Phase-Aware Remote Profiling

Priya Nagpurkar   Chandra Krintz   Timothy Sherwood

Computer Science Department
University of California, Santa Barbara
{*priya,ckrintz,sherwood*}*@cs.ucsb.edu*

Abstract

Recent advances in networking and embedded device technology have made the vision of ubiquitous computing a reality; users can access the Internet's vast offerings anytime and anywhere. Moreover, battery-powered devices such as personal digital assistants and web-enabled mobile phones have successfully emerged as new access points to the world's digital infrastructure. This ubiquity offers a new opportunity for software developers: users can now participate in the software development, optimization, and evolution process while they use their software.

Such participation requires effective techniques for gathering profile information from remote, resource-constrained devices. Further, these techniques must be unobtrusive and transparent to the user; profiles must be gathered using minimal computation, communication, and power. Toward this end, we present a flexible hardware-software scheme for efficient remote profiling. We rely on the extraction of meta information from executing programs in the form of phases, and then use this information to guide intelligent online sampling and to manage the communication of those samples. Our results indicate that phase-based remote profiling can reduce the communication, computation, and energy consumption overheads by 50-75% over random and periodic sampling.

1. Introduction

The explosive growth in Internet bandwidth and availability has precipitated a significant change in the way that software is bought, sold, used, and maintained. Users are no longer a set of disconnected individuals that passively execute disks from a shrink-wrapped box, but are instead often far more involved in the software development and improvement process. Users currently demand bug fixes, patches, upgrades, forward compatibility, and security updates served to them over the ever-present network.

This ubiquity of access offers a new opportunity for software engineers: users can now participate in the software development and evolution process. Specifically, users can dynamically transmit error reports upon program failure in modern operating systems [21]. Moreover, effective remote monitoring systems have been proposed in the literature and deployed that allow users to participate in coverage testing [20] and bug isolation [18, 7], *while* they use their software.

We are currently developing a similar system for program performance optimization and software evolution. However, unlike prior work, the platforms that we are targeting are those that have emerged as new access points to the world's digital infrastructure: mobile, resource-constrained, battery-powered devices, e.g. personal digital assistants (PDA) and web-enabled cellular phones devices and their software continue to grow in complexity and capability, techniques are needed to ensure efficient execution, user satisfaction, and minimal power consumption. Feedback-based optimization and software evolution offers potential for such systems since such techniques gather information about a program *while* it is executing, once it has been deployed in the wild.

Since mobile devices typically have neither the extra space for compilers and optimizers, nor the resources to execute them on-the-fly, we propose an alternative solution: *a distributed optimization system*. Our system will gather information about a running program, transmit this information to an *optimization center* for analysis, possible recoding, and re-compilation using feedback-based optimization, and then update the code on the end-user system when the opportunity or need arises.

Key to the success of such an approach, and the topic of this paper, is a highly efficient remote performance profiling system that is *transparent and unobtrusive*, i.e., that consumes only minimal device resources. This latter requirement is a significant challenge since profiles are commonly collected by executing instrumented versions of the software. Moreover, for deployed software, we must also communicate this information back to optimization center for analysis. This problem of overhead introduction is exacerbated for mobile devices with limited resources as this performance degradation can translate into significant battery drain.

With this work, we present a novel approach to remote program profiling that achieves efficiency and accuracy through *the exploitation of program phases*. Using program phase behavior, we can summarize a software system as a minimal but diverse set of program behaviors in a manner that is distributed, dynamic, efficient, and that accurately reflects overall program behavior. We propose a hardware-software method for general-purpose program profiling. The hardware efficiently monitors program execution behavior and makes predictions about what phase will occur next. The software system samples the program for only previously unseen phases, significantly reducing the overhead of program profiling. We evaluate both the efficiency (in terms of computation, communication, and battery power) as well as the accuracy of our approach for a number of different profiles types that have been shown previously to be important for feedback-based optimization, e.g., hot methods, hot call pairs, and hot paths.

In summary, this paper makes the following contributions:

- New architectural features that are useful for profiling and optimizing remote connected devices such as IPAQs and cell phones. These hardware hooks, guide flexible software profiling in selecting the most important parts of program execution.
- We show that these hardware guides for sampling can be built by exploiting the concept of program phase behavior, and that profile communication can be reduced by up to a factor of 6 over random sampling (where both achieve an accuracy of 10%).
- A simple online policy for deciding when to profile that is almost as effective as one with full trace knowledge
- A demonstration that phases can be used to accurately guide the profiling of multiple different types of information (basic block profiles, hot methods, and call-pair tracing
- An empirical evaluation of these techniques for all of the overheads associated with remote profiling for resource-restricted devices (communication, computation, and power).

We describe each of these components and their empirical evaluation in the sections that follow.

2. Remote Profiling

Profiling an application that is under the control of the developer is commonplace today. In a typical software development cycle, programmers test and optimize their application using some set of inputs. In this setting, the overhead introduced by profiling is not of great concern since the program is being executed solely for the purpose of testing and identification of optimization opportunities. One of the limitations of this methodology, however, is the choice of inputs used by the developer may not fully exercise a program. Moreover, testing and optimizing for all possible hardware configurations, software configurations, and use-scenarios is not feasible. These problems are exacerbated by user requirements, preferences, and devices that change over time.

To address these issues, we propose to gather profiles from *deployed software* as it executes on user devices that are connected via the network. These profiles will help developers understand how their code is being exercised *in the wild* aiding in the creation of user models, assisting in the classification of users into groups that exercise the program in a similar manner, and to enable feedback-based distributed optimization and software evolution.

2.1. Sample-Based Remote Profiling

Upon instigating this research, we assumed that the major challenge would lie in the analysis of profile data and its use for optimization. We immediately discovered that even the initial step of gathering profile data from our target platforms, wireless IPAQ devices, was a significant hurdle. It is this aspect of our system that we examine in this paper. Initially, it may appear that we can apply extant techniques from the areas of sampling and architectural performance analysis [13, 27, 3, 2, 8] to our problem of remote profiling. The problems appear similar since both are concerned with the examination of a small subset of a programs execution and the use of that information to estimate and evaluate the performance of the running application. In fact these two problems, and the solutions that address them, differ in several significant ways.

