Rapid Profiling via Stratified Sampling

S.Subramanya Sastry
Computer Sciences Dept.
University of Wisconsin-Madison
sastry@cs.wisc.edu

Rastislav Bodík
Computer Sciences Dept.
University of Wisconsin-Madison
bodik@cs.wisc.edu

James E. Smith
Dept. of ECE
University of Wisconsin-Madison
jes@ece.wisc.edu

Abstract

Sophisticated binary translators and dynamic optimizers demand a program profiler with low overhead, high accuracy, and the ability to collect a variety of profile types. A profiling scheme that achieves these goals is proposed. Conceptually, the hardware compresses a stream of profile data by counting identical events; the compressed profile data is passed to software for analysis. Compressing the high-bandwidth event stream greatly reduces software overhead. Because optimizations can tolerate some profiling errors, we allow the stream compressor to be lossy, thereby enabling a low-cost sampling-based hardware design. Because the hardware compressor is insensitive to the event content, it supports various profile types and can process multiple types simultaneously.

Basic components of our framework are periodic and random samplers, counters, and hash functions. These components are composed to form a variety of stream compressors. One design is both simple and very effective: the input stream is hash-split into multiple substreams, each of which is fed into a simple periodic sampler that selects every kth event. This stratified periodic sampler *performs better than conventional random sampling because it biases each substream towards a small number of unique events, thereby reducing sampling error, and allowing faster convergence to an accurate profile. For example, convergence to a given level of accuracy is about twice as fast for* gcc*. When sampling overhead is considered, the stratified periodic profiler achieves less than 3% error while incurring an overhead of only 3.5% for* gcc*.*

1 Introduction

Profile-directed optimizations improve program performance by analyzing a program's dynamic profile that provides information that static compile-time analysis typically cannot infer. Recent evolution of profile-directed optimizations has been shaped by two trends. First, profiles have become indispensable in a spectrum of *advanced* optimizations that include trace scheduling [20] and extend well beyond it: basic-block and path profiles [6,43] identify hot spots in the program; call-graph profiles [1] guide procedure inlining [3,9,10]; dynamic-type profiling removes indirect calls in object-oriented languages [25,26]; value-invariance profiles lead to program specialization [8,11,32,36]; and memory-conflict profiles allow aggressive load-store reordering [18,21].

The second trend is toward *dynamic* optimizations [4,7,19,29,34,40]. By optimizing the program during its execution, dynamic optimizers not only make the tedious compile/profile/re-compile cycle transparent to the user, but are also freed from some problems inherent in static optimization; namely, they gain access to the fully-linked binary and the actual input data values.

Our interest is in providing profiling support for dynamic optimizations. Ideally, a profiler suitable for dynamic optimization should have the following properties:

- *Low overhead.* Because profiling takes place during execution, its overhead must be significantly smaller than the optimization benefit. A typical benefit of about 10% speedup severely constrains the tolerable profiling overhead.
- *High accuracy.* The higher the accuracy of the profiler, the faster the profile converges to within an acceptable error. Rapid profiling, in turn, leads to earlier optimization and correspondingly longer execution in the optimized mode. In Dynamo [17], rapid selection of hot paths was important for maximizing returns from dynamic optimizations.
- *Broad applicability.* The diversity of dynamic optimizations calls for a versatile profiler that can measure diverse properties of control flow, addresses, and data values.
- *Simultaneous profiling.* Sometimes it is convenient to collect multiple profiles simultaneously. For example, if the dynamic behavior of an optimized procedure changes, the optimizer may want to trigger its reoptimization. If multiple profiles can be collected simultaneously, monitoring of changes can run on the background of continuous optimizations.
- *Low cost and complexity.* Minimal hardware support and simple software algorithms bring the well-known benefits of reduced power consumption and verifiability.

Related Work. Let us review related work with respect to the above ideal properties, focusing on three distinct implementation categories: smart software profilers, custom hardware profilers, and hybrid profilers.

Smart software profilers: The first group of software profilers instruments the program with profiling instructions. One method for reducing the overhead of executing the additional instructions is to exploit the program structure: Ball-Larus edge profiling [5] and path profiling [6] use program analysis and manage to restrict overheads to 10-30%. Other tricks for reducing the instrumentation overhead include restricting profiling to a subset of instructions [8,36] and turning off profiling after the profile stabilizes [8]. Despite recent advances, profiles that measure more than the control flow incur high overheads. For example, the best software value profiler slows down the program 10–30 times [8,32].

The second software approach is sampling. Recently, sampling techniques have been used to obtain value profiles with low overhead (about 10%) [33]. However, the low sampling rate (1 every

1063-6897/01 $10.00 © 2001 IEEE

32,000 instructions) increases the time before optimizations can be performed.

Custom hardware support: Conte *et al.* proposed a *profile buffer* [13] and Merten *et al.* described a *hotspot detector* [35]. While these specialized designs work very well for their specialized purpose, they cannot be used to collect other kinds of profiles.

Hybrid profilers: In these solutions, programmable hardware collects profiling information, potentially performs some simple profile preprocessing, and passes on the information to software which then does a more complete profile analysis. ProfileMe [15], the Relational Profiling Architecture (RPA) [42], and the programmable profiling co-processor [14] are examples of this approach. ProfileMe [15] provides mechanisms of instruction-based profiling wherein the hardware picks instructions and collects a variety of information as instructions flow down the pipeline. The information is post-processed by software. The other two solutions (RPA and the profiling co-processor) provide more flexible profiling abilities than ProfileMe, by supporting a wider range of profiles to be collected in hardware and by enabling hardware preprocessing of profile information that reduces post-profile analysis overheads in software.

Proposed Solution: We explore a method that combines the advantages of hardware and software. While hardware is suitable for high-bandwidth data processing, software is a better fit for irregular processing of small amounts of data. We choose a hybrid hardware-software approach because purely hardware approaches are usually inflexible and are targeted at specific optimizations. Purely software approaches tend to incur relatively large overheads and/or require low sampling rates which can lead to long profiling times.

Our solution is motivated by the observation that most kinds of profiles compute *execution counts* of certain events of interest (values computed by an instruction, targets of a branch, targets of a call, or addresses of a load). This observation leads to a *stream compression* profiling model. In this model, the processor generates a stream of profiled events whose type is selected by software. The stream is a sequence of data tuples: for example, a value-profiling tuple contains the PC of a load instruction and the loaded value. Dedicated hardware compresses this stream before passing it on to software. The basic compression method collapses and counts identical tuples. Consequently, profiling overhead is reduced because software processes a shorter stream.

A conceptual view of the hybrid stream compression profiling model is shown in Figure 1. The *selector* creates a tuple for each retired instruction chosen for profiling and sends the tuple to the *compressor*. The compressor, placed off the processor's critical path, consumes the selected tuple stream. The compressor summarizes the input stream and feeds it to profiling software through an intermediate buffer. In Figure 1, the input tuple stream $t_1, t_2, t_1, t_2, t_3, t_2$ may be compressed into an output tuple stream $(t_1, 2), (t_2, 3), (t_3, 1)$. The second element in the output pairs denotes the number of occurrences of the tuple.

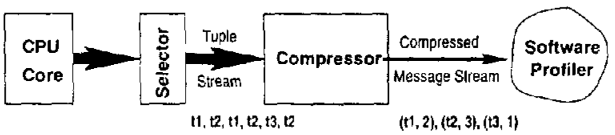


Figure 1: Abstract diagram of the stream-compression profiling model.

The variety of profiles that can be collected by the stream-compression model depends on the flexibility of the software-controlled selector. As a case study, we collect value profiles required for value reuse optimizations (software-directed instruction reuse and program specialization), a demanding profiling application. Furthermore, we will show that the hybrid profiling scheme can collect multiple profiles simultaneously. As an application, we show that edge profiles can be collected simultaneously with call target profiles with high accuracy and low overhead.

In this paper, we primarily focus on designing "efficient" compressors that incur low overhead by means of high compression of the tuple stream, while preserving adequate profile accuracy. We present a set of profiling components that can be composed in a number of ways to yield different compressors. Because optimizations can tolerate some profiling errors, we allow the stream compressor to be lossy, thereby enabling a low-cost sampling-based hardware design. Conventional simple random sampling, simple periodic sampling, stratified random sampling, and stratified periodic sampling are among the compressors considered. *Stratified sampling* is a technique in which the population is split into multiple disjoint sub-populations (strata) which are then sampled independently [22]. In our implementation, we use a hash function to split the input stream. Using load value profiling as a case study, we experimentally show that given a fixed overhead budget and a desired level of accuracy, the stratified periodic sampler achieves the desired accuracy the fastest.

