Break Dancing: Low Overhead, Architecture Neutral Software Branch Tracing

Gabriel Marin
Google
USA

Alexey Alexandrov
Google
USA

Tipp Moseley
Google
USA

Abstract

Sampling-based Feedback Directed Optimization (FDO) methods like AutoFDO and BOLT that employ profiles collected in live production environments, are commonly used in datacenter applications to attain significant performance benefits without the toil of maintaining representative load tests. Sampled profiles rely on hardware facilities like Intel's Last Branch Record (LBR) which are not currently available even on popular CPUs from ARM or AMD. Since not all architectures include a hardware LBR feature, we present an architecture neutral approach to collect LBR-like data. We use sampling and limited program tracing to capture LBR-like data from optimized and unmodified applications binaries. Since the implementation is in user space, we can collect arbitrarily long LBR buffers, and by varying the sampling rate, we can adjust the runtime overhead to arbitrarily low values. We target runtime overheads of <2% when the profiler is on and zero when it's off. This amortizes to negligible fleet-wide collection cost given the size of a modern production fleet. We implemented a profiler that uses this method of software branch tracing. We also analyzed its overhead and the similarity of the data it collects to the Intel LBR hardware using the SPEC2006 benchmarks. Results demonstrate profile quality and optimization efficacy at parity with LBR-based AutoFDO and the target profiling overhead being achievable even without implementing any advanced tuning.

CCS Concepts: • **General and reference** → **Measurement**; Performance; • **Computing methodologies** → *Simulation tools*.

Keywords: branch tracing, sampling based feedback directed optimization, debug registers, instruction decoding

ACM Reference Format:
Gabriel Marin, Alexey Alexandrov, and Tipp Moseley. 2021. Break Dancing: Low Overhead, Architecture Neutral Software Branch Tracing. In *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '21), June 22, 2021, Virtual, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3461648.3463853

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

LCTES '21, June 22, 2021, Virtual, Canada

© 2021 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-8472-8/21/06.
https://doi.org/10.1145/3461648.3463853

1 Introduction

Today's datacenter workloads, as well as web browsers running on mobile devices, are large, control-flow heavy applications. Modern CPU front-ends try to keep CPU back-ends busy by employing speculative execution, micro-operation caches, and dedicated instruction caches and TLBs. However, when the front-end makes incorrect guesses about the control flow, there is usually a large penalty as the execution pipelines are flushed and instructions on the new path must be fetched and executed. Often, due to the large application code footprints and lack of code locality, the instructions have to be fetched from the L2 cache or even from memory.

Using top-down analysis [20] previous studies [11] have shown that datacenter workloads can be over 20% CPU front-end bound. Feedback directed optimization (FDO) is effective at improving application run-time performance. It can reduce front-end stalls by improving the layout of hot code paths. It also improves runtime back-end performance through better code inlining decisions, which in turn enables other compiler optimizations. Despite its benefits, FDO is not easy to deploy. It requires a tedious dual-compilation model and representative workloads on which the profiled binary can be trained [13].

Sampling based FDO [5] has simplified the deployment process, removing the need for building a profiling binary and running synthetic workloads. In follow-up work on AutoFDO [4], the authors have shown that using LBR based profiles increases profile accuracy and improves performance. Panchenko et al. [14] proposed sampling-based profiling for post-link optimizations and they also confirm that using LBR profiles offers the best and most stable performance.

LBR [9] is a special Intel register that stores the last N taken branches in format {`from_pc`, `to_pc`} in a circular buffer. The value of N is microarchitecture dependent, but it is generally between 8 and 32 for recent Intel microarchitectures. Linux perf events can read the contents of the LBR buffer on a hardware event and store it in the events ring buffer.

While Intel architectures are common in the data centers, other architectures, including AMD Epyc and ARM, are gaining popularity. In addition, ARM architectures are

most common for mobile platforms. Neither AMD nor ARM have an equivalent LBR profiling feature at the time of this writing. Both platforms have support for instruction tracing, but these approaches come with a higher memory and CPU overhead and the hardware support is usually turned off on production machines.

In this paper, we present an architecture neutral software branch tracing tool that leverages hardware breakpoints, hardware counters and limited instruction decoding to sample short trace bursts of taken branches. Hardware performance counters and debug registers are widely available on all modern architectures. Both can be programmed in a portable way through the Linux perf events API.

Debug registers [9] are privileged resources that can be programmed to enable various debug conditions associated with a set of addresses, such as setting program breakpoints or watching reads and writes on a given memory address. Unlike PMU interrupts that can suffer from instruction skew, debug register interrupts are synchronous with respect to the instruction that triggers a breakpoint condition.