The first important difference is that in an online profiling environment, a decision has to be made at each point in time, whether or not to profile. Most of the past work requires multiple passes over the data, at least in the worst case. For example, in the SimPoint framework [24], the first pass over the data analyzes the program at a high level by finding regions of execution that are similar to one another. The next step, then examines *all* of the points and picks a small subset of the program's execution for sampling. Statistical sampling techniques suffer from a similar problem in that they may require multiple passes over the data until

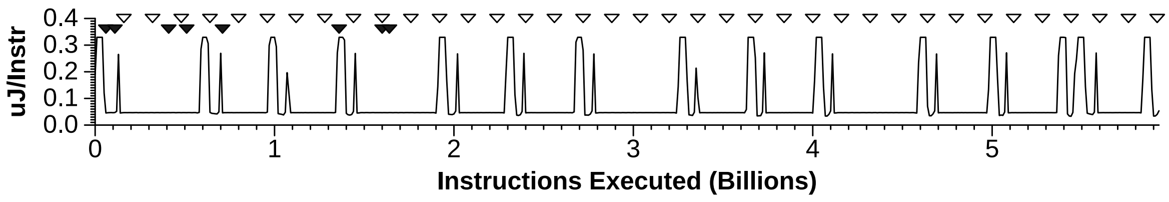


Figure 1. The figure shows the run-time power usage of the full execution of the program `mpeg encode`**.The program exhibits different phases, marked by periods of high and low power. A random or periodic sampling method (the white triangles) will continue to take samples over the full execution of the program. A more intelligent sampling technique based on phase information (shown as black triangles) can achieve the same error, by taking few key samples from each phase.**

the statistics of the results stabilize. This is not directly applicable to our problem since once a profile is *not* taken for some amount of time, there is no going back to re-take it. Furthermore, the decision as to whether a profile should be taken at time T, must be made from partial knowledge of the program execution from time 1 to $T-1$, i.e., we must *predict* when to sample. The second difference is that our goal is to develop a profile gathering technique that is general purpose enough to be used for a variety of profiling applications. As such, we cannot rely solely on hardware performance counters to provide all of the information we need. Instead we need a more widely applicable software instrumentation. Moreover, we must keep performance overhead to a minimum to reduce interference and to not degrade the user's perception of program or device performance (e.g. computation, communication, battery life).

The final key difference is the severely limited resources of end user devices that we must employ. It is acceptable for an offline technique to generate gigabytes of trace information and to spend hours analyzing it, but this is not the case for a cell phone or a PDA. Since we must impose very little computation and communication overhead on the users to maintain transparency, we must extract profile data very efficiently – while ensuring that we collect an accurate snapshot of the program's behavior.

2.2. Phase-Based Remote Profiling

The key to enabling efficient, post-deployment, remote collection of accurate profile information is the exploitation of program phase information. Phases can be used to create an intelligent profiling scheme that carefully chooses profiling points *online*. The way a program's execution changes over time is not random, but is often structured into repeating behaviors, i.e., phases. Using the description of phases from our previous work [23], a phase is a set of dynamic instructions, i.e., *intervals*, during program execution that have similar behavior, regardless of temporal adjacency. Prior work [23, 24, 25, 11, 14] has shown that it is possible to identify, predict, and create meaningful classifications of phases in program behavior accurately. Phase behavior has been used in the past to reduce the overhead of architectural simulation [24] and to guide online optimizations [11, 14, 25].

In this paper, we employ program phase behavior in a novel way: to enable efficient collection of accurate profile information from remote users. The advantage we gain by using phase information is that we need only to gather information about part of the phase and we can then use that information to approximate overall profile behavior. By carefully selecting a representative from each phase, we can drastically reduce the number of times that we need to sample and the amount of total communication required for profile transmission to the optimization center. Since an interval will be similar to all other intervals in a phase, it can serve as a representative of all other intervals in the phase. As such, only representative intervals of the program phases in the program need be collected (instrumented, communicated, and analyzed) to capture the behavior of the entire program. This will make more efficient use of those limited resources available on mobile devices. Furthermore, these low-overhead profiles will be highly accurate (very similar to exhaustive profiles of the same program).

Figure 1 exemplifies our approach using actual energy data gathered from the execution of the mpeg encoding utility. The execution of mpeg exhibits a small number of distinct phases during execution that repeat multiple times. A random or periodic sampling method will continue to take samples over the full execution of the program regardless of any repeating behavior. In Figure 1, the white triangles show where samples would be taken if sampling is done periodically to achieve an accuracy error of 5% (i.e. the resulting basic block count profile is within 5% of the exhaustive profile) . This has the unfortunate drawback that *most* of the samples will not provide any new information because they are so similar to samples seen in the past. A more intelligent sampling technique based on phase information (shown as black triangles) can achieve the same error rate with sig-

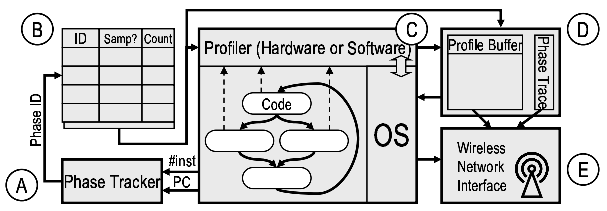


Figure 2. Overview of the phase-aware profiling scheme. Phases are tracked in hardware (A) and the results are fed to a small table that tracks the state of each phase (B). When a phase is deemed to be important , the profiler is notified and a sample is taken (C). The sample is stored, along with its importance in a small profile buffer in either hardware or software (D). This information is then transmitted back to a trace aggregation center (E).

nificantly fewer samples. This is done by taking only key samples from each phase.

