Shadow Profiling: Hiding Instrumentation Costs with Parallelism

Tipp Moseley,† Alex Shye,† Vijay Janapa Reddi,† Dirk Grunwald,† and Ramesh Peri‡

†Department of Computer Science
University of Colorado
Boulder, CO 80309

‡Intel Corporation
Hillsboro, OR

Abstract

In profiling, a tradeoff exists between information and overhead. For example, hardware-sampling profilers incur negligible overhead, but the information they collect is consequently very coarse. Other profilers use instrumentation tools to gather temporal traces such as path profiles and hot memory streams, but they have high overhead. Runtime and feedback-directed compilation systems need detailed information to aggressively optimize, but the cost of gathering profiles can outweigh the benefits. Shadow profiling is a novel method for sampling long traces of instrumented code in parallel with normal execution, taking advantage of the trend of increasing numbers of cores. Each instrumented sample can be many millions of instructions in length. The primary goal is to incur negligible overhead, yet attain profile information that is nearly as accurate as a perfect profile.

The profiler requires no modifications to the operating system or hardware, and is tunable to allow for greater coverage or lower overhead. We evaluate the performance and accuracy of this new profiling technique for two common types of instrumentation-based profiles: interprocedural path profiling and value profiling. Overall, profiles collected using the shadow profiling framework are 94% accurate versus perfect value profiles, while incurring less than 1% overhead. Consequently, this technique increases the viability of dynamic and continuous optimization systems by hiding the high overhead of instrumentation and enabling the online collection of many types of profiles that were previously too costly.

1 Introduction

Profile-guided optimization (PGO) uses dynamic information obtained from a program's execution to decide upon a strategy for optimization. PGO systems fall roughly into two categories, dynamic and static, though there is not always a clear dichotomy. Dynamic optimization systems gather profile information as a program executes and apply optimizations during the same run of the program. This technique is becoming common for managed runtime environments such as Java [8], and is being applied to native programs as well [6, 23]. More traditionally, static PGO collects profiles offline and directs the recompilation of an entire program from source. Other methods, such as Spike [13], employ static binary modification to apply feedback-directed optimization.

For any PGO system to be effective, it must be aware of both what and how to effectively perform optimization. For deciding what to optimize, sampling just the program counter (PC) at a high rate is enough information to detect hot code and even deduce which instructions stall the pipeline [1]. Hardware performance monitoring units (PMU) can be used to gather richer information such as cache miss and branch mispredict PCs. Such information is valuable to programmers when deciding where to focus optimization efforts, and it is also useful for JIT compilers to apply the most aggressive and costly optimizations to the hottest regions of code. However, deciding how to optimize is a considerably harder problem. Many profiling techniques that are used to decide how to optimize incur undesirable overhead because they require instrumentation to record temporal *traces* of events. For profiling to be worthwhile, its benefit must outweigh its cost. Therefore, much effort must be spent on ensuring that profiling has low overhead.

One of the most common feedback-directed optimizations is improving code layout and superblock formation using Ball-Larus path profiling [4]. Path profiling is popular because it delivers good performance improvement and the instrumentation overhead can be as low as 30-40% for highly optimized, compiler-based instrumentation. Other types of profiles, such as hot data stream profiling [12] and value profiling [9] have been used to apply optimizations

that achieve gains of over 20%, but they can also result in substantial overheads that are tens or hundreds of times slower than the original program.

Such high overheads can be prohibitive to profiling, so methods for sampling-based instrumentation have recently become more popular. Low-overhead instrumentation is particularly desirable for dynamic optimization, where the profiling cost must be amortized over time and by the optimizations performed. Arnold [2] and Hirzel [18] both describe techniques for reducing the cost of instrumented code via sampling. The technique, known as "bursty tracing", periodically alternates between normal and instrumented code.

Shadow profiling is a novel technique for performing instrumentation sampling of detailed traces. It differs from previous techniques in that instrumented code is executed in parallel with a program's execution. Instead of instrumenting an application directly, a *shadow process* is periodically created and instrumented while the parent continues normal execution. By leveraging the copy-on-write mechanism available in modern operating systems, the child process is a mostly-deterministic replica of the parent. Although forking replicates the virtual address space, it does not protect against other potential system side effects involving files and shared memory. Issues with safety and non-determinism are addressed in Section 3.

Shadow profiling offers the following features:

- Low overhead. The type of instrumentation injected into the shadow process doesn't affect the speed of the original application. The only sources of slowdown are paging and bus and disk contention, which are very low in practice.

- Scalability. Many traces can be collected in parallel; as the number of cores on die increases, idle resources will be increasingly available for profiling.

- Tuning. The sample length of each shadow and the average number of active shadows are both configurable, enabling a flexible method to adjust for greater coverage or lower overhead.

- Simplicity. The infrastructure leverages many present technologies, so existing profilers and instrumentation systems can be used with only slight modifications.

The rest of this paper is organized as follows. Section 2 provides background information. Section 3 describes the design and implementation of shadow profiling. Section 4 explores tuning tradeoffs and evaluates performance. Section 5 discusses related work. Section 6 outlines future work, and Section 7 concludes.

2 Background

Program instrumentation is a common way to observe very detailed dynamic execution behavior. Instrumentation can be applied in a number of ways and at a number of points in a program's life cycle. For this discussion, two techniques are particularly relevant: probing and dynamic instrumentation.

Probing, also known as code patching, is a technique that works by overwriting original instructions with a jump to a bridge that handles the saving and restoring of registers, calling analysis routines, and execution of the replaced instructions [19, 7]. For collecting events that only occur rarely, probing is a low-overhead alternative to binary translation. Therefore, probes are most commonly used at function entry points, function call sites, and system calls. Some instruction sets, such as IA32, are not amenable to probing because of variable length instructions. Therefore, great care must be taken to ensure that probe-based instrumentation does not corrupt the original instructions.

