ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors

Jeffrey Dean Jamey Hicks Carl A. Waldspurger William E. Weihl George Chrysos
Digital Equipment Corporation

Abstract

Profile information is valuable for identifying performance bottlenecks and guiding optimizations. Periodic sampling of a processor's performance monitoring hardware is an effective, unobtrusive way to obtain detailed profiles. Unfortunately, existing hardware simply counts *events*, such as cache misses and branch mispredictions, and cannot accurately attribute these events to instructions, especially on out-of-order machines. We propose an alternative approach that samples *instructions* by attaching a *ProfileMe* tag early in the processor pipeline. As the instruction moves through the pipeline, a detailed record of all interesting events and pipeline stage latencies is collected. We also propose *paired sampling*, a technique that captures interactions between instructions that may execute in parallel, revealing information about useful concurrency and the utilization of various pipeline stages while an instruction is in flight. We describe an inexpensive hardware implementation of our approach, outline a variety of software techniques to extract useful profile information from this hardware, and explain several ways in which this information can provide valuable feedback for programmers and optimizers.

1 Introduction

Processors are getting faster, yet application performance is not keeping pace. On large commercial applications, average cycles-per-instruction (CPI) values may be as high as 2.5 or 3. With 4-way instruction issue, a CPI of 3 means that only one issue slot in every 12 is being put to good use!

It is common to blame such problems on memory, and in fact most applications spend many cycles waiting for memory, but other problems, such as branch mispredictions, also waste cycles. To improve the performance of a particular application, we need to know which instructions are stalling and why.

In this paper, we describe hardware and software support for a sampling-based profiling system that provides detailed instruction-level information on processors that can execute instructions speculatively and out of order. Our approach, called *ProfileMe*, consists of two parts: an *instruction sampling* technique, which captures information about individual instructions (*e.g.*, cache miss rates for each instruction), and a *paired sampling* technique, which captures information about the interactions among instructions (*e.g.*, concurrency levels). *ProfileMe* has several key advantages over previous techniques: (1) it collects complete information about each instruction, rather than sampling a small number of events at a time; (2) it accurately attributes events to instructions; (3) it collects information about all instructions, including instructions in uninterruptible sections of code; and (4) it collects information about useful concurrency, thus helping to pinpoint real bottlenecks.

Sampling has a number of advantages over other profiling methods, such as simulation or instrumentation: it works on unmodified programs, it profiles complete systems, and it can have very low overhead. Indeed, recent work [2] has shown that low-overhead sampling-based profiling can reveal detailed instruction-level information about pipeline stalls and their causes—but that work is limited to in-order processors, and its techniques do not extend to out-of-order processors.

Most modern microprocessors, including the Alpha 21164 [8], Pentium Pro [11] and R10000 [14], provide performance counters that count a variety of events (*e.g.*, branch mispredicts or data cache misses) and deliver an interrupt when the counters overflow. Event counters provide useful aggregate information, such as the total number of branch mispredicts during a program run. However, as we discuss later, they do not give accurate information about individual instructions, such as the mispredict rate for a single branch.

Our method is a departure from traditional performance counters. Rather than counting *events* and sampling the program counter when the event counters overflow, we sample *instructions*. At random intervals, we select an instruction; as it executes, we record information about its execution in internal registers. Information recorded includes the instruction's PC, the number of cycles spent in each pipeline stage, whether it suffered I-cache or D-cache misses, the effective address of a memory operand or branch target, and whether it retired or why it aborted. After the instruction completes, we generate an interrupt and deliver the recorded information to software.

Our core instruction sampling technique captures detailed information about a single instruction, and is useful for identifying instructions that remain in the pipeline for a long time. On an in-order machine, this information is sufficient to identify bottlenecks. However, on an out-of-order machine, the concurrency provided by executing instructions out-of-order can mask some stalls.

All of the authors can be reached at DIGITAL. Dean is at the Western Research Laboratory (jdean@pa.dec.com), Hicks is at the Cambridge Research Laboratory (jamey@crl.dec.com), Waldspurger and Weihl are at the Systems Research Center ({caw,weihl}@pa.dec.com), and Chrysos is with the Advanced Development group of Digital Semiconductor (chrysos@vssad.hlo.dec.com). More information about this line of profiling research at DIGITAL can be found on the Web at `http://www.research.digital.com/SRC/dcpi/`.

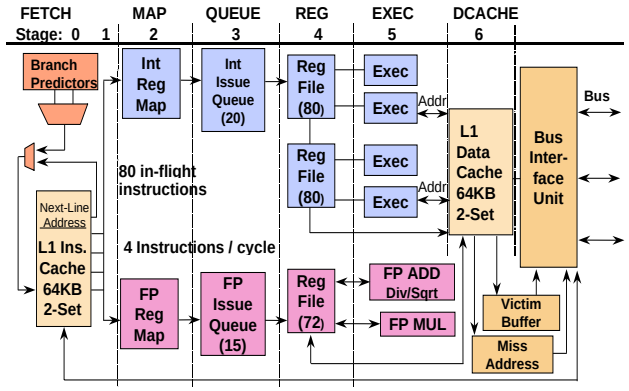

Figure 1: Alpha 21264 Processor Pipeline.

To identify real bottlenecks, instruction-level information must be combined with information about *useful concurrency* (*e.g.*, while a given instruction is in flight, how many issue slots are used by instructions that ultimately retire). We use *paired sampling*, a nested form of sampling, to measure useful concurrency: for each profiled instruction, the dynamic window of instructions that may execute concurrently with it is also randomly sampled, forming a *sample pair*. Paired sampling exposes the interactions among instructions, enabling a wide variety of interesting concurrency and utilization metrics to be computed.

The remainder of this paper describes our approach in more detail. Section 2 explains why performance on out-of-order processors is hard to understand and why event counters are insufficient. Section 3 presents an overview of our approach. Section 4 describes the hardware requirements of *ProfileMe*, while Section 5 discusses how profiling software can exploit this hardware to collect profiles and perform various analyses to extract useful information. Section 6 discusses alternative metrics for identifying bottlenecks. Section 7 discusses optimizations that can benefit from the information produced by our approach. Related work is examined in Section 8. Finally, we summarize our conclusions in Section 9.

2 Problem

The behavior of programs on an out-of-order processor can be subtle and difficult to understand. To motivate our profiling mechanism, we begin this section by reviewing the flow of instructions in out-of-order processors. Using the Alpha 21264 processor as a concrete example, we discuss the myriad ways in which instructions may be delayed. We then demonstrate the problems with using event counters to understand the performance of programs executed on processors with out-of-order and speculative execution.