2.3. Supporting Phase-Driven Profiling

Figure 2 depicts our implementation of *Phase-Aware Profiling*. Although each of the components can be implemented in either hardware or software, we take a hardware-centric approach. To predict the phase in which a future interval will be, we employ the Phase Tracker hardware that we proposed in prior work [25]. The PhaseTracker is a small, low area, low overhead hardware resource that consumes approximately 4 picojoule of energy per dynamic branch. The Phase Tracker (A) collects dynamic branch behavior of a program into intervals and segregates the intervals into phases according to a similarity threshold between interval execution characteristics. The similarity threshold governs how many phases are generated and how similar the intervals are within a phase. A higher threshold value will generate fewer phases, each consisting of more intervals – and the similarity across the intervals any single phase will exhibit more variance. Thus, we can adjust the threshold according to the number of samples needed; however the resulting samples will cover the most diverse and important sets of behaviors. The Phase Tracker uses branch program counters and the number of instructions executed between branches, to produce a prediction for the phase of the next interval, (complete details on the prediction process can be found in [25]). Prior work has shown the accuracy of PhaseTracker phase prediction to be 85-90% [25]. We assume a prediction accuracy of 100% in this work; as such, our results indicate an upper bound on the potential of phase-aware profiling performance.

Since the PhaseTracker is shared by multiple processes in a large-scale system (much like hardware performance monitors (HPMs)), we must provide a mechanism to distinguish per-process phase data. In this work, we assume that the PhaseTracker tracks a single process and that the operating system toggles phase tracking (via a register in the PhaseTracker hardware) upon a context switch. We are investigating the implementation of such techniques as part of future work.

The Phase Tracker outputs a *phase ID*, which is a unique identifier for the behavior likely to be observed in the current interval. We store the phase IDs in a small table to track each phase and identify when a sample should be taken. We have found that a table of size 20 ensures that there is a minimal number of misses if the table is fully associative with random replacement. In general, the worst case is one in which two similar behaviors are sampled more than once as a resulting table miss. However, the performance effects from such misses are negligible for tables of this size. The table tracks a list of phase IDs and stores a "sampled bit" that indicates if the phase has been sampled so far. Additionally, we record a count of the number of times this phase has been seen in the past. Our system makes a sampling decision using a representative selection policy (described in Section 3.2.1).

When a decision to sample an interval is made, the profiler (C) is informed to take a sample. The profiling system is one in which profiling can be turned on and off. Such a system can be implemented in either hardware or software. We assume a software based profiling scheme like [3, 16, 7] that switches between instrumented and non-instrumented versions of the code depending on whether a particular part of the execution is to be profiled or not. As part of future work, we are incorporating a hardware based approach to avoid code duplication based on the DISE dynamic expansion of microprocessor instructions [9]. Once the profile has been generated, we store it in a specialized profile buffer and tag the profile with the phase id (D). In addition to the profile data, we record a trace of phase IDs from intervals in *previously seen* phases. This enables us to reconstruct accurate time-varying behavior of the program if necessary.

Once the buffer fills, it must be emptied via transmission. At this point, the profile is transmitted over a wireless network (part of the IPAQs or over a cellular network) back to some data center for study (E). We evaluate the efficacy of transferring data intermittently while the program is executing (e.g. when storage is limited) as well as transferring the complete phase trace once the program terminates. Since communication consumes significant battery power in mobile devices, we must ensure that we minimize the number of bytes transfered. As such, in addition to using phase behavior to reduce profile size, we also incorporate compression of the trace prior to transmission. However, the application of compression consumes computational resources. We include the effect of this tradeoff (increased computation for decreased communication due to compression) as part of the empirical evaluation of our approach. In general, we found that the benefit from reduced communication overhead far outweighs the computational overhead we introduce – in terms of battery power.

3. Empirical Evaluation

In this section we present the empirical evaluation of our approach. We begin with a discussion of our experimental methodology. We then present results from a number of experiments we performed to evaluate the efficacy of phase-aware remote profiling for resource-constrained devices. We present empirical data for both profile accuracy and collection overhead and compare our system to two other commonly used profile collection techniques.

3.1. Experimental Methodology

Since we are targeting mobile devices such as cell phones and PDAs, we evaluated our system using six benchmarks from the MediaBench benchmark suite [17], a suite designed for the empirical evaluation of media applications. The benchmarks we used include the encoding and decoding programs for mpeg (movie), jpeg (picture), and gsm (voice). We show the basic statistics for the programs and inputs we used in this study in table (a) of Figure 3. The second column in the table is the number of static branches in the program, which correlates with the size of the branch profiles generated. The next five columns show the dynamic statistics: number of branches executed (in millions), number of instructions executed (in millions), the cache miss rate assuming a 64K, 4-way set associative, instruction and data cache, the energy consumed by executing the program (in Joules), and the execution time (in seconds). Since the inputs that are provided with MediaBench are very short, and because these applications are typically used in a streaming fashion, it was necessary to find more substantial inputs to analyze the realistic long term effects of profiling. We plan to make these inputs available via our web page.

We evaluated our remote profiling framework using simulation; we employed SimpleScalar to emulate a StrongARM processor extended with a branch tracking mechanism. We modified the simulator to emulate the capture of phase information as described in [25] with an interval size of 10 million instructions. We used the BitRaker Anvil framework [5] to collect different profile types (described in Subsection 3.2.3).

To compute energy consumption and execution time, we used a model that we generated from an actual hardware system. We computed the energy consumed per-instruction (including events such as cache misses) and per-byte-transmitted energy consumed and instructions per second. The values used are summarized in table (b) in Figure 3. We generated these values using an HP iPAQ H3835 running Familiar Linux v0.6.1, a Lucent/Orinoco Gold wireless card, and hand-coded benchmarks. We periodically (every 10 seconds) measured battery voltage and current levels using external monitors. We calibrated our model and validated it against a variety of benchmarks.

In the table ((b) in the Figure), we report the average Joules/s consumed by each of these latter single-instruction programs (IREG: integer register operations, IMEM-R: load operations that miss in the L1 cache, IMEM-W: store operations that miss in the L1 cache, and FPREG: floating point operations). We compute instructions per second of each benchmark in a similar fashion using the instructions per second measurement of each constituent instruction type (reported via simulation). To compute the power consumption for transfer, we computed the number of Joules per byte transfered using the specifications of our wireless card. We show these values and the Joules-per-byte-transfered in the final row of the table.