The contributions of this paper can be summarized as follows:

- We present a h/w-s/w hybrid profiling model based on the stream compression metaphor.
- We present a framework of components for constructing a variety of compressing profilers.
- Specifically, we propose stratified periodic sampling as the compression method of choice. Our evaluation shows that with fixed profiling overheads and accuracy levels, the stratified sampler achieves the desired accuracy at least twice as fast as a random sampler.

The rest of this paper is organized as follows. Section 2 presents more details on the tuple selector as well as the software side of the hybrid profiler. Section 3 focuses on the various sampling compressors and their properties obtained using Monte Carlo simulations. Section 4 describes a more sophisticated, tagged compressor that serves as a reference point in our experiments. Section 5 evaluates the compressors using real benchmarks. Finally, Section 6 summarizes our findings.

2 Details of the hybrid profiling scheme

There are three main components of the hybrid profiling model: the selection mechanism, the compression mechanism, and the communication of information to software. In this section, we discuss the first and third of these components. Compression mechanisms are central to our work and are discussed in greater detail in the following section.

2.1 Selecting profile information

Referring to Figure 1, the "front-end" of the profiler is a selection mechanism. Although specific selector implementations are not our main focus, one possibility is programmable selector hardware. In particular, registers in the selector can be written by profiling software via special instructions. These registers can then be used by hardware to select the specific instructions to be profiled. Heil and Smith [42] describe such mechanisms as part of their relational profiling architecture. Below, we describe a few selection mechanisms.

- **Selection by opcode**: A number of profiling applications can be implemented by selecting instructions on the basis of opcode. This strategy is a low-cost solution that does not require any modification to the ISA or the program binary. This mechanism is proposed in [42] and [14].
- **Selection by PC range**: This strategy is useful for focusing on hot segments of a program or for reducing the profile bandwidth to the hardware profiler. This mechanism was proposed in [14].
- **Binary modification**: This mechanism places explicit instructions in the binary immediately before an instruction to be profiled. This mechanism requires one extra opcode in the ISA and also introduces overheads in the instruction stream. However, this method can be used to supplement the above two selection methods because it enables profiling sets of instructions that are not otherwise easily selected.

In this paper, we simply select instructions for profiling based on opcode. Furthermore, for the common profiling applications we study, two word profile tuples are sufficient.

Because the compressor does not interpret the contents of profile tuples, the same hardware can be used without modification for collecting a variety of profiles as listed below.

- Value Profile: the tuple is `<PC, instr-output>`.
- Edge Profile: the tuple is `<PC, branch-target>`.
- Call Target Profile: the tuple is `<PC, call-target>`.
- Type Profile: the tuple is `<PC, method-table-addr>`; this profile enables feedback-directed inlining [3] and polymorphic inline caching [26].

Furthermore, because the compressor does not interpret the tuple, it is possible to run multiple profiling applications simultaneously. The software is responsible for distinguishing between the incoming tuple messages. When profiles being collected can be distinguished solely on the basis of opcode (e.g., edge profile and call target profile; the former profiles branches and indirect jumps and the latter profiles profiles direct and indirect calls), the software can distinguish between incoming messages on the basis of the message PC. On the other hand, if edge profiles and a profile of mispredicted branches are to be collected, this approach cannot be used because branches are common to both profiles. In such cases, the selector hardware can assign different tags to instructions that belong to different profiling applications. For example, the selector can assign mispredicted branches with a different tag than correctly predicted branches.

To demonstrate the feasibility of running multiple profiling applications concurrently, we collect edge profiles and call-target profiles simultaneously and present accuracy and overhead results in Section 5.

2.2 HW-SW communication mechanisms

There are two mechanisms for communicating tuple messages to software. The first mechanism is based on processor interrupts and the second is based on message passing.

Interrupt-based approach. In this approach, tuple information is stored in a hardware buffer. When the buffer fills, the main processor is interrupted. The interrupt handler reads the buffer and folds the tuple information into the profile data being collected. This approach is used in [2, 14].

Message-passing based approach. In this approach, profile information is communicated to concurrent software threads via shared queues in memory. These concurrent threads read profile messages from the shared queues and compute the profile. An example of this approach are the *service threads* outlined in [42]. Service threads running on simple service processors read and process the messages. The relevant overhead metric for this approach is the time between consecutive messages sent to the service threads. The time should be long enough that the profile service thread can completely process the message.

In our experimental evaluation, we use the interrupt-based communication model because it is readily implementable in current generation processors.

3 A framework for designing compressors

This section focuses on the lossy stream compressor, which is the most novel part of our hybrid profiling scheme. Instead of presenting a few independent compressor designs, we describe a framework of components from which various compressors can be built.

3.1 Samplers as compressors

Samplers can serve as stream compressors because selecting a subset of the input stream reduces the bandwidth of the output stream. Samplers count the input events statistically: an event t selected by a sampler operating at a rate r can be interpreted by software as r occurrences of tuple t compressed into one. Clearly, such stream compression is *lossy*, because events skipped by the sampler may have been different than t. However, when the stream is *biased* towards tuple t (i.e., the stream is dominated by t), the sampler's accuracy may be sufficient for profiling purposes. This subsection presents two basic samplers used in our framework, and the following subsection focuses on increasing the bias in the input stream.

- A *random sampler* with rate r, denoted R_r, selects an element of the input stream with probability $p = 1/r$. Note that random samplers in [2, 15, 33] are slightly different; they select an element via a countdown register initialized with a random number from the interval $\langle 1, 2r \rangle$.
- A *periodic sampler* with rate r, denoted P_r, selects every rth element of the input stream. In statistical literature, this sampler is known as a *systematic* sampler [22].

In order to design an accurate compressor, the inherent accuracy of the two samplers must be understood. In this section, we assume an idealized input stream in which tuples are randomly permuted (the reason for this assumption is that an input stream with periodic behavior may cause large errors with the periodic sampler P_r). In Section 5, we will evaluate compressors on real workloads.

We compare R_r and P_r using a very simple profiling problem. Assume the *output* stream of the compressor contains k elements. The problem is to determine how many elements were in the input stream. Clearly, the answer for both R_r and P_r is rk. More precisely, rk is the *most likely* number of elements seen in the input. The important difference between the two samplers is *how* likely it is that their input actually contained rk elements. The P_r sampler is more confident about its answer because it effectively counts the input stream; its input stream length is **guaranteed** to contain between rk and $rk + (r - 1)$ elements. On the other hand, the length of the input stream for R_r can range from k to infinitely many elements. Therefore, R_r **estimates** the lengths of the input stream. This explanation is validated experimentally. The error of the two samplers is shown in Figure 4 using an experiment described in detail in Section 3.3.

We can address the drawback of R_r by adding to it a counter that measures the length of the input stream. We use C to denote a counter component of our framework, and CR_r to denote a random sampler equipped with such a counter. With this enhancement, CR_r has access to the same information as P and the two are thus equivalent.

3.2 Stratified sampling via hashing

As mentioned in the previous subsection, the accuracy of sampling increases with the bias in the input stream: the more a tuple dominates the stream, the less likely is the sampler to make a mistake. An effective technique for increasing the bias is to stratify the input population into disjoint sub-populations which are then independently sampled. This technique, known as *stratified sampling* [22], can be conveniently implemented in the profiling context using hashing. Because the input stream is split based on tuples having the same hash signature, it can be reasoned that any given substream has a greater bias (or, smaller entropy, in information theoretic terms; lower variance, in statistical terms) than the original input stream. Hence, the samples that are selected from each substream are more accurate representatives of the input stream than a corresponding same-size sample selected from the original input stream.

We use $H[X_r]_n$ to denote a *hash-based splitter* that splits the input stream into n disjoint substreams using a hash function. Each of the substreams is independently sampled at the rate of r using any sampler X. We discuss the stratified sampler in more detail in the following subsection. Its implementation details are described in Section 3.5.

3.3 Composing profiling components

Figure 2 pictorially shows the four profiling components R_r, P_r, C, and $H[X]_n$ that we presented in the previous subsections.

