To appear at the Seventh International Symposium on High Performance Computer Architecture (HPCA-7), January 19-24, 2001

A Programmable Co-processor for Profiling

Craig B. Zilles and Gurindar S. Sohi
Computer Sciences Department, University of Wisconsin - Madison
1210 West Dayton Street, Madison, WI 53706-1685, USA
{zilles, sohi}@cs.wisc.edu

Abstract

Aggressive program optimization requires accurate profile information, but such accuracy requires many samples to be collected. We explore a novel profiling architecture that reduces the overhead of collecting each sample by including a programmable co-processor that analyzes a stream of profile samples generated by a microprocessor. From this stream of samples, the co-processor can detect correlations between instructions (e.g., memory dependence profiling) as well as those between different dynamic instances of the same instruction (e.g., value profiling). The profiler's programmable nature allows a broad range of data to be extracted, post-processed, and formatted, as well as provides the flexibility to tailor the profiling application to the program under test. Because the co-processor is specialized for profiling, it can execute profiling applications more efficiently than a general-purpose processor. The co-processor should not significantly impact the cost or performance of the main processor because it can be implemented using a small number of transistors at the chip's periphery.

We demonstrate the proposed design through a detailed evaluation of load value profiling. Our implementation quickly and accurately estimates the value invariance of loads, with time overhead roughly proportional to the size of the instruction working set of the program. This algorithm demonstrates a number of general techniques for profiling, including: estimating the completeness of a profile, a means to focus profiling on particular instructions, management of profiling resources.

1 Introduction

Understanding dynamic program behavior is the key to maximizing performance. Without a means to identify bottlenecks and inefficiencies, it is difficult to effectively optimize a program's execution. Program profiling is an important mechanism for observing dynamic program behavior.

Many program profiling systems have been proposed [1, 2, 6, 7, 16, 17, 18, 24, 25, 30, 36, 40, 41] and there is some consensus as to the desired attributes of such a system. These attributes can be grouped into four main categories:

- **Usability:** Widespread adoption of profiling necessitates that the effort required by the user be minimized and that the technique be widely applicable. Specifically, special compilation requirements should be avoided.
- **Low Overhead:** Overhead, in both space and time, should be minimized to enable profiling of long running applications with realistic data sets. Run-time optimization systems are especially sensitive to overhead.
- **Accuracy/Precision:** Behaviors should be correctly attributed (to individual instructions when possible), and the profiling system should keep result perturbation to a minimum.
- **Expressiveness:** The ideal profiling system should be able to measure any behavior.

The ideal profiling system has not yet been developed; every scheme has its strengths and limitations. In this paper we present a profiling architecture that we feel compares favorably to existing schemes, at the cost of additional hardware.

Much like the ProfileMe system [18], our profiling architecture profiles instructions; since this is performed transparently in hardware, no special application preparation is required. It stores the profiled instructions as they retire, with their dynamic information, in a *sample buffer*. Unlike the proposed ProfileMe implementation, multiple in-flight instructions can be profiled simultaneously.

Although the sample buffer could be accessed directly by the main processor, our architecture includes a *programmable profiling co-processor* that serves as an intermediary. This co-processor can distill the profiling information into a compact form before passing it to the main processor. In this way high-quality profiling information can be gathered quickly while maintaining low overhead.

The co-processor is controlled by downloading programs into it from the main processor. The co-processor's programmable nature, coupled with the richness of the profile information that can be collected, enables a broad range of program behaviors to be observed with a single piece of hardware. Programmability allows the profiling software to be specialized to the program under observation.

Because this co-processor will be used exclusively for profiling, we can tailor its design for efficiency. By implementing common profiling operations (discussed in Section 3.1) as primitives in hardware, a high performance profiling co-processor can be implemented on small area and power budgets. Moreover, because the co-processor is decoupled from the main processor through the sample buffer, it can be located where it will not significantly impact the design of the core. The hardware design is discussed in Section 3.

After a brief discussion of some profiles that could be collected by the co-processor (Section 4), we evaluate our profiling architecture through a case study of load value profiling. We demonstrate an algorithm that, in general, collects more accurate profiles, faster, and with lower overhead, than a simple sampling value profiler. This algorithm (Section 5) demonstrates a number of techniques that have applicability for profiling beyond value profiling. These techniques enable the algorithm to implicitly identify the most frequent instructions, profile these instructions until it is confident they have been characterized, mask them and then profile the set of next most frequent instructions. In this way, the profiler successively profiles instructions with the largest potential impact to those with the least, and the algorithm stops incurring overhead when the profile is complete.

2 Observations on Profiling: A Motivation

Looking forward, we see two trends that we feel will place larger demands on the rate at which profiling information will need

to be collected. In this section, we discuss these trends and explain why sampling will not be able to meet these demands without increasing overhead.

2.1 The Changing Face of Profiling

The program profiling systems proposed to date have concentrated on two topics: identifying control flow profiles [16, 32] and the instructions associated with performance degrading events [1, 2, 18, 40]. Many techniques that are likely to be employed in the future (including value-based code specialization, speculative multithreading, pre-execution, software managed caches, etc.) either require or can exploit additional types of profile information. In order to collect this larger set of profile information, the rate at which profile information is gathered must be increased.

These techniques for program optimization are still evolving rapidly, and as they are developed they will require new types of profile information to be collected. With a programmable profile engine, this profile information can be collected on existing hardware, rather than having to wait for the next hardware design cycle to include the necessary special purpose hardware.

In addition, there is a trend toward run-time optimization [6, 27, 33], whereby a program's execution is optimized as the profile information is gathered. Run-time optimization requires profile collection to be quick, to maximize the portion of the program's execution that is optimized, and low overhead, so as to not significantly impact the run time. Many profiling systems leverage sophisticated analysis to post-process profiles, but in a run-time optimization environment such post-processing may not be cost-effective.

2.2 Reducing the Overhead of Collecting Samples

Most profiling systems use sampling to maintain low overheads. Sampling is a meta-technique that can be applied to other techniques (including instrumentation [24] or interpretation [9]) to reduce overhead by decreasing the rate at which information is collected. Sampling exploits the fact that profile information can only be used as a hint and, therefore, does not need to be complete or even necessarily correct. Sampling is effective because statistically we are likely to collect information about common events, *i.e.*, the ones that provide the most potential for performance improvement. Furthermore, highly biased behaviors, again the ones that provide the most potential, can be estimated with a given confidence level with fewer samples than less highly biased behaviors [37].

Because the overhead of interrupt-based sampling is proportional to the data collection rate, higher profiling rates equate to larger overheads (as shown in Figure 1). In order to reduce the overhead of collecting each sample (*i.e.*, the constant of proportionality), our proposed profiling system delegates much of the profiling computation to a dedicated profiling co-processor. The co-processor summarizes the information contained in many samples before passing it to the main processor. By specializing the co-processor to the task of profiling, we can provide profiling computation more cheaply than can the general-purpose host processor. In the next section, we discuss the design of the profiling co-processor.

3 Hardware

Our proposed profiling architecture requires hardware support beyond that which has been included in current implementations. In light of the fact that peak processor performance has been growing more rapidly than real program performance, we feel that dedicating hardware resources to features that can close this gap — including profiling support — will be justifiable in future micro-

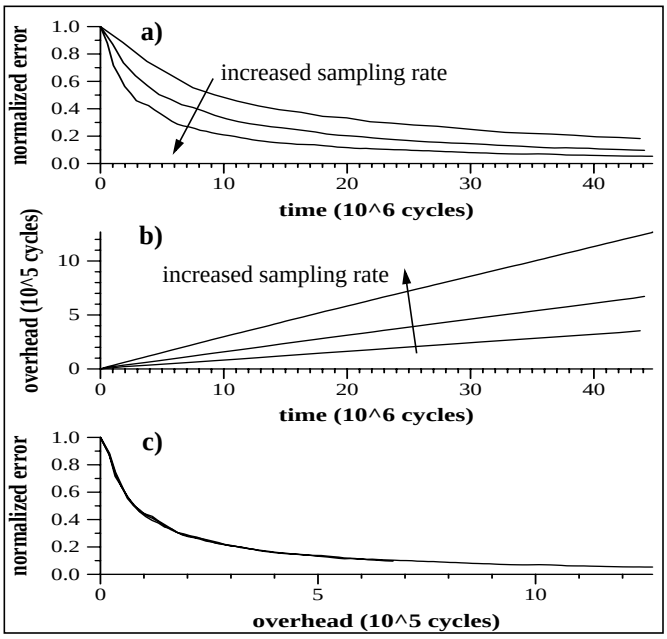

Figure 1. *The relationship between accuracy, overhead, and sampling rate for traditional interrupt-driven sampling (data shown for load value profiling on gcc, samples every 512, 1024, and 2048 loads): (a) faster sampling rate enables profile to converge faster, but (b) higher sampling rate translates to correspondingly higher overhead, leading to (c) the profile quality being a function of overhead (independent of sampling rate)*

processors. Nevertheless, an important aspect of the design of a profiling co-processor will be minimizing its impact on the main processor's performance and cost. Specifically, the co-processor should:

- Use a moderate number of transistors.
- Keep the additional circuits far from the core of the processor. Therefore, the design must be able to tolerate communication latencies from the core.
- Avoid loading critical circuit paths in order to minimally impact processor frequency.
- Not significantly increase power consumption.