3.2. Results

To evaluate the impact of remote profiling on both the power and timing and to examine its accuracy versus its overhead, we implemented four different profile collection policies:

- Exhaustive - gather an exact profile for each interval. We use this policy to evaluate the accuracy of the other policies.
- Periodic Sampling - gather a profile every Nth interval, for N in [3,100].
- Random Sampling - gather a profile for interval i with a probability of 1/N for some N
- Phase-based - gather a profile for every interval that is dissimilar from all previously gathered intervals, given some threshold of similarity.

For the periodic and random techniques, we gathered data for different sampling frequencies. The number of intervals profiled, and therefore the percentage of total execution profiled, depends on the sampling period N. We performed experiments for a range of sampling frequencies which correspond to a range of overheads and accuracies. Because a truly random technique is at the whim of chance as to whether or not it performs well, we characterized two aspects of random profiling for each of the different percent-sampled values: 1) we computed the average error across 10 runs (avg random), and 2) we computed the maximum error seen across 10 runs (max random). To get a range of

		Dynamic Statistics				
Benchmark	Static Branches	Branches (Millions)	Instructions (Millions)	Cache Miss Rate	Energy (Joules)	Time (seconds)
gsmdecode	572	182.05	1610.05	0.000	29.93	16.35
gsmencode	748	79.42	2562.29	0.000	29.38	19.93
jpegdecode	930	111.76	1421.33	0.006	46.50	21.87
jpegencode	1175	433.70	4218.60	0.002	100.65	51.41
mpegdecode	1104	309.95	3007.85	0.001	65.03	900.63
mpegencode	2216	244.25	4196.19	0.001	52.63	282.40
Average	1124	226.86	2836.05	0.002	54.02	215.43

(a)

Instr Type	Average Joules / s	Instr / s (Millions)
IREG	0.865	204.790
IMEM-R	0.973	19.462
IMEM-Rcache	0.000	137.510
IMEM-W	1.340	11.625
FPREG	0.965	0.439
Wireless Card	Specification 5V*0.285A	Max Bandwidth
Transmit	1.425	11Mb/s

(b)

Figure 3. (a) General benchmark statistics (b) Empirical data used to compute energy consumption.

accuracies and overheads, we adjusted the parameter N and examined the effect.

For phase-aware profiling, as described earlier, we began with an implementation of the phase prediction system that we developed in prior work [25]. We then picked an interval from each phase to act as the representative from that phase. We implemented various policies that identify different phase representatives. As we demonstrate in the next subsection, the policy used to pick this representative can have a large impact on the results, especially at low sampling rates. Unlike the random and periodic sampling approaches, there is no sampling frequency variable that we can vary to get different tradeoff points between accuracy and overhead. Instead, we achieve a similar effect by dynamically tuning the *similarity threshold*. The similarity threshold determines the cutoff point at which two intervals are said to be similar and hence are part of the same phase. As we lower the threshold, the system detects more unique phases, each with a fewer number of intervals. As this occurs, more samples will be taken (since there are more unique phases) which will increase both the percentage of the program's execution that is sampled and the accuracy of the profile.

To measure profile accuracy, we compared each sampled profile to the exhaustive profile. We computed *percentage error in basic block counts* as our accuracy metric. We computed this value as the element-wise difference in basic block counts between a sampled profile and the exhaustive profile. We then divided this value by the total counts in the exhaustive profile to produce percentage error.

In the subsections that follow, we first evaluate the accuracy of different policies for representative selection. We then (Subsection 3.2.2) compare the accuracy of the different profiling approaches: random, periodic, and phase-aware. In subsection 3.2.3, we demonstrate that our approach is not specific to basic block count profiles by presenting empirical data on the accuracy of hot method, hot call-pair, and hot path profiles. We then present data (Subsection 3.2.4) on the overhead of these profiling approaches.

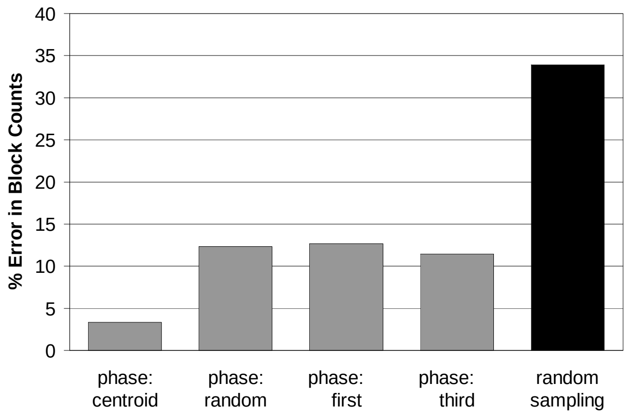


Figure 4. Evaluation of representative selection policies. The graph shows the average error in block counts at 1% sampled for different representative selection schemes. The black bar shows the performance of average random sampling (non-phase-based).

Finally, we describe in subsection 3.2.5 some extensions on how we can further reduce the overhead of phase-aware remote profiling by employing multiple users of the same program.

3.2.1. Selection of Phase Representatives

Figure 4 shows the percentage error at 1% sampled for four different representative selection policies for phase-based profiling. The y axis shows percentage error in basic block counts. The different polices for representative selection that we studied were (a) `first`: select the first interval as the representative, (b) `centroid`: select the centroid of the intervals in the phase as the representative, (c) `third`: select the third interval as the representative. (d) `random`: randomly select one representative from all intervals in the phase (we report performance for this policy as the average performance of 5 selections), For comparison, the graph also includes error produced by the random sampling method described earlier, which chooses random samples from the entire program (black bar on far right).

As expected, the centroid method, `centroid`, performs the best: its error remains low even when we sample very little of the program. `First` and `random` perform the worst. This happens since the first is not representative of the steady state (the phase is just warming up) and because selecting randomly can result in selection of a representative that is dissimilar to all others. `Third` enables accuracy that is between that of `best` and `first`/`random`. That is, `third` is able to select an interval that is more representative of the steady state of the phase than `first` and `random`. Moreover, `third` is simple and can be implemented without additional overhead. As such, we use `third` for the rest of the results in the paper.