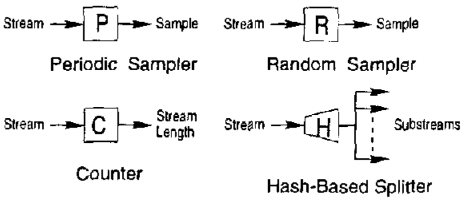


Figure 2: Hardware profiling components.

The input to each of these components is the input stream. The random and periodic samplers produce on the output a sample of the input stream. The hash-based splitter produces multiple disjoint input streams. The counter produces the length of the input stream, i.e. the number of input tuples. We now show how these components can be composed to produce different sampling schemes.

Figure 3 shows six different samplers created by combining the four basic components: P_r, R_r, CR_r, $H[P_r]_n$, $H[R_r]_n$, and $H[CR_r]_n$. All samplers compress the input stream into a stream of messages $\langle tuple, count\rangle$. For the samplers with the counter component C, the value $count$ equals the count since the last sample was taken. For samplers without the counter component, the value $count$ is implied and equals the sampling rate r.

We will show that the *stratified periodic sampler* $H[P_r]_n$ sampler is the most successful design. In statistical literature, such a sampler is called stratified systematic sampler [22]. We evaluate the accuracy of these samplers using the problem of estimating the number of times, t, that a given tuple p occurs in the input stream of length N (note that this is a slightly harder problem than the one introduced in Section 3.1). We evaluate the samplers using Monte Carlo simulations. We generated the input stream by randomly permuting a sequence of N tuples containing t copies of the tuple p ($t = 0.3N$). The randomly generated stream is input to each sampler and the output stream is used to estimate the value of t, as follows. For each sampler without a counter, if the output stream contains i copies of tuple p, then t is estimated to be $EST = ir$. For each sampler with a counter, t is estimated to be the sum of $count$'s from all messages that contain the tuple p. We plot the error in the estimate $ERR = |100(t - EST)/EST|$.

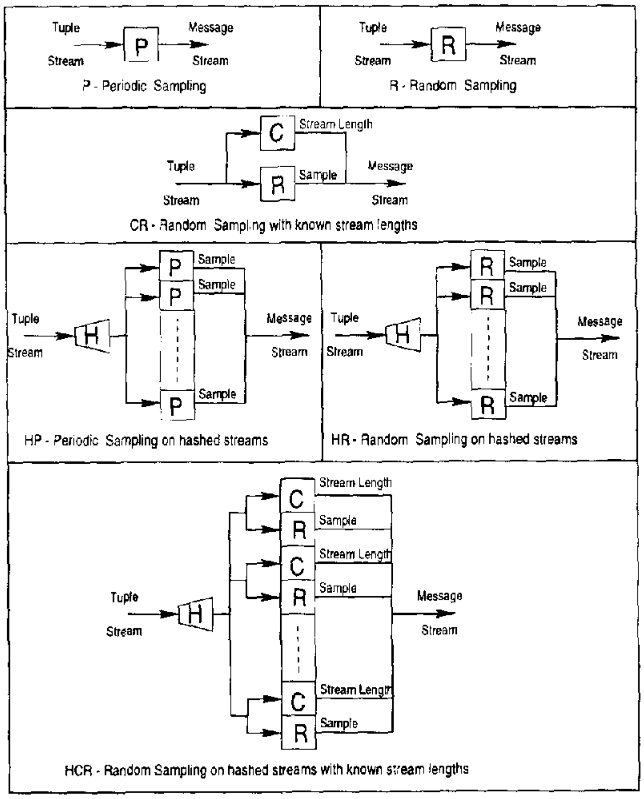


Figure 3: Six samplers built from the basic components.

Figure 4 shows the error in estimating the frequency of the tuple for stream lengths ranging from 1000 to 20000, for a sampling rate of 10. For each input stream length N, we performed 2500 experiments and plotted the mean value of ERR. As expected, the graph shows that with increasing stream length, the error decreases.

Note that all the samplers in Figure 4 have the same sampling rate and hence incur the same overhead. It is therefore interesting to compare their convergence rate. Assuming that we fix the maximum tolerable error at 4%, Figure 4 shows that R_r needs to sample almost three times longer than $H[P_r]_n$ (12000 versus 4600). This experiment shows that, with a fixed profiling overhead, the stratified periodic sampler reduces the error below a maximum tolerable error faster than a random sampler. Most importantly, experiments with real benchmarks yield similar results, as we will show in Section 5.

Furthermore, the graph shows that, based on their accuracy, the samplers divide into three equivalence classes. In decreasing error order, the three classes are $\{R_r, H[R_r]_n\}$, $\{P_r, CR_r\}$, and $\{H[P_r]_n, H[CR_r]_n\}$. Let us now explain why some samplers have the same accuracy. The intuition behind the equivalence of P_r and CR_r was explained in Section 3.1. This intuition also explains the equivalence of $H[P_r]_n$ and $H[CR_r]_n$. Perhaps more surprising is the equivalence of R_r and $H[R_r]_n$. Intuitively, it may appear

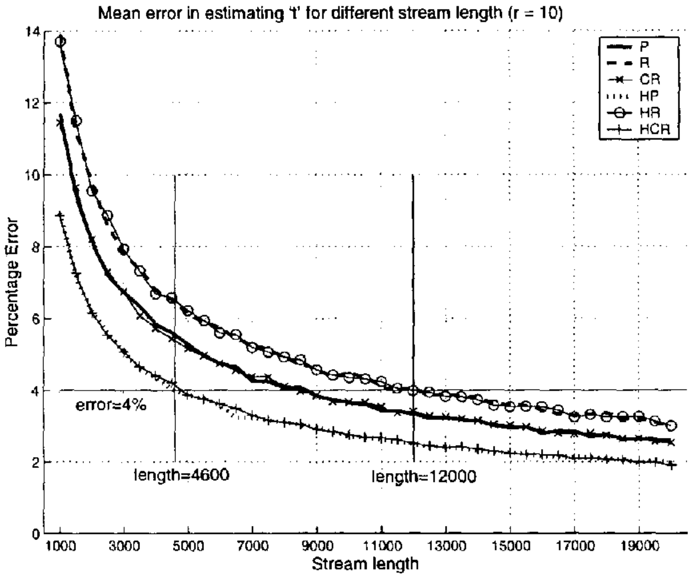


Figure 4: Estimating the frequency of a tuple: Mean error for different input stream lengths.

that $H[R_r]_n$ should perform better than R_r. The reason why this is not the case stems from the fact that our random sampler R_r is stateless, i.e., each element in the input stream is selected *independently* from the other elements, with a probability $1/r$. This observation allows us to think of R_r, equivalently, as if each of its input elements was directed into a separate substream (of length one), which is sampled with a separate R_r sampler. This alternative view is equivalent to $H[R_r]_n$, which explains why stratification with our proposed R_r is of no benefit.[1]

Finally, let us intuitively explain why the most successful design, $H[P_r]_n$, is superior to P_r. Consider the simple example of an input stream consisting of eight 1's and eight 0's, randomly permuted. A P_8 sampler produces a sample size of 2. Using this sample, we would estimate the ratio of 1's and 0's accurately only *half the time*. Now assume that $H[P_8]_2$ splits the input stream into separate 1 and 0 substreams. $H[P_8]_2$ will also produce two messages. But, since the substreams are fully biased, the messages will *always* convey the correct ratio of the number of 1's and 0's.

While we have shown six specific compressor designs, it should be clear that within this framework, other combinations of components are possible. A complete and systematic study of other sampling schemes is outside the scope of the current work.

3.4 Two-Level Compressors

While sampling techniques are lossy stream compression methods, a straightforward lossless stream compression method can be built using a cache-like associative table of counters that accumulate frequency counts of the input tuples. When a table entry must be replaced, or when the counter reaches a maximum value, the tuple is dispatched to software with its corresponding frequency count. Although this method works, it is hardware-intensive. First of all, the table entries can be quite wide. Assuming a 4-byte word (32 bits), two words of profiling information per tuple and a 1-byte counter yields 9 bytes per entry. Furthermore, to avoid frequent tuple replacements (and frequent communication with software), the table needs to capture the working set of the profiling application. For a table with as few as 4K entries the total is 36 KB of storage. With 64-bit words, this grows to 68 KB. Finally, accessing such a table associatively would require compare/match logic the width of the tuples being held in the table. Hence, any practical implementation of a compressor would not use this straightforward approach. However, we will show that the associative counter table is nevertheless a useful profiling component besides being an useful starting point for exploring other compression schemes. We use A_k to denote an associative counter table with k entries.

