Continuous Path and Edge Profiling*

Michael D. Bond Kathryn S. McKinley

Dept. of Computer Sciences, University of Texas at Austin

E-mail: {mikebond,mckinley}@cs.utexas.edu

Abstract

Microarchitectures increasingly rely on dynamic optimization to improve performance in ways that are difficult or impossible for ahead-of-time compilers. Dynamic optimizers in turn require continuous, portable, low cost, and accurate control-flow profiles to inform their decisions, but prior approaches have struggled to meet these goals simultaneously.

This paper presents PEP, a hybrid instrumentation and sampling approach for continuous path and edge profiling that is efficient, accurate, *and* portable. *PEP uses a subset of Ball-Larus path profiling to identify paths with low overhead, and uses sampling to mitigate the expense of storing paths. PEP further reduces overhead by using profiling to guide instrumentation placement. PEP improves profile accuracy with a modified version of Arnold-Grove sampling. The resulting system has 1.2% average and 4.3% maximum overhead, 94% path profile accuracy, and 96% edge profile accuracy on a set of Java benchmarks.*

1. Introduction

Traditional static optimization is limited by factors such as dynamically linked libraries, lack of runtime microarchitecture information, and unpredictable or phasic behavior. Recent work shows that dynamic optimization can overcome these limitations by using dynamic control-flow information to inform optimizations [2, 5, 9, 11, 15, 16, 17, 20, 22, 32]. Effective dynamic recompilation depends on control-flow information that is *accurate*, *continuous*, and *low overhead*. Existing approaches struggle to meet all these goals at once.

Instrumentation-based approaches generally profile every path or edge [6, 8, 14] but are too expensive for all-the-time use, or their accuracy relies on relatively predictable program behavior. Sample-based approaches profile a representative fraction of dynamic paths or edges [1, 4, 18, 26, 27] but either cannot collect path profiles or require switching between uninstrumented and instrumented code. Some software-based systems achieve low overhead by limiting profile collection to initial program execution [2, 30], giving up the goal of continuous profiling and risking performance degradation if behavior changes [31]. Platform-specific hardware [10, 23, 28] achieves low runtime overhead but forsakes portability.

This paper presents PEP, a low overhead, high accuracy continuous path and edge profiling approach. PEP is a hybrid: it combines both instrumentation and sampling. It uses a low-overhead subset of Ball-Larus path profiling that correctly and continuously computes paths, but only stores paths at sample points to mitigate this expense. PEP requires only (1) a thread-switching mechanism that is common to Java virtual machines and operating system support for timer interrupts, or (2) an existing sampling mechanism. PEP further lowers instrumentation overhead using *profile-guided profiling*: it places instrumentation using the edge profile collected so far. To our knowledge, PEP is the first profiling technique to combine all-the-time instrumentation with sampling, and the first to employ path profiling as an efficient means for edge profiling.

We evaluate PEP in Jikes RVM, a high performance Java-in-Java virtual machine with an adaptive sample-based compiler. We demonstrate a configuration of PEP with 1.2% average and 4.3% maximum overhead, and collects path and edge profiles with 94% and 96% average accuracy using SPEC JVM and DaCapo benchmarks. This combination of accuracy, continuity, and efficiency creates new opportunities for dynamic optimizers to be more speculative and aggressive.

*This work is supported by NSF CCR-0311829, NSF ITR CCR-0085792, NSF CCR-0311829, NSF CISE infrastructure grant EIA-0303609, DARPA F33615-03-C-4106, and IBM. Any opinions, findings and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

2. Related Work

This section compares PEP to previous work. It first discusses systems that reduce costs by collecting only an initial profile. It then presents continuous approaches that use instrumentation, sampling, and hardware.

2.1. One-Time Profiling

Some online profiling techniques achieve low overhead by profiling only part of a program's execution. Jikes RVM's baseline compiler collects a *one-time* edge profile for unoptimized code, but its optimizing compiler does not [2]. *Structural path profiling* collects a one-time path profile with dynamic instrumentation, which it later removes [30]. A one-time profile may not capture whole-program behavior, and performance will consequently suffer. The effects can be devastating in systems with high misspeculation penalties (e.g., rePLay [20] and MSSP [32]). Aggressive, speculative dynamic optimizers must respond quickly to changing program behavior to avoid significant performance degradation [31].

2.2. Instrumentation-Based Profiling

Continuous instrumentation-based profiling tends to have high overhead. Ball and Larus path and edge profiling add 31% and 16% on average [6].

Practical path profiling (PPP) [8] and targeted path profiling [14] use *profile-guided profiling* to reduce overhead. PPP simplifies path profiling instrumentation using an existing edge profile. PEP borrows PPP's *smart path numbering* algorithm to place instrumentation on cold edges (Section 3.4). If the edge profile is unrepresentative, path profile accuracy does not suffer, although performance may. PPP however uses other techniques that *do* sacrifice accuracy, although not much for SPEC C and Fortran programs. PEP has lower runtime overhead than PPP (1.2% vs. 5% on average), with about the same path profiling accuracy (94% vs. 95% on average).

2.3. Sample-Based Profiling

Sample-based approaches avoid the high runtime overhead of instrumentation. The DIGITAL Continuous Profiling Infrastructure (DCPI) collects a basic block profile (less information than a path or edge profile [13]) by occasionally sampling the program counter [1]. DCPI has 1-3% overhead on average, which is similar to PEP's.

Arnold and Ryder sample instrumentation by switching between uninstrumented and instrumented versions of program code [4]. The switching framework alone has 4.9% average overhead, which drops to 1.4% by moving *yieldpoints* from uninstrumented to instrumented code. In contrast, PEP adds a small amount of instrumentation instead of duplicating code, and does not need to switch between uninstrumented and instrumented code. Its instrumentation (without sampling) executes all the time with an average overhead of 1.1%.

Dynamic instrumentation can emulate sampling by repeatedly adding and removing instrumentation [18, 26, 27]. The relatively high cost of this swapping limits how quickly dynamic systems can respond to changes in program behavior. Because this approach is complex, not many dynamic optimizers use it.

2.4. Profiling with Hardware Support

Hardware support includes using existing branch prediction hardware to collect edge profiles with low overhead (0.4-4.6%) [10], and a programmable, hardware-based path profiler with low overhead (0.5% on average) and high accuracy (above 90% on average) using a sufficiently large hardware-based path table [28]. Shye et al. collect path profiles by sampling the Itanium 2's branch trace buffer (BTB), achieving 1% overhead on average and 88% path accuracy on average [23]. PEP avoids specialized hardware but has overhead and accuracy comparable to hardware-based profilers.