3.2.2. Comparison of Sampling Techniques

To evaluate the efficacy of the different sampling techniques that we considered, we calculated error in basic block counts for different percentages of program execution sampled – the higher the percentage, the more accurate the profile should be. The best performing profile technique is the one that produces the least amount of error for the least percentage of the program that is sampled. Figure 5 shows the average error in block counts across benchmarks using the third interval of each phase as the phase representative. The graph compares the accuracy of each of the different sampling techniques, `avg random`, `max random`, `periodic`, and `phase-aware`. The y-axis is error and the x-axis is percent of the program that was sampled for a given parameterization of each technique.

The graph shows that on average, phase-aware profiling results in significantly lower error for a very small percent sampled. The error for periodic sampling approaches that of phase-aware sampling for larger percent sampled values.

However, we can achieve very high accuracy using phases, e.g., less than 5% error, by sampling a very small amount of the program's execution (4%). To achieve the same accuracy, periodic sampling requires that 11% be sampled, average random sampling requires that 20% be sampled, and max random is never able to achieve an error of less than 5%.

All of the results presented up to this point are across all benchmarks and do not show how the performance compares across individuals. Figure 6 shows the impact of the error on each individual benchmark. We consider the performance of each technique when we constrain the error to be less than 1% and less than 5%. In the bottom graph (5%) we can see that under these assumptions the basic block error for most programs is quite low (under 10%). The only exceptions are jpeg-encode and mpeg-encode, both of which have errors in the range of 30%. The phase-aware technique performs far better across the board. The only

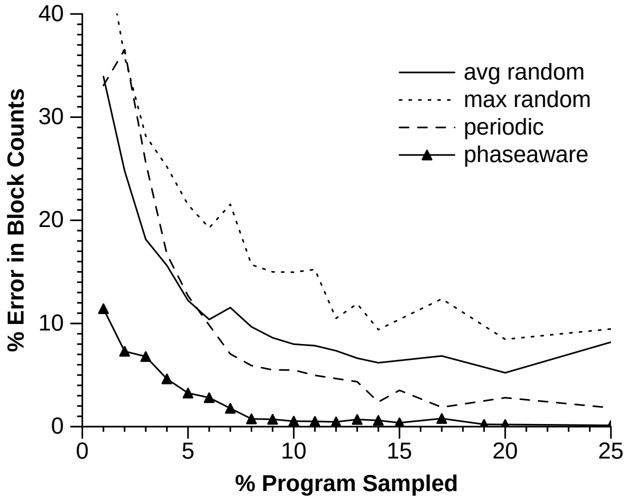
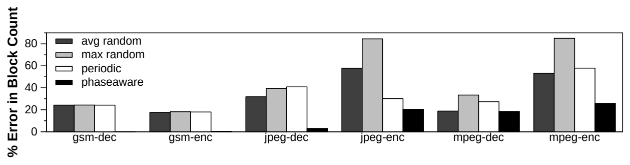
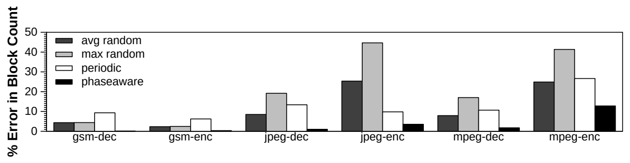


Figure 5. Average error in block counts for various sampling percentages.



(a)



(b)

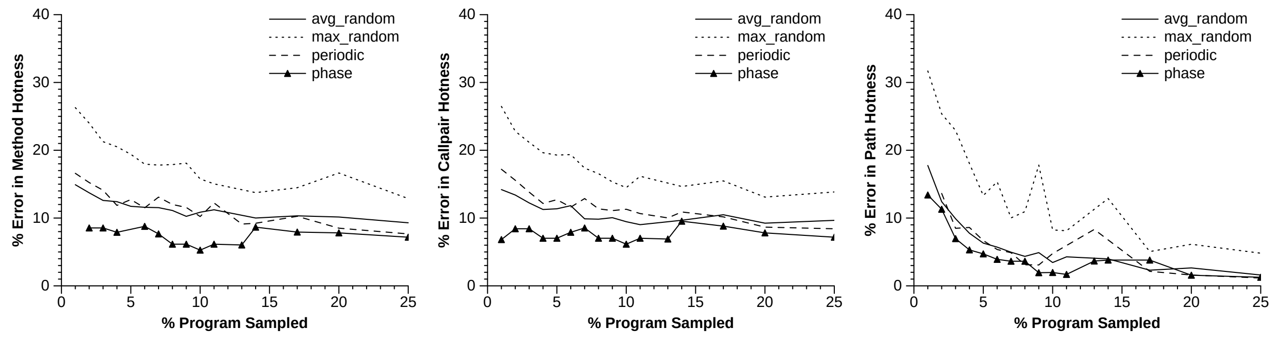
Figure 6. Error rates for the four different techniques across all benchmarks when the profiling overhead is limited to 1% (top graph) or 5% (bottom graph).

benchmark with significant error is mpeg-encode which has an error of 9%.

If we reduce the amount of profiling that be done by another factor of 5 we get the top graph. In the top graph only 1% of the program can be sampled. At these lean profiling rates, all of the programs have error rates at of above 20% for both periodic and random. For some programs, such as mpeg-encode, almost no useful information can be gathered (the error rate is above 50%). Phase-aware sampling can help to minimize this effect across all of the benchmarks.

3.2.3. Efficacy of Phase Profiling Across Different Profile Types

We next demonstrate that phase-aware sampling is not restricted to any one profile type by showing that it performs well for three different types of profiles. We evaluated the efficacy of each of the sampling techniques in iden-



(1) Hot Method Identification (2) Hot Call-pair Identification (3) Hot Path Identification

Figure 7. Efficacy of different sampling techniques for different profile types

tifying frequently executing parts of the program. Profiles that capture frequently executing parts are commonly used for feedback directed optimization, e.g., hot basic blocks, hot methods, hot call-pairs, and hot paths, and as such will be important profile types for our distributed optimization system. We measured the error produced by each of the profiling techniques for these profile types. We report data on all but hot blocks due to space constraints, however the performance trend is very similar.

A hot method profile represents time spent in methods. We generated this profile by counting method invocations and the number of times loop backedges within the method are executed. A hot call-pair profile captures frequency of call edges, indicating the number of times a particular call-site was executed. We generated path profiles using the Ball-Larus approach [4]. We consider a “hot” event to be one in the top 15% of the most frequent events.