We feel that the design presented in this section abides by these constraints. We estimate that our baseline design can be implemented in approximately one-half million transistors; we estimated 300,000 transistors for the memory arrays and believe the core is simpler than that of the StrongARM [34], which only required 250,000 transistors. This is substantially smaller than a modern microprocessor (*e.g.*, AMD's Athlon (w/o L2 cache) is 22 million transistors) and future processors are expected to be even larger. The transistors that make up the co-processor can be located at the core's periphery.

Additional hardware in the core is required to collect and export the profile information to the co-processor. This hardware is similar to that required for the ProfileMe proposal [18] except additional storage is required because multiple in-flight instructions can be profiled. Many types of information could be collected for an instruction, including the instruction's PC, register values, memory address, and any micro-architectural events associated with the instruction (*e.g.*, the instruction caused a branch misprediction), as well as the instruction itself. The generality of the profiler design is dependent on what information the core makes available about an instruction. Exporting this information to the

co-processor requires additional datapaths[1]. For the studies done a 128-bit datapath was sufficient, and these signals are latency-tolerant, so the interconnect can be pipelined over many cycles.

In this section, we describe the major features of our profiling architecture. We start by describing the requirements for such a co-processor, in Section 3.1. Each of the following sub-sections covers a portion of the design: instruction filtering and the sample buffer (Section 3.2), co-processor datapath (Section 3.3), co-processor control (Section 3.4), and interactions with the main processor (Section 3.5). A high-level block diagram of the profiling architecture is shown in Figure 2.

3.1 Characteristics of Profiling Applications

By analyzing many profiling algorithms we have identified some common operations. In order to efficiently execute profiling applications, these operations are provided by the hardware as primitives. Below is a list of these common operations with a brief description of the hardware that implements them:

- **Implicit loop-based structure:** A routine is executed for every sampled instruction. The co-processor provides a special branch target that fetches the next instruction to sample and jumps to the routine for processing that type of instruction (described in Section 3.4).
- **Opcode filtering:** Only certain classes of instructions are considered (*e.g.*, only branch instructions are considered for control flow profiling). In the decode stage of the main processor, there is a configurable hardware filter that can filter instructions by opcode (described in Section 3.2).
- **Field extraction:** Processing instructions usually requires extracting one or more fields from the instruction (*e.g.*, branch target PC, or register identifiers). The co-processor includes a hardware instruction decoder/field extractor that provides bit-fields without the need to shift and mask the instruction bits in software (described in Section 3.4).
- **Lookups/matching:** The current sample has to be paired with previous related samples. The co-processor has an associative array that provides match operations to the software (described in Section 3.3).
- **Counter manipulation:** Counters are used to summarize repeated events and for management of profiling resources. The co-processor's data memory provides read-modify-write operations and its ALUs support saturating arithmetic (described in Section 3.3).
- **Data dependent control flow:** Profiling applications are often control intensive, and many of the branches are hard to predict because their outcomes differ from sample to sample. The profiling co-processor is capable of executing a branch every cycle (in parallel with other operations), and its short pipeline minimizes stalls due to branch misprediction (discussed in Section 3.4).

By specializing the co-processor design to the needs of profiling applications, we can provide computation for profiling inexpensively.

1. The information on these wires is largely a subset of what is required from the core by a DIVA-style checker architecture [4]. If the processor design employs such a checker, the impact of these wires can be amortized over both features.

3.2 Instruction Filtering and the Sample Buffer

Because the profiling co-processor does not have the resources to profile every retired instruction (nor is profiling every instruction required for useful profiling information) the main processor only needs to collect profile information at the rate that the co-processor consumes it. A configurable hardware filter, which is accessed at decode time, allows instructions to be tagged for profiling in a controllable way. By pro-actively filtering, rather than tagging instructions randomly, we can focus on a subset of instructions; this increases the locality in the sample stream, enabling better utilization of the co-processor's memory resources.

The filter considers two of the instruction's characteristics: its opcode class and its program counter (PC). Opcode filtering is provided because most profiling applications only monitor a subset of instructions (*i.e.*, loads or branches). Two PC-based filters provide the ability to consider only a fraction of the program at a time, as well as to exclude particular instructions from consideration. These filters are shown in Figure 2, and their uses are described in Section 5.1.3 and Section 5.1.4, respectively.

These three filters are used in conjunction to determine whether an instruction should be profiled. These filters must be able to support the decode width of the processor. The small opcode filter and the first PC-based filter can easily be replicated. Replicating the larger PC filter would be too costly, but it can be constructed to exploit the fact that blocks of instructions have consecutive PC's, much in the same way that instruction caches are built to fetch multiple instructions.

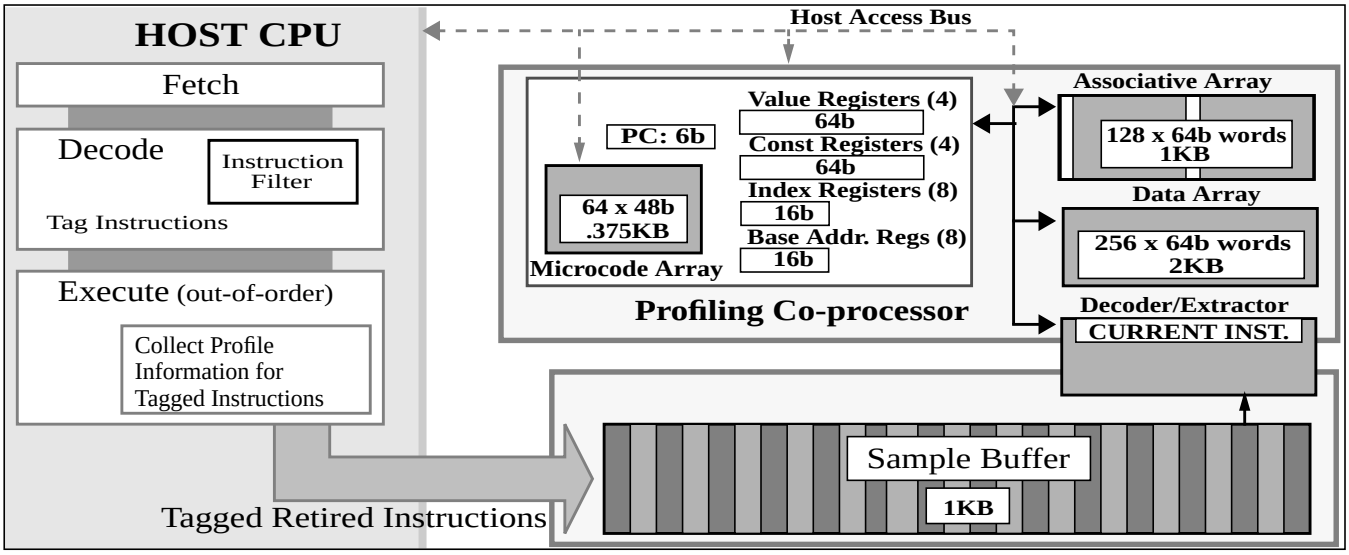

Figure 2. *Block diagram of profiling co-processor hardware, showing major arrays with size estimates for a baseline design.*

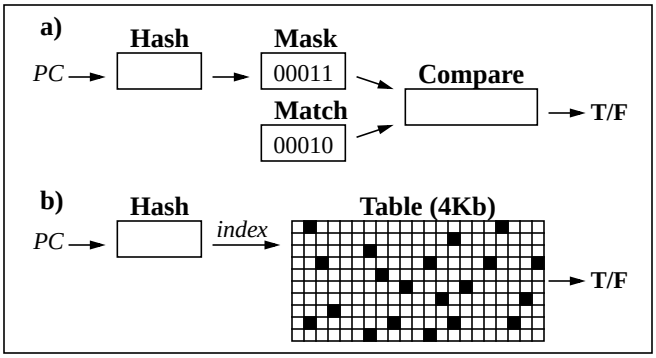

Figure 3. *Two PC-based filters: a) a hashed version of the PC is masked by a variable-width mask and compared to a programmable value (*Match*); in this way the program can be sub-divided into 2^n regions to be profiled independently (where n is the number of ones in the mask register), and b) a hashed version of the PC indexes into a table of bits which can be set independently, enabling individual instructions to ignored during profiling. In both cases multiple hash functions are provided to reduce conflict problems.*

The actual collection of the profile information is highly dependent on the underlying micro-architecture. For our simulation micro-architecture, the process is similar to that of ProfileMe [18].