3. Continuous Path and Edge Profiling

This section describes our mechanism for continuous path and edge profiling (PEP) in dynamic optimizers. We first review Ball-Larus path profiling since PEP extends it. We then describe PEP instrumentation, sampling, and profile-guided profiling.

3.1. Ball-Larus Path Profiling

Ball-Larus path profiling (BLPP) instrumentation counts how many times each acyclic, intraprocedural path executes [6]. Paths begin on method entry and end on method exit. Paths also begin and end on loop back edges. A procedure call starts a new path and defers the caller's path until the callee returns. BLPP adds instrumentation that computes a unique path number for each possible path, and stores it at the end of the path.

To enumerate acyclic paths, BLPP first converts a routine's control flow graph (CFG) to a directed acyclic graph (DAG) by *truncating* each loop back edge. BLPP truncates the back edge by removing it and adding two *dummy* edges: (1) one from method entry to loop header and (2) one from loop tail to method exit. Figures 1(a) and (b) illustrate truncation.

The Ball-Larus path numbering algorithm (Figure 2) then assigns values to edges such that, for each of the N acyclic paths in the DAG, the sum of the edge values is a unique number in $[0, N-1]$. Figure 1(c) shows a DAG labeled with edge values where $N = 8$.

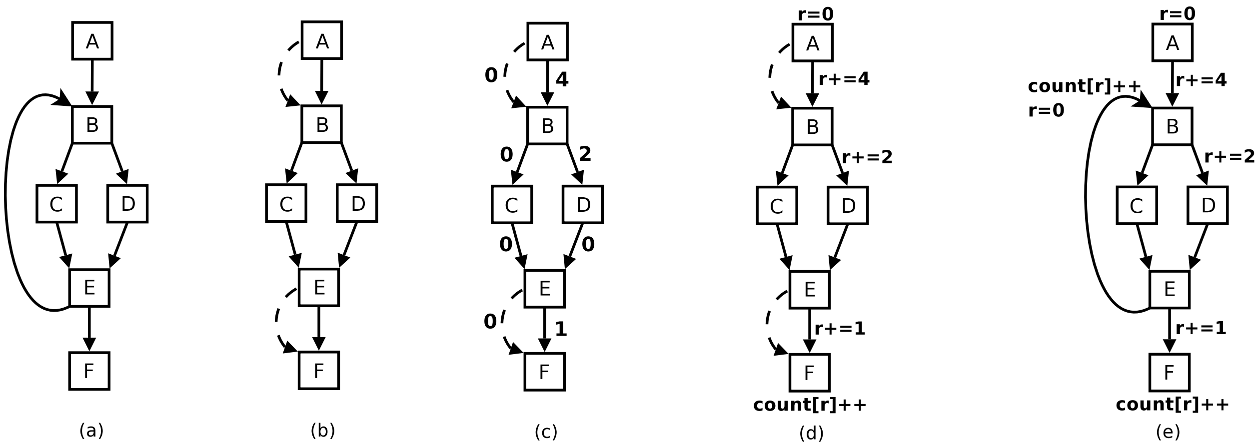

Figure 1. An example routine instrumented by Ball-Larus path profiling. (a) Original routine; (b) after conversion from the CFG to a DAG; (c) after path numbering; (d) after instrumentation placement; and (e) after conversion back to the CFG.

```
foreach basic block v in reverse topological order
  if v is the exit block
    NumPaths(v) = 1
  else
    NumPaths(v) = 0
    foreach edge e = v→w
      Val(e) = NumPaths(v)
      NumPaths(v) = NumPaths(v)+ NumPaths(w)
```

Figure 2. Ball-Larus path numbering algorithm.

BLPP next places instrumentation on edges. It uses a variable `r`, called the *path register*, to compute the unique number for each path as the path executes. The instrumentation

1. initializes `r` to zero (`r=0`) on method entry;
2. adds nonzero values to `r` (`r+=val`) on edges; and
3. updates the frequency for the taken path in a path frequency table (`count[r]++`) on method exit.

Figure 1(d) shows the example DAG with path-counting instrumentation. The final step of the BLPP algorithm converts the DAG back to a CFG. It removes dummy edges, restores back edges, and moves instrumentation from dummy edges to the corresponding back edges, including copying `r=0` from method entry and `count[r]++` from method exit to the back edge. Figure 1(e) shows the example converted back to a CFG.

Ball and Larus showed that BLPP's runtime overhead is 31% on average on SPEC95, but is as high as 73% for `perl` and 97% for `gcc`.

3.2. PEP Instrumentation

PEP relies on the simple observation that Ball-Larus instrumentation can be separated into two steps:

1. computing the current path's path number, and
2. incrementing the frequency for that path.

This separation matters because step (1) is extremely inexpensive, and step (2) is not. Step (2) is a load-increment-store memory operation or a hash function call [6], while step (1) is a sequence of register additions (plus the cost of any register spills). Our measurements confirm that step (2) represents the bulk of the runtime overhead of Ball-Larus instrumentation.

PEP avoids storing all paths by *sampling* the value of the path register `r` at exactly the same locations BLPP would update the path profile (i.e., it samples the `count[r]++` instrumentation). There are a variety of ways to implement this feature in a dynamic optimizer that uses operating system or hardware support to sample programs. Our implementation piggybacks on Jikes RVM's thread-switching system [2]. This system supports a general mechanism that the VM uses to support multithreading, thread scheduling, JIT compilation, garbage collection, and method and call sampling. To gain control of a thread quickly, Jikes RVM inserts *yieldpoints* on loop headers, method entry, and method exit. Each yieldpoint examines a global status flag that is true only when the RVM needs to take a sample or perform some other system function.