We accumulated each profile for each of the sampling techniques into a single profile vector. For basic blocks for example, we gathered basic block frequencies in a single basic block vector for each of the sampling techniques, sorted it in decreasing order. We then identified the top 15% most frequently executed basic blocks (“hot blocks”) in each profile and compared them to the top 15% hottest blocks in the exhaustive profile. We counted the number of blocks that were different from the exhaustive set, normalized the value by the number of blocks in the top 15% and multiplied by 100. The top 15% cut-off is with respect to the exhaustive profile, though the sampled approaches might not capture all events and therefore have fewer total events in the profile. We also adjust the cut-off to include all events at the cut-off point that have the same frequency.

Figure 7 shows the results. The x axis is the percent of the program that was sampled, and the y axis is the percentage error in identifying hot methods, hot call-pairs, and hot paths, averaged across all benchmarks. The graphs show that the phase-aware technique performs considerably better for all three profile types. For hot methods and hot call-pairs identification, the phase-aware technique only needs to sample 1% of the program’s execution for an error of less than 10%. Other techniques have to sample 7 to 20 times more to achieve similar accuracy. The difference is not as pronounced for hot paths, where the phase-aware technique has to sample 4% of the program’s execution for an error of 5%, and the other techniques have to sample 7%. The error at higher values of percent sampled is sometimes more than that at lower values because the relative ordering of events changes, affecting the hotness classification, though the error in terms of absolute counts is not high. For the benchmarks we considered, the average number of hot methods and hot call-pairs were 22, and the average number of hot paths was 50. In summary, phase-aware profiling performs considerably better than the other techniques for all three types of profile indicating that its performance is independent of profile type.

3.2.4. Impact on Power

We next evaluate the impact of phase-based remote profiling on resource performance (Figure 8). In particular, since we are interested in making remote profiling of handhelds feasible, we study the impact on power consumption. We measured the overall power consumption for *all of the required remote profile collection functions*: computation overhead for instrumentation, communication overhead (using compression), and computation overhead for applying compression.

We assume a maximum accuracy error of 5%. To achieve this level of accuracy, phase-aware sampling requires that on average 4% of the program be sampled, periodic sampling requires that 11% be sampled, average random sampling requires that 20% be sampled, and max random is never able to achieve an error of less than 5%. As such, we focus on the former three techniques.

Sampling Overhead For Accuracy Error of 5%				
% Sampled:	**Periodic: 11% AvgRand: 20% Phase: 4%**			
		Energy		
	Joules		Percent Reduction	
Protocol	Periodic	AvgRandom	vs Periodic	vs ARand
At End	7.75	15.02	58.97	78.83
Interleaved	8.24	16.03	58.90	78.86
		Computation Overhead		
	Instructions Executed (Millions)		Percent Reduction	
Protocol	Periodic	AvgRandom	vs Periodic	vs ARand
At End	265.61	514.20	58.84	78.74
Interleaved	280.48	545.46	57.58	78.19
		Communication Overhead (Compressed)		
	Bytes Transfered		Percent Reduction	
Protocol	Periodic	AvgRandom	vs Periodic	vs ARand
At End	2217.83	2217.83	0.00	0.00
Interleaved	27095.50	51683.97	51.36	74.50

Figure 8. Energy consumption of sampling methods given 5% error

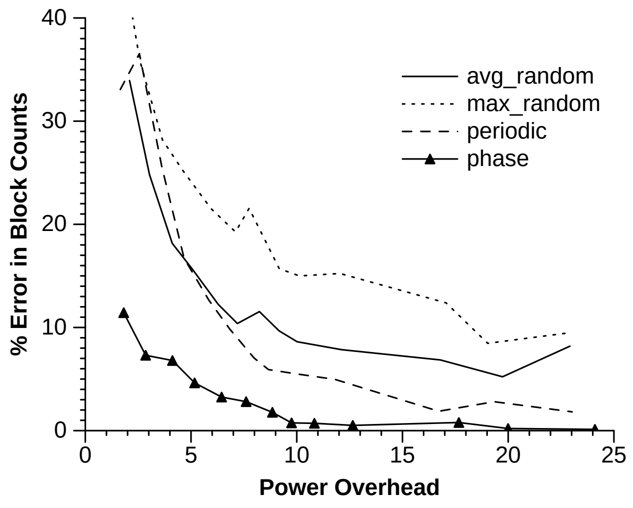


Figure 9. Power (in joules) versus error.

We show how 5% error translates into energy, computation, and communication overhead in the three sections of the table in Figure 8. In each section, we show the average overhead for each metric across benchmarks for periodic and average random sampling in columns 2 and 3. In column 4 and 5, we show the percent reduction enabled by phase-based profiling over each of these techniques, respectively.

Each section in the table contains two rows of data for the two different communication protocols that we studied. For "At End", we combine the basic block vectors of each profiled interval into a single vector; upon program termination, we compress the vector and transmit it. Using this protocol, phase-aware profiling reduces energy consumption by 75% over random sampling. Phase-aware profiling reduces computation overhead by requiring 72% fewer instructions for instrumentation over random sampling.

Given this "At End" approach, the communication cost is the same across profiling techniques since we are communicating a single profile vector in either case (though the counts will be different). However, we investigated another protocol, one in which we compress and transmit the basic block vector after each interval. This protocol reduces the amount of device storage required (which may be highly constrained for real devices); as such, it is a realistic alternative that we should consider. Using this "Interleaved" protocol, phase-based remote profiling can also reduce communication overhead since fewer intervals are communicated to achieve the same 5% accuracy. These results are shown in the second row of each section. The reductions in overhead for energy and computation are similar to the "At End" protocol. However, phase-based profiling requires that less than $1/4$ the number of bytes be transmitted *to communicate the same information* as the random approach.

The graph in Figure 9 summarizes our results using the "At End" protocol. The graph shows the percent energy overhead imposed on the program versus error for each of the competitive remote profiling techniques. The data indicates that by reducing the percentage of the program sampled, we are able to achieve significant power savings over random and periodic profiling. Moreover, we are able achieve these savings while collecting profile information that is very similar to exhaustive profiling.

