Ephemeral Instrumentation for Lightweight Program Profiling

Omri Traub, Stuart Schechter, and Michael D. Smith
Harvard University
{otraub,stuart,smith}@eecs.harvard.edu

Abstract

Program profiling is a mechanism that is useful for performance evaluation and code optimization. Profiling techniques that provide detailed information with extremely low overhead are especially important for systems that continuously monitor or dynamically optimize running executables. In this paper, we describe an approach for program profiling called *ephemeral instrumentation* and show that it collects useful profiles with low overhead. This approach builds on ideas from both program instrumentation and statistical sampling; it produces binaries that are able to periodically record aspects of their executions in great detail. It works because program behavior is predictable and because we are able to convert ephemeral profiles into traditional formats. This paper describes an ephemeral instrumentation system for gathering branch biases and post-processing that data into a traditional edge profile. We evaluate the usefulness of such profiles by using them to drive a superblock scheduler. Our experimental results show that we can gather ephemeral profiles with extremely low overheads (1-5%) while acquiring profile data that rivals the usefulness of complete profiles gathered at much higher overheads.

1 Introduction

Program profiles are important tools for understanding the dynamic behavior of programs. To the programmer, profiles provide insight into a program's resource utilization and help to identify performance bottlenecks. To the compiler writer, profiles are used to guide code optimization. There is a growing demand for flexible profiling systems that are capable of gathering increasingly detailed amounts of profile data with extremely low overheads. To this end, we propose a new approach to profiling, called *ephemeral instrumentation*, that satisfies this wide range of demands.

Programmers have always found the traditional way of gathering profile information rather cumbersome. The programmer first builds an instrumented copy of the program, then executes this copy on one or more representative program inputs, and finally re-optimizes the entire application using the profile data. Systems of this kind collect profile information for every execution of an instrumented program point or structure, which we call *complete profiles*. A program instrumented for complete profiling often suffers a significant slowdown due to the overhead of profile gathering. Even efficient systems that require post-processing of the profile data like QPT [2] and PP [3] incur significant run-time overheads (16.3–52.8% [3]).

A profiling system that is transparent to the user and collects profile data with minimal overhead would be a significant improvement. This line of thought helped to spur the development of profiling systems like DCPI [1] and Morph [17], which achieve extremely low overheads through the use of statistical sampling, and the recent investigations into hardware-based approaches to profile gathering [6,10]. Though the overhead in these approaches is nearly unnoticeable (Anderson et al. [1] report overhead of 1–3% and Conte et al. [6] report an overhead of 0.4–4.6%), researchers are still investigating the usefulness of statistical and hardware-generated profiles in code optimization.

This paper introduces a new technique for program profiling called *ephemeral instrumentation*. The goal of this software-based instrumentation technique is to produce profile information that is as useful and detailed as that obtained from complete profiling while incurring an overhead that is comparable to that of sampling- and hardware-based techniques. To achieve this goal, ephemeral instrumentation uses a mix of ideas from complete profiling and statistical sampling. We use the term “ephemeral” in describing our approach because the instrumentation associated with a program point or structure will “come and go” as the profiled program executes.

We focus our discussion on an implementation of ephemeral instrumentation for obtaining edge profiles, although our technique can be extended to gather more complex kinds of profiles. As with existing lightweight instrumentation techniques, our approach gathers a limited amount of profile data (branch[1] biases in particular) and then invokes a post-processing step in order to produce a traditional edge profile.

We can profile branches in an ephemeral manner because many studies have shown that the majority of branches in programs exhibit very predicable behavior, with biases that tend to be highly skewed towards one direction or another (e.g., see Young et al. [16]). Thus, it is not necessary to record every execution of a branch in order to accurately predict that branch’s bias. Rather, we simply want, for each static branch site, to statistically sample the bias of that branch over the program run. Note that it is not possible to sample a branch’s bias directly using current statistical sampling systems like DCPI or Morph.