Since yieldpoints are on loop headers, rather than loop back edges, we implement PEP to end paths at headers rather than back edges. PEP splits paths that traverse a header into two paths at the header. We believe this difference from BLPP is minor because it only affects the first path through a loop. We could avoid this difference by modifying Jikes RVM to place yieldpoints on back edges rather than headers. Figure 3 shows how BLPP works when paths end at loop headers instead of back edges. Figures 3(a) and (b) show how we truncate a loop header: we split it immediately after the yield-

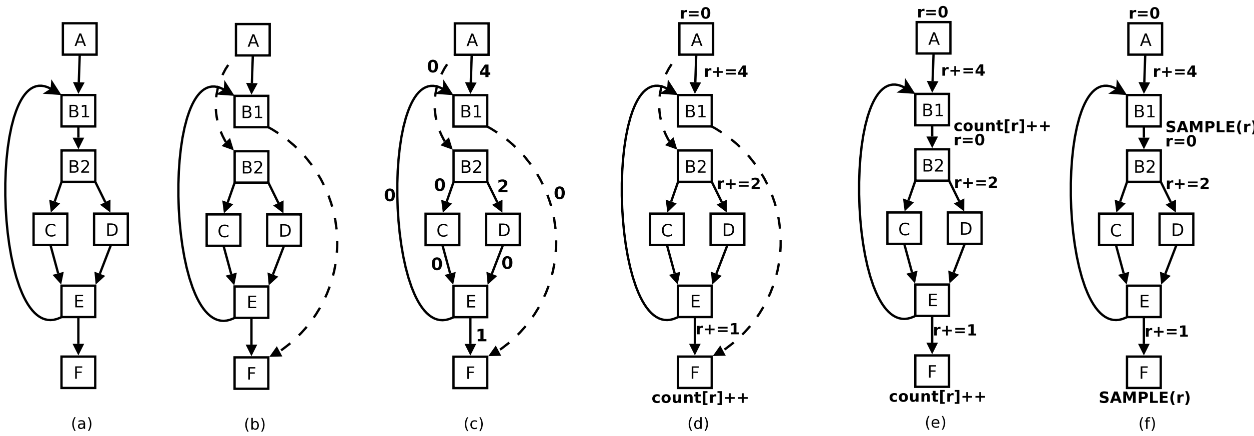

Figure 3. (a)-(e) An example routine instrumented by BLPP when paths end at loop headers rather than back edges. (f) PEP instrumentation and the points at which PEP samples the path register.

point (in *B1*), and truncate the edge between the two sub-blocks by removing it and adding two dummy edges. Figures 3(c)-(e) show path numbering, instrumentation, and conversion back to a CFG. Figure 3(f) shows the instrumentation PEP adds to the example CFG, and where PEP samples the value of the path register. PEP stores the directed acyclic graph (DAG) representation labeled with edge values (e.g., Figure 3(c)), which we call the *P-DAG*, for later use during sampling.

We believe PEP could be implemented in a system without thread-switching points. First, this approach requires compiler analysis to determine the location of the current path value (i.e., which register or stack location) at each program point, which is similar to *GC maps* in JVMs [2]. Second, this approach requires a way to reconstruct the partial path from the partial path number. Conveniently, a partially taken path can be identified from the partial path number using the same greedy reconstruction algorithm for full paths [6].

3.3. PEP Sampling

This section describes how PEP updates the path and edge profiles when it samples the value of the path register. The value of the path register at each sample point is the path number for the most recently taken path, so PEP updates the path profile simply by incrementing the frequency of the path in the method's path profile.

To update the edge profile, PEP uses an efficient greedy algorithm [6] to trace the sequence of edges in the P-DAG (see previous section) that make up the taken path. It then updates the frequencies of each edge in the edge profile.

3.4. Profile-Guided Profiling

PEP uses existing edge profile information to further reduce the cost of executing the path profiling instrumentation PEP inserts. This *profile-guided profiling* is a good fit for staged dynamic optimization, in which a method is successively recompiled at higher optimization levels in response to profile information [8, 14].

PEP uses *smart path numbering*, one of the profile-guided profiling techniques from practical path profiling (PPP) [8]. The smart path numbering algorithm (Figure 4) modifies the Ball-Larus path numbering algorithm (Figure 2) to assign zero to the hottest outgoing edge of each basic block, so PEP does not place *any* instrumentation on these edges.

foreach basic block v in reverse topological order
  if v is the exit block
    $\mathit{NumPaths}(v) = 1$
  else
    $\mathit{NumPaths}(v) = 0$
    foreach edge $e = v \rightarrow w$ in decr. order of exec. freq.
      $\mathit{Val}(e) = \mathit{NumPaths}(v)$
      $\mathit{NumPaths}(v) = \mathit{NumPaths}(v) + \mathit{NumPaths}(w)$

Figure 4. Smart path numbering algorithm. Changes to Ball-Larus numbering (Figure 2) are underlined.

Profile-guided profiling provides only modest performance improvement. If we instead use profile-guided profiling to place instrumentation on *hot* edges, PEP instrumentation's execution overhead increases only 1.4% (i.e., from 1.1% to 2.5%; full results omitted for space). PEP's overhead is much lower than BLPP's mainly due to its novel instrumentation-sampling mix.

4. Implementation

This section describes our implementation of PEP in Jikes RVM.

4.1. Adaptive Compilation in Jikes RVM

Jikes RVM is a high performance Java-in-Java virtual machine that uses an adaptive compilation system triggered by sampling [2]. The compilation system has two completely distinct compilers and four levels of optimization. When Jikes RVM first loads a class, it invokes the *baseline* compiler, which translates bytecode to unoptimized, native code. All subsequent compilation uses the *optimizing* compiler at one of three progressively higher levels of optimization.

To support multithreading, garbage collection, and other Java language features, Java VMs provide thread switching points. Since VMs need to gain control of threads quickly, these points almost always include loop back edges, method entries, and method exits. Jikes RVM uses exactly these yieldpoints to perform thread switching and sampling. Our focus here is sampling. Jikes RVM schedules regular operating system timer interrupts, and the interrupt handler sets a flag. Every yieldpoint examines this flag and executes the yieldpoint handler if the flag is set. This handler examines the stack, computes method invocation counts, and updates the dynamic call graph.

4.2. One-Time Edge Profiling

In addition to method sampling, the baseline compiler collects an intraprocedural edge profile. When the baseline compiler translates bytecode to unoptimized, native code, it also inserts instrumentation on each branch that updates a *taken* or *not-taken* counter. This instrumentation is expensive, but is not executed for long since Jikes RVM quickly recompiles frequently executed methods.

The optimizing compiler computes branch biases from this edge profile and uses them to drive optimization such as Pettis-Hansen code reordering [21], loop invariant code motion, and register allocation. The optimizing compiler does not add edge profile instrumentation, so subsequent recompilations of the method instead use the out-of-date edge profile generated by the baseline compiler.

4.3. Implementing PEP in Jikes RVM

Since (1) programs execute mostly optimized code by design, (2) the baseline compiler already has edge profiling, and (3) the baseline compiler is a separate compiler altogether, we implement PEP in the optimizing compiler only. The PEP instrumentation pass

1. builds the P-DAG representation (Section 3.2);
2. assigns values to edges (Section 3.4);
3. inserts instrumentation (Section 3.2); and
4. modifies yieldpoints to pass the value of the path register to the yieldpoint handler.