Alternately, other instrumentation techniques recompile an entire application to interleave instrumentation with original program instructions. Some tools, such as ATOM [33], employ static object file rewriting to accomplish this goal. Others, such as Pin [24], Dynamo [6], and Valgrind [26], actually have much in common with a virtual machine. The shadow profiling infrastructure is implemented using the Pin system. Pin is a just-in-time (JIT) dynamic compiler that is capable of inserting arbitrary instrumentation into unmodified binary programs for Intel® IA32, EM64T, Itanium® , and ARM instruction set architectures. Pin does not rely on static compiler support and also has other advanced features, such as a low-overhead PinProbes mode, the ability to begin JITing at any point during execution, and the ability to attach to a program that is already running.

Since Pin is a dynamic compiler, it must store traces into a code cache as they are compiled. The instrumentation code is aggressively optimized with register reallocation and inlining of analysis routines. As a result, the instrumented code is very fast. One drawback to this approach is that short-running programs take noticeably longer to execute because most of the time is spent compiling traces that are only executed a few times. For longer-running programs, the compilation cost is amortized as the code cache is filled and hot code is reused.

To somewhat mitigate this overhead, Pin has a persistence infrastructure developed by Janapa Reddi [27]. Persistence uses an on-disk cache to store compiled traces for future runs of a program. When a new process begins, it loads in an old code cache from disk, so traces that have been previously executed don't have to be recompiled. Then when the process exits, it updates persistent copy on disk. If there are multiple processes active, there may be race con-

ditions for the persistent cache, so it is protected by a read-write file lock. This method has been shown to dramatically improve the startup time of applications running under Pin.

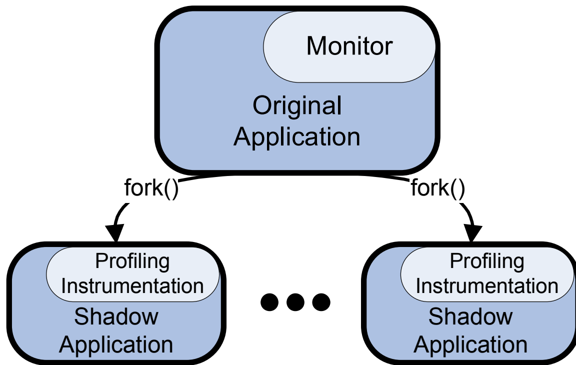


Figure 1. Components of the shadow profiling system.

3 Shadow Profiling Infrastructure

The components of the system are shown in Figure 1. A probe-based application monitor is injected into the original application when it begins. The probes allow the monitor to observe certain system calls, yet still permit the application to execute natively without incurring overhead. Periodically, the monitor will fork a shadow process that is to be instrumented and profiled. After the fork occurs, Pin switches from probe mode to JIT mode and after that point every instruction is executed from the code cache.

Figure 2 shows a chronological view of how a profiled run might proceed. In this example, the instrumented code is three times slower than native. At time 1, the first shadow is forked and is migrated to CPU 2 by the operating system. While the shadow is duplicating the behavior that occurred during that timeslice, the original application proceeds ahead 3 time units. At the beginning of time 4, the application monitor recognizes that shadow 0 has completed, so another shadow is created. In this example, shadows 0 and 1 are shown to be scheduled on different CPUs, but they could also be scheduled sequentially on the same CPU at the operating system's discretion. Some programs may require greater coverage, while others may require less. Therefore, many shadow processes may execute in parallel, or they can instead be created less frequently.

The rest of this section describes the technical details involved in creating shadow processes and making them transparent to the original application.

Ideally, after a fork, two processes would be exact replicas of each other and both could continue execution without interference. The most significant problem, duplicating the address space, is conveniently handled by the copy-on-write paging system common to most modern operating systems. However, there are a number of other issues that must be considered: shared memory, memory-mapped files, system calls, and threading. The handling of each issue is described in context below.

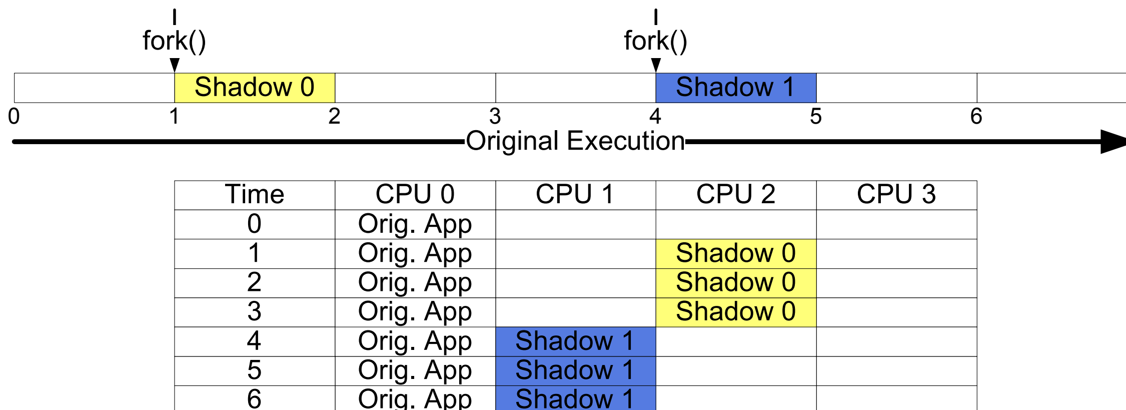
3.1 Application Monitor

The application monitor first takes control of an application from invocation or by attaching to an existing process. Since the shadow is not allowed to execute certain system calls, their effects need to be logged so that they may be emulated. The original program runs natively, so each time an image is loaded, probes must be placed on particular system calls to log their effects.

Using just probes, the monitor is only activated when a system call occurs. However, it is desirable to be able to create shadows at arbitrary points in time, not just at system call boundaries. After the monitor gains control, it registers a signal handler that will be invoked on a timer interrupt. By default, the SIGPROF signal is used, but since applications can use signals in unconventional ways, the signal number used to profile is configurable via a command-line parameter. Likewise, a regular timer interrupt must be configured. One option is to use the setitimer or alarm system calls, but this could interfere with many applications' normal behavior. Instead, a separate process, called the signaler, is created to signal the monitor and then sleep for a predefined period. The signaler remains active until the original application completes. Note that the monitor itself has an unmeasurably small effect on the program's performance.

