue, Toshio Suganuma, Hideaki Komatsu, and Toshio 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 03)*, Sept-Oct 2003.

[18] Sun Microsystems Inc. UltraSPARC User's Manual. 1997.

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

[20] Thomas M. Conte, Burzin Patel, Kishore N. Menezes, and J. Stan Cox. Hardware-Based Profiling: An Effective Technique for Profile-Driven Optimization. In *International Journal of Parallel Programming*, pages 187–206, Vol 24, No. 2, April 1996.

[21] Jeffery Dean, James E. Hicks, Carl A. Waldspurger, William E. Weihl, and George Chrysos. ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 292–302, December 13 1997.

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

[23] Craig Zilles and Gurindar Sohi. A Programmable Co-processor for Profiling. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, pages 241–253, January 2001.

[24] Thomas Ball, Peter Mataga, and Shmuel Sagiv. Edge Profiling versus Path Profiling: The Showdown. In *Symposium on Principles of Programming Languages*, pages 134–148, 1998.

[25] David Brooks, Vivek Tiwari, and Margaret 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.

[26] Ashutosh S. Dhodapkar and James E. Smith. Managing Multi-configuration Hardware via Dynamic Working Set Analysis. In *29th Annual International Symposium on Computer Architecture*, pages 233–244, 2002.

[27] Rajeev Balasubramonian, David H. Albonesi, Alper Buyuktosunoglu, and Sandhya Dwarkadas. Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures. In *International Symposium on*

Microarchitecture, pages 245–257, 2000.

[28] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the 10th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, pages 45 – 57, 2002.

[29] Timothy Sherwood, Suleyman Sair, and Brad Calder. Phase Tracking and Prediction. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 336 – 349, 2003.

[30] T. S. P. E. Corporation. SPEC CPU2000 benchmarks. http://www.spec.org/cpu2000/.

[31] Doug Burger, Todd M. Austin, and Steve Bennett. Evaluating Future Microprocessors: The SimpleScalar Toolset. In *Technical Report CS-TR-96-1308*. University of Wisconsin-Madison, July 1996.

[32] AJ KleinOsowski and David J. Lilja. MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research. In *Computer Architecture Letters*, page Volume 1, June 2002.

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

[34] 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.

[35] 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.

[36] James E. Smith Ashutosh S. Dhodapkar. Comparing Program Phase Detection Techniques. In *Proceedings of the 36th International Symposium on Microarchitecture*, 2003.

Appendix A

Results of experiments based on Plackett-Burman methodology that determine the importance of HPT paramters

Figure 20 shows results of experiments based on the Plackett-Burman design with the complete path execution count overlap percentage as the output metric. Numbers in the columns indicate the rank obtained and importance of the corresponding input parameter. The last column, which is obtained by summing up ranks obtained by a parameter for all benchmarks, reflects its overall importance. Figure 21 and 22 show the rankings for the HPT parameters when L2 cache miss and branch mis-prediction overlap percentages are used as output metrics. We find that HPT size has the highest importance for a path execution count and branch mis-prediction profile whereas HPT associativity is important while gathering a l2 cache miss profile. The HPT parameters, HPT size, associativity and the indexing scheme are consistently more important than the other parameters. As aforementioned, the low importance of L2 cache and branch predictor configuration indicates that the overlap percentages are independent of the machine configuration.

	gcc	gzip	mcf	parser	vortex	art	equake	bzip	ammp	mesa	perl	twolf	Total
HPT Size	3	2	2	1	1	3	1	2	1	1	1	1	**19**
HPT Associativity	1	1	1	2	2	2	5	1	2	2	3	3	**25**
Indexing Scheme	2	3	5	3	3	1	2	3	4	3	2	2	**33**
L2 Size	6	6	6	4	5	6	3	6	3	6	5	4	**60**
L2 Associativity	4	5	3	6	6	5	4	4	6	5	4	6	**58**
Branch Predictor	5	4	4	5	4	4	6	5	5	4	6	5	**57**

Figure 20: Results of the Plackett-Burman study with the complete path execution count overlap percentage used as the output metric. Numbers indicate ranks obtained by the input parameters.

	gcc	gzip	mcf	parser	vortex	art	equake	bzip	ammp	mesa	perl	twolf	Total
HPT Size	3	3	2	4	1	4	2	2	4	2	2	1	**30**
HPT Associativity	1	1	1	1	2	1	3	1	1	1	1	2	**16**
Indexing Scheme	5	2	3	3	3	2	1	3	2	3	3	4	**34**
L2 Size	4	4	6	5	6	5	4	6	6	5	4	6	**61**
L2 Associativity	2	5	5	6	5	6	5	4	3	4	6	5	**56**
Branch Predictor	6	6	4	2	4	3	6	5	5	6	5	3	**55**

Figure 21: Results of the Plackett-Burman study with the complete L2 cache miss overlap percentage used as the output metric.

	gcc	gzip	mcf	parser	vortex	art	equake	bzip	ammp	mesa	perl	twolf	Total
HPT Size	2	4	2	1	1	3	1	1	1	1	2	1	**20**
HPT Associativity	1	1	1	2	2	1	4	2	2	2	3	6	**27**
Indexing Scheme	6	2	4	4	3	2	3	4	3	3	1	2	**37**
L2 Size	3	6	5	5	5	6	2	6	6	6	4	5	**59**
L2 Associativity	5	3	6	6	6	4	6	5	5	5	5	4	**60**
Branch Predictor	4	5	3	3	4	5	5	3	4	4	6	3	**49**

Figure 22: Results of the Plackett-Burman study with the complete branch misprediction overlap percentage used as the output metric.