The sample buffer serves to decouple the retiring of tagged instructions by the main processor from their processing by the co-processor. The sample buffer helps tolerate burstiness of retirement without letting the co-processor go idle. To conserve space in the sample buffer and to conserve bandwidth between the main processor and the co-processor, the profiling hardware is programmed to collect only those fields that will be used by the current profiling application. In our experiments, we have found a 1kB sample buffer and a 128-bit datapath (between the processor and co-processor) to be sufficient.

3.3 Profiling Co-processor Datapath

Like most processors, the profiling co-processor is made up of memories, register files, and arithmetic/logic units (ALUs). Each of these structures has been tailored to support profiling applications with a minimum of resources.

Three memory arrays are included: a microcode array for co-processor program storage, an associative array for efficient matching, and a data array for general purpose data storage. These memories, unlike those in the main processor, are not caches backed up by main memory. Such a design avoids the size and complexity associated with cache tags, miss logic, and coherency logic, and it enables efficient static code scheduling because all operations have a known, fixed latency. Because profiling information need not be complete, or even correct, data can be dropped and algorithms can be simplified to fit these constrained resources. Communication to main memory (discussed in Section 3.5) is performed through the main processor via the *host access bus* (shown in Figure 2).

The associative array is implemented as a content-addressable memory (CAM), and it provides inexpensive hash table-like functionality for lookups and matching. Each entry of the associative array has a valid bit that is set when the entry is written and can be cleared through an invalidate operation. The data memory has special support for read-modify-write operations, like incrementing counters. All three memories are single-ported, and we have found that having twice as much data memory as associative memory is a good compromise between cost and functionality.

In order to minimize the number of ports on each structure and reduce register specifier size, the register file is partitioned into five separate special-purpose register files: *value*, *index*, *constant*, *base address*, and *branch target*. The size, number, and port configuration of these register files is shown in Table 1. Value registers are wide enough hold data from retired host instructions (*e.g.*, register values, addresses, and PCs). The narrower index registers hold offsets into memory arrays or counters for monitoring the profiler's status. To facilitate implementing circular buffers and saturating counters of different sizes, the width of index registers is configurable at program load time.

The other three types of registers — constant, base address and branch target — are read-only to the co-processor; they are configured when the program is loaded. Because few constants are used, they are stored in a constant register file instead of requiring immediates in instructions. Base address registers are used to subdivide the co-processor's memory into sub-arrays. Only aligned, power-of-two size sub-arrays are supported to allow addresses to be generated without arithmetic. Branch target registers are discussed in Section 3.4.

The main ALU in the co-processor has limited functionality compared to that of a traditional processor. Because multiplication and division are seldom used in profiling, they are not supported, and only limited shifts are available. The ALU does provide a mechanism for generating pseudo-random numbers, through the use of a linear feedback shift register (LFSR) [21], for use in resource management decisions; this helps to avoid pathological behaviors caused by repetition in the program. In addition to the main ALU, an incrementer is provided specifically for manipulating index register values. Because information can become distorted if counters roll over, the ALUs support saturating arithmetic. The ALUs also provide control conditions, through comparisons and based on whether their results are saturated, for use as branch predicates. The control of the co-processor is described in the next section

3.4 Profiling Co-processor Control

The co-processor executes a short routine for each instruction in the sample buffer, in the order that the instructions were retired. As each instruction comes to the head of the sample buffer, it is copied

Structure	Size	Ports	Transistor Count
PC Filter bit-mask	4Kb	1	25k
Sample Buffer	1kB	1W/1R	65k
Microcode Array	64 x 48b	1	18k
Associative Array	1kB	1	74k
Data Array	2kB	1	98k
Value Registers	4 x 64b	2R/1W	3k
Index Registers	8 x 16b	1R/1W	2k
Constant Registers	4 x 64b	1	2k
Base Address Registers	8 x 16b	1	1k
Jump Target Registers	4 x 6b	1	<1k
Decoder/Field Extractor	21 x 6b	1	1k
Total Transistor Count			289k

Table 1. *Estimated sizes and number of ports for major structures in baseline co-processor design.*

to the decoder/field extractor (DFE). The DFE provides access to the information associated with the current retired instruction (including efficient field extraction) and supplies the starting PC to be used for processing this instruction. Typically, when instructions from different opcode classes are profiled by a single profiling application, they are treated differently. To avoid a performance-degrading, multi-way branch in software, the DFE contains a table that associates each opcode class with a starting co-processor PC.

There is a significant amount of parallelism in profiling applications, but it needs to be exploited in a cost-effective manner. Our co-processor executes a single instruction per cycle, but this instruction encodes multiple operations. The co-processor's instruction set is much like microcode, in that each instruction is only a simple encoding of the co-processor's control signals. This allows a large fraction of the resources to be used each cycle and reduces instruction decode time. A short decode time is necessary to maintain a high clock rate for our short in-order pipeline. A two-stage (fetch/decode, execute) pipeline is used to minimize the microcode branch misprediction penalty.

Because profiling applications are extremely control intensive, every microcode instruction includes a branch slot. Often 50 to 75 percent of the branch slots are used. To predict the direction of these branches, each instruction has a 2-bit branch predictor. Most branches in profiling applications are short, so only a small set of short forward and backward immediate offsets are included in the instruction set. For the rare cases when these are insufficient, the architecture includes *branch target* registers that enable jumps to any location.

In addition, two special branch targets are included: **done** and **interrupt**. A routine branches to the target **done** when it has finished processing a host instruction; this increments the head pointer in the sample buffer and jumps to the routine associated with the new head instruction's type. When the co-processor branches to the **interrupt** target, execution is halted and the main processor is interrupted.

3.5 Interactions with the Main Processor

Although providing the co-processor a direct path to memory would make it more powerful, we fear that it could impact the circuit speed of the main processor's memory access path. Instead, to read and write main memory, the profiling co-processor leverages the existing memory system hardware in the main processor. Using loads and stores to special address ranges, the main processor can read and write the co-processor's state. In this way, the main processor can copy programs and initial state into the co-processor and copy profile data from the co-processor into main memory.

The baseline co-processor, not counting the sample buffer, has about 4kB of state. To completely transfer this state requires about 500 64-bit memory operations, but a full transfer is seldom required. Special operations are provided to invalidate the associative array and clear the data array, because these are the desired initial states for many profiling applications. Because those arrays dominate the co-processor's state, the number of stores necessary to program the co-processor is drastically reduced. If the program is already loaded (*e.g.*, when the co-processor continues collecting the same type of information after an interrupt), the co-processor can typically be re-initialized in less than 10 stores. When reading profile state from the co-processor, it is only necessary to read data that could have changed (*e.g.*, it is not necessary to read the microcode array), but it is not uncommon to read most of the associative and data arrays (about 3kB).

Once the co-processor has been programmed, it performs the profiling autonomously until it desires external communication. When it has filled its arrays with useful profiling information or requires guidance from the profiling application, it interrupts the main processor.

The overhead observed during profiling is roughly the product of: (1) the number of interrupts, (2) the number of samples that are recorded per interrupt, and (3) the average time required to record a sample. To minimize the total overhead, we attempt to minimize all three, while maximizing the quality of information extracted (which is often a function of the number of samples recorded). The first component (the number of interrupts) is minimized through the design of the profiling algorithm that strives to interrupt only when the slots have a high information content that complements or enhances the information already collected (discussed in Section 5.1). The second component (samples per interrupt) can be slightly reduced by having the co-processor post-process the information collected to remove samples that have low information content. The overhead per sample (the third component) is reduced by hand-assembling the interrupt handler to minimize instruction count and maximize instruction-level parallelism and organizing data structures to minimize the number of cache misses, as is described in [2]. In addition, since we are processing multiple samples per interrupt, modulo scheduling can be used to overlap the cache misses associated with the hash table lookup. Using these techniques, we have written interrupt handlers that require as little as 10-30 cycles to process each sample, depending on the complexity of the handler.

4 Example Applications

In addition to the case study that follows, we briefly discuss some applications which could be implemented with the profiling co-processor. These profiles are not new, but can potentially be implemented in a low-overhead manner without specialized hardware.

Edge Profiling. A two-pass profiler could be built that first profiles direct branches counting how many times the branch was taken and not-taken, then profiles indirect branches to identify their targets and count their frequencies. In addition, by observing a pair of branch samples in close proximity we can potentially identify correlations between branch outcomes.

Call Stack Monitoring. Call and return instructions could be profiled to maintain the current call stack, enabling events (*e.g.*, branch mispredictions) to be correlated to calling contexts.