3.1.1 Managing Overhead

When the monitor receives a signal, its only action is to decide if a shadow should be created. To control the amount of processor time given to shadow processes, the user may specify the *load* parameter, given as a floating point number. For example, a load value of 1.0 means that there will always be a single shadow actively profiling. The user may also specify values like 0.5, meaning that only half of the time a shadow will exist, or 2.0, meaning that two shadows will exist concurrently. Each time the monitor receives a signal, it updates a running average of the number of active shadows. When the average falls below the desired load, a new shadow is forked. After the fork, the parent returns control to the running program.



Time	CPU 0	CPU 1	CPU 2	CPU 3
0	Orig. App			
1	Orig. App		Shadow 0	
2	Orig. App		Shadow 0	
3	Orig. App		Shadow 0	
4	Orig. App	Shadow 1		
5	Orig. App	Shadow 1		
6	Orig. App	Shadow 1		

Figure 2. A chronological example of shadow profiling execution.

3.2 Shadow Profiler

After a fork, the child process must switch from probe to JIT mode in Pin. This is done with the PIN_ExecuteAt API, which directs Pin to begin instrumenting from an arbitrary processor context. Since this is all done from inside the signal handler, the architectural context, including the PC, is conveniently available. Once Pin begins instrumenting the program, profiling begins. The shadow will execute a predefined number of instructions, usually many thousands or millions, before exiting. Execution and instrumentation proceed as a normal Pintool, but the shadow must be restricted from actions that cause side effects in the original process or the rest of the system.

One way the shadow can affect the original process is through shared memory pages (attained in UNIX via mmap or shmat system calls) and memory-mapped files that have write access. Thus, after a fork, write access to such pages is revoked in the child process. Since the shadow will fault when attempting to write to shared pages, it can either chose to skip the faulting instruction or terminate and create a new shadow. In practice, this issue has not arisen since none of the SPEC CPU2000 benchmarks use shared pages. They do tend to be more common in database and webserver applications, but they are still written to infrequently.

A much more common way that the shadow process can cause side-effects is through system calls, which occur approximately 35 times per second on our experimental configuration. Fortunately, the majority of system calls are benign, can be emulated, or can be simply ignored. In our experiments, around 5% of dynamic system calls cannot be handled safely and terminate the shadow process. The following is a list of dynamic execution frequency, examples, and solutions for different groups of system calls.

- Benign (49%): e.g. getrusage, time, brk, munmap, fstat, etc. These system calls may be allowed to execute since they do not have side effects on the system.
- Output (39%): e.g. write, ftruncate, writev, rename, unlink, etc. These system calls are not allowed to execute because they change the state of the system. They can be emulated with success assumed, and execution continues.
- Input (5%): e.g. read, readv, etc. These depend on the type of descriptor that is being accessed. Since the kernel's file pointer is not duplicated across a fork, open files must be closed, reopened, and "seeked" to the previous position. Pipes and sockets are more complicated; the original process must transmit data read from pipes and sockets to the shadow process so that the calls can be emulated.
- Miscellaneous (2%): e.g. open, creat, mmap, mprotect, fcntl, lseek, etc. There are some special cases, depending on the arguments, but usually these may be allowed to execute.
- Unsafe (5%): e.g. ioctl, because its behavior is device-specific. Also some cases of mmap, which is often used to dynamically load libraries. In this case, the shadow terminates and signals the process monitor.

The key concept to keep in mind is that the goal is to attain many small samples of execution, so each shadow will only execute for at most a few hundred million of instructions.

Therefore, many shadows will not encounter a single system call, and most will only encounter one or two. Since unsafe system calls are rare, most shadow processes terminate themselves when the desired number of instructions has been profiled. In the event that an unsafe (or unknown) condition is encountered, the monitor simply creates a new shadow to replace it.

Since Pin's code cache is local to the process executing, each shadow process begins compiling from scratch and compiled traces are lost when a process exits. This results in a profiling tradeoff between the number and length of shadows. Shorter-running shadows will have high JIT overhead and result in more total profiled segments, but a lower total number of profiled instructions. Conversely, longer-running shadows will take advantage of hot code in the code cache, allowing for more instructions to be executed and fewer unique segments. To somewhat alleviate this overhead, we enable Pin's code cache persistence feature.

3.3 Multithreaded Programs

Threads are a particularly complicated issue. In our development system, a Linux 2.6 kernel, with the native POSIX thread library (NPTL), if one thread in a multithreaded program calls fork, only that thread survives in the new process. This behavior is common to all other threading libraries that we are familiar with. Following a fork, the address space is duplicated as usual, but the architecture context and corresponding kernel threads are lost. This is a bit unintuitive, but it is reasonable since a multithreaded fork would require the synchronization of all threads. Synchronization could be very slow, and the typical usage model is to call execve shortly after forking, so it is not surprising that this is how fork naturally behaves. Unfortunately, different behavior is necessary for shadow profiling to work on multithreaded programs. It would be relatively simple to implement a multithreaded fork in the operating system, but that approach would be cumbersome and non-portable. Instead, we are currently working on a userspace solution to emulate the desired behavior by synchronizing each thread in a signal handler and reproducing the context in the child process. This method will work as follows:

1. Barrier all threads in the program and store their CPU state.

2. Fork the process and recreate (clone) threads for those that were destroyed. Remember, the address space is identical. Only the CPU contexts are lost.