The yieldpoint handler uses hash tables to keep track of path frequencies, and it updates the edge profile as described in Section 3.3. Our implementation saves time by computing a path's edges only when PEP samples a path for the first time. When PEP samples the same path again (the common case), it looks up the sequence of edges in the path profile.

For consistency and simplicity, PEP collects an edge profile that is directly comparable to the edge profile collected by the baseline compiler. The baseline compiler uses a taken counter and a not-taken counter for each bytecode-level branch. The optimizing compiler's internal representation (IR) stores a map from the original bytecode branch to the corresponding IR branch. Note that multiple branches in the IR may map to the same bytecode branch due to method inlining, loop unrolling, and other optimizations. PEP uses the same taken and not-taken counters for each of these branches. The compiler may eliminate a bytecode-level branch altogether via constant propagation and dead code elimination. In this case, PEP cannot collect a profile for it, but since it never executes, accuracy is not compromised.

PEP can lose accuracy when the optimizing compiler does not insert yieldpoints. This occurs when (1) a method is a leaf with no branches or (2) a method is marked as *uninterruptible*, indicating that Jikes RVM should not switch threads during the method. In the first case, the method has no branches, so its profile is trivial. The second case occurs in internal Jikes RVM methods only. The compiler occasionally inlines such a method into an application method. If the inlined uninterruptible method contains a loop, the compiler does not add a yieldpoint to the loop header. PEP instruments the application method, but loses information about paths that end at the uninterruptible loop's header.

4.4. Arnold-Grove Sampling

This section explains two problems with timer-based sampling, how Arnold and Grove address this problem, and how we adapt their solution for PEP.

Jikes RVM uses a timer-based interrupt to set the yieldpoint flag, triggering a call to the yieldpoint handler (Section 4.1). Timer ticks occur relatively infrequently (e.g., every 20 ms on IA32), which is not often enough to collect representative path and edge profiles for our short-running programs, although it may be sufficient for long-running server programs. Furthermore, because sampling is timer-based, and the time between consecutive yieldpoints varies considerably throughout a program, some yieldpoints are more likely to be sampled than others, resulting in sampling bias [3].

Arnold and Grove identify these two problems and present a solution that improves profile accuracy for call graph profiles [3]. They collect `SAMPLES` samples at successive yieldpoints by setting, rather than resetting,

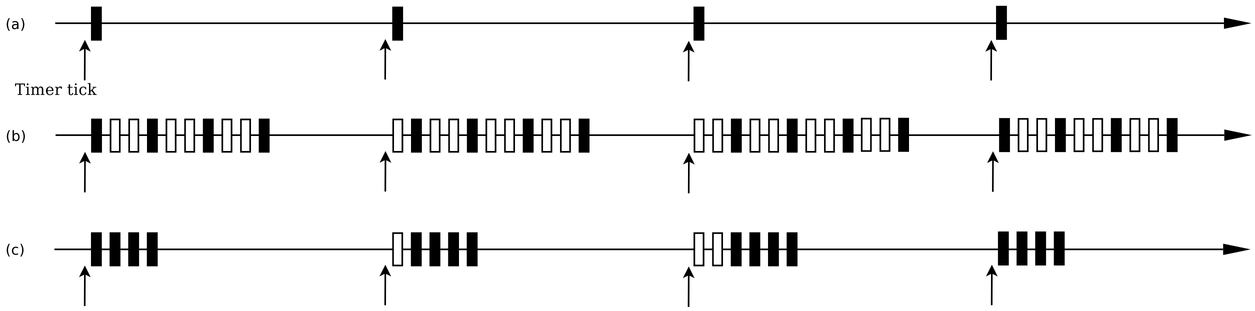

Figure 5. Example comparing (a) timer-based sampling, (b) Arnold-Grove sampling, and (c) simplified Arnold-Grove sampling. Boxes represent sampling opportunities. Filled boxes represent taken samples, while unfilled boxes are skipped samples. For (b) and (c), `SAMPLES` is 4, and `STRIDE` is 3.

the thread-switch flag at the end of the yieldpoint handler. To alleviate timer-based sampling bias, they *stride*, taking a sample every `STRIDE` yieldpoints. Before the first sample, they stride by an amount that rotates among the values in [1, `STRIDE`]. Figures 5(a) and (b) compare timer-based sampling to Arnold-Grove sampling.

We observe that in Jikes RVM, taking a sample is almost as expensive as striding over a sample [3]. We find that the most important time to stride is before the first timer tick, and that striding after the first sample is not a good overhead-accuracy trade-off, at least for PEP. Thus, we modify Arnold-Grove sampling to stop striding after the first sample of a timer tick. We refer to this modification as *simplified Arnold-Grove sampling*. Figures 5(b) and (c) compare simplified and regular Arnold-Grove sampling.

We use the notation *PEP(*`SAMPLES,STRIDE`*)* to refer to a PEP configuration that uses simplified Arnold-Grove sampling. For example, *PEP(1,1)* is equivalent to timer-based sampling: it takes one sample per timer tick and does not stride. *PEP(64,17)* strides by up to 17 (i.e., it skips between 0 and 16 samples) after a timer tick, and then it takes 64 samples without striding.

5. Methodology

We now describe our experimental platform, benchmarks, and methodology. We evaluate PEP in Jikes RVM (CVS HEAD, 2005/02/19 04:02:09 UTC) [2].

Platform. We perform our experiments on a 3.2 GHz Pentium 4 with hyper-threading enabled. It has a 64-byte L1 and L2 cache line size, an 8KB 4-way set associative L1 data cache, a 12Kμops L1 instruction trace cache, a 512KB unified 8-way set associative L2 on-chip cache, and 1GB main memory, and runs Linux 2.6.0.

Benchmarks. We use the SPEC JVM98 benchmarks, a fixed-workload version of SPEC JBB2000 (70000 transactions) called `pseudojbb`, and the DaCapo benchmarks that execute on Jikes RVM [19, 24, 25]. We omit the DaCapo benchmark `hsqldb` because we could not get it to run correctly with Jikes RVM, with or without PEP.

Methodology. We use two methodologies for our experiments. (1) The *adaptive* methodology lets the adaptive compiler behave as intended and is non-deterministic. (2) The *replay compilation* methodology (previously known as pseudo-adaptive) is deterministic and eliminates variations due to non-deterministic application of the adaptive compiler. Without it, exactly when the timer interrupt goes off changes compilation decisions. Replay compilation forces Jikes RVM to compile the same methods in the same order on different executions and thus prevents high variability.