Memory Dependence Profiling. By storing the PCs and memory addresses of recent stores and comparing them to the memory addresses of loads, dependences between stores and loads can be identified.

Cache Conflict Profiling. By storing the memory addresses of cache missing loads we can identify when a block has been brought into the cache multiple times in a short duration, likely indicating a conflict. From this address we can compute its set and identify the instructions that have been accessing this set.

5 Case Study: Value Profiling

The predictability and invariance of data values has been actively studied recently. For mechanisms that exploit this program behavior (*e.g.*, value prediction [12, 29, 35, 38] and dynamic specialization [3, 15, 19]), the identification of candidate invariant values is either a strict requirement or an enhancement. Because requiring the programmer to identify these values is tedious and error prone, value profiling [9, 10, 11, 20] has been proposed to automatically characterize value invariance (the frequency at

which the value is seen) as well as to identify the most common values.

Early implementations of value profiling using instrumentation or simulation are not feasible for large programs (Calder, *et al*. reports an average slowdown of a factor of 10). More recently, a sampling implementation of value profiling [9] has been demonstrated with only 10% slow down, by sampling every 32,000 instructions[2]. Because their implementation interprets the program to collect the values, the cost of collecting each sample is high (hundreds to thousands of cycles per sample).

Our profiling co-processor can be used to emulate this *simple sampling* algorithm and achieve comparable results at lower overheads. Because values are captured in hardware, interpretation is not required, and multiple samples can be buffered allowing the cost of the interrupt to be amortized over many samples. Our simulation results show an equivalent sampling frequency can be achieved with only a 0.3 percent overhead. With such a low overhead, many behaviors could be profiled simultaneously without significantly impacting performance.

To demonstrate the benefit of the other features of our hardware profiler, we examine performing value profiling to support dynamic optimization, a more challenging profiling scenario. In this context, we have the additional requirement that the profile be collected at a high rate, to maximize the portion of the execution that can be optimized. Although this can be acheived by greatly increasing the sampling rate, the limits on overhead are more strict because performance lost to profiling reduces the benefit achieved by dynamic optimization. This challenging scenario motivated the profiling co-processor approach.

Our algorithm improves over existing algorithms in two respects: (a) by providing mechanisms and policies for targeting the profile to particular instructions, and (b) by summarizing multiple related samples in the co-processor, fewer updates to the in-memory profile result tables are required. In the next sub-section (5.1) we describe our algorithm. Our experimental methodology is described in Section 5.2. Quantitative results on profiling accuracy and overhead, comparing it to the simple sampling approach, are provided in Section 5.3.

5.1 The Algorithm

At a high level, our algorithm performs the following steps:

1. Find the N most frequently executed, unmasked instructions
2. Collect and summarize profile information on those instructions
3. Interrupt the processor
4. Copy this information to a data-structure in memory
5. If an instruction has been sufficiently characterized, mask it
6. Repeat, until all instructions have been masked

There are a number of nice things about this structure. First, it profiles instructions in the order of their importance, from most frequently executed to least frequently executed. In Section 5.1.2, we discuss a replacement policy that statistically retains the most frequently executed instructions without prior knowledge. Second, by predicting when we have sufficiently profiled an instruction (which is discussed in Section 5.1.4), we can stop profiling that instruction and focus our profiling resources on the remaining instructions. In this way we can profile even infrequently executed instructions, which may be difficult with a traditional sampling scheme. Finally, when no unmasked instructions are found, the profile is declared complete and the co-processor can be used to profile other behaviors.

2. To be exact, their proposal samples 4 instructions, on every other interrupt, at an interrupt frequency of about every 64,000 instructions.

In the section that follows, we present some of the important components of the algorithm including: maximizing the benefit of the limited co-processor resources (Section 5.1.1), thrash detection (Section 5.1.3), and how invariances are estimated (Section 5.1.5). Additional detail can be found in Appendix A.

5.1.1 Slots

Given that we have a limited set of resources with which to monitor the program, much of the algorithm is designed to maximize the utilization of those resources. In order to simultaneously track the maximum number of static load instructions, we minimize the storage allocated to each static load, by only tracking one value per load at a time.

Each load that is being monitored by the profiling co-processor is allocated a portion of the co-processor's storage resources, which we call a *slot*. The manner that this storage is allocated is configured in software, and hence part of the co-processor's programmability. A slot (shown in Figure 4) consists of storage for the PC, an active value, two 1-byte counters (hit and miss), and a 2-byte counter (total). Although we performed our experiments in the 64-bit Alpha architecture, we found that the upper half of a word often had very little information content; 32 bits was usually sufficient to discriminate between PCs (where we select bits [34:2]) and values (where the low 32 bits are stored).

Unlike previous value profiling techniques, each slot only keeps one active value at any time (although more than one value is associated with a static instruction in the data structure in main memory). We can accurately estimate the invariance of the active value by keeping track of the number of times we see a matching value (hits) and dividing by the total number of values seen (hits + misses). We simply must select the top (most invariant) values to be the active values.

Although identifying the top value cannot be done *a priori*, when we select a random sample for the active value, we are statistically most likely to select the top value. A similar observation was made by Bala *et al*. [6] with regards to path profiles. Since we are not assured to get the right value, we need to periodically re-select if the active value's invariance is low.

5.1.2 Informational Replacement

When an instruction that is not currently allocated to a slot is encountered, we have to decide whether or not to replace one of the current slots. The cost of replacing a slot is the loss of any information that has been gleaned about the associated instruction. Therefore, the amount of information stored in the slot should be considered when making the replacement decision. We use the number of samples observed (*i.e.*, a total counter that is incremented every time a sample is observed) as an estimate of the information content of a slot. In this way, infrequently executed instructions are likely to be replaced by frequently executed instructions.

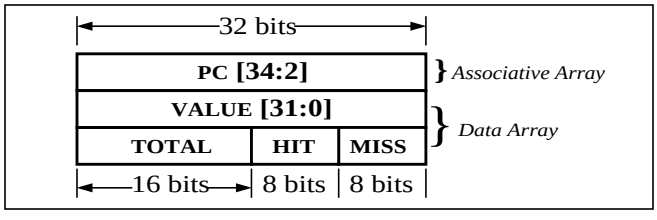

Figure 4. *For each static load that it is profiling, the co-processor maintains a data structure (slot) that contains the instruction's PC, an active value, and some statistics about the instruction's past behavior. The PC is stored in the associative array for fast matching.*

5.1.3 Thrash Detection and Filtering

The algorithm that has been described so far is sufficient when the working set of unfiltered loads fits in the slot array. But if there is a working set of unfiltered loads whose frequencies are roughly equal and that is much larger than the slot array, the algorithm thrashes, constantly replacing entries before they have a chance to observe multiple samples. When this happens, the co-processor is throwing away information at the same rate as it is collecting it.

Thrashing can be quickly and easily detected by monitoring the number of times that we have performed slot replacements. When thrashing is occurring, the number of replacements will be a large fraction of the samples observed. To avoid thrashing, we use a divide-and-conquer technique, using the first PC-based filter to partition the program. Each subset is profiled in turn, and this partitioning artificially increases the locality in the sample stream.

5.1.4 Confidence Estimation

Once enough samples have been collected to estimate the invariance of an instruction within the necessary accuracy, collecting additional samples for this instruction does not provide any benefit. The difficulty in such an approach is determining when enough samples have been collected. For a truly random process, given a set of samples and a desired accuracy, statistical methods can be used to compute a confidence that the set of samples represents the process to the desired accuracy [37], but program behaviors are not random processes.

Calder, *et al*. [10] investigated a scheme, called *convergent profiling*, that compares a recent set of samples for an instruction to its profile as a whole to determine if the profile had converged. Upon converging collecting samples was discontinued. Once the rest of the algorithm has been streamlined, such a comparison can become expensive, and we have found it to be unnecessary.

We found that many confidence decisions can be made with much less information. Statistical confidence increases with measured bias of the sample. Thus it is easy to be confident of highly invariant and highly variant instructions. Hence, when we are storing data into memory (in the interrupt handler) we test if the invariance is above a high threshold or below a low one, and if so we mask off the instruction, using the second, larger PC-based filter. Otherwise the instruction should be profiled multiple times to attempt to characterize all of the dominant values; we allow the instruction to be sampled 4 times before masking it.

Because of phase behavior, it is necessary to periodically re-sample to ensure that all important behaviors are observed. Our time-based re-sampling could be improved by including an explicit phase detector. A special case of phase behavior is what has been referred to as glacial values [5]. Some benchmarks (*e.g.*, *gap* and *vpr)* have loads that will load a single value for a long interval (thousands of instances) and then change to repeatedly loading a different value. Because we may not sample the phases equally, our profiler can have trouble determining what fraction of the execution is associated with each value. We can, however, easily diagnose that this behavior is occurring (multiple profiled values with many good counts and zero or few bad counts) which is likely to be more important than knowing the exact contributions. Performing this diagnosis could be difficult for a traditional sampling profiler.