While the associative counter table alone is an unrealistic compressor, it can be combined with the samplers we presented in the previous section as a second-level compressor. Compressing the messages generated by the samplers will further reduce overheads without affecting accuracy. The accuracy is not affected since the compression performed by A_k is lossless. Therefore, the $H[P_r]_n A_k$ and the $H[P_r]_n$ compressors have equivalent error characteristics, but the $H[P_r]_n A_k$ compressor incurs lower overhead than the $H[P_r]_n$ compressor.

Furthermore, since the tuple stream that is input to the A_k component is already a compressed stream, the compression requirements of the A_k component are not stringent. Therefore, a table as small as 16 entries suffices to achieve a further compression ratio between 1.15 and 2.5 for our benchmarks.

3.5 Stratified periodic sampling

We now look at the stratified sampling scheme ($H[P_r]_n$) in greater detail.

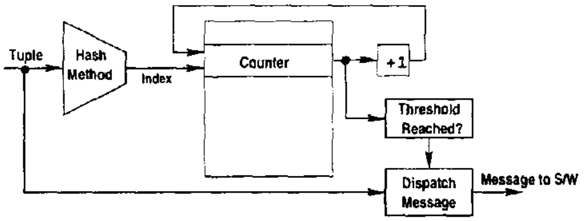


Figure 5: Stratified Sampling Technique

Figure 5 shows the design of a stratified sampler. Each cycle, a tuple is picked from a tuple queue (that absorbs burstiness in the incoming tuple stream). The hash function computes a *signature* of the tuple. The signature is used as an index to select a counter in the counter table. The selected counter is then incremented. If the counter reaches its maximum threshold value r, the counter is reset to zero, and a message consisting of the complete tuple (with an implied occurrence count of r) is sent to profiling software via the output queue.

In our implementation, we use the following hash function. Given a tuple `<pc, w>`, the index is computed as follows:

```
npc = flip(randomize(pc)); nw = randomize(w);
index = xor-fold(npc xor nw, index-size).
```

The function `randomize(w)` looks up a 256-entry random number table for each of the individual bytes of w and composes the new bytes together. `flip(w)` reverses the bytes of w. `xor-fold(w, n)` splits w into n-bit chunks and xors the chunks together.

We used this elaborate hashing function since it provides the best accuracy of all the hashing functions that we experimented with. We have not systematically studied different hashing functions and the trade-offs between profile accuracy and hardware cost. In practice, an actual implementation of the stratified sampler might use a cheaper hash function.

[1] Stratification of the input population improves performance when the size of the subpopulations is known ([22], CR_r, P_r) or when the samples are not picked independently (P_r, countdown random sampling used in [2, 15, 33]).

4 Reducing Collisions: Adding tags to stratified sampling

The stratified periodic sampling described in Section 3.3 makes no effort to resolve tuples that hash to the same substream. At the other extreme, the associative table of counters described in Section 3.4 avoids *all* aliasing, by comparing tuples against complete tags. Unfortunately, the latter design is relatively expensive. This section presents a compromise solution that reduces (but does not eliminate) aliasing by maintaining *partial tags*.

4.1 Design detail

In this design, the signature generated by the hash function has more bits than are required for indexing into the table. The more additional bits, the better the ability to discriminate amongst tuples. The signature is subsequently divided into an index and a tag.

Given a tuple `<pc, w>`, the index is computed as described in Section 3.5. The tag is computed as `tag = xor-fold(pc xor w, tag-size)`. The index is used to select a table entry; if the tags match, there is a hit in the table, otherwise there is a miss. Like a cache memory, the table can be direct-mapped, set-associative, or fully-associative.

Each entry in the table contains the tag, a hit counter, and a miss counter. The hit counter keeps track of the number of occurrences of a tuple. The miss counter is used in making replacement decisions and is discussed below, following an informal description of the replacement process.

At some point, it becomes necessary to evict a tuple's entry from the table, either because its hit count has reached a maximum threshold or because another tuple (with a different tag) maps to the same table entry. If it reaches the maximum threshold, the current tuple is reported to software. In the case of an eviction, the replaced tuple and its occurrence count *should* be passed to software, but this is not possible because only the hashed signature is available in the entry, not the complete tuple. This problem is solved by deferring eviction and placing the to-be-evicted entry into an *eviction state* in which it waits for the same signature to be seen once again.

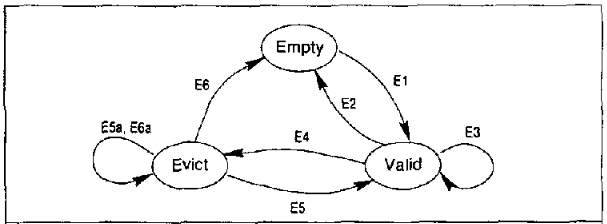
If the to-be-evicted tuple occurs frequently, then it is likely to occur again soon, and the entire tuple is available so complete information can be passed to the software. If the to-be-evicted tuple occurs only rarely, and it does not occur again soon, its value will eventually be discarded.

Figure 6 shows a state machine diagram that describes the detailed operation of a table entry, including the states that an entry goes through, and conditions under which an entry is allocated, evicted, or replaced.

Initially, all entries in the table are `Empty`. When an incoming profile tuple accesses an entry, the entry is allocated for the tuple and the entry transitions to a `Valid` state (Transition E1). In this state, the entry accumulates matching tuples in the hit counter (Transition E3). When the hit counter saturates, a summary message containing the counter value and the tuple is sent to the message queue and the entry transitions to `Empty` (Transition E2).

If there is a miss in the `Valid` state, some other tuple is seeking access to the entry. Because the necessary tuple information for the to-be-evicted entry is not available, the entry transitions to an `Evict` state (Transition E4). The entry is left in the `Evict` state until the matching tuple is seen again (Transition E6). However, the state machine waits until "enough" hits have accumulated in the counter (Transition E6a). This reduces the flood of messages that can be caused by repeatedly aliasing tuples. The *ShouldEvict* condition in the state diagram specifies the exact value of "enough".

In the `Evict` state, misses to the entry will accumulate in the miss counter (Transition E5a), but if "too many" misses are accumulated, the state machine gives up and replaces the current tuple with the new conflicting tuple (Transition E5). In this case the original entry is lost. The *ShouldReplace* condition in the state diagram specifies the exact value of "too many". This replacement policy is similar to the replacement policy in the value profiling algorithm presented in [14].



```
T_orig  : Tuple in table entry
T_new   : Incoming tuple
Hit     : Signatures of T_new and T_orig match
Miss    : Signatures of T_new and T_orig do not match
```

```
E1 : Condition: Miss for T_new
     Action   : Allocate Entry to T_new

E2 : Condition: Hit & hit counter saturates
     Action   : Send summary message to s/w
                Reset Entry (hits = misses = 0)

E3 : Condition: Hit & hit counter does not saturate
     Action   : Increment hit counter

E4 : Condition: Miss
     Action   : Increment miss counter

E5 : Condition: Miss && Should-Replace
     Action   : Allocate Entry to T_new
                Reset Entry (hits = misses = 0)
                T_new.hits = T_orig.hits + T_orig.misses;

E5a: Condition: Miss && !Should-Replace
     Action   : Increment miss counter

E6 : Condition: Hit && Should-Evict
     Action   : Evict T_orig & send eviction message to s/w
                Reset Entry (hits = misses = 0)
                Entry.hits = T_orig.misses;

E6a: Condition: Hit && !Should-Evict
     Action   : Increment hit counter
```

```
Should-Evict    : ((hits >= triggerThreshold/4) && (misses >= triggerThreshold/hits))

Should-Replace : (    Miss counter saturates
                   || (hits ==1)
                   || (hits >  1) && (hits < 2*misses))
```

Figure 6: State diagram describing the states of a profile entry

The unusual manipulation of the hit and miss counters by Transitions E5 and E6 illustrates an important point – it is important not to drop counts when assigning the entry to a new tuple. We found after extensive experimentation that resetting counters almost always leads to lower accuracies because valuable information concerning the input stream is discarded. For every entry, the hit and miss counters together maintain the length of the stream that access that entry. The examples in Section 3.1 implied that it is this exact knowledge of the stream length that leads to increased accuracy. Resetting the hit/miss counters to zero would obscure this valuable information.