3. In each new thread, revive the previously stored CPU state.

4. Continue execution and virtualize thread IDs for system calls.

4 Performance Evaluation

Shadow profiling, like any sampling technique, balances a tradeoff between information and overhead. The balance between these can be controlled by two parameters to the profiling tool. First, the *sample size*, controls the number of consecutive instructions that a shadow should execute before exiting. A small sample size allows for more unique samples to be taken, but because of the initial overhead of creating a shadow process, the total number of instructions profiled is lower. Larger samples have less overhead, but since they have a coarser granularity, they may miss important program phases. Another parameter that affects the profiling overhead is *load*, which is defined as the average number of active shadow processes. A specific configuration is defined as the tuple $< sample\ size, load >$. In these experiments, we evaluate sample sizes of 1M, 10M, and 100M instructions, and load values of 0.0625, 0.125, 0.25, 0.5, 1.0, and 2.0, resulting in a total of 18 different configurations tested.

4.1 Experimental Configuration

All experiments were run on a 4-CPU system with Intel® Xeon™ 3.0GHz processors with 4096kB L3 cache. The system has 6GB of memory and is running Linux kernel version 2.6.9. The framework is evaluated using the SPEC CPU2000 integer benchmark suite. The floating point benchmarks are omitted because they generally exhibit highly repetitive behavior that is not as interesting from the perspective of profiling,

Each experiment presented is the average of 3 repeated trials. Even so, there still exists some degree of variability in performance and accuracy due to the random points at which shadows are created.

Pin's ability to persistently store its code cache across multiple runs is very important in reducing the overhead required to profile. However, there are a number of options for how to apply this feature. JIT compilation could be entirely obviated by creating persistent caches prior to running experiments, but this would unfairly ignore one of the most significant design challenges. For these experiments, a new persistent cache is created each time a program is invoked. The cache starts out empty, and each time a shadow process terminates, it writes it code cache back out to disk. Using this policy, the first few shadows execute very slowly because they must populate the code cache, but later shadows execute faster because they are able to benefit from previous traces that have already been instrumented.

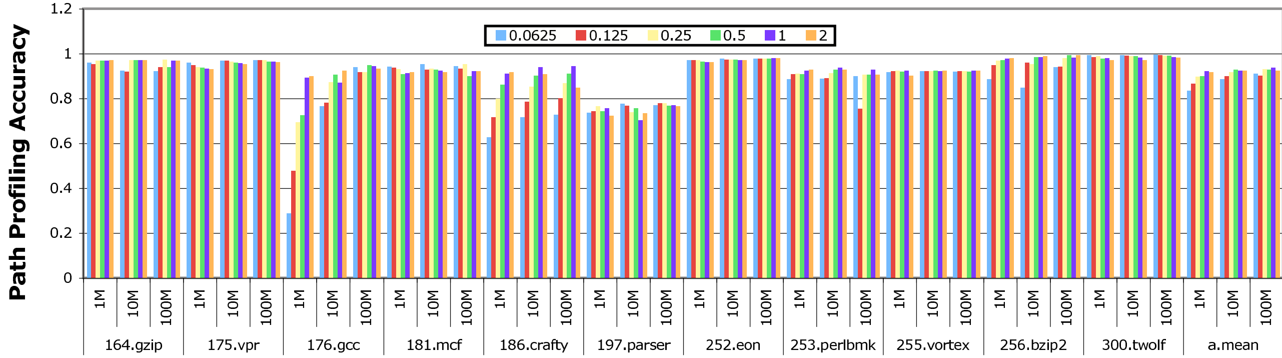


Figure 3. Path profiling accuracy relative to a perfect profile. Higher numbers indicate a closer match. The x-axis denotes configuration combinations of *load* and *sample size*.

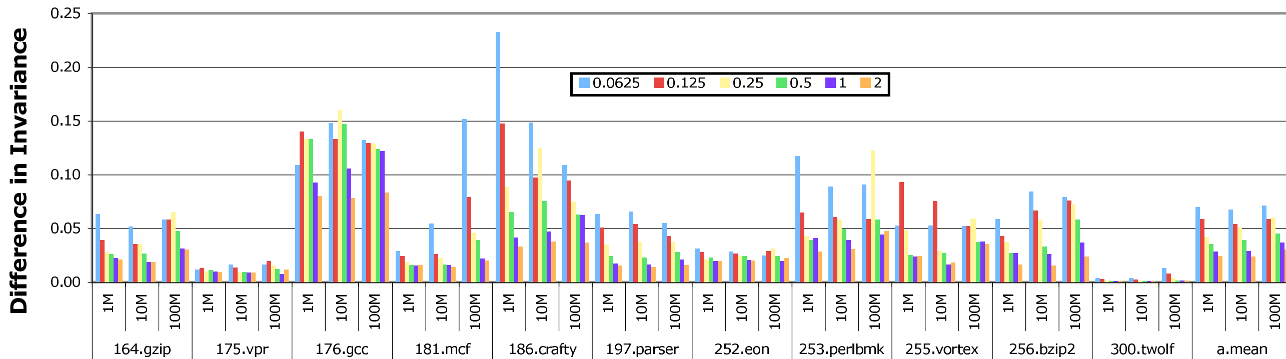


Figure 4. Value profiling difference in invariance. Lower numbers indicate a closer match. The x-axis denotes configuration combinations of *load* and *sample size*.

4.2 Interprocedural Path Profiling

Path profiling, first described by Ball and Larus [4], can be used to direct the compiler or a dynamic optimizer to improve code layout and create superblocks. When the instrumentation is inserted by the compiler, it can be highly optimized because the entire control flow graph is known. This type of instrumentation typically results in overheads of 30-40%. In a dynamic instrumentation system, indirect branch targets are not known, so every basic block must be instrumented and the overhead is a bit higher. Furthermore, Ball and Larus only examine intraprocedural acyclic paths. This study examines interprocedural paths, or those that span multiple functions. Our Pintool for collecting these paths incurs an overhead of about 3000%. While there are opportunities for optimizing the profiler itself, evaluating a naive implementation exemplifies one of the key benefits of shadow profiling: simplicity.