We sample a branch’s bias by periodically inserting instrumentation code to capture a small and fixed number of the branch’s executions. This implies that we affect the completeness of the pro-

1. We use the generic term “branch” to refer to any conditional transfer of control; the use of the term subsumes both conditional branches and multiway branches. We use the more precise terms only where the context demands us to differentiate between conditional and multiway branches. If the transfer of control is unconditional, we use the term “unconditional jump.”

file data and the overhead of the instrumentation process by varying two parameters: one that controls the *periodicity* of instrumentation insertion and the other that defines the *persistence* of the instrumentation. We refer to the time between instrumentation insertions as an *epoch*. For example, increasing the values of the periodicity and persistence parameters would provide more detailed profiles at the cost of higher run-time overheads. As our results show, however, it is possible to obtain quite useful profiling information by infrequently recording branch behavior over very short time intervals. We evaluate the usefulness of the profiling information gathered by our system in guiding a superblock scheduler [9].

The rest of this paper is organized as follows: Section 2 describes the implementation of ephemeral instrumentation for edge profiling. It also briefly describes how we could use ephemeral instrumentation to gather a kind of path profile. Section 3 discusses a post-processing algorithm that is needed in order to transform the profile data gathered by the instrumentation system into an edge profile that a profile-driven optimizer could effectively use. This section also shows that branch biases are sufficient to construct an edge profile that is a normalized version of the profile gathered under complete profiling. Section 4 describes our experimental methodology and presents our results. Section 5 discusses related work, while Section 6 offers some conclusions.

2 Ephemeral instrumentation

This section describes an implementation of ephemeral instrumentation for gathering branch biases. We begin in Section 2.1 by describing how we instrument a conditional branch to gather bias information and cause the instrumentation to "unhook" itself dynamically after a constant number of executions; instrumentation of multiway branches is a simple extension of this technique. Section 2.2 then motivates the need for the dynamic insertion of instrumentation code and describes how we achieve this effect. Section 2.3 presents a few of the important implementation details that affect the results presented in Section 4. Finally, Section 2.4 explains why we believe that other kinds of profiles, such as path profiles [3,13], can also be gathered ephemerally.

2.1 Instrumenting conditional branches

It is expensive to change the layout of executables at run time. Since we wish to insert and remove instrumentation code dynamically, we want our instrumentation system to use techniques that do not change the code layout. For example, it is not easy to separate two consecutive basic blocks and insert some instrumentation code in between. We therefore use a technique, similar to the one employed by the creators of PatchWrx [5], that requires us to modify only single instructions in the original executable.

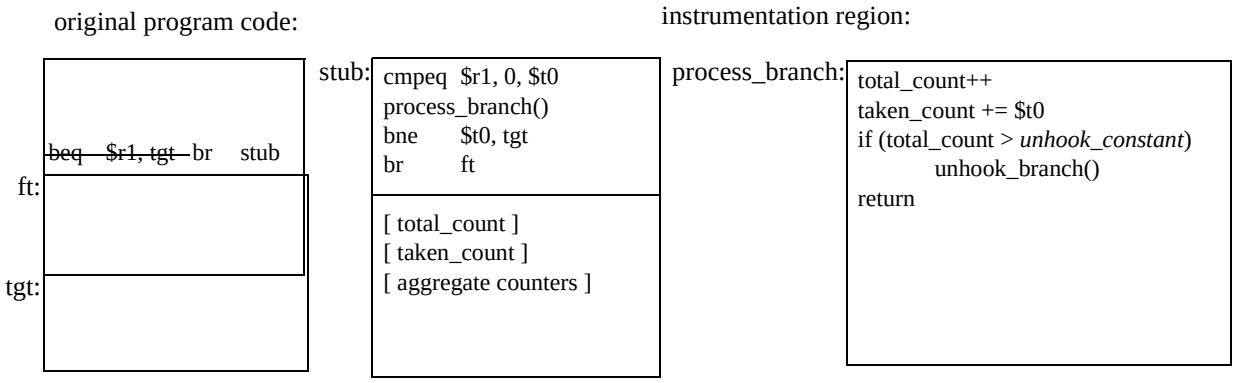

Figure 1. Hooking a conditional branch. The original conditional branch is overwritten by an unconditional branch to a stub customized for this instrumentation point. The 8-word, customized stub records the outcome of the branch and then calls a general subroutine to update the instrumentation state. Note that each branch-specific stub contains both instructions and data counters.