Transition E5 represents replacement of a previous tuple by a new tuple. The hit counter is *not* reset as might be expected. Instead, the hit count of the evicted tuple is assigned to the new tuple along with the miss count. Although the hit count does not belong to the new tuple, this avoids losing track of the stream size.

During Transition E6, we initialize the hit counter of the entry with the miss count of the evicted entry. The next tuple to hit in this entry (hopefully the tuple that caused the miss count to be incremented via Transitions E4 and E5a) will assume the new count. Again, the exact stream size is maintained.

4.2 Hardware cost

The tagged profiler incurs extra space overhead beyond the stratified sampler. Besides the hit counter, every entry requires a tag, a miss counter, and bits to code the current state of the FSM. Using a 1-bit tag to encode the FSM state (the Evict state can be inferred by a non-zero value in the miss counter), a 3-bit miss counter, and a 1-bit tag, this is 5 extra bits per counter. With the same sized counter table, and a 8-bit hit counter, this is at least a 62% increase in space requirements over the stratified sampler, in addition tot he the logic required to implement the state machine.

5 Experimental Results

We evaluate the proposed profiling hardware designs through a timing simulation of the value profiling application described below. The timing simulation is needed for measuring the profiling overhead.

5.1 Example Application: Value Profiling

Recently, a number of studies have demonstrated that programs exhibit significant *value locality*, the phenomenon that a small number of values occur repeatedly in the same register or memory location [23, 27, 31, 38, 41]. In the compiler domain, it has been known for some time that value locality can be used to speed up programs by exploiting the fixed/invariant inputs. Partial evaluation [28], data specialization [30], DyC [24], Tempo [12], 'C [37], and code specialization using value profiles [36] are different software techniques for exploiting value locality. Collectively, we refer to these as *value reuse* techniques because the input/output behavior involving the invariant values can be repeatedly reused as opposed to being recomputed each time they occur.

While there exist different techniques for exploiting value reuse, all techniques rely on some kind of value profiling to identify the value locality of instructions [8, 11, 32, 36]. Value profiling support is very important if value reuse optimizations are to be deployed at runtime. We focus on value profiling of loads to evaluate our proposed profiler and compare them with other profilers.

5.2 Methodology

We used the Simplescalar toolset [16] to model a 4-way machine with 64-KB L1 data and instruction caches, 1-MB unified L2 data cache, and a gshare branch predictor. The cycle-level timing model is used for computing profiling overheads and does not affect the actual sampling algorithms.

We use a collection of SpecINT95 benchmarks and Java programs listed in Table 5.2. The SpecInt95 benchmarks were compiled for the SimpleScalar ISA by *gcc* with optimization flags "-O3". The Java programs (including *strata*) were compiled by *strata* [40], a bytecode-to-simplescalar compiler for the SimpleScalar ISA. For all programs, simulation was performed after skipping the initialization phases. For the SpecInt95 benchmarks, the recommendations of Sherwood and Calder [39] were used in determining the simulation starting points. For the Java benchmarks, the starting points were determined empirically (by examining the source code, knowledge of benchmark and output, and experimentation).

5.3 Evaluation Metrics

5.3.1 Profiling Error

The errors in our value profiles is computed using an ideal value profile. Examining the same stream as our profilers, the ideal profile accumulates *all* generated events, rather than just the samples.

Benchmark	Comment	Input
go	SpecInt95	5stone21 files (Ref)
li	SpecInt95	8-queens.lsp (Test)
m88ksim	SpecInt95	Ref input
gcc	SpecInt95	cccp.i
perl	SpecInt95	primes.pl, primes.in
raytrace	SpecJVM98	Speed 100
strata	bytecode to Simplescalar compiler	Some class file
jess	SpecJVM98	speed 100
jack	SpecJVM98	Jack.jack
db	SpecJVM98	speed 100

Table 1: Benchmarks used in this study

When computing the error, we remove from both profiles all (*load, value*) tuples that are highly unlikely to be used by a realistic value-reuse optimizer. Taking these tuples into account would introduce error that is irrelevant to the optimizer. Analogous to the error metric in [14], from both profiles, we discard loads that execute infrequently, as well as values that are infrequent for a given static load. Namely, we select for profiling a static load only if it executes at least 1000 times. Furthermore, only sufficiently invariant tuples are selected. A (*load, value*) tuple is *sufficiently invariant* if it accounts for at least 10% of the executions of the load. Finally, we select only those loads that have at least 40% of their dynamic execution accounted for by sufficiently invariant tuples.

The profiling error is computed as follows. Let $v = (pc, val)$ be a tuple that has been selected from the ideal value profile. The ideal invariance of v is computed as $I_i(v) = n_i(v)/n_i(pc)$ where $n_i(v)$ is the number of times v occurs in the ideal profile, and $n_i(pc)$ is the execution count of pc. Let $I_p(v) = n_p(v)/n_p(pc)$ be the tuple's invariance estimated by our profiler; $n_p(v)$ and $n_p(pc)$ denote the number of times our profiler sees the tuple v and the load pc, respectively. Then, the error in profiling the invariance is $e(v) = |i_i(v) - i_p(v)|$. We compute the error for the entire profile by taking a frequency-weighted average of $e(v)$ over all selected tuples, i.e., the error $e = \sum f(v)/f \times e(v)$ over all selected tuples v, where $f(v)$ is the cumulative execution frequency of v, and f is the execution frequency of all selected tuples.

5.3.2 Profiling Overhead

Our evaluation assumes interrupt-driven communication between the compressor and the software profiler (see Section 2.2). We use an analytical model to compute profiling overhead. The overhead is dependent on:

- *Per-interrupt fixed costs*: This is a fixed cost that depends on the OS and the specific processor. Anderson *et al* [2] show that for their system, per-interrupt fixed costs are about 214 cycles. We use this value in our model.
- *Number of messages processed per interrupt*: In our model, we assume at least 100 messages per interrupt to amortize the per-interrupt fixed cost.
- *Processing time per message*: This is the time required to fold the message into the profile; the actual value is specific to the profiling application. We assume a fixed cost in our model, as described below.

In their paper, Zilles and Sohi [14] state that with careful assembly coding of their interrupt handler, every message can be processed in 10-30 cycles. Because we do not perform convergent profiling checks as in [14], at least 3 cycles per message are saved. On the other hand, by processing at least 100 messages per interrupt, we

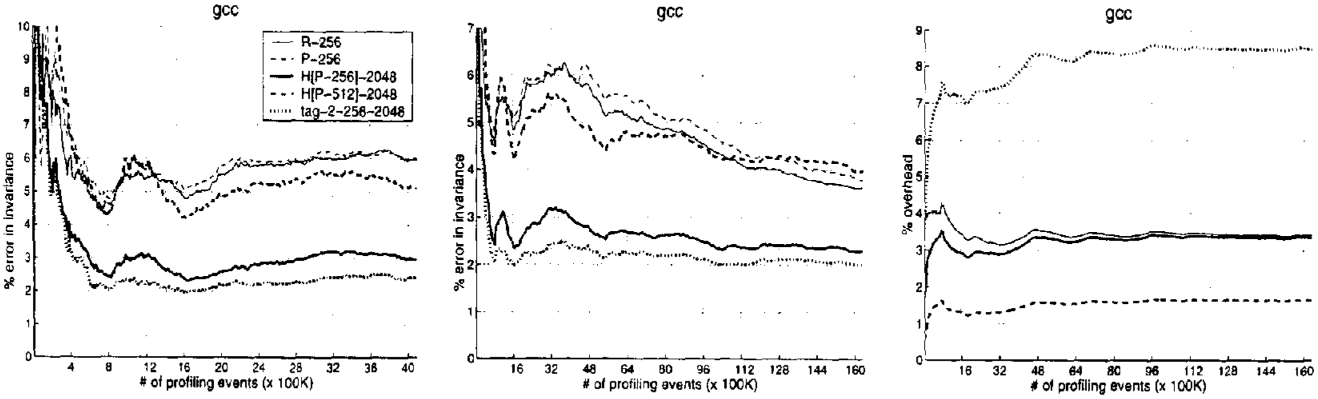


Figure 7: Results for *gcc*: The first graph shows the variation of % error with program progress (up to 4M events). The second graph shows errors for a longer duration (up to 16M events). The third graph shows the variation of cumulative % overhead with increasing time (up to 16M events). In the third graph, the plots for the R_{256} and P_{256} compressors overlap.

incur under 3 cycles in fixed interrupt costs per message (214 cycles per 100 messages). Therefore, in the worst case, we assume the interrupt overheads will be 30 cycles per message. Based on this number, we estimate the profiling overhead to be $Overhead = 30 \times NumMessages$, where $NumMessages$ is the number of messages dispatched. The percentage overhead is computed with respect to the total simulation time.