The accuracy of the partial path profile can be attained by comparing the sampled profile with a perfect profile. We use a method similar to Wall's weight matching scheme [36]. In this paper, accuracy is defined as

$$Accuracy\ of\ P_{est} = \frac{\Sigma_{p \in (H_{est} \cap H_{act})} F_{act}(p)}{\Sigma_{p \in H_{act}} F_{act}(p)} \quad (1)$$

In this equation, $F(p)$ is the flow of a path. This is defined as the path's count divided by the count of all the paths added together, and represents the weight that path p accounts for. H_{act} is the set of paths in the perfect path profile which are above a set threshhold. We use the top 5% as the threshhold in this study. H_{est} is then defined by selecting the hottest paths in the partial profile equal to the number of paths in H_{act}.

4.3 Value Profiling

To further demonstrate the robustness of the shadow profiling framework, we choose an instrumentation-based profile that has a considerably higher overhead. Value profiling can be used to locate variables and instructions that have predictable or invariant values at run-time, but cannot be identified with compiler analysis [9]. A value profile can

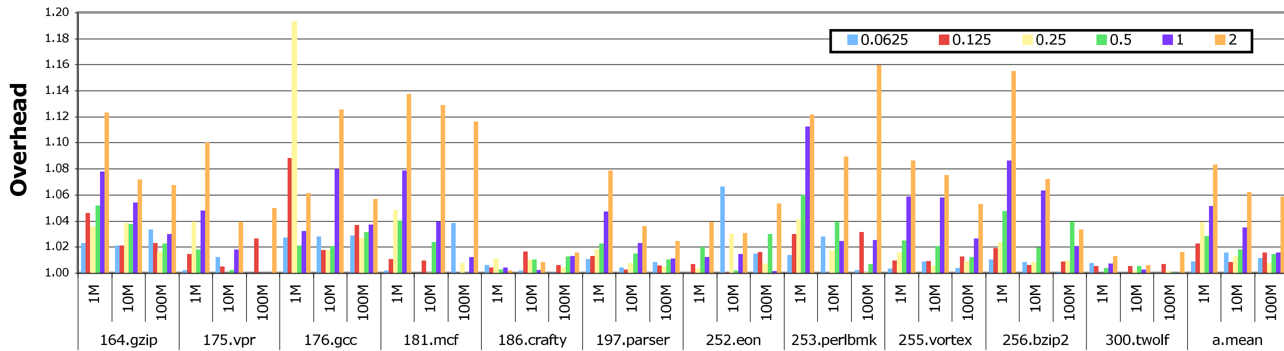


Figure 5. Slowdown versus native execution for shadow-based value profiling. The x-axis denotes configuration combinations of *load* **and** *sample size***.**

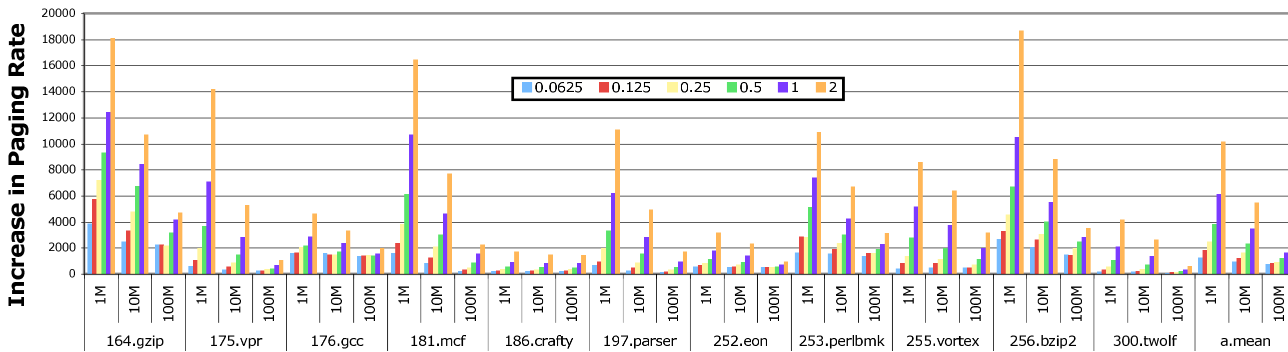


Figure 6. Increase in the page fault rate for shadow-based value profiling. The x-axis denotes configuration combinations of *load* **and** *sample size***.**

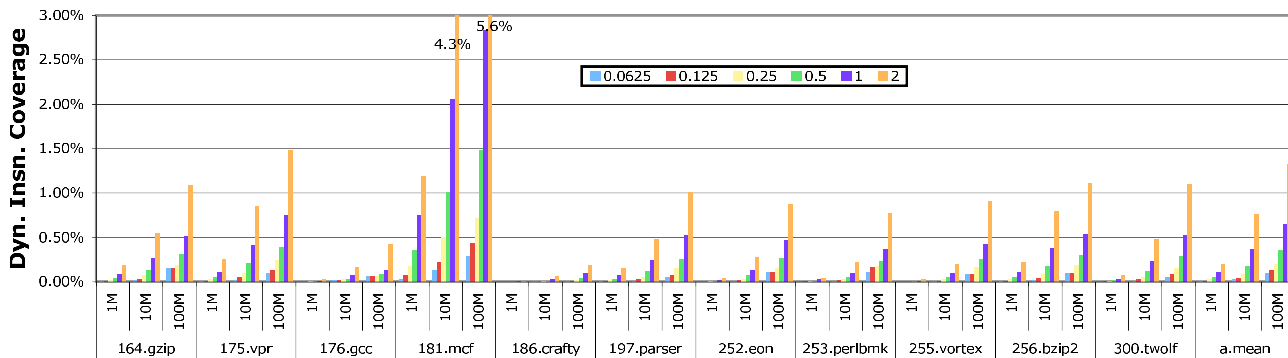


Figure 7. Dynamic instruction coverage for shadow-based value profiling. The x-axis denotes configuration combinations of *load* **and** *sample size***.**

be used to create specialized code for the observed common case of execution, and has been observed to achieve up to 21% speedup improvements in programs. However, collecting a full value profile is very expensive; our implementation has an overhead of up to 10,000%.