To instrument a conditional branch, we overwrite it with an unconditional branch[2], the target of which is an instrumentation stub containing instructions and data specific to the conditional branch. Figure 1 illustrates this process for an integer conditional branch; the procedures for floating-point and multiway (indirect) branch instrumentations are similar. The branch-specific stub computes the branch direction based on the condition and stores the result in a temporary register. The stub then calls a routine shared by all instrumentation points of that type. The common routine updates the counters in the stub to reflect the current branch execution, and then typically returns to the stub where control is directed back into the executable at the appropriate target. If however we have recorded a number of executions of this branch equal to the persistence parameter, the common routine will first “unhook” this branch’s instrumentation by overwriting the patch point in the program code with the original conditional branch. The unhooking code also executes instructions that maintain instruction cache consistency.

Since our instrumentation routine only calls the unhook code once per epoch, it is extremely lightweight on average. In fact, the common path through both the stub and the common routine requires a mere nine to ten instruction executions. Since the only hard-to-predict conditional branch in the common case is the stub branch that simulates the original branch instruction, the instrumentation will not overly burden the instruction fetch unit. Further, if an instrumentation point is not reached during an epoch, no profiling, unhook, or rehook code will be executed.for that point.

2. Given the experimental platform and benchmarks in this study, we can always reach the instrumentation code with a simple unconditional branch. A general solution to the problem of limited branching distances is beyond the scope of this paper.

2.2 Inserting instrumentation dynamically

The previous section described how instrumentation code can be dynamically removed. In this section, we describe how and why instrumentation code is dynamically inserted.

The execution of an application begins with all branches instrumented. Currently, our system instruments each branch at program load time. In the future, our system will instrument branches on-demand, catching page faults and executing the stub-creation routine for the branches on those pages.

For a highly- and consistently-biased program branch, it works well to instrument the branch only for its initial executions (up to the persistence value, after which instrumentation is unhooked). All but the most trivial of programs however have a number of dynamically-important branches that are data-driven and exhibit phased behavior. For these branches, their initial biases may not match their long-term biases.

As an example, consider the dominant loop in the SPECint benchmark *compress*, which manages a hash-table. The code starts by accessing the hash table and checking for a match. If the match fails, the code checks to see if the entry was empty, and if it was, inserts the element in that location. At the beginning of the program execution, the hash table is sparsely populated and the lookup usually finds an empty slot. As the execution progresses however, the hash table fills up. In general, the conditional branch that checks for an empty slot succeeds early in the program and fails during the rest of the program. Over the entire program run, this branch fails more than it succeeds. If we only record this branch's behavior for a short time early in the program's execution, the instrumentation system would require a large persistence threshold to capture the correct long-term bias.

In order to capture long-term behavior with a low overhead, we employ a sampling approach. At the end of every epoch, the instrumented program is interrupted, and all of the branches unhooked in the previous epoch are re-hooked.[3] We achieve this by having the unhooking routine perform two more tasks in addition to restoring the original instruction to the patch point. We first aggregate the epoch-specific counters into larger counters, also maintained at the branch-specific stub. We then add the current branch address to a global list, which is examined at the start of the next epoch to determine which branches need to be rehooked. By using this sampling technique and a

3. Note to the reviewers: For the current results, we use a fixed epoch to set the interrupt timer. We are aware that it is a good idea to randomize the length of the sampling period to minimize correlation between the application and the profiling system. We plan to implement this feature before gathering the final results.

small unhook constant, we can spread branch profiling over the entire program run, thus capturing the aggregate bias of each branch in a lightweight manner.