3.2.5. Extending Phase-Aware Remote Profiling to Multiple Users

Given a large connected user base, we can further reduce the overhead of phase-based remote profiling by providing feedback to users about phase discovery. If a user executes a program using the same input as that used by another user for which phase data already has been collected, the second execution will provide us with no new information, wasting resources needlessly. As part of our phase based remote profiling system, we performed a preliminary investigation into the use of phase IDs (those described in Section 2.3) as a *feedback mechanism to other users* so that they may avoid unnecessary profiling. Such a technique requires that the Phase Tracker and device architecture be homogeneous so that a phase ID identified by the Phase Tracker for a particular program interval on one device is the same as that for the same interval on another device.

To provide *dynamic feedback* to users, we communicate phase IDs periodically, in the reverse direction. The remote profiling system on the user's device adds these phase IDs to the phase ID table in the Phase Tracker (if they are not already present) and sets their "sampled bit". As such, when

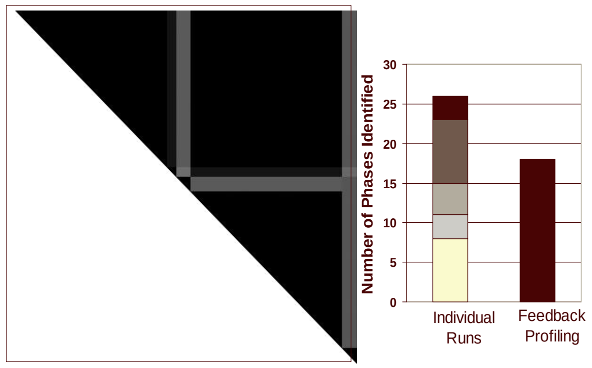


Figure 10. Similarity matrix (left) for the gsmencode MediaBench benchmark across 5 different executions using different inputs. The right graph shows the number of phases identified if we profile this benchmark with each input separately (left bar broken down by input). The right bar graph shows the number of *unique* phases. Only unique phases need to be sampled using our feedback-directed phase profiling technique.

a previously unseen phase is predicted, it is *only* sampled if it also has not been provided by the feedback mechanism.

By communicating phase IDs of known phases to users (so that their systems avoid sampling them), we can reduce the overhead of phase-aware sampling when users execute the same program with the same inputs. However, it may also be possible to use this technique when the same phase occurs *across inputs*. A more realistic scenario is one in which software is deployed and users execute it with a wide range of diverse inputs. Each input will cause the program to exhibit phased behavior; for some programs, some phases may be the same across inputs.

To evaluate the potential of feedback-directed remote phase profiling, we analyzed many different inputs for one of our benchmarks, `gsmdecode`. Figure 10 shows the similarity matrix for this benchmark. A similarity matrix is a 2-dimensional array all of the intervals in a program; each entry is the similarity value between two intervals encoded as a grayscale value with dark values identifying similar intervals (in the same phase), e.g., the points on the diagonal are black since an interval is exactly the same as itself. The x-axis and y-axis of figure are increasing interval id's. We omit data in the lower triangle for clarity, since it is symmetric with the upper triangle. We read the figure by first selecting an interval on the the diagonal and then traversing the row. By doing so, we can visualize how similar the row interval is compared to all others that follow it during execution. By traversing the column above, we can visualize how similar the row interval is compared to all other intervals that came before it during execution.

Commonly, similarity matrices are used to analyze the execution of a benchmark running a single input [19, 25]. However, we use it here to visualize execution of *five* different inputs. We concatenate the intervals from each input and then compute the similarity between each interval in the entire set. The dark regions indicate that even across inputs there are many intervals that are very similar, i.e., there are phases that span inputs.

The graph on the right in the figure shows the number of total intervals in all five executions of `gsmdecode`. The total height of the left bar is 28, indicating the number of different phases identified if we were to execute the program with each input individually. The left bar is broken up into pieces, indicating the number of phases found for each input. The right bar shows the number of intervals that we must sample, across all five inputs, to gather all of the *unique* phase behavior: 18 phases.

The data shows that there are 10 phases (36%) that overlap across all of the inputs. This indicates that there is potential for reducing the overhead of phase-driven remote profiling further using feedback-directed profile collection. For this benchmark, we can communicate the phase IDs to the user base as each is discovered by individual users. For programs that execute a phase that has already been identified, we can avoid collection and communication of the profile.

4. Related Work

Our work builds upon and extends a body of related research on program phase behavior [23, 24, 25, 11, 14, 12, 19, 15]. Our work is novel in that it is the first, to our knowledge, to investigate the efficacy of remote performance profiling. Moreover, we use program phase behavior to significantly improve the efficiency of remote profiling and as such, we make it feasible to gather performance characteristics about software for mobile devices post-deployment.

The other area of research that is related to our work is that of sample-based profiling techniques. Many other researchers have identified that an entire program need not be profiled to extract accurate execution behavior information from it. Instead, many sample-based approaches have been proposed [13, 27, 3, 2, 8]. Sample-based profiling is used to gather performance statistics about a program for use on the same device (as opposed to remotely), in compiler and runtime optimization.

Extant sample-based profiling techniques that couple hardware support for performance profiling include those that employ hardware performance counters [1, 10], and others that use special-purpose hardware to guide sampling [28, 22]. The work in [22] is somewhat related to the research herein in that it describes a performance profiling approach that couples hardware and software in an attempt to reduce profiling overhead by using programmable hardware to capture and compress profile information before passing it on to software for analysis and exploitation. The generated profile is in the form of a single or multiple event streams. Dedicated hardware performs lossy compression on this stream, by using hardware-based low-cost sampling mechanisms, thereby reducing the amount of information that the software profiler has to process. This approach is completely orthogonal to ours, in that, it uses specialized hardware to capture and pre-process profile information as dictated by the software profiler. We are interested in using specialized hardware to drive our profiling policies.