A general value profile is collected by instrumenting any instruction that has a destination register, including loads, arithmetic, and comparison operations. The profiler keeps a 50-entry top-N-value (TNV) table for each instruction. Each entry in the TNV table is a tuple containing the value and the number of times it has been observed. Each time a relevant instruction occurs, its TNV table is updated. To

allow for new values to make their way into the table, the lower half of the table is periodically flushed.

Sampled profiles are compared to perfect profiles by calculating the difference in invariance of top values (*Diff-Top*). This metric describes the weighted difference in invariance between two profiles for the top-most value in the TNV table for instructions that exist in both profiles. The invariance of an instruction is calculated as the number of times the top-most value was observed divided by the number of times that instruction executed. Each instruction's difference is included in an averaged weighted by the execution frequency in the perfect profile.

4.4 Profiling Accuracy

The profiling accuracy for shadow-based path profiling is shown in Figure 3. For all configurations tested, the average accuracy ranges from 83-93%, and most of the benchmarks achieve above 90% coverage in every experiment. The main trend that can be observed is that increasing load and sample size has a positive impact on accuracy. For these applications, all of the sample sizes do well, and a configuration of $< 100M, 0.25 >$ results in 93% accuracy with about 1% overhead.

One benchmark that exhibits these trends very well is 176.gcc. At $< 1M, 0.0625 >$, it achieves only 28% accuracy, but it quickly increases to over 94% with longer sample sizes. This effect is caused by the fact that 176.gcc has a very large code footprint, and therefore Pin spends more time compiling traces than in other applications. Longer samples and higher load increase code cache efficiency. Another interesting benchmark is 186.crafty because it has millions of unique paths. In fact, the perfect profiler cannot complete without pruning a large number of cold paths because it runs out of memory. 186.crafty exhibits trends similar to 176.gcc, though slightly less amplified.

Overall, path profiling accuracy shows that shadow profiling achieves good, representative coverage of a program's dynamic instruction mix. To further demonstrate its effectiveness, we evaluate value profiling, which requires much more rigorous instrumentation that previously could not have been considered for a dynamic optimization system. Figure 4 shows the difference in invariance versus a perfect profile. A lower difference in invariance indicates a greater similarity to the perfect profile. Again, nearly all of the benchmarks perform very well, and there is slightly less variation than with path profiling. On average, the accuracy remains within 4-8% of a perfect profile.

In both examples, the data shows that by sampling a relatively small portion of execution, representative profiles can be collected. Interestingly, the average accuracy isn't greatly affected by the varying profiling parameters, although individual benchmarks that are more difficult to profile are more dependent on the abundance of profile data collected.

4.5 Performance Overhead

Now that shadow profiling has been shown to achieve good, representative coverage versus perfect profiles, we examine the overhead incurred by the framework. Figure 5 shows the slowdown experienced by the original program for value profiling, defined as the profiled execution time divided by the native execution time. The overhead for path profiling is not presented, but it highlights the same trends and is slightly higher.

The two trends that are expected to occur can be observed in the figure. First, increasing the number of active shadow processes, or profiling load, results in an increase in overhead. Second, lengthening the sample size results in a decrease in overhead because the application will fork less often. Hence, low profiling load combined with long samples will result in the lowest overhead.

Although shadow profiling can be tuned to incur very low average overhead, some of the configurations tested caused the original program to slow down by up to 20%. In the experimental system, there are two man sources of overhead: copy-on-write paging and bus bandwidth contention. Disk contention is not a factor since all of the applications easily fit in memory. Figure 6 shows the increase in page fault rate (page faults per second) observed by the original program when shadow profiling is enabled. Unlike the overall performance results, this graph is very regular and predictable, yet it corresponds well with performance overhead. For example, 186.crafty and 300.twolf have among the lowest paging rate increases and also the lowest overhead. Note that the increase in page faults is due to memory copies, not faults to disk.

We find that we can achieve very high accuracy (over 93%) with overhead measuring around 1%. This shows that the framework can be tuned so that profiling overhead is negligible, yet the accuracy is still within 93% of a perfect profile. Therefore, shadow profiling is a viable tool for gathering information in dynamic optimization systems.

4.6 Coverage

Another measurement important to instrumentation sampling is dynamic instruction coverage. Figure 7 shows the percentage of dynamic instructions captured by value profiling instrumentation. A configuration of $< 100M, 0.25 >$ captures only 0.2% of total execution, yet still is enough to achieve over 90% accuracy for both path and value profiles, Interestingly, 181.mcf receives four times better coverage than average. This is because 181.mcf has a very small code

footprint and frequent misses in the data cache allow the instrumentation costs to be hidden by long latencies.

5 Related Work

Replica processes have traditionally been used for providing redundancy in fault tolerant systems. Systems have been proposed that implement replica processes using a scheme similar to shadow profiling [30].

Maintaining determinism among replica processes with interrupts and multi-threaded execution is challenging and still an open research problem [5, 40]. Shadow profiling sidesteps the issue of determinism by simply not requiring deterministic execution. The shadow processes exist only to collect meaningful profile information and are not required for program correctness. Also, if any unsafe or undefined execution occurs, the shadow process can be safely exit and be recreated. To the best of our knowledge, this is the first use of replica processes for profiling.

Software tools such as Pin [24], ATOM [33], and DynamoRIO [6] can be used to instrument an application and collect full execution profiles. However, the overhead of instrumenting an entire execution is prohibitive and infeasible when collecting detailed profiles on long-running programs.