2.3 Implementation details

[Note to reviewers: In the final version of the paper, we will include a detailed discussion of our implementation for our Alpha 21164 machine. We will describe the following issues: how we find registers for our instrumentation routines; how and where we allocate memory for the instrumentation stubs and common routines; and how we maintain cache consistency and deal with multi-threaded applications.]

2.4 Gathering other kinds of profiles

The key to ephemeral instrumentation is to identify the set of statistics and instrumentation points that, for consistency reasons, want to be hooked and unhooked together. In the case of collecting branch biases, we created a single instrumentation point for each branch site. Since the work done by an instrumentation point in this scheme is independent of the other instrumentation points, unhooking dealt with only that instrumentation point.

Path profiling as described by Ball and Larus [3] employs two kinds of instrumentation points: a lightweight instrumentation point that simply updates a path identifier in a dedicated register, and a heavyweight instrumentation point that increments the execution count for the current path identifier. As above, the counters that we need to compare to generate a “path bias” are all updated by a single instrumentation point. To understand how unhooking would work, let us assume that, for simplicity of explanation, no profiled path is a sub-path of another profiled path. Under this assumption, we can associate each lightweight instrumentation point with a single heavyweight instrumentation point. We can easily cause the unhooking of a heavyweight instrumentation point to unhook its associated lightweight instrumentation points as well. We are investigating this as our next application of ephemeral instrumentation.

3 Processing ephemeral profiles

Our ephemeral instrumentation technique gathers data that we can use to determine branch biases. However, superblock scheduling [6] requires a weighted CFG, where each CFG node/edge is annotated with its execution frequency. As an example, Figure 2 contains a CFG first annotated with branch biases and then with a set of edge and node weights that yield the same branch biases. This section focuses on the reasons why the left-hand labeling in Figure 2 is insufficient for superblock scheduling, and it explains how we can obtain the right-hand labeling from the left-hand

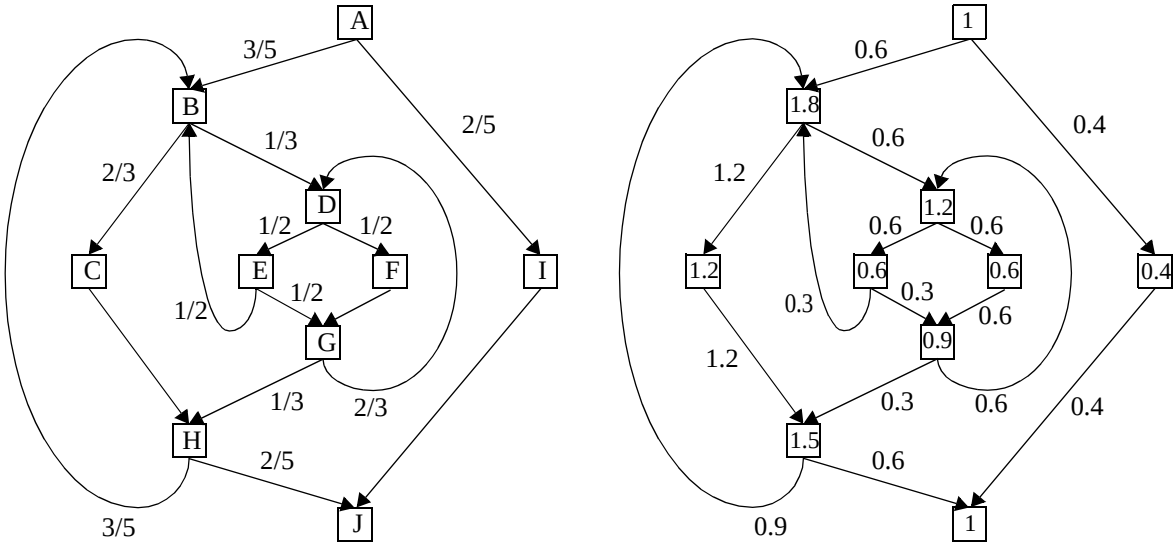

Figure 2. Example CFG from the compress() procedure in the SPECint benchmark compress. *The CFG on the left is annotated with branch biases that were gathered during ephemeral instrumentation. The CFG on the right is annotated with weights that were obtained by our post-processing algorithm.*