2.1 Superscalar Out-of-Order Architecture

An out-of-order execution processor fetches and retires instructions in order, but executes them according to their data dependencies. Figure 1 depicts the pipeline of the Alpha 21264 processor [12]. Each cycle, the first stage of the pipeline fetches and decodes a group of instructions from the instruction cache. The instruction decoder identifies which instructions in the fetched group are part of the instruction stream. Because it takes multiple cycles to resolve the PC of the next instruction to fetch, the next PC is predicted by a branch or jump predictor. If the prediction is incorrect, the processor will abort the mispredicted instructions (the *bad path*) and will restart fetching instructions on the *good path*.

To allow instructions to execute out of order, registers are renamed to prevent write-after-read and write-after-write conflicts. This renaming is accomplished by *mapping* logical to physical registers. Thus two instructions that write the same architectural register can safely execute out of order because they will write different physical registers, and consumers of those architectural registers will get the proper values. Instructions are fetched and mapped in order.

A mapped instruction resides in the issue queue until its operands have been computed and a functional unit of the appropriate type is available. After instructions have executed, they are marked as ready to retire and will be retired by the processor when all previous ready-to-retire instructions in program order have been retired. Upon retirement, the processor commits the changes of the instruction to the architectural state and releases resources consumed by the instruction.

In some cases, such as when a branch is mispredicted, instructions must be trapped or discarded. When this occurs, the speculative architectural state is rolled back and fetching continues after the most recent untrapped instruction, *e.g.*, the actual branch target.

Numerous events may delay the execution of an instruction. In the front of the pipeline, the fetcher may stall due to an I-cache miss or may fetch bad-path instructions due to a misprediction. The mapper may stall due to a lack of free physical registers or of free slots in the issue queue. Instructions in the issue queue may wait for their register dependences to be satisfied or for the availability of functional units. Instructions may stall due to data cache misses. Instructions may trap because they were speculatively issued down a bad path, or because the processor took an interrupt.

Many of these events are difficult to predict statically, and all of them can degrade performance. At the same time, the ability of an out-of-order processor to issue a later instruction while an earlier instruction is stalled can mean that some delays are hidden. Our approach identifies which instructions are delayed and how that affects the running time of a program, enabling programmers or optimization tools to improve performance.

2.2 Event Counter Limitations

As mentioned earlier, many existing processors provide event counters to help measure the performance of programs. Unfortunately, event counters do not accurately attribute events to instructions: the instruction that caused

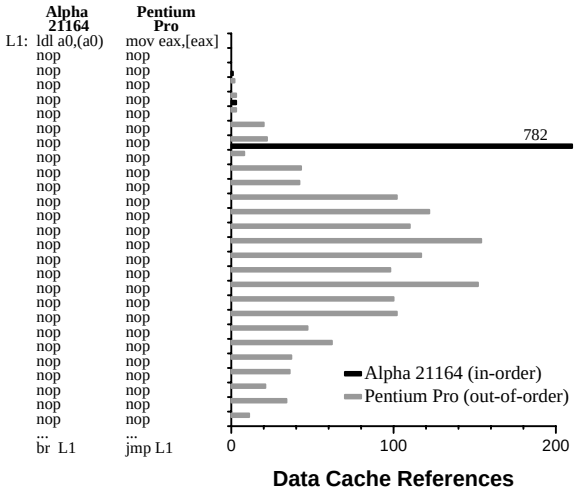

Figure 2: Distribution of PC values delivered to performance counter interrupt routines on in-order and out-of-order processors

an event resulting in an event-counter overflow is usually earlier, by an unpredictable amount, than the instruction whose PC is delivered to the interrupt handler. Out-of-order and speculative execution amplify this problem, but it is present even on in-order machines.

Figure 2 compares the program counter value delivered to the performance counter interrupt handler when monitoring D-cache-reference events on the in-order Alpha 21164 and on the out-of-order Pentium Pro. The example program, shown on the left side of the figure, consists of a loop containing a single memory read instruction, followed by hundreds of `nop` instructions.

On the in-order Alpha, almost all performance counter events are attributed to the instruction that is executing six cycles after the event, resulting in a large peak of samples on the seventh instruction after the load. This skewed distribution is not ideal, but static analysis can sometimes work backwards from the single large peak to identify the instruction that caused the event.

On the Pentium Pro, the event samples are widely distributed over the next 25 instructions. This smeared distribution of samples makes it nearly impossible to attribute an event to the instruction that caused it. Similar behavior occurs when counting other hardware events.

Aside from the complicated distribution of event samples, hardware event counters suffer from several additional problems. First, performance-counter interrupts may be deferred when running non-interruptible or high-priority system code, such as Alpha PALcode [8]. As a result, samples for events will be incorrectly attributed to the instruction following the high-priority code, resulting in undesirable "blind spots".

In addition, there are typically many more events of interest than there are hardware counters, making it impossible to concurrently monitor all interesting events. The increasing complexity of processors is likely to exacerbate this problem. Moreover, event counters only record the fact that an event occurred; they do not provide any context about the event. For many kinds of events, additional information, such as the latency to service a cache miss, would be extremely useful.

Our approach avoids these problems on both in-order and out-of-order processors. Profiling instructions instead of events completely eliminates the difficulties with event attribution. Similarly, sampling information about instructions in hardware eliminates blind spots. Instruction-based profiling also permits a complete set of correlated events to be collected for each instruction, avoiding the need to process and correlate interrupts from multiple event counters.

3 Overview of Approach

To understand a program's performance, we would like to gather information at two levels:

- *Aggregate information*, summarizing performance statistics over an entire workload, an individual program, a procedure, or a smaller unit such as a loop.
- *Instruction-level information*, showing the average behavior of each instruction.

We are interested in gathering the key performance metrics that aid our identification of performance bottlenecks. As described earlier, *ProfileMe* involves *sampling instructions*: randomly choosing instructions to be profiled and recording information about their execution. By aggregating samples from repeated execution of the same instruction, we can estimate many interesting metrics for each instruction (such as I-cache miss rate, branch mispredict rate for branches, D-cache miss rate for memory operations, average latency in cycles from fetch to retire, etc.) Our approach makes gathering this instruction-level information both possible and relatively inexpensive. Information about individual instructions can easily be aggregated to summarize the behavior of larger units of code.

ProfileMe consists of both hardware and software. Hardware is needed to select instructions to be profiled, to record information about profiled instructions, and to generate an interrupt when a profiled instruction completes so that the recorded information can be delivered to software. Software is needed to control the sampling interval to ensure a random sample of the instruction stream, to field the interrupts generated by the hardware so that information about profiled instructions can be saved in a profile database, and to analyze profile data to identify performance problems.