5.4 Evaluating compressors

In this section, we evaluate the following compressors: R_{256}, P_{256}, $H[P_{256}]_{2048}$, $H[P_{512}]_{2048}$ (presented in Section 3.3), and a 2-bit tagged compressor (presented in Section 4) with an 8-bit hit counter (equivalent to a sampling rate of 256) and a 2-way set associative 2048-entry counter table. We do not present results for other tagged compressors since our results indicate that for our tagged implementation, the 2-bit tagged compressor has the best error-overhead behavior among all tagged compressors.

Figure 7 shows for *gcc* the variation of % error and % overheads with program progress. We selected *gcc* because it is one of the hardest programs to profile and has a much bigger working set than the other programs. The first graph in Figure 7 is plotted for 4M profiling events (17M instructions). The other two graphs are plotted for 16M profiling events (70M instructions).

We first compare the random and periodic samplers. The graphs show that the two are almost identical in performance both in error and overhead. It is interesting to compare these results with the Monte Carlo simulation results presented in Section 3.3. For randomly generated input streams, Monte Carlo simulations showed that the periodic sampler performs better than the random sampler. However, in practice, programs do not generate random input streams and this seems to explain why P_{256} does not perform better than R_{256} on real workloads. For the rest of the discussion, we discuss the random profiler only.

Next, we discuss the tagged compressor. The error graphs show that the tagged compressor has the best accuracy of all compressors, but the low error comes at the cost of almost three times the profiling overhead, as shown in the third graph. Furthermore, from the graphs, we conjecture that for *gcc*, the improvement in accuracy over the much simpler stratified sampler is not significant enough to merit the higher overheads and the higher hardware complexity of the tagged compressor design. At this juncture, we wish to point out that the tagged compressor design presented in Figure 6 is only one of many possible implementations of a tagged compressor. Preliminary experimentation indicates that other advanced implementations (based on the sampling idea) are possible and these may improve on the stratified sampler both in error and overheads. These more sophisticated designs are left for future work.

We now compare the $H[P_{256}]_{2048}$ and the R_{256} compressor. Assuming a maximum tolerable error of 5%, the first two graphs in Figure 7 show that $H[P_{256}]_{2048}$ reaches this error threshold after 200K events whereas R_{256} reaches this threshold after 600K-700K events. However, if one requires the error to drop below 5% and stay below that limit, the R_{256} compressor reaches the 5% error threshold only after about 7M profiling events, about 23 times longer than the $H[P_{256}]_{2048}$ compressor which reaches the 5% threshold after 300K events. The overhead graph shows that the stratified sampler always has a lower overhead than the random sampler because the $H[P_{256}]_{2048}$ compressor retains up to $256 \times 2048 = 512K$ tuples in the counter table, which means that up to $2K$ fewer messages are dispatched to the software when compared to the random sampler. However, if the programs are profiled for a long time, this advantage vanishes as shown by the converging overhead plot.

Let us now examine how to reduce profiling overheads for the stratified samplers while maintaining the same accuracy as a random profiler. If we compare the $H[P_{512}]_{2048}$ and the R_{256} compressors, we find both have similar accuracy, but the $H[P_{512}]_{2048}$ incurs half the overhead when compared to R_{256}. This shows that with the stratified sampler, we can achieve the same accuracy as a random sampler at a lower sampling rate, and hence at lower profiling overheads. The factor by which the sampling rate can be reduced is benchmark-specific as can be seen from Figure 8.

One final conclusion that we can draw from the error graphs is that the stratified sampler and the tagged compressor stabilize more quickly than the random and periodic samplers, i.e. they converge to their "final" errors much more quickly than the random and periodic samplers.

By using profile convergence checks and instruction filtering techniques proposed in [14], the performance of the stratified sampler can be improved further. Instruction filtering reduces aliasing in the hashed substreams and can lead to faster convergence of profiles.

The preceding discussion focused on *gcc*. Figure 8 shows the error curves for the five compressors for all the other benchmarks.

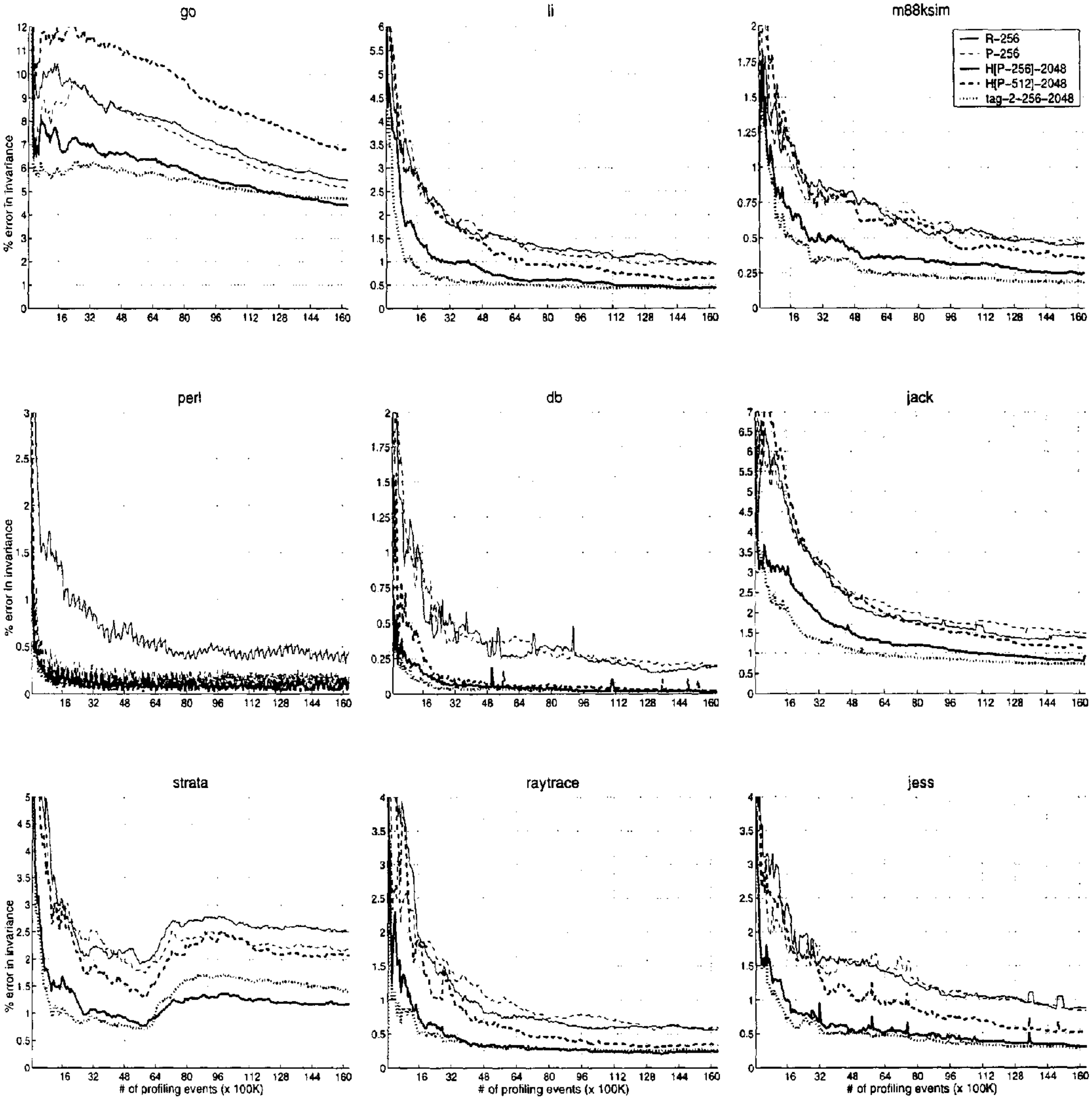


Figure 8: Results for all the other benchmarks: The graphs show the variation of % error with program progress (up to 16M events).