labeling. The final version of the paper will present the details of our algorithm for this transformation.

In a complete profiling scheme, the weights on a weighted CFG correspond to the exact execution frequencies of the nodes and edges in the CFG. Since these weights are exact, an optimizer can directly compare the magnitudes of two weights to determine which CFG element was executed more frequently. Referring to the example CFG in Figure 2, if the block *D* has a higher weight than block *B*, then an optimizer would know that *D* executed more frequently than *B*. For superblock scheduling in particular, the algorithm compares the execution frequencies of blocks when choosing a seed block for a superblock, and the execution frequencies of edges when deciding whether to add a block to a superblock. If these frequencies are not comparable, the superblock scheduler will create superblocks that do not correspond to the frequently executed program paths.

A superblock scheduler cannot directly use the data gathered by the ephemeral instrumentation system in Section 2 for two reasons. First, ephemeral instrumentation, like other low-overhead instrumentation techniques, does not instrument every node and edge in a CFG. Post-processing based on conservation of flow is therefore required to transform a profile containing weights for just some of the CFG edges/nodes into a *fully-weighted* CFG (one with weights on every edge and node). Second, ephemeral instrumentation greatly restricts the kinds of comparisons that one can make with the raw profile data. Unhooking an instrumentation point is equivalent to the use of a

saturating counter at that instrumentation point. Obviously, if the counters for two unique branches have saturated, we cannot compare the magnitudes of these counters to determine which branch executed more frequently. The only counter values that we can compare are those that were “hooked” and “unhooked” at the same time. We can, for example, compare the magnitude of a conditional branch’s not-taken counter to the magnitude of its execution counter to determine the probability that it is not taken. We cannot, however, meaningfully compare the magnitude of its not-taken counter to the magnitude of any other branch’s not-taken counter. Looking at the example on the left-hand side of Figure 2, it is not immediately obvious whether block *D* executes more frequently than *B*.

We observe, however, that the problem of obtaining a weighted CFG from a CFG annotated with branch biases is equivalent to the problem of finding the limiting probabilities on an irreducible, finite-state Markov chain [11]. We have formulated the solution in a post-processing algorithm that fits neatly into most compilers. Space limitations prohibit us from presenting this formulation in detail, and so we simply discuss the equivalence of the two problems here.

If we add a directed edge from a CFG’s exit node to its entry node (from node *J* to node *A* for the example CFG in Figure 2) and consider all unannotated edges as having probability 1 and missing edges has having probability 0, we can interpret the CFG annotated with branch biases as an irreducible, finite-state Markov chain. The theorem of limiting probabilities says that there exists a unique, non-negative solution which associates a probability π_j with each state *j* in the chain. A state *j* has a probability π_j if the proportion of time that the stochastic process represented by the Markov chain spends in the long run in state *j* is equal to π_j. For state *j*, this proportion of time π_j is within a constant scaling factor *s* of the execution count of the corresponding CFG block *j*, for all *j* between 0 and the number of nodes in the CFG minus one. The execution count of the CFG edges follows directly by knowing the execution count of each CFG node and the exit probability for each edge emanating from the node. The computation of limiting probabilities is equivalent to solving a system of *n* equations in *n* unknowns, where *n* is the number of nodes in the CFG. We could solve this simultaneous system of equations using back substitution, but we solve it with a post-processing algorithm that takes advantage of the common properties of CFGs.