There are many sample-based, software-only performance profiling techniques, e.g., [13, 27, 3, 2, 8]. These approaches are intended to be used within extant dynamic optimization systems. Duesterwald et al [13] present online path profiling to enable hot path prediction in dynamic optimization systems, and [27] examines several sampled based techniques to gather profiles within a Java Virtual Machine to enable feedback-directed dynamic optimization. In [3], the authors present an online, software only mechanism for sampling executing code. They use code duplication (methods both with and without instrumentation) and transfer execution between the two based on method invocation and taken backward branch (backedges) counts. They show that for sampling method call-pair frequencies and field accesses that their technique exhibits very low overhead. In our work, we show that phase-aware profiling can capture a wide range of profile types, e.g., basic block frequencies, hot blocks, hot methods, hot call-pairs, and hot paths, with high accuracy. As part of future work, we intend to compare our approach to extant sampling techniques. Since such techniques have no concept of interval, we therefore must first identify a meaningful way to empirically perform the comparison.

In addition, sample-, and instrumentation based program monitoring is used to collect code coverage information [26, 20, 6] and to identify errors in deployed programs [18]. The focus of our work is on performance profiling; however, we believe that phase-awareness can be used to gather code coverage information and to aid in bug isolation. We plan to investigate such uses as part of future work.

5. Conclusions and Future Work

In this paper, we couple hardware and software techniques to enable efficient collection of remote profiles from resource-restricted devices. The key to our approach is the exploitation of program phase behavior. We show that by using special phase tracking hardware to guide sample-based profiling, we generate highly accurate profiles with low overhead. Moreover, we demonstrate the generality of phase-aware profiling by evaluating it for three different types of profiles: hot methods, hot call-pairs, and hot paths.

We also investigate the identification of phase representatives – an execution interval that most accurately reflects the behavior of the phase. Of the online methods we investigate, we find that the third interval is the best representative (as opposed to the first interval and random selection). Our simulation results indicate that phase based profiling enables a 50-75% reduction in overhead (communication, computation, and battery power) over periodic and random sampling.

Phase-based remote profiling requires that users be willing to allow transparent sampling of the execution of their software. Though there are many security and privacy concerns for such a system, we believe that users will be incentivized to participate since doing so will enable software vendors to automatically improve performance, fix bugs, and upgrade software transparently. As part of future work, we plan to investigate novel techniques that ensure that the information transmitted is obfuscated.

Also as part of future work, we are investigating the impact of PhaseTracker prediction inaccuracy, the impact of and solutions for OS and multi-process interference on the PhaseTracker, the efficacy and efficient implementation of phase ID feedback, and techniques for turning on and off profiling efficiently in resource-constrained devices. We plan to employ each of these mechanisms in the complete implementation of our phase-aware distributed optimization system.

References

[1] J. Anderson, W. Weihl, L. Berc, J. Dean, S. Ghemawat, M. Henziger, S. Leung, R. Sites, M. Vandevoorde, and C. Waldspurger. Continuous Profiling: Where Have All the Cycles Gone? *ACM Transactions on Computer Systems (TOCS)*, 15(4):357–390, 1997.

[2] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive optimization in the jalapeño jvm. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Oct. 2000.

[3] M. Arnold and B. Ryder. A Framework for Reducing the Cost of Instrumented Code. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2001.

[4] T. Ball and J. Larus. Efficient path profiling. In *29th International Symposium on Microarchitecture*, Dec. 1996.

[5] BitRaker INC. Anvil. `http://www.bitraker.com/Anvil.htm`.

[6] J. Bowring, A. Orso, and M. Harrold. Monitoring Deployed Software Using Software Tomography. In *Proceedings of ACM SIGPLAN-SIGSOFT Worshop on Program Analysis for Software Tools and Engineering*, pages 2–9, 2002.

[7] T. Chilimbi and M. Hauswirth. Low-overhead memory leak detection using adaptive statistical profiling. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 2004.

[8] M. Cierniak, G. Lueh, and J. Stichnoth. Practicing JUDO: Java Under Dynamic Optimizations. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 13–26, June 2000.

[9] M. Corliss, E. C. Lewis, and A. Roth. DISE: a programmable macro engine for customizing applications. In *30th Annual International Symposium on Computer Architecture*, May 2003.

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

[11] A. Dhodapkar and J. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *29th Annual International Symposium on Computer Architecture*, May 2002.

[12] A. Dhodapkar and J. Smith. Comparing program phase detection techniques. In *36th Annual International Symposium on Microarchitecture*, Dec. 2003.

[13] E. Duesterwald and V. Bala. Software Profiling for Hot Path Prediction: Less is More. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, Oct. 2000.

[14] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *International Conference on Parallel Architecture and Compilation Techniques*, Sept. 2003.

[15] M. Hind, V. Rajan, and P. Sweeney. The Phase Shift Detection Problem is Non-monotonic. Technical Report RC23058, IBM, 2003.

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

[17] C. Lee, M. Potkonjak, and W. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture (Micro-30)*, pages 330–335, 1997.

[18] B. Liblit, A. Aiken, A. Zheng, and M. Jordan. Bug Isolation via Remote Program Sampling. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2003.

[19] P. Nagpurkar and C. Krintz. Visualization and Analysis of Phased Behavior in Java Programs. In *ACM International Conference on the Principles and Practice of Programming in Java*, June 2004.

[20] A. Orso, D. Liang, M. Harrold, and R. Lipton. GAMMA System: Continous Evolution for Software After Deployment. In *Proceedings of International Symposium on Software Testing and Analysis*, pages 65–69, 2002.

[21] M. X. E. Reporting. `http://support.microsoft.com/default.aspx?scid=kb;en-us;310414`.

[22] S. Sastry, R. Bodík, and J. Smith. Rapid Profiling via Stratified Sampling. In *Annual International Symposium on Computer Architecture*, pages 278–289, July 2001.

[23] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2001.

[24] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *10th International Conference on Architectural Support for Programming*, Oct. 2002.

[25] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *30th Annual International Symposium on Computer Architecture*, June 2003.

[26] M. Tikir and J. Hollingsworth. Efficient Instrumentation for Code Coverage Testing. In *International Symposium on Software Testing and Analysis*, 2002.

[27] J. Whaley. A Portable Sampling-based Profiler for Java Virtual Machines. In *Proceedings of ACM JavaGrande Conference*, pages 78–87, 2000.

[28] C. Zilles and G. Sohi. A programmable co-processor for profiling. In *HPCA*, pages 241–, 2001.