Instruction decoding has no hardware restrictions, but most decoding libraries have support for a limited number of instruction set architectures (ISAs). The proposed tracing tool can interface with multiple decoding libraries as needed.

The rest of this paper is organized as follows. Section 2 describes the architecture of the branch tracing profiler. Section 3 analyzes the runtime overhead of having the profiler turned on with the SPEC benchmarks. Section 4 compares the similarity of the FDO profiles produced by our profiler against hardware LBR based profiles, and computes the speedups achieved with each profile type on the SPEC benchmarks. Section 5 reviews existing control flow profiling techniques and other sampling tools that use debug registers to enable analyses not possible with the PMU alone. Section 6 summarizes our findings and concludes the paper.

2 Software Based Branch Tracing

Modern production fleets are very large, so even small overheads can add up to significant infrastructure costs. A primary requirement of our software based profiler is to have zero runtime overhead when the profiler is off. This excludes the use of any type of program instrumentation, which has a non-zero cost even when profiling is disabled. Instead, we use sampling on perf events to select a random starting point for tracing, and we use look-ahead instruction decoding and hardware breakpoints to advance through the control flow of the application for a short burst of taken branches.

2.1 High Level Design

Algorithm 1 shows the main steps of the profiler. The `SamplingHandler` is invoked on a perf events sample. It represents the starting point of a profile trace. We configured perf events to issue an interrupt on the first sample taken and

Algorithm 1 Software branch tracing algorithm.

```
1:  thread_local branch
2:  function StartProfiler(ucontext)
3:      for all threads do
4:          EnablePerfSampling(event, period)
5:      end for
6:  end function

7:  function StopProfiler
8:      for all threads do
9:          DisablePerfSamplingAndBreakpoints()
10:     end for
11: end function

12: function SamplingHandler(ucontext)
13:     pc ← GetPC(ucontext)
14:     if FindNextUnresolvedBranch(pc, ucontext) then
15:         SetBreakpoint(branch.Address())
16:     else
17:         EnablePerfSampling(event, period)
18:     end if
19: end function

20: function BreakpointHandler(ucontext)
21:     [target, taken] ←
            RecordBranchIfTaken(branch, ucontext)
22:     if taken & LBRBufferFull() then
23:         EnablePerfSampling(event, period)
24:         return
25:     end if
26:     if FindNextUnresolvedBranch(target, ucontext) then
27:         SetBreakpoint(branch.Address())
28:     else
29:         EnablePerfSampling(event, period)
30:     end if
31: end function

32: function FindNextUnresolvedBranch(pc, ucontext)
33:     repeat
34:         [branch, ok] ← decoder.FindNextBranch(pc)
35:         if !ok then return false end if
36:         if branch.RequiresBreakpoint() then
37:             return true
38:         end if
39:         [target, taken] ←
                RecordBranchIfTaken(branch, ucontext)
40:         if taken & LBRBufferFull() then
41:             return false
42:         end if
43:         pc ← target
44:     until false
45: end function

46: function EnablePerfSampling(event, period)
47:     if buffer.available() < maxLBRSize then
48:         SwapBuffer()
49:     end if
50:     PerfEventsEnable(event, period)
51: end function
```

to not rearm itself. On the entrance to `SamplingHandler`, perf events sampling is disabled. It is explicitly rearmed by the profiler after the required number of taken branches is collected.

We make no assumptions about the event used for sampling, and there is no requirement to use precise events if available. The PMU sampling is used purely to select uniformly distributed starting points for traces along the sampled dimension. Once the profiler selects a starting address for tracing, it calls `FindNextUnresolvedBranch` to look for upcoming control transfer instructions, until it reaches a branch instruction that it cannot fully resolve statically. In the process, it may find zero or multiple control transfer instructions that it can resolve just by decoding the instruction, such as unconditional relative branches and direct function calls. We record all taken branches in the simulated LBR buffer in the usual {`from_pc`, `to_pc`} format.

We implemented a limited instruction decoder on top of the XED [10] x86 instruction decoding library. We also have a partial implementation using DynamoRio [2], which has support for more architectures. While iterating over instructions, the decoder only needs to look at the instructions' opcodes to decide if they are control transfer instructions and to determine their sizes, so that it can iterate through the instruction stream. However, once it reaches a control transfer instruction, it performs full decoding of the operands and computes a machine independent representation of the target address and any eventual condition code. The profiler can evaluate the machine independent representation using an arbitrary CPU context. This avoids the need to decode the control transfer instruction again at evaluation time inside the breakpoint handler.