4 Results

This section compares scheduling results using ephemeral profiles against those obtained from complete profiling. Section 4.1 briefly overviews our experimental methodology, while Section 4.2 presents the experimental results.

4.1 Experimental methodology

We are interested in two questions: how costly is it to gather profile information using a particular system, and how useful is the gathered profile information for profile-driven optimization? To answer the first question, we measure the run-time overhead of our ephemeral system as a ratio of the time it takes to execute a profiled version of a benchmark program divided by the time it takes to execute the unprofiled version of that program. We measure time using the UNIX *gettimeofday* command and take the lowest of 5 runs. We encourage the reader to consult Ball and Larus [2] and Anderson et al. [1] for the best overhead numbers available for the other profiling systems.

To answer the second question, we measure the performance improvement of an application optimized with a superblock scheduler driven by profile information. The scheduler, implemented by Young [13], runs under the Machine-SUIF compiler. The scheduler expects a profile consisting of execution counts for the CFG edges emanating from the branches in the CFG. We use compiled simulation to compute cycle counts and to validate the output of the scheduling process. We use a test data set that differs from the training data set when measuring performance improvements. The cycle counts of the executions resulting from the optimization are normalized against a baseline obtained by doing only local scheduling. We will provide more details in the final paper.

We gather *complete* profiles using the HALT tool [14]. We gather *ephemeral* profiles using the system and the post-processing algorithm described in Sections 2 and 3. For each profiling system, we run that system's edge profile through a simple filter that generates the edge-profile format expected by our superblock scheduler.

Table 1 lists the benchmarks used in this study. The final paper will provide results for the entire SPECint suite. The results we report are representative of the range of results for the entire SPECint suite. For example, *compress* is a difficult benchmark for a superblock scheduler because of the significant number of weakly-biased branches in this benchmark. Vortex provides us with insight into how the technique works for large benchmarks with many important branches.

Benchmark	Description	Data set	
		Training	Testing
wc	UNIX word count program	compress92 ref input	PostScript for a conference paper
compress	Lempel/Ziv file compression utility	compress92 ref input; source code	MPEG movie data (6MB)
eqntott	Translates boolean eqns to truth tables	fixed to floating point encoder	priority encoder, SPEC92 ref input
espresso	Boolean function minimizer	ti, part of SPEC92 ref inputs	tial, part of SPEC92 ref inputs
vortex	Object-oriented database	SPEC95 training input	SPEC95 test input (longer)

Table 1: Benchmarks and data sets used in this study.

4.2 Experimental results

In this section, we present one table of results for each benchmark studied. Each table contains the results for ephemeral and complete profiling. For each approach, we report a normalized cycle count. This is the ratio of the cycle counts of the benchmark scheduled using the given profiling information to the cycle counts of the benchmark using local scheduling. For ephemeral instrumentation, we fix the epoch length at 10,000 micro-seconds. We then vary the persistence parameter from 10 to 10,000. Tables 2–6 present our results.

Complete	Ephemeral		
Normalized Cycle Count	Persistence Parameter	Normalized Cycle Count	Overhead
72.3%	10	72.3%	3.9%
	100	72.3%	4.0%
	1000	72.3%	3.8%
	10000	72.3%	6.6%

Table 2: Results for wc.

Complete	Ephemeral		
Normalized Cycle Count	Persistence Parameter	Normalized Cycle Count	Overhead
70.8%	10	78.9%	0.7%
	100	78.9%	0.9%
	1000	78.9%	0.9%
	10000	78.9%	3.9%

Table 3: Results for compress.

Complete	Ephemeral		
Normalized Cycle Count	Persistence Parameter	Normalized Cycle Count	Overhead
61.1%	10	59.0%	0.0%
	100		0.6%
	1000		0.5%
	10000	60.3%	3.2%

Table 4: Results for eqntott.