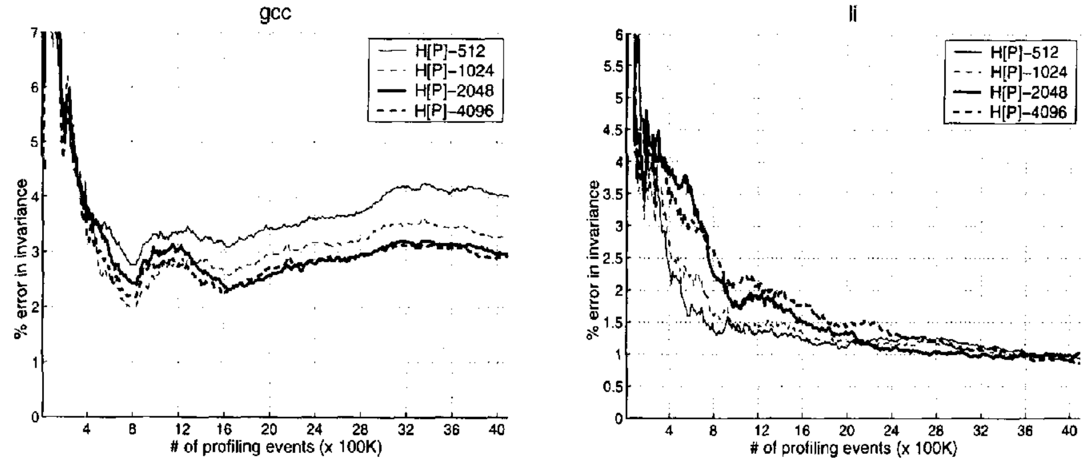


Figure 9: Sensitivity results of the stratified sampler for four different table sizes for *gcc* and *li* (up to 4M profiling events)

5.5 Sensitivity study of the stratified sampler

We now present results of a sensitivity study of the stratified sampler by considering four different table sizes (512, 1024, 2048, and 4096). Figure 9 shows error plots for the $H[P_{256}]_{512}$, $H[P_{256}]_{1024}$, $H[P_{256}]_{2048}$, and $H[P_{256}]_{4096}$ compressors for *gcc* and *li*.

The graphs show that with increasing number of table entries, the performance gets better but only *after* a sufficient number of tuples have been seen. Because the counter table accumulates tuples that are not sampled until the counters overflow, during the initial phases, when the ratio of stream length to the output messages is small, more messages are sent out from a smaller table. Hence, for some benchmarks (*li*, for example, shown in Figure 9) the accuracy is better with fewer counters during the initial phases. As the program progresses, this effect diminishes.

5.6 Two-level compression

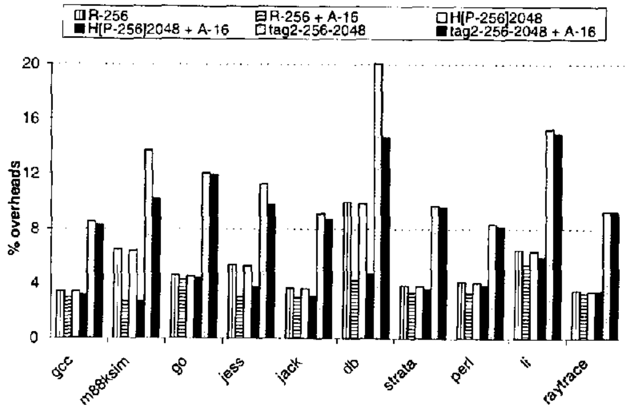


Figure 10: Reduction in overheads due to the A_{16} component

If a small associative counter table is added, additional reductions in overhead can be achieved. Figure 10 shows the profiling overheads for the R_{256}, $H[P_{256}]_{2048}$ and the tagged profilers with and without a 16-entry associative buffer. The figure shows that for all these compressors, adding the second compressor results in a reduction of overheads for all benchmarks. The reduction is pronounced for *m88ksim*, *db* and *jess*. For these programs, the output message stream is dominated by a few prominent tuples. It is these repetitive output messages that enables the high second-level compression of the message stream. This result shows that the associative array of counters is a useful component in building efficient compressors.

5.7 Simultaneous profiling: Edge and Call target profiling

In Section 2.1, we stated that the hybrid profiler can collect multiple profiles simultaneously. To demonstrate the feasibility of this application, we collected call-target and edge profiles simultaneously using the $H[P_{256}]_{2048}$ compressor. These profiles are used by the runtime optimizer to select inline candidates and to build traces, respectively. Call target profiles are especially important for Java programs because, unlike C programs, the presence of virtual calls in Java makes it difficult to statically determine the call targets at a call site. Call target profiles enable the optimizer to determine the likely call targets and inline sites and implement optimizations such as feedback-directed inlining [3] and polymorphic inline caching [26]. Edge profiles enable the optimizer to build traces and perform code layout optimizations.

Figure 11 shows the results of simultaneously collecting edge and call target profiles. The graphs show that for all benchmarks, the error is less than 3%. For six of the benchmarks, the error is negligible. Furthermore, the overheads are also very low; the profiling overheads are under 4.5% for all benchmarks. The accuracy is significantly better than the accuracy in the preceding value profiling application because branches, jumps, and calls take values from a much smaller value set than load instructions.

6 Summary

Profile-driven optimizations require flexible profiling support that can collect a variety of profiles accurately, rapidly, and with low overheads. In this paper, we presented a hybrid hardware-software profiling scheme. In the proposed hybrid profiling scheme, the hardware compresses the input profile stream to generate a smaller stream of output messages that is processed by software to construct the required profile. Since optimizations can tolerate profiling errors, we show that the compression can be lossy in that the occurrence counts reported for tuples in the input stream need not be exact.

On the basis of the lossy compression metaphor, we presented a framework of profile components that can be composed in a mul-

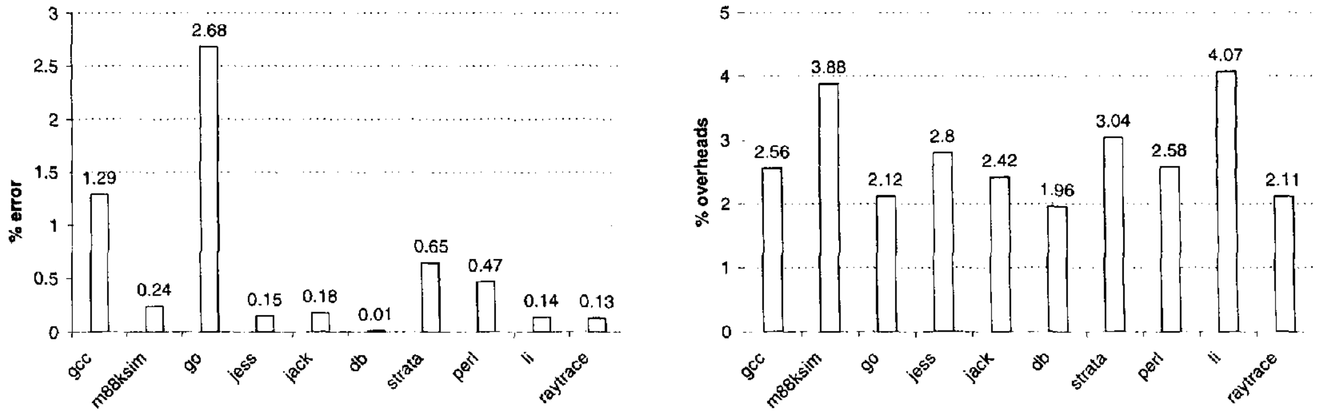


Figure 11: Final error and cumulative overhead for simultaneous call and edge profiling after 4M profiling events

tiple ways to build hardware compressors. Conventional random sampling, periodic sampling, and stratified sampling were proposed and studied. The stratified sampling scheme uses a hashing scheme to split the input stream into multiple disjoint streams that index into a table of counters. In addition, we proposed a more sophisticated compression scheme that builds on the stratified sampling scheme by using tags to detect aliasing in the counter table.

We used load value profiling as an example application for evaluating the proposed compressors. We showed that the stratified sampling scheme has the best error-overhead performance among all the compressors we studied. For *gcc*, we showed that with the same or smaller profiling overheads, the stratified sampler achieves a desired accuracy twice as fast as a random sampler. We also showed that for *gcc*, if the profiling time and the desired accuracy level are fixed, the stratified sampler can achieve these thresholds with half the overheads as a random sampler. The overhead factors are benchmark-specific (more than two for *perl*, *db* and other benchmarks).