When we reach a branch that we cannot resolve statically, we set a breakpoint at its address and let the program execute natively until we hit the breakpoint. Inside the breakpoint handler, we evaluate the branch condition and target, and we start looking again for upcoming branch instructions. We repeat this process until we collect the specified number of consecutive taken branches.

Figure 1 demonstrates how the profiler behaves on a sample code segment. Let's assume we get a sampling interrupt while the program is at address `cc2461`. The profiler starts decoding instruction until it finds a control transfer instruction at address `cc24ae`. We determine that the instruction is a conditional branch of type `JZ` and its target is a sum of two constant values. While the branch target can be evaluated statically, we need to set a breakpoint to understand if the branch is taken or not. When it hits the breakpoint, the profiler determines that the branch is taken and it starts parsing instructions from its target address. It eventually finds a call instruction at address `cc2541` and computes its target as an 8-byte load from address `REG13 + 0x128`. The call instruction also requires a breakpoint to resolve its target. Register names in the machine independent representation don't necessarily match the architectural names. They correspond to entries in the CPU context structure for an easier evaluation.

```
                                            Type        Target
   cc2460: push   %rbp
⇒  cc2461: mov    %rsp,%rbp
   cc2464: push   %r15
   ⋮
   cc24ab: test   %rcx,%rcx
•  cc24ae: je     cc24f8                    COND (JZ)   0x48 + 0xcc24b0
   ⋮
   cc24f8: mov    (%r14,%r13,1),%rsi
   cc24fc: movdqa -0x8f7394(%rip),%xmm0
   ⋮
   cc2532: movdqa %xmm0,0x140(%rax)
   cc2539:
   cc253a: mov    0x120(%rax),%rdi
•  cc2541: callq  *0x128(%rax)              CALL        [REG13 + 0x128] (8B)
```

Figure 1. Branch tracing example. The figure presents a code segment with control transfer instructions annotated with the machine independent representation of their branch types and targets. The red color denotes the branch attributes that cannot be resolved statically.

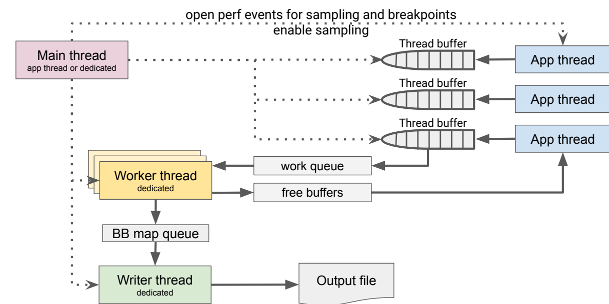


Figure 2. The branch tracing profiler diagram

Once we collect the required number of consecutive taken branches, or if we encounter any situation that we cannot handle, we finalize the current LBR buffer and restart sampling with the given event and interval. Section 2.3 describes some of the cases when we abort an existing trace early. Every time we enable sampling, we check that the local buffer has enough free space to store the simulated LBR size number of entries. The buffer is swapped out for an empty buffer if available space is insufficient. Checking for space here eliminates the need for a check every time we record an entry, and avoids ending a trace collection early due to a full buffer. The profiler can also be turned off completely. In this case, the code executes natively with no overhead.

2.2 Implementation Details

Figure 2 shows a high level diagram of the profiler architecture. The profiler may be turned on and off from an application thread or from a dedicated control thread, depending on

how the profiler is integrated with the application. On profiler start, one or more worker threads and a writer thread are created. The `StartProfiler` routine discovers all the application threads and opens perf events file descriptors for each thread, to use for sampling and setting breakpoints.

As seen in Algorithm 1, most of the thread profiling work is done inside signal handlers. To avoid deadlocks, this code must be async-signal safe. In other words, it must avoid calling any library routines that acquire locks that can also be acquired by the application code that is being profiled. For example, it should avoid allocating memory using the system allocator, which may acquire a lock on the slow path. On profile start, the control thread preallocates a pool of memory buffers where threads can write collected branch tracing data, preallocates thread specific state, and initializes the instruction decoder.

During profiling, application threads write branch data to their assigned buffers without locking. When a buffer gets full, its thread adds it to a queue of filled buffers and then grabs a free buffer from a different queue. Only the buffer swap requires synchronization with the other application threads. Buffers can be sized to accommodate an arbitrary number of LBR traces. In our experiments, we allocated two 16KB buffers for each application thread. One buffer is assigned to the thread and the second one is placed in the global free buffers pool. Each buffer can hold about one thousand branch entries with two {`from_pc`, `to_pc`} 64-bit addresses per entry, and lock contention has not been an issue even with hundreds of threads.