5.1.5 Invariance Estimation Algorithm

To estimate a value's invariance, we need to reconstruct the data we did not collect from the data we did collect. Each static instruction is post-processed in isolation by the main processor. For each value that accumulated hit counts, we can compute its invariance — hits / (hits + misses) — for the regions it was selected as the active value by the profiler. Because of clustering in the stream of values loaded by an instruction, this computed invariance often over-estimates the value's invariance for the whole program. To get a more accurate estimate, we also consider the counts recorded when other values were active as well as the total number of samples observed.

5.2 Methodology

In order to evaluate the capabilities of our proposed profiling architecture, we built a cycle-accurate simulator of the co-processor. The profiling algorithm described in this section was implemented in the co-processor's microcode and executed by the simulated co-processor. This co-processor model is included in a timing simulator derived from the Alpha version of SimpleScalar [8] that simulates the main processor. In addition to the benchmark program, the main processor executes the profiling system code (including the interrupt handler) that is responsible for configuring the co-processor and storing the co-processor's results into memory. A timing simulation is required to determine which instruction samples are dropped because the sample buffer is full and to estimate the profiling overhead.

The simulated main processor is a 4-way superscalar, dynamically-scheduled processor, roughly modelled after the Alpha 21264 [26]. The processor has 64kB L1 caches, a shared 1MB L2 cache (10 cycle access), and an 80 cycle main memory access. We simulated 3 co-processor configurations to observe the co-processor's sensitivity to the sizes of the associative and data memories; these memories are a substantial portion of the co-processor's cost. The baseline configuration has a 1kB associative memory and a 2kB data memory. In addition, we simulated co-processors with half (.5kB/1kB) and twice (2kB/4kB) the memory resources. We simulate the co-processor executing at the same frequency as the main processor, but explore its sensitivity to this in Section 5.3.

Our benchmarks are from the SPEC2000 integer benchmark suite. We used modified reference inputs that attempt to maintain the reference data set size while reducing execution duration; the modified inputs ran between 9 and 44 billion instructions. A 100 million instruction region, selected from a dominant execution phase of each program, was used for simulation. Typically, the profiler characterizes a program in a much shorter interval.

In this set of experiments, we only concern ourselves with identifying load values that account for at least 200 dynamic instances of a particular load and have invariances of at least 25%. In general, such loads are unlikely to benefit from value-based optimizations, and are difficult to accurately characterize with anything other than a complete profile. When a single static instruction has multiple values that exceed these requirements, we attempt to estimate the invariance of all such values.

5.3 Results

For each run we collect a complete profile, and compare it to our estimated profile. Figure 5 shows estimated invariance of values versus the true invariance for *gcc*, the program with the largest instruction working set. Each point in the scatter plot represents a single <static instruction, value> pair; the shape of the point roughly indicates the frequency of the value (x's for values loaded less than one-thousand times, triangles for between one-thousand and ten-thousand, and squares for values loaded more than ten-thousand times). Ideal results would have all points on the dashed line; the pair of dotted lines are to aid visualization and represent a 5 percent over- and under-estimate. The points along the x-axis are values which were not sampled and are largely due to infrequently executed instructions. In general, the estimates are very accurate; most estimates are within 5 percent. In fact the results are even better than they appear because points near the diagonal often obscure other points at the same location. In general

we more accurately predict values with high invariances and the outliers tend to be the infrequently executed values.

Quantitative Results. To obtain a quantitative estimate of the quality of the profiles, we compute the root-mean-square error (weighted by the value's frequency) of the estimations. We compare our algorithm against the hardware implementation of the simple sampling approach described in the beginning of Section 5; we have increased its sampling rate to once every 1024 loads to enable faster profile collection.

Figure 6(a) compares the profile quality generated by the two algorithms, demonstrating that there is a lot of variation in accuracy. The accuracy of the simple sampling approach is largely inversely proportional to the size of the instruction working set; *gcc* and *perl* have the largest number of loads in the simulated region. The co-processor implementation has better accuracies for most of the benchmarks. *Gap* is worse because our algorithm has inaccurately characterized some of the glacial values mentioned in Section 5.1.4. *Bzip2* and *parser* have lower accuracy because the algorithm was satisfied with the accuracy and discontinued profiling; for these benchmarks the overhead is substantially lower for the co-processor algorithm.

Crafty is a special case; its accuracy suffers in both algorithms because it has many 64b bit-flags (for tracking the state of a chess board) whose values are aliased together because we only store the bottom 32 bits of the value. We are currently working on an extension to our algorithm to detect this condition, by taking a second pass over invariant instructions to capture 64-bit values.

Overhead. The overhead of profiling, as discussed in Section 3.5, comes from transferring state to and from the co-processor and the associated cache pollution. It is difficult to compare the overhead of the two algorithms; the simple-sampling approach has a fixed overhead per unit time, whereas the co-processor algorithm will cease execution when it is satisfied with the profile collected, and, hence, its overhead is largely a function of the size of the program's instruction working set. Percentage slowdowns, although not the most meaningful way to represent the overhead of the co-processor algorithm, provide a means to compare the two algorithms.

The measured overheads, shown in Figure 6(b), include all initialization and the processing required to set the filter bits but not the final estimation algorithm (described in Section 5.1.5). This final estimation overhead is linear with the number of static loads profiled. For both mechanisms all overheads are less than 2%, and in all cases, the overhead of the co-processor algorithm is lower. This is largely due to the co-processor interrupting the program fewer times and, therefore, making fewer updates to the in-memory data structure. The overhead of each update is higher for the co-processor algorithm (because typically multiple counters have to be updated and the decision as to whether to mask must be made), but because the number of updates is so much lower, the overhead is lower.

Convergence Rate. Potentially more important for a dynamic optimization scenario is the rate at which the profiles converge. Figure 7 shows the relationships between accuracy, overhead and time for both the co-processor algorithm and simple sampler on *gcc*. It can be seen that the co-processor's profile converges much more quickly. It interrupts frequently in the beginning because it can quickly characterize the frequent instructions. This leads to higher overhead initially, but as it shifts to profiling the less frequent instructions, the interrupt frequency drops, leading to a decrease in overhead. It can be seen in Figure 7c that the co-processor has better accuracy (lower error) for any given overhead; this is true for all of the benchmarks.

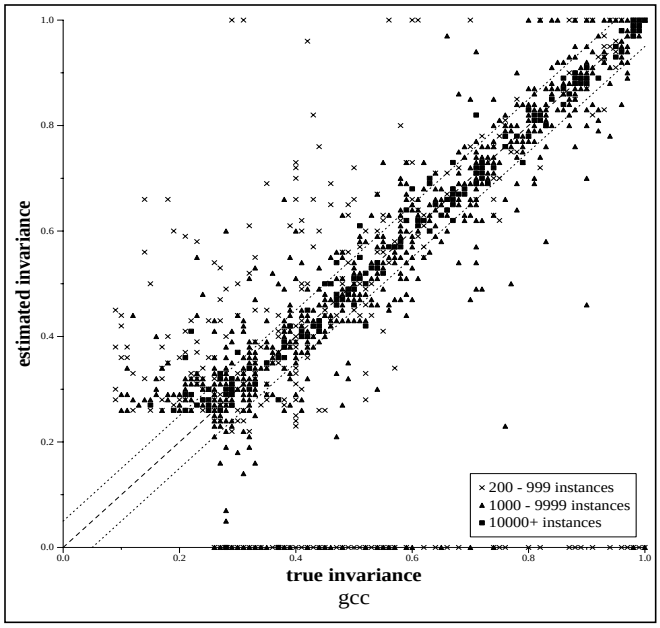

Figure 5. *Scatter plot that shows the accuracy of our invariance estimates on a per-value basis. Data is shown for the benchmark gcc with the baseline array size (256 slots). Because of the sheer number of such points, we chose not to plot points with estimated and actual invariance below 25 percent; this causes the empty box in the lower left.*