Sampling-based instrumentation approaches can significantly lower the overhead of profiling. For example, Arnold and Sweeney periodically sample the call stack to to approximate the calling context tree [3]. Whaley uses similar samples for a partial calling context tree [38]. The Arnold-Ryder [2] instrumentation framework and ephemeral profiling [34] reduce instrumentation overhead by only sampling bursts of execution. The Arnold-Ryder framework, implemented in the Jalapeno JVM, creates two versions of each procedure. Execution alternates between one version that contains the original code with *checks* at procedure entry points and loop back edges, and another that is an *instrumented* version of the same code. When a counter decrements to zero in the checking version, control transfers to the instrumented version. Then an intraprocedural, acyclic trace is collected, the counter is reset, and execution transfers back to the checking version. Hirzel and Chilimbi coined the term "bursty tracing" for this technique and extended it to allow longer, interprocedural traces. Also, instead of Java bytecode, they apply it to IA32 binaries using the Vulcan binary rewriting tool [18]. The sampled execution time can be optimistically estimated by:

$$T_{total} = T_{checking} * Overhead_{checking} + T_{inst} * Overhead_{inst} \quad (2)$$

We may assume that $Overhead_{checking}$ is negligible. Now suppose that $Overhead_{inst}$ is 10. Then if 1% of the program executes in instrumented mode, T_{total} will be 10% higher than without instrumentation. Of course, as the overhead of instrumentation and the time spent in instrumented code increase, so does the total execution time.

Shadow profiling is partially inspired by bursty sampling-based techniques. However, instead of executing the instrumentation code along with the original application code, shadow profiling shifts the instrumentation to the shadow process which is able to run in a different hardware context. The original application can proceed natively without any slowdown due to instrumentation. This benefit allows shadow profiling to scale and allow better coverage for higher overhead instrumentation techniques.

Low-overhead software profiling techniques are often studied for the collection of detailed profiles such as path profiles [4, 16], value profiles [9], memory stream profiles [11] and whole program paths [22]. Shadow profiling is orthogonal to profile-specific software techniques and can be used to further improve the overhead of such techniques.

Specialized profiling hardware such as Sastry's rapid profiling via stratified sampling [28], Merten's hot spot detection [25], Vaswani's programmable hardware path profiler [35], and Conte's profile buffer [14] have also been proposed for profiling with low overhead. Zilles [41] and Heil [17] propose co-processors specifically for profiling. ProfileMe [15] introduces hardware for profiling the path of specific instructions through the pipeline.

Recent microprocessors have been designed with onchip performance monitoring units (PMU) [20, 21, 31] containing a set of performance counting registers which trigger software interrupts for sampling. DCPI [1] samples the program counter to detect stalling instructions. PMU sampling has been shown to be effective in collecting path profiles [10, 29], cache miss profiles [23], and value profiles.

Hardware profiling mechanisms are promising but have a few drawbacks. While software instrumentation is flexible and portable, hardware mechanisms are limited to the features they are designed with. Furthermore, PMU functionality is often very processor-specific making portability of profiling tools difficult. More importantly, hardware designers are often reluctant to include profile-specific hardware in microprocessor designs. Profiling hardware impacts the processor design cycle. With the increasing time-to-market pressures, the top design priorities are hardware validation, and processor performance. Thus, if hardware profiling is given any attention, is is often as a "second-class citizen" in processor design [32].

With the microarchitectural trends moving toward massive multi-threaded and multi-core processors, shadow profiling aims to leverage the abundance of extra hardware is already available in the form of extra hardware threads or cores. Shadow profiling simply creates shadow processes for instrumentation sampling and then allows the operating system to schedule the processes and leverage the available hardware resources. SuperPin [37] uses a similar approach,

but instead aims to deterministically replicate full execution by creating "slices" of execution between each system call.

6 Future Work

Since architectural trends are forcing programming paradigms to shift more towards multithreaded programming, it is imperative that profiling research adjust accordingly. To address this issue, we are currently developing a userspace implementation that transparently recreates thread contexts since they do not live after a fork system call. Due to both job and instruction scheduling, the shadow will not be a deterministic replica after a fork, but it is still suitable for the purpose of profiling.

Currently, no discretion is given to *when* a shadow should be created or *how much* information is enough. There are several options to explore in this domain. First, some consideration should be given to the load on the rest of the system. Under high system load, the profiling load should be scaled back. To better choose phases to profile, the compiler could provide insight into which regions of code would benefit the most from profiling, and spawn shadow processes accordingly. Alternately, a performance counter monitoring system could detect phase changes and profile only after phase transitions occur. In a continuous optimization system, as more profiles are collected the data is likely to converge. Once the profile stabilizes, the sampling rate can be reduced so as only to detect when a program's behavior deviates from the existing profile. A statistical model, similar to that used by SMARTS [39] for architectural simulation, could likely be applied for profiling purposes to direct sampling frequencies and lengths. Calder also proposed a method for convergent profiling of value profiles [9].

7 Conclusion

This paper presents shadow profiling, a novel technique for performing expensive, instrumentation-based analysis in parallel with a running program. The method demonstrates the best qualities from both instrumentation- and sampling-based approaches; it is shown to achieve high accuracy for interprocedural path profiles and value profiles, while incurring negligible overhead on the running program. For value profiles, shadow profiling achieves 93% average accuracy compared to a perfect profile. Path profiling achieves similar results, with an average of 2-7% difference in invariance versus a full profile. In both cases, ideal coverage is achieved with around 1% overhead. The tool requires no special operating system or hardware support, and greatly increases the viability of dynamic and continuous optimization systems. Since profiling is done in parallel and the number of processor cores is likely to increase considerably in coming years, instrumentation slowdown is no longer a major concern. Designers can now focus less on minimizing the overhead of profiling, and more on applying optimizations.

Acknowledgements

The authors would like to thank Robert Cohn and the rest of the Pin team for the exceptional support provided and answering many questions.

References

[1] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: where have all the cycles gone? In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 1–14, New York, NY, USA, 1997. ACM Press.

[2] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 168–179, 2001.

[3] M. Arnold and P. Sweeney. Approximating the calling context tree via sampling, 2000.

[4] T. Ball and J. R. Larus. Efficient path profiling. In *International Symposium on Microarchitecture*, pages 46–57, 1996.