To get a statistically meaningful estimate of program behavior, the profiling software requires a random sample of the instruction stream. For many analyses, the profiling software needs a random sample of just the retired instructions. As described in more detail later, our approach

involves randomly sampling fetched and decoded instructions, discarding in hardware the instructions not on the current control path. Note that this does not impact the random sampling of retired instructions: selecting only retired instructions from a random sampling of fetched instructions yields a random sampling of retired instructions, just as if the hardware were providing a random sampling of retired instructions directly.

Note that an instruction chosen for profiling may later abort rather than retiring (*e.g.*, due to speculative execution down a bad path), yet our profiling hardware will deliver performance information even for aborted instructions. At first glance, this may seem counterproductive, but bad path instructions consume machine resources and can cause performance bottlenecks. Providing samples for all instructions (along with retired/aborted status information) permits an analysis of which instructions are aborting and why, rather than making aborted instructions completely invisible to profiling.

As mentioned earlier, sampling individual instructions is important, but is not sufficient to accurately identify bottlenecks in out-of-order processors. We augment our core instruction sampling mechanism with *paired sampling*, which permits the sampling of multiple instructions that may be in flight concurrently. Paired sampling provides essential information for analyzing the interactions among instructions. Data captured from pairs of profiled instructions has many valuable uses. It can be analyzed to measure useful concurrency levels, making it possible to find true bottlenecks, identified by long stalls coupled with low useful concurrency. Paired samples can be used to measure edge frequencies of the control-flow-graphs and call-graph of a program and can also improve the accuracy of sampling-based path-profiling. Finally, it may be possible to statistically reconstruct detailed processor pipeline states, by clustering pairs of samples based on recent branch history, stalls, or other events.

There are other possible techniques for obtaining concurrency information. One naive approach would be to snapshot the entire pipeline state, directly revealing the set of concurrently executing instructions at a given point in time. However, this would be extremely costly in hardware, time, and space. Worse yet, this approach is insufficient, since only those instructions that retire should be counted as useful, and instructions that are currently in flight may still be aborted. Paired sampling yields better information about useful concurrency than a complete pipeline snapshot, with significantly less cost and complexity.

In the next section, we describe the hardware needed for *ProfileMe*, including both the core instruction sampling mechanisms and paired sampling. Section 5 discusses in more detail how this hardware can be used to provide useful profiling information for a variety of performance understanding and optimization tasks.

4 ProfileMe Hardware

The hardware required for sampling instruction execution is extremely modest, and scales linearly with the number of in-flight instructions that may be sampled simultaneously. By restricting the number of instructions simultaneously profiled—to one or two instructions instead of all in-flight instructions—we reduce the hardware overhead of profiling. The run-time profiling overhead may be decreased arbitrarily by reducing the sampling rate, although previous work has shown that high frequency sampling can be implemented with relatively low overhead through careful programming [2].

In the subsections below we describe the hardware needed to sample the execution of a single instruction, the additional hardware needed for paired sampling, and how replicating the hardware can reduce the software overhead substantially.

4.1 Instruction Sampling

Four pieces of hardware are needed to sample the execution of a single instruction: a way of selecting instructions to be profiled, a way of tagging profiled instructions in the processor pipeline, registers to record data about a profiled instruction during its execution, and a way of generating an interrupt when a profiled instruction completes so that the recorded information can be captured by profiling software.

4.1.1 Choosing Profiled Instructions

In the front of the pipeline, we need hardware to choose instructions to be profiled. To ensure that instructions are chosen randomly, we add a software-writable *Fetched Instruction Counter* to the processor's instruction fetcher. At the beginning of each sampling interval, the profiling software writes a random value to the counter. The counter decrements once for each instruction fetched on the current control path; the instruction fetched when the counter reaches zero is selected for profiling.

Counting fetched instructions on the current control path is actually somewhat complicated, since a variable number of instructions (zero to four on the Alpha 21264) on the current control path are fetched each cycle. This complexity can be avoided by instead counting "fetch opportunities" and selecting a particular fetch opportunity to be profiled. A given fetch opportunity may contain an instruction on the current control path, an instruction not on the current control path (but in the same fetch block as instructions that are on the current control path), or no instruction at all (*e.g.*, if the fetcher is stalled waiting for an I-cache miss). Choosing instructions to be profiled based on counting fetch opportunities simplifies the hardware for triggering profiling, but may result in a significant number of samples that do not contain instructions on the current control path, effectively reducing the sampling rate for most purposes.

4.1.2 The ProfileMe Tag

We augment the decoded instruction state with a *ProfileMe Tag* that is passed through the processor pipeline

Measured Latency	Explanation
Fetch→Map	Stalls due to lack of physical registers or issue queue slots
Map→Data ready	Stalls due to data dependences
Data ready→Issue	Stalls due to execution resouce contention
Issue→Retire ready	Execution latency
Retire ready→Retire	Stalls due to prior unretired instructions
Load issue→ Completion	Memory system latency (Alpha allows loads to retire before value returns, so this may be later than Issue to Retire-Ready)

Table 1: **Latency Measurements.** Pipeline stage latencies are useful for identifying and diagnosing stalls and delays.

with every in-flight instruction. The ProfileMe Tag is set for an instruction when it is chosen to be profiled. In the lowest-cost implementation, the ProfileMe Tag is set for at most one in-flight instruction at a time. In this case, a single bit suffices for the tag; for paired sampling or, in general, sampling N potentially concurrent instructions, more bits are needed.

4.1.3 Instruction-Level Data Collection

When the ProfileMe Tag is set for an instruction, the profiling hardware records events, latencies, addresses, *etc.*, associated with that instruction, in a set of internal processor Profile Registers indexed by the tag. The information collected for profiled instructions will vary across processor implementations. This subsection sketches the information that is important for profiling in current out-of-order execution processors and the hardware needed to gather it. Note that it is relatively easy to have the hardware record additional events or other information about the instruction in the Profile Registers.

The *Profiled PC Register* records the address of the profiled instruction. The *Profiled Address Register* records the effective address of load and store instructions and the target address of indirect jump instructions. The *Profiled Context Register* records the address space number or other identification of the process or thread executing the profiled instruction.

A *Profiled Event Register* is a bit-field that records whether various events were experienced by the instruction. Events include: I-cache and D-cache misses, branch taken, branch mispredictions, various resource conflicts, memory traps, whether the instruction retired, trap reason, *etc*. When the fetcher selects an instruction it clears the Profiled Event Register.