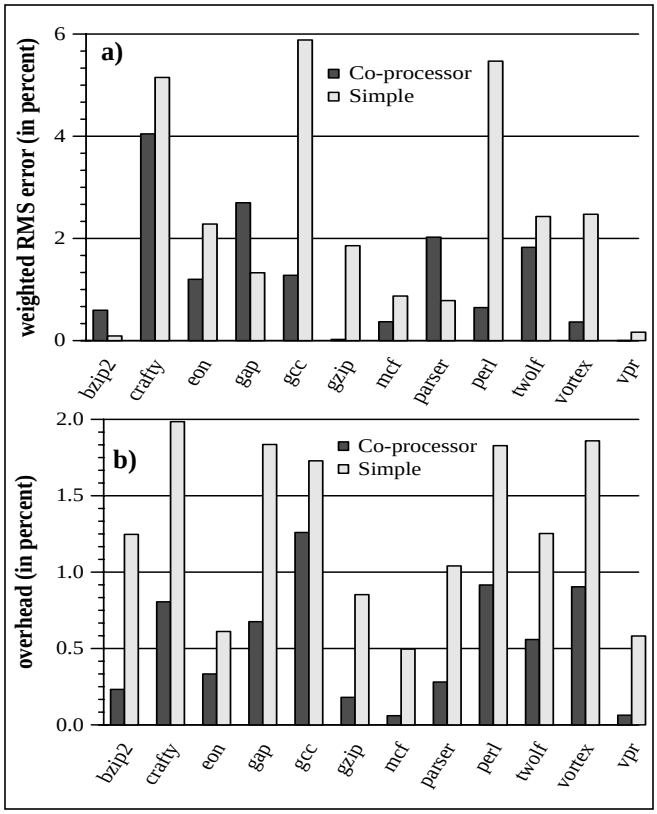

Figure 6. *Results for all benchmarks: (a) weighted root-mean-square error in invariance estimation, (b) overhead. In both cases, smaller numbers are better.*

Gcc and *perl*, the largest benchmarks and, hence, the most likely to represent real workloads, best demonstrate the capabilities of our co-processor algorithm. Both see a substantial increase in accuracy (almost 4 and 8 times, respectively) with a significant reduction in overhead (30% and 45% less, respectively).

Sensitivity Analysis. To explore the sensitivity of our profiler on the co-processor's design, we ran the same experiments with two other configurations: one with twice the storage resources and one with half as much. The major differentiation between the configurations is that the rate at which the profiles can be captured is almost linearly related to array size (shown for *gcc* in Figure 8). The accuracies and overheads, on the other hand, are less dramatically affected (shown in Figure 9). In general, the accuracies are better with more resources, but the relationship is sub-linear. In some cases, smaller arrays provide better profiles. This is due to second-order effects including when interrupts are performed and the interaction between filtering and the rate at which samples can be processed. More interestingly, smaller arrays tend to have slightly lower overhead because the increase in interrupt frequency is less than the corresponding decrease in samples per interrupt.

The frequency at which the co-processor executes does not appear to have a substantial impact on its results. Comparing results between co-processors clocked at 1, 1/2, 1/4 and 1/8 the main processor's frequency, we found that all were able to achieve comparable profile accuracies eventually, and achieve those accuracies with roughly equivalent overheads. The rate at which the profiles were collected was somewhat affected for the 1/4 and 1/8 cases in the largest benchmarks (*e.g., gcc, perl*), but it appears to be largely due to poor decisions by the profile software on when to perform interrupts. We expect that further tuning of the algorithm could mitigate much of this penalty.

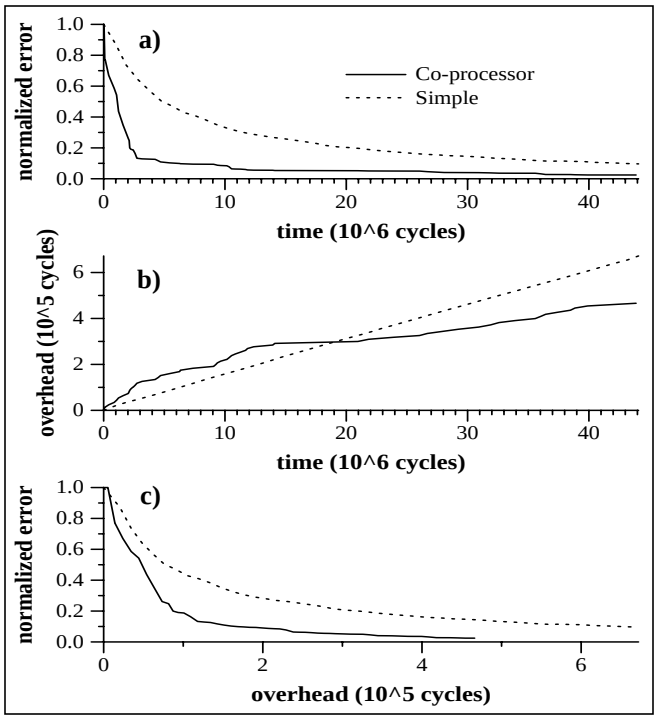

Figure 7. *Relationship between accuracy, overhead, and time for co-processor and simple algorithms. The co-processor algorithm converges faster (a) by taking much of its overhead early (b) and by having better accuracy for a given amount of overhead. Data shown for gcc.*

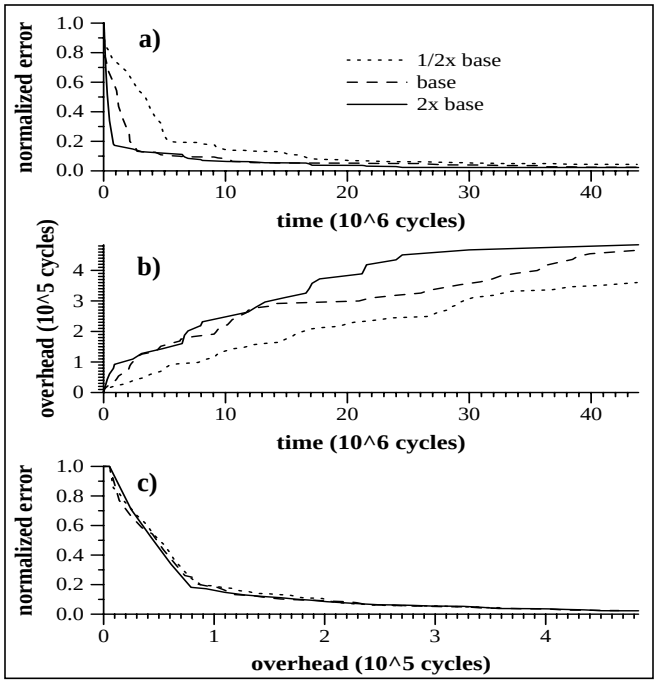

Figure 8. *The rate at which a profile can be collected is sensitive to the storage resources in the co-processor (a), although the overhead for collecting a profile of a given accuracy is not (c). Data shown for gcc.*

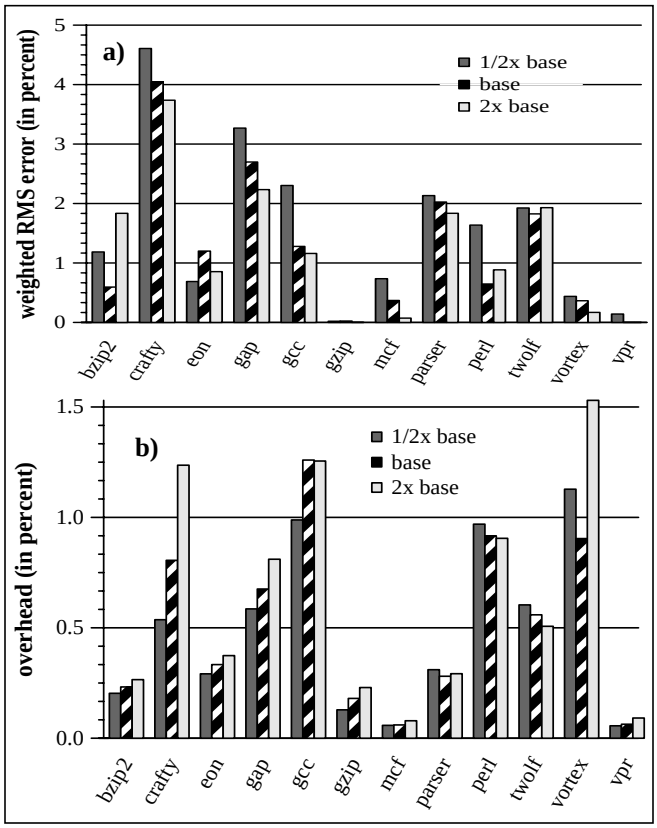

Figure 9. *Profiling accuracy (a) and overhead (b) have only moderate sensitivity on amount of storage resources available to the co-processor.*

6 Related Work

Our proposed profiling architecture builds upon previous work in instruction-based sampling [18, 39]. In these schemes, an instruction is selected for profiling at fetch time and, as the instruction proceeds down the pipeline, each stage captures information about it. In addition to micro-architectural events, our system assumes that the architectural state associated with the instruction (*e.g.*, register values) is captured also, as is proposed in [14]. Unlike these approaches that interrupt the processor after every sample, our scheme can buffer multiple samples as well as condense the profile data into a reduced form. In addition, we include a filter that constrains which instructions are selected for sampling.