Replay compilation uses *advice files* produced by a previous well-performing adaptive run (best of five). The advice files specify (1) the optimization level for compiling each method, (2) the dynamic call graph profile, and (3) the edge profile produced by baseline-compiled code. We execute two consecutive iterations of the application. During the first iteration, Jikes RVM optimizes code using the advice files. The second iteration executes only the application at a mix of optimization levels.

We report benchmark execution time as the minimum of 10 trials (shown as dots; crosses represent trials off the graph), since it is the least disturbed by system variability. The median produces almost exactly the same results. We use 25 trials for compress because its times vary a lot. We report profile accuracy as the median of 10 trials (shown as dots) to obtain a representative accuracy not swayed by outliers.

5.1. Collecting Perfect Profiles

To collect perfect profiles for comparison with PEP, we implement pure instrumentation-based path and edge profiling in the optimizing compiler. Our instrumentation-based path profiling mimics PEP's instrumentation, except that it updates the path profile at every yieldpoint via an inserted hash call. Like

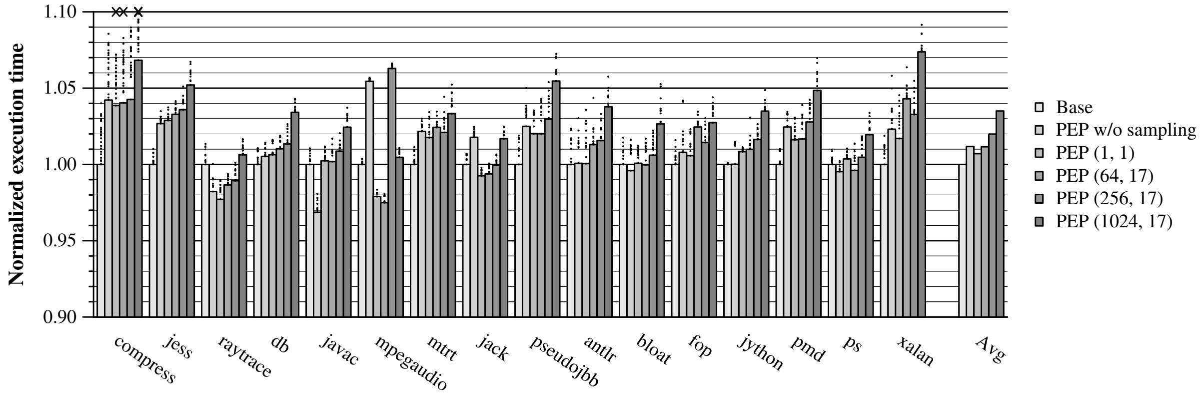

Figure 6. The execution overhead of PEP. We use the second iteration of replay compilation.

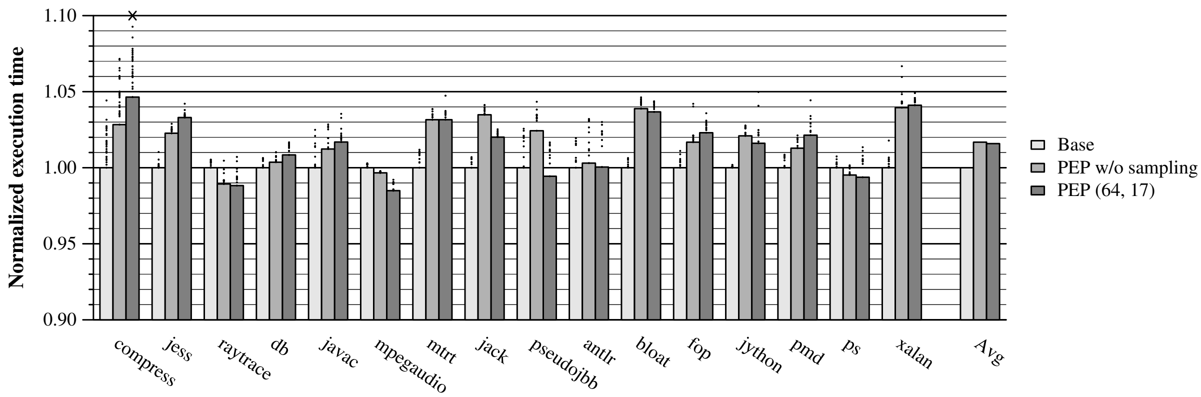

Figure 7. The compilation and execution overhead of PEP. We use the first iteration of replay compilation.

PEP, instrumentation-based path profiling does not profile paths that end at uninterruptible loop headers (Section 4.3). Instrumentation-based edge profiling mimics the existing edge profiling in the baseline compiler (Section 4.2). It adds instrumentation on each IR branch that increments a taken or not-taken counter for the corresponding bytecode-level branch. Instrumentation-based path and edge profiling have 92% (8 to 407%) and 10% (0 to 34%) average overhead, respectively. This overhead is tolerable because we only use instrumentation-based profiling to collect perfect profiles for comparison purposes.

6. Results

This section evaluates the overhead and profile accuracy of PEP. We demonstrate a sampling configuration, *PEP(64,17)* (Section 4.4), that has 1.2% runtime overhead, 94% path profile accuracy, and 96% edge profile accuracy. We measure the performance potential of continuous profiles by driving edge profile-guided optimizations in Jikes RVM with PEP.

6.1. Overhead

Figure 6 presents execution times for PEP instrumentation alone and with various sampling configurations. We use the second run of replay compilation to deterministically measure application execution time only. Execution times for each benchmark are normalized to the *Base* (i.e., without PEP) time.

The overhead of PEP's instrumentation alone is on average 1.1% and at most 5.4%. Timer-based sampling (*PEP(1,1)*) adds no detectable overhead. *PEP(64,17)* adds 0.1% overhead on average, yielding on average 1.2% and at most 4.3% total runtime overhead. The other sampling configurations add 0.8-2.3% additional overhead on average.

For some programs, overhead decreases when instrumentation is added or when the sampling rate increases. This result is counterintuitive because instrumentation and increased sampling rate only give the processor more work to do. These results must be due to sensitivities at the microarchitecture and JVM levels. For example, PEP instrumentation perturbs code layout, which could improve instruction cache performance.