A *Profiled Path Register* is used to capture recent branch-taken information from the processor's global branch history register. This information can be used to determine the code path taken in reaching the profiled instruction, as described in Section 5.3.

A set of *Latency Registers* records the number of cycles spent by the instruction in each pipeline phase. Table 1 lists some of the latencies of interest on the Alpha 21264, along with a description of the problems they help diagnose.

4.1.4 Capturing Profile Data

The ProfileMe Tag remains set for a profiled instruction until it retires or aborts. After all processor activity pertaining to the instruction or instruction pair has completed, an interrupt is generated. Profiling software fields the interrupt, reads the Profile Registers, and resets the Fetched Instruction Counter to a pseudo-random value.

Note that some information to be recorded in the ProfileMe registers needs to travel a long distance on the chip. Instead of increasing cycle time to accommodate long wires to deliver such information, latches can be inserted to pipeline the signal delivery to the ProfileMe registers. In this case, the interrupt that signals the collection of a ProfileMe sample must be delayed by the hardware until all the appropriate signals have had time to reach the ProfileMe registers.

4.2 Paired Sampling

Paired sampling requires the ability to sample two potentially concurrent instructions. In addition, we need information about the overlap between the instructions in a sample pair. We make the following extensions to the core instruction sampling mechanisms.

To choose instructions in a sample pair, we need to be able to specify major and minor sample intervals. The major interval specifies the number of fetched instructions until the first instruction of a pair is chosen. The minor interval specifies the number of fetched instructions between the first and second profiled instructions in a sample pair. As before, the profiling software randomizes the major interval; in addition, it randomizes the minor interval to ensure a random sample of pairs of instructions that might be in flight at the same time.

To tag profiled instructions, we need a ProfileMe tag with at least two bits to distinguish two instructions that may be simultaneously in flight.

To record information about both instructions in a sample pair, we need two sets of Profile Registers, indexed by the ProfileMe tag, and the signals carrying information to the registers must also carry the tag along.

To allow profiling software to capture the recorded information, an interrupt should be generated after both sampled instructions have finished executing and all relevant data has been recorded in their profile registers.

Finally, we need to capture the latency between the two sampled instructions (*i.e.*, the number of cycles between the times when the two sampled instructions were fetched). This information is needed to determine how the instructions overlapped in the processor pipeline.

4.3 Amortizing Interrupt Delivery Costs

Previous work has shown that the cost of delivering and dispatching performance counter interrupts is one of the most significant sources of overhead in sampling-based profiling systems [2]. The ProfileMe approach makes it possible to reduce this overhead by providing additional

hardware copies of profile registers and buffering multiple samples before delivering a performance counter interrupt. Software can then read the profile records for several samples at once, thereby amortizing the performance counter interrupt delivery cost.

5 Profiling Software

The hardware mechanisms presented in the previous section can be utilized in various ways. One approach is to gather samples several thousand times per second, logging them in memory or on disk for later processing. Space consumption can be reduced by processing some of the information as the samples are gathered, such as by aggregating samples for the same instruction, as is done for event-counter-based samples in the DCPI system [2]. Overhead can be further reduced by ignoring certain fields of the profile information except when gathering data for specific optimizations. Once the profile information has been collected, it can be analyzed to extract useful information. Several analyses are described in the following subsections.

5.1 Estimating Event Frequencies

Samples for individual instructions can be used to estimate various instruction-level event frequencies as follows. Assume an average sampling rate of one sample every S fetched instructions. Suppose that instruction I is fetched N times and a fraction f of those have a given property P (*e.g.*, "retired," or "missed in the D-cache"). Let the random variable k be the number of samples that have property P. We estimate the actual number of fetches of I with property P as kS.

The expected value of kS is fN, *i.e.*, the actual number of fetches of I with property P. The coefficient of variation of kS is $\sigma_{kS}/E[kS] = \sqrt{1/N}\sqrt{(S-f)/f)}$, which is approximately $\sqrt{S/fN}$ (since $S \gg f$). Note that this is simply $\sqrt{1/E[k]}$. In other words, relative accuracy improves with the square root of the (expected number of) samples with property P. Infrequent events or long sampling intervals require longer runs to get enough samples for accurate estimates.

Continue working on this part

We extended a cycle-accurate simulator for the Alpha 21264 processor to gather ProfileMe samples. We sampled every 10^4 to 10^6 fetched instructions from a traces of 10^7 to 10^9 instructions from a suite of benchmarks including GO, POVRAY, LI, and PERL. By sampling retired instruction samples via ProfileMe, we were able to accurately estimate the execution frequency of each instruction of each of the SPECint95 programs.

Figure 3 illustrates how the estimated counts for each PC converge on the measured value as the number of samples increases. The left column shows the results for the retire count for each instruction and the right column shows the results for D-cache miss counts. Each point represents a single instruction. All graphs show the ratio of the estimated value to the actual value on the y-axis; the top two rows use a log scale, and the bottom row uses a linear scale. In the top row, the x-axis shows the total number of samples for each instruction; this is typically more than the number of samples in which the instruction retired or suffered a D-cache miss. In the bottom two rows, the x-axis shows the number of samples with the relevant property (retired in the left column, D-cache miss in the right). These two graphs show a clear improvement in accuracy as the number of samples with the relevant property increases.

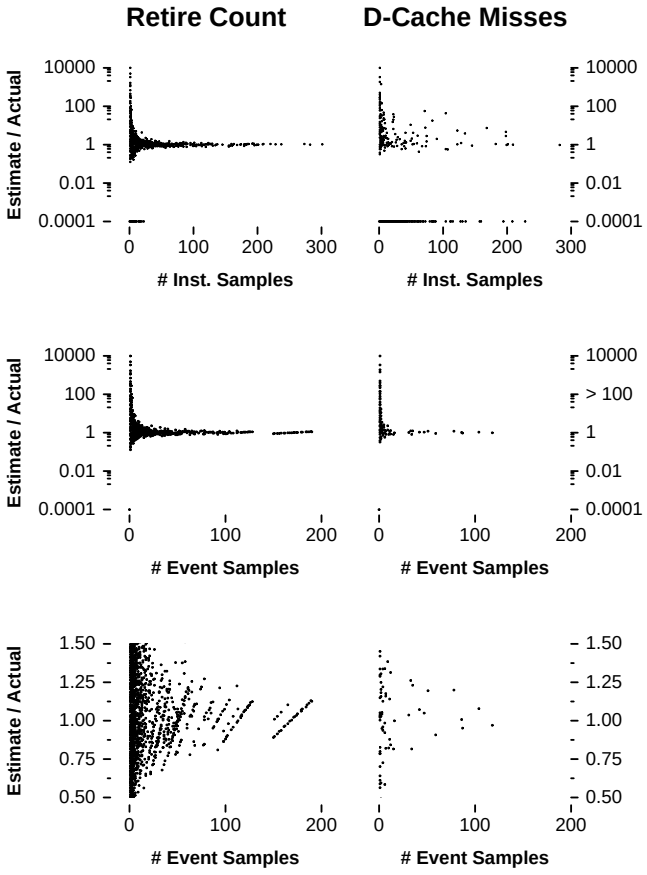

Figure 3: Convergence of Retire Count and D-Cache Miss Rate

Note to us: if we plot $y = (1/\sqrt{x}) + 1$ (and reflect it as well around y=1 to get the lower envelope), that will give us the envelope corresponding to a single standard deviation from the expected value – and 2/3 of the points should be within that envelope. Any envelope that includes a fixed percentage of the points will follow a $1/\sqrt{x}$ curve.