Other previous work has proposed mechanisms to coalesce profile information from multiple samples for specific types of profiling, typically control flow profiling. Conte *et al*. proposed the *profile buffer* [16], which tracks retired branch outcomes by incrementing counters based on branch address. Merten *et al*. propose a slightly more complicated structure, the *hot spot detector*, which uses retired branch outcomes to identify program hot spots and captures branch biases [32]. In [33], they extend the hot spot detector to enable autonomous re-layout of frequently executed program regions. In these schemes, the merging algorithm is implemented in hard-wired logic, while our proposed profiling approach trades off efficiency for flexibility by using a programmable co-processor to perform the reduction.

Recently, Chou *et al*. described an *instruction path co-processor* (I-COP) [13] another programmable co-processor that observes the retirement stream. They describe how their array of processors can be used for trace construction and optimization for trace caches, and estimate that the I-COP is roughly the size of 256KB of fast SRAM. Because the profiling co-processor exclusively performs profiling, the hardware can be tailored to provide high profiling rates with resources more than an order of magnitude smaller. In addition, because a richer set of information is provided with the retirement stream, the profiling co-processor can collect a broader range of profiles.

Concurrently with this work, Heil and Smith proposed the *relational profiling architecture* (RPA) [23]. This architecture shares much in common with the profiling co-processor approach, but is designed to exploit characteristics of their underlying co-designed virtual machine model. For example, instructions include an extra bit field that can be set to control whether or not instructions should be profiled (serving the same purpose as our PC filter bit array). Rather than post-processing samples in a dedicated co-processor, the RPA sends messages to *service threads*, which execute on small, general-purpose, peripheral processors. In addition, they demonstrate that the datapath for communicating profile information out of the core should be feasible, and propose the concept of assured sampling, where every instance of a set of instructions is monitored, enabling the RPA to be used to ensure correctness in the presence of speculative optimizations.

Programmable co-processors have been previously proposed to manage other sub-tasks on the processor's behalf. Kuskin *et al*. proposed using a programmable co-processor, MAGIC, to implement cache coherence [28], and they demonstrated its use for coherence performance monitoring [31]. More recently, Hallnor and Reinhardt proposed a software-managed cache that uses a co-processor to implement the replacement policy [22].

7 Conclusion

We propose a new profiling architecture based around a profiling co-processor that performs local post-processing on the profiling data. This post-processing condenses sample data before passing it to the main processor, thereby reducing the overhead of collecting each data sample. This post-processing can be performed efficiently because the co-processor implements common profiling operations in hardware as primitives. The co-processor should only modestly impact the design of the main processor because we estimate that it can be implemented in less than a half million transistors, the majority of which can be far away from the main processor's core.

Because the co-processor is programmable, it is capable of collecting a wide range of profile data. As a demonstration we implemented load value profiling. This application can be demanding because it has a large state space of values in which to detect patterns. Our profiling algorithm monitors its own results and terminates profiling an instruction when it is confident that it can estimate its invariance. In this way, the algorithm successively profiles instructions from most frequent to least frequent, and terminates when it is satisfied it has a good enough profile. We believe that the structure of this profiling algorithm and the techniques to implement it are applicable to other types of profiling.

8 Acknowledgements

The authors would like to thank Amir Roth, Milo Martin, Ras Bodik, and the anonymous reviewers for their comments on earlier drafts of this paper. This work was supported in part by National Science Foundation grants MIP-9505853, CCR-9900584 and EIA-0071924, donations from Intel and Sun Microsystems, and the University of Wisconsin Graduate School. Craig Zilles was supported by an Intel Foundation Graduate Fellowship and Wisconsin Distinguished Graduate Fellowships during the academic years 1999 and 2000, respectively.

9 References

[1] Intel Corporation. Vtune: a visual tuning environment. http://support.intel.com/support/performancetools/vtune/.

[2] J. Anderson, et al. Continuous Profiling: Where have all the cycles gone? In *Proc. 16th Symposium on Operating System Principles*, Oct. 1997.

[3] J. Auslander, M. Philipose, C. Chambers, S. Eggers, and B. Bershad. Fast, effective dynamic compilation. In *Proc. SIGPLAN'96 Conference on Programming Language Design and Implementation*, pages 149–159, May. 1996.

[4] T. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *Proc. 32nd International Symposium on Microarchitecture*, pages 196–207, Nov. 1999.

[5] T. Autrey and M. Wolfe. Initial results for glacial variable analysis. In *Proc. 9th International Workshop on Languages and Compilers for Parallel Computing*, August 1996.

[6] V. Bala, E. Duesterwald, and S. Banerjia. Transparent Dynamic Optimization. Technical Report HPL-1999-77, Hewlett Packard Labs, June 1999.

[7] T. Ball and J. Larus. Efficient Path Profiling. In *Proc. 29th International Symposium on Microarchitecture*, pages 46–57, Dec. 1996.

[8] D. Burger and T. Austin. The SimpleScalar Tool Set, Version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin-Madison, Jun. 1997.

[9] M. Burrows, et al. Efficient and Flexible Value Sampling. In *Proc. 9th Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.

[10] B. Calder, P. Feller, and A. Eustace. Value Profiling. In *Proc. 30th International Symposium on Microarchitecture*, pages 259–269, Dec. 1997.

[11] B. Calder, P. Feller, and A. Eustace. Value Profiling and Optimization. *Journal of Instruction Level Parallelism*, March 1999.

[12] B. Calder, G. Reinman, and D. Tullsen. Selective Value Prediction. In *Proc. 26th International Symposium on Computer Architecture*, pages 64–74, Jun. 1999.

[13] Y. Chou, P. Pillai, H. Schmit, and J. Shen. PipeRench Implementation of the Instruction Path Coprocessor. In *Proc. 33rd International Symposium on Microarchitecture*, Dec 2000.

[14] G. Chrysos, J. Dean, J. Hicks, C. Waldspurger, and W. Weihl. Apparatus for sampling instruction operand or result values in a processor pipeline. US Patent 5923872, July 1999.

[15] C. Consel and F. Noel. A general approach for run-time specialization and its application to C. In *Proc. Conference on Principles of Programming Languages*, pages 145–156, Jan. 1996.

[16] T. Conte, K. Menezes, and M. Hirsch. Accurate and practical profile-driven compilation using the profile buffer. In *Proc. 27th International Symposium on Microarchitecture*, pages 36–45, Dec. 1997.

[17] T. Conte, B. Patel, and J. Cox. Using branch handling hardware to support profile-driven optimization. In *Proc. 27th International Symposium on Microarchitecture*, pages 11–21, Nov. 1994.

[18] J. Dean, J. Hicks, C. Waldspurger, W. Weihl, and G. Chrysos. ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors. In *Proc. 30th International Symposium on Microarchitecture*, pages 292–302, Dec. 1997.

[19] D. Engler, W. Hsieh, and M. Kaashoek. 'C: A language for high-level, efficient, and machine-independent dynamic code generation. In *Proc. Conference on Principles of Programming Languages*, pages 131–144, Jan. 1996.

[20] F. Gabbay and A. Mendelson. Can program profiling support value prediction? In *Proc. 27th International Symposium on Microarchitecture*, pages 270–280, Dec. 1997.

[21] S. W. Golumb. *Shift Register Sequences*. Aegean Park Press, revised edition, 1982.

[22] E. Hallnor and S. Reinhardt. A Fully Associative Software-Managed Cache Design. In *Proc. 27th International Symposium on Computer Architecture*, June 2000.

[23] T. Heil and J. Smith. Relational Profiling: Enabling Thread Level Parallelism in Virtual Machines. In *Proc. 33rd International Symposium on Microarchitecture*, Dec 2000.

[24] J. Hollingsworth, B. Miller, and J. Cargille. Dynamic Program Instrumentation for Scalable Performance Tools. In *Proc. Scalable High Performance Computing Conference '94*, May 1994.

[25] M. Horowitz, M. Martonosi, T. Mowry, and M. Smith. Informing Memory Operations: Providing Memory Performance Feedback in Modern Processors. In *Proc. 23rd International Symposium on Computer Architecture*, pages 260–270, May 1996.

[26] R. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2), Mar./Apr. 1999.

[27] A. Klaiber. The Technology Behind Crusoe Processors. Technical report, Transmeta Corporation, Jan. 2000.

[28] J. Kuskin, et al. The Stanford FLASH Multiprocessor. In *Proc. 21st International Symposium on Computer Architecture*, pages 302–313, Apr. 1994.

[29] M. Lipasti, C. Wilkerson, and J. Shen. Value Locality and Load Value Prediction. In *Proc. 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, Oct. 1996.

[30] M. Martonosi, D. Clark, and M. Mesarina. The SHRIMP Hardware Performance Monitor: Design and Applications. In *Proc. 1996 SIGMETRICS Symposium on Parallel and Distributed Tools*, May 1996.