6.2. The Cost of Adding Instrumentation

In addition runtime overhead, PEP adds compilation overhead by inserting instrumentation. Qualitatively, PEP adds little to compilation time: it performs three

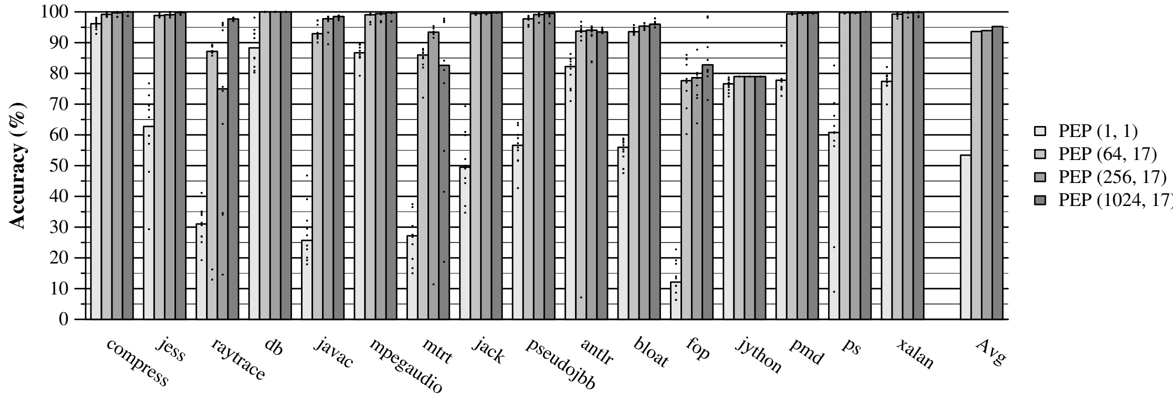

Figure 8. The path profile accuracy of PEP.

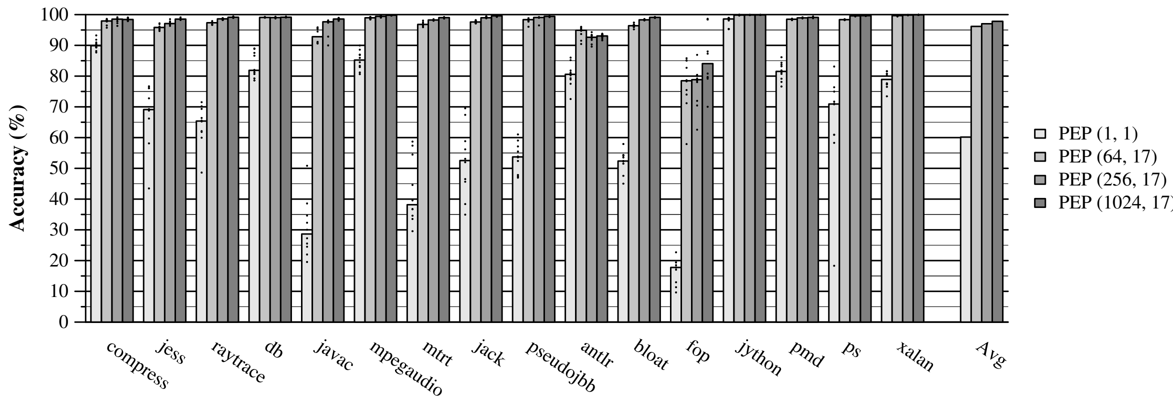

Figure 9. The edge profile accuracy of PEP.

quick passes over each method compiled by the optimizing compiler to (1) construct the P-DAG (Section 3.2), (2) perform smart path numbering (Section 3.4), and (3) insert instrumentation. In this section, we evaluate this claim quantitatively. We report run times for the first iteration of replay compilation, which includes both compilation and program execution, yielding overheads that reflect the combined cost of both inserting and executing PEP instrumentation.

Figure 7 shows the compilation and execution overhead of PEP. This overhead is 1.6% on average and 4.6% at most, which is higher than execution overhead alone (Section 6.1), suggesting that, as a percentage, PEP adds more overhead to compilation than to program execution. However, because PEP's compilation overhead is modest, it is only likely to affect performance for shorter-running programs, which the dynamic optimizer spends a greater proportion of time compiling. These benchmarks are generally short-running: median execution time (including compilation) is 11 seconds. For example, `jack` executes in four seconds, and its PEP compilation and execution overhead is relatively high (3.5%) compared to execution overhead alone (1.8%).

6.3. Path Profiling Accuracy

This section evaluates PEP's accuracy at predicting programs' hot paths. We use the Wall weight-matching scheme [29], as does previous work [7, 8, 23]. It measures PEP's ability to identify a program's hot paths, given a threshold, but does not compare relative frequencies. We use this measure because accurate hot path identification is exactly what most path-based optimizations need.

We compute flow, which is a measure of the amount of execution along a path, using the branch-flow metric [8]. Branch flow computes the flow $F(p)$ of a path p by weighting p's frequency *freq*(p) by its length in branches, b_p:

$$F(p) = \mathit{freq}(p) \times b_p$$

The flow on a set of paths P is the sum of the flows:

$$F(P) = \sum_{p \in P} (\mathit{freq}(p) \times b_p)$$

The Wall weight-matching scheme first identifies a program's *actual* hot paths, H_{actual}, using a perfect path profile collected with instrumentation. A path is hot if its flow is above 0.125% of total program flow, the same threshold used in previous work. The scheme then constructs the set of *estimated* hot paths, $H_{estimated}$, by selecting the top $|H_{actual}|$ hottest paths from PEP's estimated path profile. Accuracy is the fraction of actual hot path flow that the estimated path profile predicts:

$$Accuracy = \frac{F(H_{estimated} \cap H_{actual})}{F(H_{actual})}$$

Figure 8 shows the accuracy of PEP at predicting programs' hot paths. Multiple samples per timer tick and striding yield high accuracy, with small improvements from higher sampling rates. *PEP(64,17)* has 94% path profile accuracy. Timer-based sampling, with 53% accuracy, is not sufficient for predicting hot paths.

6.4. Edge Profiling Accuracy

This section evaluates PEP's ability to collect a representative edge profile. We compare PEP's edge profile to a perfect edge profile generated by instrumentation-based *path* profiling to avoid measuring loss of accuracy due to uninterruptible loop headers (Section 5.1). However, if we compare to instrumentation-based edge profiling instead, accuracy falls by only 2% on average (e.g., 96% to 94% for *PEP (64, 17)*), indicating that uninterruptible loop headers execute rarely in application code (full results omitted for space).