5.2 Estimating Interaction Frequencies

Paired samples can be used to estimate a wide range of concurrency and utilization metrics. This section explains in more detail how paired sampling works and how paired samples can be analyzed to derive statistical estimates of concurrency and resource utilization levels while an instruction is in flight.

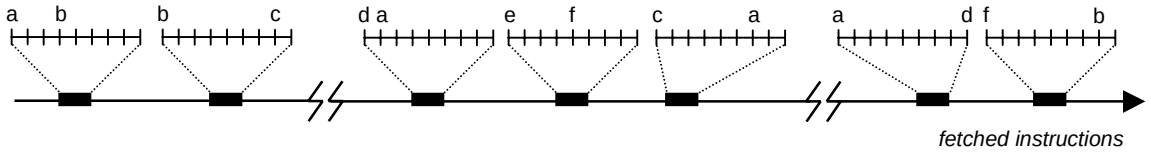

Figure 4: **Recursive Sampling Example.** Two levels of sampling are depicted: (1) a major inter-pair sampling interval between windows (black regions), and (2) a minor intra-pair sampling interval within each window (expanded view).

5.2.1 Recursive Sampling

Paired sampling enables ProfileMe records to be collected for two instructions that may be in flight simultaneously. A key application of paired sampling hardware is *recursive sampling*: for each profiled instruction, the set of other instructions that can potentially execute concurrently with it is directly sampled. Recursive sampling is based on the same statistical arguments that justify ordinary sampling. Because it involves two levels of sampling, it will be most effective for heavily executed code.

Figure 4 illustrates an example of recursive sampling. The arrow indicates the sequence of instructions that are fetched (in program order) during some dynamic execution. The *first* level of sampling is represented by the small black regions of fetched instructions; their spacing corresponds to the *major* sampling interval.

The *second* level of sampling is depicted by the expanded window of instructions shown above each black region. The first labelled instruction in each window represents the instruction selected by the first level of sampling. The second labelled instruction in each window is determined by the *minor* sampling interval.

Denote the size of the window of potentially concurrent instructions by W. For each paired sample $\langle I_1, I_2 \rangle$, recursive sampling is implemented by setting the intra-pair fetch distance to a pseudo-random number uniformly distributed between 1 and W. The window size is fixed, and its value is conservatively chosen to include all subsequent instructions that may be simultaneously in flight with the first-level sample. In general, an appropriate value for W depends on the maximum number of in-flight instructions supported by the processor. (On most processors, this is at most a few hundred instructions.) However, the minor intra-pair sampling interval will typically be orders of magnitude smaller than the major inter-pair interval.

5.2.2 Analyzing Sample Pairs

For a given profiled instruction I, the set of potentially concurrent instructions are those that may be co-resident in the processor pipeline with I during any dynamic execution. This includes instructions that may be in various stages of execution *before* I is fetched, as well as instructions that are fetched *after* I.

Figure 5(a) shows how the sample pairs from Figure 4 can be analyzed to recover information about instructions in a window of $\pm W$ potentially concurrent instructions around I. In this example, we consider all pairs $\langle I_1, I_2 \rangle$ containing the instruction labelled a. When $I_1 = a$, I_2 is a random sample in the window *after* a; when $I_2 = a$, I_1 is a random sample in the window *before* a. By considering each pair twice, random samples are uniformly distributed over the set of all potentially concurrent instructions.

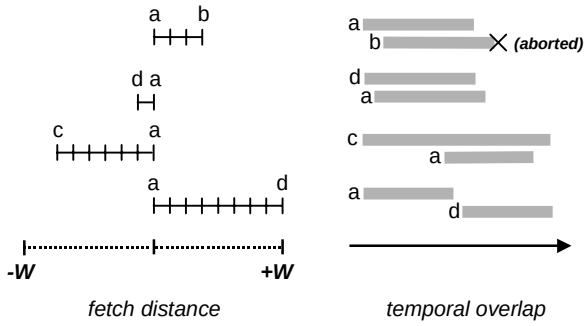

Figure 5: **Paired Sample Analysis.** (a) Sample pairs containing instruction a form a random sample of instructions in the window of $\pm W$ potentially concurrent instructions around a. (b) Execution timings for the instructions in each pair enables their temporal overlap to be determined.

The ProfileMe data recorded for each paired sample $\langle I_1, I_2 \rangle$ includes latency registers that indicate where I_1 and I_2 were in the processor pipeline at each point in time, as well as the intra-pair fetch latency that allows the two sets of latency registers to be correlated. The ProfileMe records for I_1 and I_2 also indicate whether they retired or aborted. This information can be used to determine whether or not the two instructions in a sample pair overlapped in time, as illustrated in Figure 5(b). For example, the data associated with sample pairs $\langle d, a \rangle$ and $\langle c, a \rangle$ reveal varying degrees of execution overlap, and there was no overlap for $\langle a, d \rangle$. Similarly, the data for $\langle a, b \rangle$ indicates that while the executions of a and b overlapped, b was subsequently aborted.

The definition of *overlap* can be altered to focus on particular aspects of concurrent execution. The subsection below uses a particular definition to estimate the number of issue slots wasted while a given instruction was in flight. Other useful definitions of *overlap* include: one instruction issued while the other was stalled in the issue queue; one instruction retired within a fixed number of cycles of the

other; and both instructions were using arithmetic units at the same time.

5.2.3 Example Metric: Wasted Issue Slots

To pinpoint bottlenecks, we need to identify instructions with high execution counts, long latencies, and low levels of useful concurrency. One interesting measure of concurrency is the total number of issue slots "wasted" while an instruction was in progress. To compute this metric, we define *useful overlap* to mean that while an instruction was in progress, the instruction paired with it in a sample pair issued and subsequently retired. Here we define "in progress" to mean the time between when an instruction is fetched and when it becomes ready to retire; we do not include time spent waiting to retire, since such delays are purely due to stalls by earlier instructions.