[31] M. Martonosi, D. Ofelt, and M. Heinrich. Integrating Performance Monitoring and Communication in Parallel Computers. In *Proc. 1996 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1996.

[32] M. Merten, A. Trick, C. George, J. Gyllenhaal, and W.-M. Hwu. A Hardware-Driven Profiling Scheme for Identifying Program Hot Spots to Support Runtime Optimization. In *Proc. 26th International Symposium on Computer Architecture*, pages 136–147, Jun. 1999.

[33] M. Merten, A. Trick, E. Nystrom, R. Barnes, and W.-M. Hwu. A Hardware Mechanism for Dynamic Extraction and Re-layout of Program Hot Spots. In *Proc. 27th International Symposium on Computer Architecture*, June 2000.

[34] J. Montanaro et al. A 160-Mhz, 32-b, 0.5-W CMOS RISC Microprocessor. *IEEE Journal of Solid-State Circuits*, 31(11):1703–1712, Nov. 1996.

[35] T. Nakra, R. Gupta, and M. Soffa. Value Prediction in VLIW machines. In *Proc. 26th International Symposium on Computer Architecture*, pages 258–269, Jun. 1999.

[36] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proc. SIGPLAN'94 Conference on Programming Language Design and Implementation*, 1994.

[37] P. Tryfos. *Sampling Methods for Applied Research: Text and Cases*. John Wiley & Sons, Inc., 1996.

[38] D. Tullsen and J. Seng. Storage-less Value Prediction using Prior Register Values. In *Proc. 26th International Symposium on Computer Architecture*, pages 270–279, Jun. 1999.

[39] D. Westcott and W. White. Instruction sampling instrumentation. US Patent 5151981, Sept. 1992.

[40] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz. Performance Analysis Using The Mips R10000 Performance Counters. In *Proc. 1996 Conference on Supercomputing*, Nov. 1996.

[41] X. Zhang, Z. Wang, N. Gloy, J. Chen, and M. Smith. System Support for Automatic Profiling and Optimization. In *Proc. 16th Symposium on Operating System Principles*, Oct. 1997.

Appendix A Value Profiling Algorithm Details

This section covers details that are important for achieving the results presented for our value profiling algorithm, but aren't required by a casual reader. A flow-chart (which is referred to in the text) for the value profiling algorithm is shown in Figure 10.

A.1 Value Re-sampling

In our implementation, if the measured invariance drops below 20 percent (tested in state D of Figure 10), we re-select the active value (in state H of Figure 10) and set the hit and miss counters to 1 and 0, respectively. In addition to updating hit and miss counters, each sample increments the total counter that enables us to track how many samples have been dropped when the active value is re-selected.

A.2 Counter Saturation

When a slot has learned something useful, which could be damaged by additional updates, we turn it off (by clearing its valid bit in the associative array, so no further matches occur). The most

important case is the saturation of either of the hit or miss counters (state F, in Figure 10). When this occurs, no further updates are added to the slot, in order to preserve the ratio of hits to misses. Also, if we decide to select a new active value, but the previous active value has accrued a non-trivial number of samples (at least 16 in our current implementation), then we turn off the slot to retain that information (state I). Lastly, in the uncommon case that the total counter saturates, we turn off the slot to maintain an accurate count of the number of samples observed. When a static instruction's slot has been turned off, new instances of that instruction have to compete for the remaining slots like instructions that never had a slot.

A.3 Informational Replacement

Because it would be expensive to consider all slots for replacement, we only consider one (which is denoted as *rslot* in Figure 10). Our current scheme has an index into the array of slots that tracks the next slot up for replacement. When a slot miss occurs, the total counter is compared to a pseudo-random number between 0 and 15, inclusive. If the random number is greater, the slot is replaced, *i.e.*, the new instruction's value is stored as the new active value and the hit and miss counters are reset (state M in Figure 10). Otherwise, the replacement fails and the current sample is ignored (state K). In this way relatively infrequently executed instructions might not make it into the array until the frequent ones have been masked. Whether the entry is replaced or not, the index of the next slot up for replacement is incremented, giving a new slot some time with which to take hold before being re-considered for replacement.

A.4 Interrupt Strategy

Eventually, most of the slots will become un-replaceable (when their total count exceeds 15). As long as the current working set of instructions is represented in the slots we can continue collecting data. When we start ignoring many samples from instructions that are not allocated to slots, it is time to copy the profiler state to main memory and recommence with a clean slate. To identify when to perform an interrupt, our algorithm includes a saturating failed counter, which is incremented every time a replacement fails and is decremented every time one succeeds. The counter is initialized to zero, and when it hits its threshold, it interrupts the processor. In many cases, this mechanism implicitly detects program phase changes because the new instruction working set is not represented in the slot array.

A.5 Thrash Detection and Filtering

Our algorithm counts the number of replacements, and it interrupts the processor when it exceeds a threshold. Because replacements are infrequent when not thrashing, a low threshold (four times the number of slots) allows thrashing to be detected quickly with a minimum number of false positives.

When thrashing is detected, we attempt to estimate the extent of it to decide how finely the program should be partitioned. Our algorithm looks at the number of samples processed (an indicator of how fast thrashing was detected) to decide if the current partition should be sub-divided in 2 or 4 ways. When an interrupt is caused by thrash detection, we do not copy the captured samples to memory because they generally have low information content.

A.6 Estimation Algorithm

Our algorithm assumes that the captured samples are representative of the full program and estimates invariances by first estimating the number of captured samples that belong to each value. Each value gets samples from three sources: (1) its own hits counter, (2) a portion of the miss counters from other values, and (3) a fraction of the remaining samples measured by the total counter. Each value gets all of its associated hit samples. The miss samples for each value are divided between the other values proportionally according to the number of hit counts those values have. Finally, each value gets a share of the counts measured by the total counter that are un-accounted for. These counts are divided equally between values, but no value is given more than 20 percent of them because these counts are derived from regions where the active value had an invariance of less than 20 percent. The estimated counts are divided by the total number of samples recorded to compute each value's invariance.

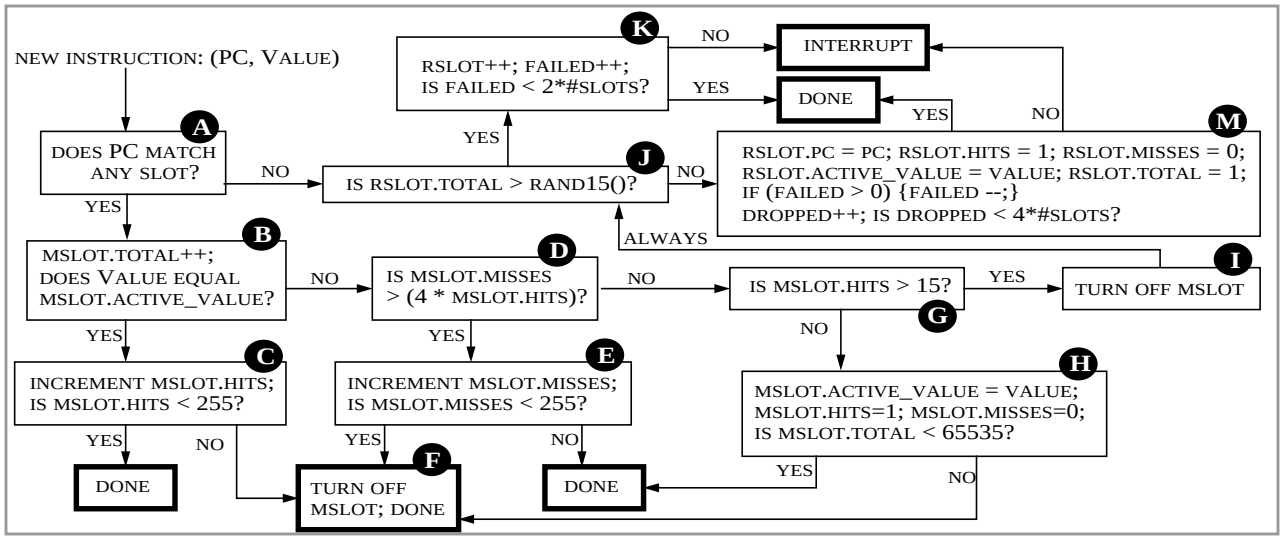

Figure 10. *Flowchart of the co-processor value profiling algorithm: Each instruction processed by the co-processor has a PC and a VALUE; MSLOT is the slot with a matching PC, if one exists; RSLOT is the next slot up for replacement; FAILED and DROPPED are counters maintained to determine when to interrupt the main processor; RAND15() generates a random number between 0 and 15; final states (DONE, INTERRUPT) have a darker border*