We compare the actual and PEP's estimated edge profiles using *relative overlap*. Relative overlap measures how well PEP predicts the taken/not-taken bias for each branch. We use this measure because Jikes RVM uses only bias, rather than absolute edge frequency, for optimizations. The accuracy for branch b is 1 minus the difference between b's actual taken bias, $taken_{actual}(b)$, and the taken bias that PEP predicts, $taken_{estimated}(b)$:

$$Accuracy(b) = 1 - |taken_{actual}(b) - taken_{estimated}(b)|$$

To compute accuracy of an edge profile, relative overlap weights the accuracy of each branch b by its actual execution frequency $freq_{actual}(b)$:

$$Accuracy = \frac{\sum_{b \in Branches}(freq_{actual}(b) \times Accuracy(b))}{\sum_{b \in Branches}(freq_{actual}(b))}$$

Figure 9 shows PEP's accuracy at predicting a representative edge profile. Multiple samples per timer tick and striding yield high accuracy, with more samples per tick producing slightly more accurate edge profiles on average. *PEP (64, 17)* provides 96% edge profile accuracy.

An alternative measure of edge profile accuracy that we call *absolute overlap* (previous work simply calls it *overlap* [3, 4, 12]) is concerned with branch *frequency* rather than just branch bias. PEP accuracy computed using absolute overlap remains high but is lower, which is not surprising since it is more difficult to predict an edge's frequency relative to the entire edge profile. *PEP(64,17)* has 83% accuracy using absolute overlap; *PEP(256,17)* has 87% accuracy; and *PEP(1024,17)* has 88% accuracy. (We omit individual results for brevity.)

6.5. Driving Optimization with PEP

This section evaluates the impact of using continuous edge profiles to drive optimization in Jikes RVM.

We first compare the accuracy of one-time edge profiling (see Section 4.2) to perfect continuous edge profiling (see Section 5.1). We omit full results to save space. The accuracy (using *relative overlap*) of one-time profiling is very high: 97% on average and 86% at worst. Initial behavior is a good predictor of whole-program behavior for these programs, so there is likely to be little room for improvement from continuous profiling.

Next we evaluate the performance pontetial of using continuous edge profiles to drive optimizations in Jikes RVM (e.g., code reordering, loop invariant code motion, and register allocation). Figure 10 compares driving optimization with a perfect continuous edge profile and a one-time profile (first two bars). We use the second iteration of replay compilation to measure only program execution. The graph shows an average 0.9% performance improvement using a continuous profile over a one-time profile. This only modest improvement is not surprising, given that the programs we evaluate have very high one-time profile accuracy. Programs with less predictable behavior may benefit more from continuous profiles.

The third bar uses a *flipped* continuous profile, where each probability is flipped (e.g., a 90% taken branch becomes 10% taken). Performance degradation is significant, indicating that Jikes RVM's edge profile-guided optimizations are in fact sensitive to profile accuracy.

Finally, we evaluate the costs and benefits of PEP instrumentation using the adaptive methodology. Figure 11 evaluates the performance of using *PEP(64,17)* both to collect an edge profile and to drive optimization with it. The comparison point *Base* is a normal adaptive run that uses only the one-time edge profile generated by baseline-compiled code. We take the median of 25 trials because of the high variability between executions due to the non-deterministic application of the optimizing compiler.

The figure shows that using PEP in Jikes RVM adds 1.3% average and 3.2% maximum overhead. This result means that the costs of PEP (compilation and execution overhead) outweigh the benefit (continuous profiling information), which is not surprising given the limited po-

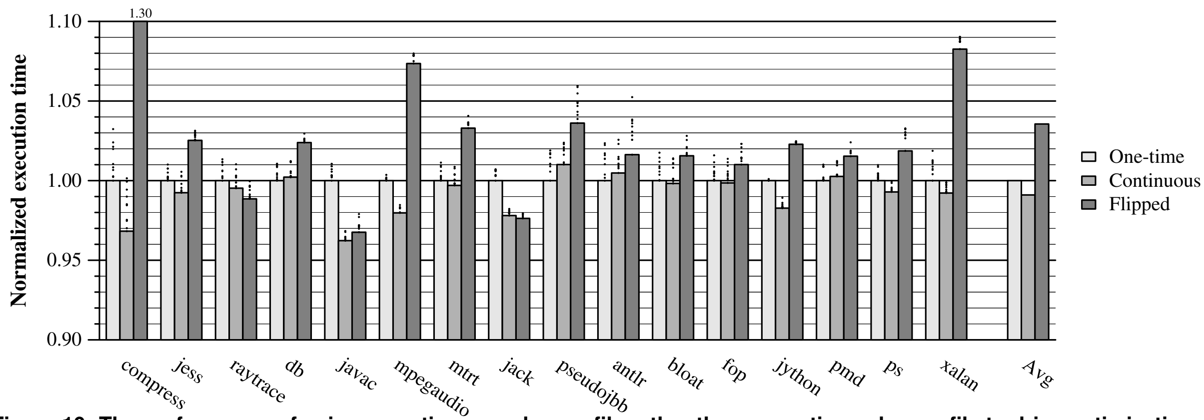

Figure 10. The performance of using a continuous edge profile rather than a one-time edge profile to drive optimization. We use the second iteration of replay compilation.

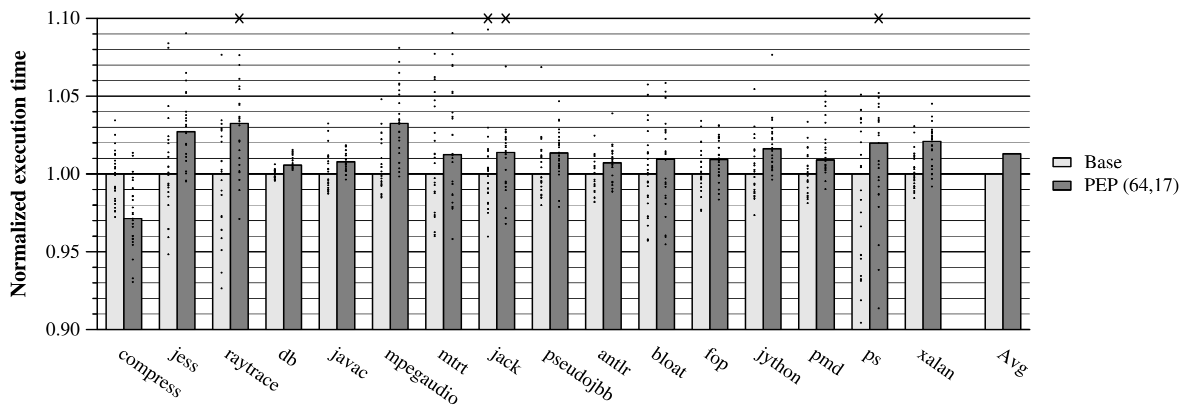

Figure 11. The performance of using PEP to collect profiles and drive optimization. We use the adaptive methodology.

tential in these programs. This result occurs in part because of the predictability of the programs we use, and in part because Jikes RVM does not aggressively speculate on runtime information. Continuous profiling is ideal for aggressive, speculative dynamic optimizers that take big gambles on runtime information, and for programs with unpredictable, phased behavior.