Fix an instruction I. Let U_I denote the number of samples of the form $\langle I_1, I_2 \rangle$, such that $I_1 = I$ or $I_2 = I$, and the executions of I_1 and I_2 exhibited useful overlap. We can now statistically estimate the number of useful instructions that issued while I was in flight by multiplying the number of matching samples by the size of the sampled window of potential concurrency; *i.e.*, the number of productive issue slots is $W \times U_I$.

By additionally estimating the cumulative latency L_I of instruction I measured in issue slots (*e.g.*, four per cycle sustainable on the Alpha 21264), we can estimate the total number of wasted issue slots during all executions of I as $L_I - W \times U_I$. This value can easily be scaled to express the average number (or percentage) of issue slots wasted per execution of I. Alternative definitions of useful overlap could account for the duration (in cycles) of each overlap.

Fortunately, the components of metrics such as wasted issue slots can be aggregated incrementally, enabling compact storage during data collection (as in the DCPI profiling system [2]).

5.2.4 Flexible Support for Concurrency Metrics

Many other concurrency metrics can be estimated in a similar manner, such as the number of instructions that retired while I was in flight, or the number of instructions that issued around I. IPC levels in the neighborhood of I can be measured by counting the number of pairs in which both instructions retire within a fixed number of cycles of each other.

More detailed information can also be extracted or aggregated, such as the average utilization of a particular functional unit while I was in a given pipeline stage. It is also possible to use ProfileMe information to separate out interesting cases when aggregating concurrency information. For example, it may be useful to compute the average concurrency level when I hits in the cache, and then compare it to the case where I suffers a cache miss. Other interesting aspects to examine for correlation with concurrency levels include register dependencies, branch-mispredict stalls, and recent branch history.

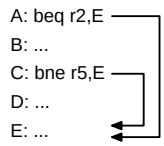

Figure 6: Ambiguous Control Flow Merge.

In general, paired sampling provides significant flexibility, allowing a variety of different metrics to be computed statistically by sampling the value of any function that can be expressed as $f(I_1, I_2)$ over a window of W instructions. In contrast to *ad hoc* hardware mechanisms, this flexibility makes paired sampling an attractive choice for capturing concurrency information on complex, out-of-order processors, because it leaves the door open for the design of new metrics and analysis techniques.

5.3 Path Profiles

Many compiler optimizations, such as trace scheduling [9] and hot-cold optimization [5], rely on predicting which paths through the program will be heavily executed. Frequently executed paths were conventionally estimated by gathering basic block or control-flow graph edge counts and then using these counts to infer the hot paths. More recently, Ball and Larus [3] and Young *et al.* [19] proposed more advanced profiling methods to gather detailed path information directly. Although such profiles give exact path counts, they require instrumenting the program and are therefore expensive and intrusive. By capturing information about the processor's global branch history and combining this with static analysis of a program's control flow graph (CFG), we can use the ProfileMe hardware to perform sampling-based profiling of program execution path segments.

Most modern microprocessors store the directions of the last N conditional branches in a *global branch history register* as part of their branch prediction hardware. By capturing the contents of this register at instruction fetch time as part of the profile record, we can analyze the CFG starting at a sampled instruction and infer the possible paths through the last N branches that the processor executed. There may be multiple paths that reach the sampled instruction and are consistent with the sampled branch history, because the history register only contains the directions of the branches and not their PCs[1]. For example, consider the CFG shown in Figure 6. Both paths AE and $ABCE$ are possible given a sample whose PC is E and whose global branch history ends in a 1 (*i.e.*, branch taken).

To explore the effectiveness of this analysis in identifying the true program path given the PC and global branch history contained in a ProfileMe sample, we traced each

[1] Asynchronous events that cause code with branches to be executed, such as interrupts or context switches, also pollute the branch history bits, but these events should be relatively infrequent. Since the goal is to identify high frequency paths, low frequency paths generated by "noisy" branch history bits will be largely ignored.

of the programs in the SPECint95 benchmark suite. For each instruction in the trace, we computed the value of the branch history bits at that point, and walked backwards through the program's CFG to identify path segments that could have been executed (*i.e.*, where the particular branch directions on the path are consistent with the branch directions indicated in the history bits). Ideally, this analysis would identify just one potential path segment corresponding to the true execution path.

We compared three different schemes for constructing paths: *Execution counts*, which ignores the branch history bits, using the execution frequencies at each control-flow merge point to identify the most likely path (trace scheduling compilers use a similar technique to construct traces from basic-block execution-count profiles); *History bits*, which uses the global branch history bits to restrict the set of paths that are examined; and *History bits + paired sampling*, which augments *History bits* by discarding paths that do not contain the other PC from a paired sample (with the intra-pair distance randomly varied between 1 and 50 fetched instructions).

The results are shown in Figure 7. The graphs depict the accuracy of each of the three different schemes, as a function of the length of the branch history that was examined. The vertical axis shows the success rate of the reconstruction over the entire SPECint95 suite (using traces of 33 to 83 million instructions for each benchmark), where success is defined as a case where only one path is produced by the analysis and the path corresponds to the actual execution path. The left graph of Figure 7 depicts an intraprocedural experiment, where we finished a path when either the path had grown backward to include a fixed number of branches corresponding to the length of the branch history being examined or when the path reached the beginning of the routine. This corresponds to the kinds of paths that might be used to guide an intraprocedural trace scheduler. The right graph of Figure 7 shows an interprocedural experiment, where the analysis continued to the callers of a routine when the beginning of the routine was reached, so that a path was only complete if it contained a number of branches equal to the length of the branch history being examined[2].

In general, the accuracy of all three methods decreases as we attempt to infer longer execution path segments, but using the branch history noticeably improves the accuracy of paths compared to using just execution frequencies (especially for longer paths). Paired sampling is able to further improve the accuracy. All three methods are considerably less accurate when trying to construct interprocedural paths, but the results still indicate that branch history bits significantly improve accuracy and that paired sampling further improves accuracy. How much this more accurate path information will translate into improved generated code remains to be seen.

[2]Note that in either case, when a procedure call instruction is encountered during the backwards traversal of the CFG, the analysis continues at the exits of the called procedure and can eventually return to the calling procedure if there is sufficient branch history to work backwards through the entire called routine. The difference between the intraprocedural and interprocedural versions is whether the analysis continues at all of the callers of the routine if the beginning of the routine in which we started our analysis is reached.