We also showed that while the tagged compressor has better accuracy than the stratified sampling scheme, the improved accuracy comes at significantly higher overheads (twice or more) and higher hardware complexity.

We then proposed an enhancement to the different compression schemes by introducing a second-level lossless compression that can further reduce profiling overheads without affecting accuracy. We show that additional lossless compression between 1.15 and 2.5 can be achieved by using an associative buffer of 16 entries that summarize the messages from the first level compressor.

On a higher level, we also showed that our hybrid profiling scheme is capable of collecting a variety of profiles (type, edge, call-target, value) and that it is capable of collecting profiles simultaneously with high accuracy (3% error) and low overheads (4.5%).

Future work involves a systematic study of other compressors that can be built using the proposed component framework. Furthermore, while we have proposed one specific implementation of a tagged compressor, more advanced sampling-based implementations are possible that improve over the stratified sampler in both accuracy and overheads.

7 Acknowledgements

This work was supported by two IBM Faculty Partnership Awards, two National Science Foundation grants CCR-9900610 and EIA-0071924, by IBM Corporation, Sun Microsystems, and Intel Corporation. This support is gratefully acknowledged.

We greatly benefited from the many interesting discussions with Timothy Heil about statistical sampling techniques and their behavior. Special thanks are due to him for helping us nail down the intuition behind the good performance of the sampling compressors. We are also grateful to Manoj Plakal, Craig Zilles, and the anonymous referees for their numerous comments that helped us clarify and better present the subject matter of this paper.

References

[1] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the ACM SIGPLAN '97 Conf. on Prog. Language Design and Impl.*, pages 85–96, 1997.

[2] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems*, 15(4):357–390, November 1997.

[3] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F.Sweeny. Adaptvie Optimization in the Jalapeno JVM. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA '00)*, October 2000.

[4] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A Transparent Runtime Optimization System. In *Proceedings of the 2000 ACM SIGPLAN Conference on Prgramming Language Design and Implementation (PLDI)*, pages 1–12, 2000.

[5] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.

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

[7] M.G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M.J. Serrano, V.C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Dynamic Optimizing Compiler for Java. In *ACM Java Grande Conference*, June 1999.

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

[9] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, and Wen-Mei W. Hwu. Profile-guided automatic inline expansion for C programs. *Software—Practice and Experience*, 22(5):349–369, May 1992.

[10] Pohua P. Chang, Scott A. Mahlke, and Wen-Mei W. Hwu. Using profile information to assist classic code optimizations. *Software: Practice and Experience*, 21(12):1301–1321, December 1991.

[11] D. Connors and W. Hwu. Compiler-Directed Dynamic Computation Reuse: Rationale and Initial Results. In *32nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 158–169, 1999.

[12] Charles Consel, Luke Hornof, Francois Noel, Jacques Noye, and Nicolae Volanschi. A Uniform Approach for Compile-time and Runtime Specialization. Technical Report RR-2775, Inria, Institut National de Recherche en Informatique et en Automatique, 1996.

[13] Thomas M. Conte, Kishore N. Menezes, and Mary Ann Hirsch. Accurate and practical profile-driven compilation using the profile buffer. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 36–45, Paris, France, dec 1996. ACM Press.

[14] Craig Zilles and Gurinder Sohi. A Programmable Co-processor for Profiling. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture (HPCA-7)*, January 2001.

[15] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos. ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-97)*, pages 292–302, Los Alamitos, December 1–3 1997. IEEE Computer Society.

[16] Doug Burger, Todd M. Austin, and Steve Bennett. Evaluating Future Microprocessors: The SimpleScalar Tool Set. Technical Report CS-TR-96-1308 (Available from http://www.cs.wisc.edu/trs.html), University of Wisconsin-Madison, July 1996.

[17] Evelyn Duesterwald and Vasanth Bala. Software profiling for hot path prediction: Less is more. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, November 1–3 2000.

[18] Carole Dulong. The IA-64 architecture at work. *Computer*, 31(7):24–32, July 1998.

[19] Kemal Ebcioğlu and Erik Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In 24th *Annual International Symposium on Computer Architecture*, pages 26–37, 1997.

[20] Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):478–490, July 1981.

[21] David M. Gallagher, William Y. Chen, Scott A. Mahlke, John C. Gyllenhaal, and Wen mei W. Hwu. Dynamic Memory Disambiguation Using the Memory Conflict Buffer. In Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 183–193, San Jose, California, 1994.

[22] William G.Cochran. *Sampling Techniques*. John Wiley and Sons, 1977.

[23] A. Gonzalez, J. Tubella, and C. Molina. Trace-Level Reuse. In *Proceedings of the the International Conference on Parallel Processing*, September 1999.

[24] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. Eggers. DyC: An Expressive Annotation-Directed Dynamic Compiler for C. Technical Report TR-97-03-03, University of Washington, Department of Computer Science and Engineering, March 1997.

[25] Urs Holzle. Adaptive Optimization for SELF: Reconciling High Performance with Exploratory Programming. Thesis CS-TR-94-1520, University of Stanford, August 1994.

[26] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches. In Pierre America, editor, *ECOOP '91: European Conference on Object-Oriented Programming*, volume 512 of *Lecture Notes in Computer Science*, pages 21–38. Springer-Verlag, 1991.

[27] Jian Huang and David J. Lilja. Exploiting Basic Block Value Locality with Block Reuse. In *Proceedings of the Fifth International Symposium on High-Performance Computer Architecture*, pages 106–114, Orlando, Florida, January 9–13, 1999. IEEE Computer Society TCCA.

[28] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, International Series in Computer Science, June 1993. ISBN number 0-13-020249-5 (pbk).

[29] Alexander Klaiber. The technology behind Crusoe(tm) Processors, January 2000.

[30] Todd B. Knoblock and Erik Ruf. Data Specialization. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 215–225, Philadelphia, Pennsylvania, 21–24 May 1996.

[31] Mikko H. Lipasti and John Paul Shen. Exceeding the Dataflow Limit via Value Prediction. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 226–237, Paris, France, December 2–4, 1996. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[32] M. U. Mock and C. Chambers and S. J. Eggers. Calpa: A Tool for Automating Selective Dynamic Compilation. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-33)*, December 2000.

[33] M.Burrows, U.Erlingson, S.T.A.Leung, M.T.Vandevoorde, C.A.Waldspurger, K.Walker, and W.E.Weihl. Efficient and Flexible Value Sampling. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 160–167, November 1–3 2000.

[34] Steve Meloan. The Java HotSpot (tm) Perfomance Engine: An In-Depth Look. Article on Sun's Java Developer Connection site, 1999.

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

[36] Robert Muth, Scott Watterson, and Saumya Debray. Code Specialization Based on Value Profiles. In *Proceedings of the 7th International Static Analysis Symposium (SAS 2000)*, pages 340–359. Springer LNCS vol. 1824, June 2000.

[37] Massimiliano Poletto, Dawson R. Engler, and M. Frans Kaashoek. tcc: A System for Fast, Flexible, and High-level Dynamic Code Generation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI-97)*, volume 32, 5 of *ACM SIGPLAN Notices*, pages 109–121, New York, June 15–18 1997. ACM Press.

[38] Y. Sazeides and J. E. Smith. The Predictability of Data Values. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-97)*, pages 248–258, Los Alamitos, December 1–3 1997. IEEE Computer Society.

[39] Timothy Sherwood and Brad Calder. Time Varying Behavior of Programs. TechReport CS99-630, University of California-San Diego, August 1999.

[40] James E Smith, Subramanya Sastry, Timothy Heil, and Todd Bezenek. Achieving High Performance via Co-Designed Virtual Machines. In *International Workshop on Innovative Architecture*, October 1999.

[41] Avinash Sodani and Gurindar S. Sohi. Dynamic instruction reuse. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97)*, pages 194–205, June2–4 1997.

[42] Timothy Heil and James E Smith. Relational Profiling: Enabling Thread-Level Parallelism in Virtual Machines. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-33)*, December 2000.

[43] C. Young and M. D. Smith. Better global scheduling using path profiles. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-98)*, pages 115–126, Los Alamitos, November 30–December 2 1998. IEEE Computer Society.