7. Conclusions

This paper presents PEP, a low-cost approach for continuous path and edge profiling that requires only an existing thread-switching or sampling mechanism. PEP is a hybrid that, to our knowledge, is the first to combine all-the-time instrumentation with sampling and the first to use path profiling as an efficient way to collect an edge profile. We demonstrate a sampling configuration for PEP with 1.2% runtime overhead that collects continuous path profiles with 94% accuracy and edge profiles with 96% accuracy in Jikes RVM. This cost and accuracy combination points to a future where highly aggressive and speculative dynamic optimizers can readily depend on continuous and accurate runtime information.

8. Acknowledgements

We thank Xianglong Huang for developing replay compilation and for adding features to it in time for the submission. We thank Samuel Guyer, Xianglong Huang, David Grove, and Steve Blackburn for their help with Jikes RVM. We thank Steve Blackburn, Emmett Witchel, David Grove, and the anonymous reviewers for their helpful suggestions for improving the paper.

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 *Symposium on Operating Systems Principles*, 1997.

[2] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive Optimization in the Jalapeño JVM. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2000.

[3] M. Arnold and D. Grove. Collecting and Exploiting High-Accuracy Call Graph Profiles in Virtual Machines. In *International Symposium on Code Generation and Optimization*, 2005.

[4] M. Arnold and B. G. Ryder. A Framework for Reducing the Cost of Instrumented Code. In *Conference on Programming Language Design and Implementation*, 2001.

[5] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. *ACM SIGPLAN Notices*, 35(5):1–12, 2000.

[6] T. Ball and J. R. Larus. Efficient Path Profiling. In *International Symposium on Microarchitecture*, 1996.

[7] T. Ball, P. Mataga, and M. Sagiv. Edge Profiling versus Path Profiling: The Showdown. In *Symposium on Principles of Programming Languages*, 1998.

[8] M. D. Bond and K. S. McKinley. Practical Path Profiling for Dynamic Optimizers. In *International Symposium on Code Generation and Optimization*, 2005.

[9] D. Bruening, T. Garnett, and S. Amarasinghe. An Infrastructure for Adaptive Dynamic Optimization. In *International Symposium on Code Generation and Optimization*, 2003.

[10] T. M. Conte, B. A. Patel, and J. S. Cox. Using Branch Handling Hardware to Support Profile-Driven Optimization. In *International Symposium on Microarchitecture*, 1994.

[11] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The Transmeta Code Morphing™ Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges. In *International Symposium on Code Generation and Optimization*, 2003.

[12] P. T. Feller. Value Profiling for Instructions and Memory Locations. Master's thesis, University of California, San Diego, 1998.

[13] R. Gupta, E. Mehofer, and Y. Zhang. Profile Guided Compiler Optimizations. In *The Compiler Design Handbook: Optimizations and Machine Code Generation*. CRC Press, 2002.

[14] R. Joshi, M. D. Bond, and C. Zilles. Targeted Path Profiling: Lower Overhead Path Profiling for Staged Dynamic Optimization Systems. In *International Symposium on Code Generation and Optimization*, 2004.

[15] T. Kistler and M. Franz. Continuous Program Optimization: A Case Study. *ACM Transactions on Programming Languages and Systems*, 25(4):500–548, 2003.

[16] J. Lu, H. Chen, R. Fu, W.-C. Hsu, B. Othmer, P.-C. Yew, and D.-Y. Chen. The Performance of Runtime Data Cache Prefetching in a Dynamic Optimization System. In *International Symposium on Microarchitecture*, 2003.

[17] M. C. Merten, A. R. Trick, E. M. Nystrom, R. D. Barnes, and W. mei W. Hmu. A Hardware Mechanism for Dynamic Extraction and Relayout of Program Hot Spots. In *International Symposium on Computer Architecture*, 2000.

[18] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Performance Measurement Tools. *IEEE Computer*, 28(11):37–46, 1995.

[19] J. E. B. Moss, K. S. McKinley, S. M. Blackburn, E. D. Berger, A. Diwan, A. Hosking, D. Stefanović, and C. Weems. The DaCapo Project. Technical report, 2004. http://ali-www.cs.umass.edu/DaCapo/.

[20] S. J. Patel and S. S. Lumetta. rePLay: A Hardware Framework for Dynamic Optimization. *IEEE Transactions on Computers*, 50(6):590–608, 2001.

[21] K. Pettis and R. C. Hansen. Profile guided code positioning. In *Conference on Programming Language Design and Implementation*, 1990.

[22] R. Rosner, Y. Almog, M. Moffie, N. Schwartz, and A. Mendelson. Power Awareness through Selective Dynamically Optimized Traces. In *International Symposium on Computer Architecture*, 2004.

[23] A. Shye, M. Iyer, T. Moseley, D. Hodgdon, D. Fay, V. J. Reddi, and D. A. Connors. Analysis of Path Profiling Information Generated with Performance Monitoring Hardware. In *Workshop on Interaction between Compilers and Computer Architecture*, 2005.

[24] Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*, release 1.03 edition, 1999.

[25] Standard Performance Evaluation Corporation. *SPECjbb2000 Documentation*, release 1.01 edition, 2001.

[26] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A Dynamic Optimization Framework for a Java Just-in-Time Compiler. In *Conference on Object Oriented Programming, Systems, Languages, and Applications*, 2001.

[27] O. Traub, S. Schechter, and M. D. Smith. Ephemeral Instrumentation for Lightweight Program Profiling. Technical report, Harvard University, 2000.

[28] K. Vaswani, M. J. Thazhuthaveetil, and Y. N. Srikant. A Programmable Hardware Path Profiler. In *International Symposium on Code Generation and Optimization*, 2005.

[29] D. W. Wall. Predicting Program Behavior Using Real or Estimated Profiles. In *Conference on Programming Language Design and Implementation*, 1991.

[30] T. Yasue, T. Suganuma, H. Komatsu, and T. Nakatani. Structural Path Profiling: An Efficient Online Path Profiling Framework for Just-In-Time Compilers. *The Journal of Instruction-Level Parallelism*, 6:1–24, 2004.

[31] C. Zilles and N. Neelakantam. Reactive Techniques for Controlling Software Speculation. In *International Symposium on Code Generation and Optimization*, 2005.

[32] C. Zilles and G. Sohi. Master/Slave Speculative Parallelization. In *International Symposium on Microarchitecture*, 2002.