We see that for *wc*, *eqntott*, and *vortex*, the ephemeral system quickly converges on the performance gain obtained under complete profiling. This is expected for *wc* and *eqntott*, since the majority branches in these benchmarks are highly biased.

As Table 3 shows, the ephemeral system does not quite converge on the full gain from complete profiling on *compress*. We found that the scheduler identified the same superblocks under each

Complete	Ephemeral		
Normalized Cycle Count	Persistence Parameter	Normalized Cycle Count	Overhead
72.2%	10		3.4%
	100		4.0%
	1000		8.0%
	10000		23.4%

Table 5: Results for espresso.

Complete	Ephemeral		
Normalized Cycle Count	Persistence Parameter	Normalized Cycle Count	Overhead
68.1%	10	68.1%	0.0%
	100	68.2%	4.9%
	1000	68.1%	22.6%
	10000	68.2%	58.3%

Table 6: Results for vortex.

profiling approach; the difference is due to the heuristic that decided how much to unroll and peel the superblock loops. It so happened that, for this benchmark, the edge counts produced by complete profiling made for better unrolling and peeling decisions than the equivalent counts from ephemeral instrumentation.

On the other hand, the results for *eqntott* under ephemeral instrumentation actually surpass the gains obtained from complete profiling. Because the cycle counts are computed using basic-block frequencies and instruction counts that result from optimization decisions based on the input profile data, a difference in cycle counts truly reflects a difference in the usefulness of the profile data. As Young and Smith [15] discuss in the context of path-based superblock scheduling, the use of detailed profile information can lead to executables that have been over optimized with respect to a particular input set. With respect to the ideas in this paper, the cycle-count results for *eqntott* challenge the implicit assumption that complete profiles are the "correct" answer. Overall, the scheduling results for *compress* and *eqntott* demonstrate the sensitivity of unroll and peel heuristics to the profile information.

Most of the profiling overheads reported in Tables 2–6 are competitive with those under statistical sampling, especially when considering persistence parameters of 100 or less. The overheads become much larger, especially at the larger persistence parameters, for *espresso* and *vortex*. These are our two larger applications, and as we increase the persistence parameter, it becomes less likely that a branch will be executed frequently enough in an epoch to become unhooked. Since the work done at one of our instrumentation points is greater than the work done at an

instrumentation point in an instrumentation scheme like QPT [2], we see that the overheads will grow more quickly in the larger, more complex applications.

Overall, these results show that it is possible to obtain the optimization gains of complete profiling with a lightweight, software-only approach and that the overhead of ephemeral instrumentation is often as low as that found in sampling-based approaches.

5 Related Work

Instrumentation-based techniques like QPT [2] are relatively mature, while the systems employing sampling- and hardware-based approaches are only beginning to emerge as viable alternatives. Furthermore, researchers are just beginning to quantify the usefulness of sampling- and hardware-based profiles in optimization [6,17] and discuss the limitations of these approaches. ProfileMe [7] is an example of how one group of researchers believes that sample-based profiling systems should evolve to support program monitoring on out-of-order issue machines.

Paradyn [8] and KernInst [12] are two systems for dynamic instrumentation. Paradyn is based on a metrics description language that lets a user cleanly insert instrumentation points before or after procedure entries, exits, and call sites. It was not however developed to be a lightweight instrumentation tool. KernInst lets a user dynamically instrument a running kernel by replacing individual kernel instructions with unconditional jumps to patch code, much like our system. The primary motivation for both of these systems is to assist a programmer in performance debugging; no attempt was made to use the data collected from either instrumentation system in profile-driven optimization. Though both of these systems allow for the removal of instrumentation code, we argue that the removal of instrumentation can be actively used in obtaining low overhead and show how we can still benefit from incomplete information in optimization.

Finally, Calder et al. [4] describe a system that predicated the instrumentation calls with a convergence check. They used this approach to reduce the time required to gather value profiles.