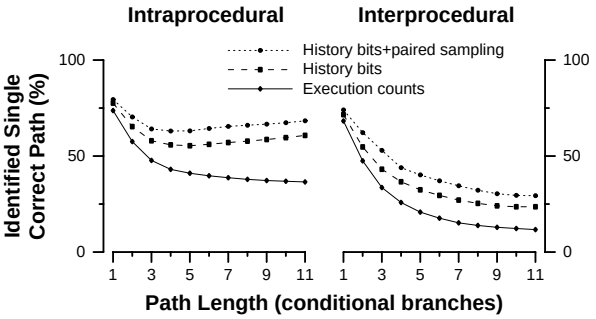

Figure 7: Effectiveness of path reconstruction strategies

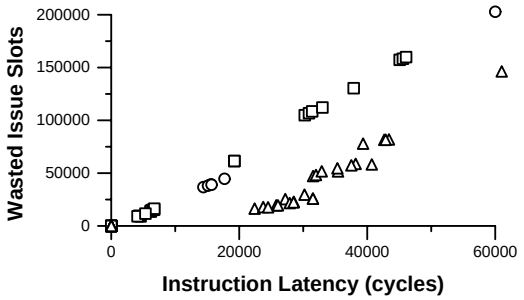

Figure 8: **Identifying Bottlenecks.** Instruction latency alone cannot accurately identify bottlenecks due to out-of-order execution that masks stalls.

6 Metrics for Identifying Bottlenecks

When we started this work, we believed that information about concurrency would be needed to identify bottlenecks accurately; this motivated us to invent paired sampling. Given that paired sampling imposes some additional costs, one might ask whether it is really necessary. To explore this question, we examined whether the cumulative latency of each instruction (which can be estimated from individual instruction samples, without paired sampling) would pinpoint bottlenecks as effectively as would the total number of issue slots wasted while each instruction was in progress.

Figure 8 shows results from running a simple program consisting of three separate loops on an Alpha 21264 simulator. Each instruction in the program is represented by a symbol (circle, square, or triangle, corresponding to the three separate loops).

In the figure, an instruction's X coordinate gives the total latency from fetch to retire-ready experienced by the

instruction over the execution of the program.[3] An instruction's Y coordinate gives the total number of issue slots wasted while the instruction was in progress.

The results in the graph show that latency is a poorer metric for pinpointing bottleneck instructions than wasted issue slots, due to varying levels of useful concurrency in the different loops. For example, the instruction with the highest latency (rightmost triangle) actually wastes fewer issue slots than instructions with lower latencies (rightmost circle and squares). However, when concurrency is fairly constant, latency is highly correlated with wasted issue slots. In the figure, intra-loop concurrency is similar across instructions, as indicated by the slopes of instructions in the different loops.

Data collected for several SPEC95 benchmarks using the same simulator indicate that real applications also exhibit varying levels of useful concurrency. Instructions-per-cycle (IPC) levels were measured by counting the number of instructions that retired during a fixed 30-cycle time window. The ratio of the maximum and minimum of these windowed IPC levels ranged from 3 to 30 across the various benchmarks; the standard deviation of the windowed IPC, weighted by retire count, varied from 20–42% of the mean for each of the benchmarks, with an overall value of 31% of the mean.

7 Potential Optimizations

This section briefly outlines some ways in which information collected by *ProfileMe* could be used in compilers and operating systems to improve performance. We are currently exploring these and other directions.

Guiding traditional compiler optimizations: Execution frequencies, branch mispredict rates, and instruction cache miss rates derived from the samples can be used to guide register allocation spilling decisions, inlining decisions, code generation, and the rearrangement of procedures and basic blocks to improve instruction cache locality.

Improved instruction scheduling: One important aspect of instruction scheduling is the insertion of prefetches and the scheduling of loads and stores. The lack of information about actual latencies forces most compilers to schedule loads and stores assuming that they will hit in the data cache. Abraham and Rau [1] have experimented with using average load latencies to drive compiler optimizations, and more recently Luk and Mowry [13] have explored the use of path information to identify loads whose cache miss behavior is correlated with the execution path taken to reach the load. *ProfileMe* provides a cheap way of gathering the data needed to drive these optimizations.

Cache and TLB hit rate enhancement: Recent studies have shown that dynamically controlling the operating system's virtual-to-physical mapping policies using information about dynamic reference patterns can reduce conflict misses in large direct-mapped caches [4, 15], lower TLB miss rates through the creation of superpages [16], and decrease the number of remote memory references in NUMA-based multiprocessors through replication and migration of pages [17]. All of these schemes gather reference pattern information through either specialized hardware for gathering cache miss addresses or specialized software schemes (*e.g.,* flushing the TLB and observing the miss pattern that results). By capturing the virtual addresses of memory references that miss in the cache or TLB, *ProfileMe* provides the information needed to guide these policies, without additional hardware complexity.

8 Related Work

The work most closely related to *ProfileMe* is a patent by Westcott and White, who also proposed a hardware mechanism for instruction-based sampling in an out-of-order execution machine [18]. Their system allows profiling of an instruction when its execution is assigned a particular internal *inum* in the processor's pipeline. During its execution, information associated with the instruction (such as whether it suffered a data cache miss, and its latency from fetch time to completion time) is recorded in internal processor registers. When the instruction retires, the information is logged to a specific area of memory, and when this memory area fills up, an interrupt is generated.

There are three key differences between the Wescott and White approach and our proposal. First, Westcott and White allow an instruction to be profiled only when it are assigned a particular *inum*. In contrast, our approach allows any instruction to be sampled; this is essential for obtaining a random sample of the entire instruction stream. Second, our system keeps information for all sampled instructions and provides a bit in the profile record indicating the instruction's retirement status. This allows software to decide how to handle unretired instructions, rather than transparently discarding them in the hardware. Third, the information we collect is geared towards understanding a wider range of performance issues. For example, we collect branch directions, global branch histories, branch mispredict information, and paired samples, all of which are useful for identifying the sequence of execution and the concurrency levels in a program's execution. The information collected by the Westcott and White mechanism focuses on an individual instruction's attributes (*e.g.,* cache miss information) and does not provide any support for determining inter-instruction relationships.