[5] C. Basile, Z. Kalbarczyk, and R. Iyer. Preemptive deterministic scheduling algorithm for multithreaded replicas. In *In Proceedings of the International Conferenace on Dependable Systems and Networks*, 2003.

[6] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *1st International Symposium on Code Generation and Optimization (CGO-03), March 2003.*, 2003.

[7] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[8] M. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. Sreedhar, H. Srinivasan, and J. Whaley. The jalapeno dynamic optimizing compiler for java. In *Proceedings ACM 1999 Java Grande Conference*, pages 129–141, San Francisco, CA, United States, June 1999. ACM.

[9] B. Calder, P. Feller, and A. Eustace. Value profiling and optimization, 1999.

[10] H. Chen, W.-C. Hsu, J. Lu, P.-C. Yew, and D.-Y. Chen. Dynamic trace selection using performance monitoring hardware sampling. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 79–90, Washington, DC, USA, 2003. IEEE Computer Society.

[11] T. M. Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 191–202, New York, NY, USA, 2001. ACM Press.

[12] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 199–209, New York, NY, USA, 2002. ACM Press.

[13] R. S. Cohn, D. W. Goodwin, and P. G. Lowney. Optimizing alpha executables on windows nt with spike. *Digital Tech. J.*, 9(4):3–20, 1998.

[14] T. M. Conte, B. A. Patel, and J. S. Cox. Using branch handling hardware to support profile-driven optimization. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 12–21, San Jose, CA, USA, 30 November–2 December 1994. ACM Press.

[15] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Z. Chrysos. Profileme : Hardware support for instruction-level profiling on out-of-order processors. In *International Symposium on Microarchitecture*, pages 292–302, 1997.

[16] E. Duesterwald and V. Bala. Software profiling for hot path prediction: Less is more. In *Proceedings of the 9th International Conference on Architectural Suport for Programming Languages and Operating Systems*, November 2000.

[17] T. H. Heil and J. E. Smith. Relational profiling: enabling thread-level parallelism in virtual machines. In *International Symposium on Microarchitecture*, pages 281–290, 2000.

[18] M. Hirzel and T. Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4).*, 2001.

[19] G. Hunt and D. Brubacher. Detours: Binary interception of Win32 functions. In *The 3rd USENIX Windows NT Symposium*, 1999.

[20] IBM. *PowerPC 740/PowerPC 750 RISC Microprocessor User's Manual*, 1999.

[21] Intel Corporation. Intel Itanium 2 processor reference manual: For software development and optimization. May 2004.

[22] J. R. Larus. Whole program paths. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 259–269, 1999.

[23] J. Lu, H. Chen, P.-C. Yew, and W.-C. Hsu. Design and implementation of a lightweight dynamic optimization system. In *Journal of Instruction-Level Parallelism 6(2004)*, pages 1–24, April 2004.

[24] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM Press.

[25] M. C. Merten, A. R. Trick, C. N. George, J. C. Gyllenhaal, and W. W. Hwu. A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. In *Proceedings of the 1999 International Symposium on Computer Architecture*, May 1999.

[26] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *Electr. Notes Theor. Comput. Sci.*, 89(2), 2003.

[27] V. J. Reddi, D. Connors, R. Cohn, and M. D. Smith. Persistent code caching: Exploiting code reuse across executions and applications. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, San Jose, CA, USA, 2007.

[28] S. S. Sastry, R. Bodik, and J. E. Smith. Rapid profiling via stratified sampling. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 278–289, 2001.

[29] A. Shye, M. Iyer, T. Moseley, D. Hodgdon, D. Fay, V. J. Reddi, and D. A. Connors. Analyis of path profiling information generated with performance monitoring hardware. In *INTERACT '05: Proceedings of the 9th Annual Workshop on Interaction between Compilers and Computer Architectures (INTERACT'05)*, pages 34–43, Washington, DC, USA, 2005. IEEE Computer Society.

[30] A. Shye, V. J. Reddi, T. Moseley, and D. A. Connors. Transient fault tolerance via dynamic process redundancy. In *Proceedings of the 2006 Workshop on Binary Instrumentation and Applications (WBIA)*, 2006.

[31] B. Sprunt. Pentium 4 performance-monitoring features. In *IEEE Micro 22(4)*, pages 72–82, 2002.

[32] B. Sprunt. Performance monitoring hardware will always be a low priority, second class feature in processor designs until.., February 2005. In Workshop on Hardware Performance Monitors in conjuction with HPCA-11.

[33] A. Srivastava and A. Eustace. Atom: a system for building customized program analysis tools. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, New York, NY, USA, 1994. ACM Press.

[34] O. Traub, S. Schechter, and M. Smith. Ephemeral instrumentation for lightweight program profiling, 2000.

[35] K. Vaswani, M. J. Thazhuthaveetil, and Y. N. Srikant. A programmable hardware path profiler. In *CGO '05: Proceedings of the International Symposium on Code Generation and Optimization*, pages 217–228, Washington, DC, USA, 2005. IEEE Computer Society.

[36] D. W. Wall. Predicting program behavior using real or estimated profiles. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 59–70, New York, NY, USA, 1991. ACM Press.

[37] S. Wallace and K. Hazelwood. Superpin: Parallelizing dynamic instrumentation for real-time performance. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, San Jose, CA, USA, 2007.

[38] J. Whaley. A portable sampling-based profiler for java virtual machines. In *Java Grande*, pages 78–87, 2000.

[39] R. Wunderlich, T. Wenisch, B. Falsafi, and J. Hoe. Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling. In *30th Annual International Symposium on Computer Architecture, June 2003.*, 2003.

[40] W. Zhao, L. Moser, and P. Melliar-Smith. Deterministic scheduling for multithreaded replicas. In *In Proceedings of the IEEE Workshop on Real-Time Object-Oriented Dependable Systems, Sedona, AZ, February 2005. 35*, 2005.

[41] C. B. Zilles and G. S. Sohi. A programmable co-processor for profiling. In *Proceedings of the International Symposium on High-Performance Computer Architecure*, 2001.