6 Conclusions

Ephemeral instrumentation works, efficiently producing profiles that are useful for optimization. We describe a system for obtaining edge profiles from branch biases gathered by ephemeral instrumentation. A post-processing algorithm turns branch bias information into a fully-weighted CFG. The system’s run-time overhead is comparable to that of the best published sampling- and hardware-based schemes. We also demonstrate the usefulness of ephemeral profiles and meaningfully compare them with complete profiles. In particular, we drive a superblock scheduler with

each kind of profile and find that the speedups using ephemeral profiles are competitive with those obtained using complete profiles.

7 References

[1] J. Anderson, et al., "Continuous Profiling: Where Have All the Cycles Gone?," *Proc. of 16th ACM Symposium on Operating Systems Principles (SOSP)*, Saint-Malo, France, October 1997.

[2] T. Ball, and J. Larus, "Optimally Profiling and Tracing Programs," *ACM TOPLAS*, 16(4):1319-1360, July 1994.

[3] T. Ball, and J. Larus, "Efficient Path Profiling," *Proc. of the 29th Annual International Symposium on Microarchitecture*, Paris, France, pp. 46-57, 1996.

[4] B. Calder, P. Feller, and A. Eustace. "Value Profiling," *Proc. of the 30th Annual International Symposium on Microarchitecture*, pp. 259–269, December 1997.

[5] J. Casmira, D. Hunter, and D. Kaeli, "Tracing and Characterization of Windows NT-based System Workloads," *Digital Technical Journal*, 10(1):6-21, December 1998.

[6] T. Conte, B. Patel, and J. S. Cox, "Using Branch Handling Hardware to Support Profile-Driven Optimization," *Proc. of the 27th Annual International Symposium on Microarchitecture*, San Jose, CA, 1994.

[7] J. Dean, J. Hicks, C. Waldspurger, W. Weihl, and G. Chrysos, "*ProfileMe*: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors," *Proc. of the 30th Annual International Symposium on Microarchitecture*, 1997.

[8] J. Hollingsworth et al., "MDL: A Language and Compiler for Dynamic Program Instrumentation," *Proc. of the International Conference on Parallel Architectures and Compilations Techniques*, November, 1997.

[9] W. Hwu et al., "The Superblock: An Effective Technique for VLIW and Superscalar Compilation," *The Journal of Supercomputing* 7(1/2):229-248, Kluwer Academic Publishers, May 1993.

[10] M. Merten, A. Trick, C. George, J. Gyllenhaal, and W. Hwu, "A Hardware-Driven Profiling Scheme for Identifying Program Hot Spots to Support Runtime Optimization", *Proc. of the 26th International Symposium on Computer Architecture*, Atlanta, GA, May 1999.

[11] S. Ross, *Introduction to Probability Models*, Academic Press, Inc., San Diego, CA, 1989.

[12] A. Tamches and B. Miller, "Fine-grained Dynamic Instrumentation of Commodity Operating System Kernels," *3rd Symposium on Operating Systems Design and Implementation*, pp. 117–130, February 1999.

[13] C. Young, "Path-based Compilation," Ph.D. thesis, Division of Engineering and Applied Sciences, Harvard University, Cambridge, MA, January 1998.

[14] C. Young and M. Smith, "Branch Instrumentation in SUIF," *Proc. of the First SUIF Compiler Workshop*, Stanford, CA, January 1996.

[15] C. Young and M. Smith, "Better Global Scheduling Using Path Profiles," *Proc. of the 31st Annual International Symposium on Microarchitecture*, pp. 115-123, December 1998.

[16] C. Young, N. Gloy, and M. Smith, "A Comparative Analysis of Schemes for Correlated Branch Prediction," *Proc. of the 22nd Annual International Symposium on Computer Architecture*, pp. 196-205, 1995.

[17] X. Zhang et al., "System Support for Automatic Profiling and Optimization," *Proc. of the 16th ACM Symposium on Operating Systems Principles*, 1997.