More recently, Horowitz *et al.* [10] proposed a hardware mechanism called *informing loads*, in which a memory op-

[3] We use this definition of latency instead of the fetch-to-retire latency to avoid penalizing instructions that issue around a stalled instruction and execute quickly but stall waiting to retire because the earlier instruction is not ready to retire; as with other out-of-order processors, the Alpha 21264 retires instructions in order. This is consistent with our definition of wasted issue slots, which considers only slots wasted while an instruction is in progress—i.e., between the time it is fetched and the time it becomes ready to retire.

eration can be followed by a conditional branch operation that is taken only if the memory operation misses in the cache. This permits reacting to cache misses at a fine-grained level (such as by branching to code that is scheduled for the case of a cache miss, rather than for the case of a cache hit). In contrast, our approach is geared more towards off-line performance feedback. However, the information provided by our approach is more detailed, since it includes other information such as the latency incurred in servicing a miss and other aspects of an instruction's execution. In many respects, these two designs are complementary: informing memory operations permit software to gain control very quickly after a cache miss, while our profile record contains more detailed information about an instruction's execution that can be used for offline analysis.

Bershad *et al.* [4] proposed specialized hardware called a cache miss lookaside (CML) buffer to identify virtual memory pages that suffer from a high L2 cache miss rate. Using the effective addresses and the latency information for loads and stores captured by our samples, we can maintain the information found in a CML buffer in software.

Some processors, such as the Intel Pentium, permit software to read the contents of the branch predictor's branch target buffer (BTB). By periodically reading the BTB in software, Conte *et al.* [7] developed a low-overhead technique to estimate edge execution frequencies of a program. The branch direction information contained in a *ProfileMe* record yields similar information. *ProfileMe* also captures the history of the past N branches leading to a sampled instruction, yielding path segment information. More recently, Conte *et al.* [6] proposed additional hardware called a *profile buffer*, which counts the number of times a branch is taken and not-taken. Again, *ProfileMe* yields similar information.

9 Conclusions

The performance of modern processors is becoming increasingly difficult to understand. The dynamic nature of speculative and out-of-order execution, coupled with the complexity of deep memory hierarchies, makes it impossible to predict program behavior solely through static analysis. Sampled profile information offers an inexpensive, unobtrusive way to collect detailed information for identifying bottlenecks and improving performance. However, this potential cannot be realized using the hardware performance counters found in existing processors, which cannot even accurately attribute events to instructions.

ProfileMe enables the collection of accurate, detailed information with modest hardware by sampling instructions instead of events. A complete record of interesting events, such as cache misses and branch mispredictions, is directly associated with each profiled instruction. A wealth of additional information is also collected, including pipeline stage latencies, branch history data, and effective addresses for memory operations. By additionally allowing a pair of in-flight instructions to be simultaneously profiled, a variety of interesting information can be derived about the interactions between instructions, including useful concurrency levels and pipeline stage utilizations. Together, these mechanisms enable the construction of a powerful, low-overhead profiling system that offers unprecedented instruction-level feedback to programmers and optimization tools.

Acknowledgments

We would like to thank Sanjay Ghemawat, Shun-Tak Leung, Susan Eggers, Bill Gray, and the anonymous referees for their helpful feedback on earlier versions of this paper. Special thanks to Mitch Lichtenberg for performing experiments illustrating how counters behave on the Intel Pentium Pro processor and to Michael Mitzenmacher for helping with the stochastic analysis in Section **??**. We also had valuable discussions with many people at DIGITAL, including Robert Cohn, Bruce Edwards, Joel Emer, Kourosh Gharachorloo, John Henning, Dan Liebholz, Ed McLellan, Rahul Razdan, and Steve Root.

References

[1] S. G. Abraham and B. R. Rau. Predicting load latencies using cache profiling. Technical Report HPL-94-110, Hewlett Packard Laboratories, Nov. 1994.

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

[3] T. Ball and J. R. Larus. Efficient path profiling. In *Proc. 29th Annual Intl. Symp. on Microarchitecture*, pages 46–57, Dec. 1996.

[4] B. N. Bershad, D. Lee, T. H. Romer, and J. B. Chen. Avoiding conflict misses dynamically in large direct-mapped caches. In *Proc. Sixth Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 158–170, Oct. 1994.

[5] R. Cohn and P. G. Lowney. Hot cold optimization of large Windows/NT applications. In *Proc. 29th Annual Intl. Symp. on Microarchitecture*, pages 80–89, Dec. 1996.

[6] T. M. Conte, K. N. Menezes, and M. A. Hirsch. Accurate and practical profile-driven compilation using the profile buffer. In *Proc. 29th Annual Intl. Symp. on Microarchitecture*, pages 36–45, Dec. 1996.

[7] T. M. Conte, B. A. Patel, and J. S. Cox. Using branch handling hardware to support profile-driven optimization. In *Proc. 27th Annual Intl. Symp. on Microarchitecture*, pages 12–21, Nov. 1994.

[8] Digital Equipment Corporation. *Alpha 21164 Microprocessor Hardware Reference Manual*. Maynard, MA, 1995. Order Number EC-QAEQB-TE.

[9] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. on Computing*, 30(7):478–490, July 1981.

[10] M. Horowitz, M. Martonosi, T. C. Mowry, and M. D. Smith. Informing memory operations: Providing memory performance feedback in modern processors. In *Proc. 23nd Annual Intl. Symp. on Computer Architecture*, pages 260–270, May 1996.

[11] Intel Corporation. *Pentium(R) Pro Processor Developer's Manual*. McGraw-Hill, June 1997.

[12] D. Leibholz and R. Razdan. The Alpha 21264: A 500 MHz Out-of-Order Execution Microprocessor. In *IEEE CompCon'97*, Feb. 1997.

[13] C.-K. Luk and T. C. Mowry. Predicting data cache misses in non-numeric applications through correlation profiling. In *Proc. 30th Annual Intl. Symp. on Microarchitecture*, Dec. 1997.

[14] MIPS Technologies, Inc. *MIPS R10000 Microprocessor User's Manual*. Mountain View, CA, 1995.

[15] T. H. Romer, D. Lee, B. N. Bershad, and J. B. Chen. Dynamic page mapping policies for cache conflict resolution on standard hardware. In *Proc. First Symp. on Operating Systems Design and Implementation*, pages 255–266, 1994.

[16] T. H. Romer, W. H. Ohlrich, A. R. Karlin, and B. N. Bershad. Reducing TLB and memory overhead using online superpage promotion. In *Proc. 22nd Annual Intl. Symp. on Computer Architecture*, pages 176–187, June 1995.

[17] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating system support for improving data locality on CC-NUMA compute servers. In *Proc. Seventh Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 279–289, Oct. 1996.

[18] D. W. Westcott and V. White. Instruction sampling instrumentation, Sept. 1992. U.S. Patent #5,151,981, assigned to International Business Machines Corporation.

[19] C. Young and M. D. Smith. Improving the accuracy of static branch prediction using branch correlation. In *Proc. Sixth Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 232–241, Oct. 1994.