The worker threads grab filled buffers from the work queue, aggregate the branch traces into basic blocks, and post the processed buffers to the free queue. To reduce the size of the resulting profile, instead of saving individual traces, each worker thread maintains a map of executed application basic blocks and frequency counts. When this data structure grows above a certain threshold, it is passed on to the writer thread via a different queue. The writer thread annotates each basic block with information about the load module of origin, and writes the data out to an output profile file.

The profiler handles only application threads that exist at the time a profiling session is started. It ignores any new application threads created during the profiling session. The typical use case for profiling production applications, either inside data centers or on mobile devices in the field, is to enable the profiler for a few seconds every once is a while using an external trigger. Another approach is to use a dedicated control thread that periodically turns the profiler on and off, to collect data from new application threads.

In `StopProfiler`, at profiling session end, worker threads process the partially filled buffers assigned to the application threads. Thus, traces from partially filled buffers, including data from terminated threads, are still accounted for.

2.3 Limitations

The profiler can observe and capture all user space control flow transfers, including C++ exception handling. However, because the profiler runs in user space, it has inherent limitations in understanding control flow changes triggered from kernel space. As a result, it won't follow execution inside asynchronous signal handlers and it cannot use breakpoints to trace execution inside restartable sequences [15], since any context switch off the CPU or an interrupt inside a restartable sequence region causes it to be aborted and possibly restarted.

We handle restartable sequences by parsing the defined critical section ranges from the special ELF section where they are described. Before setting a breakpoint, we check if the breakpoint address is inside one of the restartable sequences. If it is, we end the current LBR trace early, and enable sampling to get a new trace start address. This approach causes a small blind spot behind `rseq` regions, which end up getting traced less frequently. A better approach is to analyze the control flow inside a restartable sequence, find its exit point, and restart tracing the thread's control flow from that point. While we would still miss capturing the control flow inside restartable sequences, this omission has little impact on FDO builds. Restartable sequences are generally short, hand crafted and optimized code sequences.

Signal handlers are invoked on asynchronous events. Their execution is not deterministic with respect to surrounding code. We don't aim to trace code execution inside signal handlers or include them in FDO profiles. Most signal handlers return to the point where the handler interrupted the main program and the profiler will continue tracing as if nothing happened. However, sometimes, signal handlers can interact badly with the profiler. Assume that the profiler is tracing inside libc with a breakpoint set inside a commonly used function. If an asynchronous signal handler calls the same function, it can trigger the breakpoint we've set. Depending on what signals are masked during the handler execution, the breakpoint handler can interrupt the original handler, or its execution is delayed until we return from the first signal handler. In the first case, a check of the stack pointer can tell us if we are at the expected stack frame or if the breakpoint was triggered by a different control flow path. If the stack pointers don't match, we end the current LBR trace early and restart sampling, to avoid an infinite loop.

In the second case, when the signal used by the breakpoint handler is masked, the breakpoint handler is invoked after we return from the first signal, but the program context points to the instruction where the main program was interrupted by the original signal. We can catch such situations by verifying that the `pc` where the program was interrupted matches the breakpoint address we have set. If they don't match, we can either return from the breakpoint handler without taking

any action to have the breakpoint triggered again, or we can end the current LBR trace and restart sampling.

Finally, there may be situations where a signal handler doesn't return to the point where the main program was interrupted. In such cases, the profiler loses access to the respective application thread temporarily, as the breakpoint that the profiler set before the signal fired will not be reached. As mentioned in the previous section, the common use case in production environments is to enable the profiler for a few seconds at a time. This avoids disturbing the application execution for too long, and ensures that the profiler sees any new application threads that may be dynamically created. In `StopProfiler`, after a profiling session, we close all perf events file descriptors, which disables sampling and any active breakpoints. Next time `StartProfiler` is called, it enables tracing for all the active application threads at that time.

In the following sections, we evaluate the collection overhead and the quality of the FDO profiles produced by our profiler using the SPEC CPU2006 benchmark suite [8]. We used the SPEC benchmark suite because it provides a broad set of applications with coverage of various code patterns, without overfitting any particular ones.

3 Collection Overhead

We evaluated the runtime overhead of the branch tracing profiler on two systems, a workstation with dual socket Intel(R) Xeon(R) CPU E5-2690 v4 @ 2.60GHz (Broadwell microarchitecture), and a server with dual socket AMD EPYC 7B12 processors (Rome microarchitecture). We disabled the Turbo Boost feature to eliminate one potential source of noise. We compiled the SPEC benchmark suite using `llvm 12.0.0-rc1`, and optimization flags `-fno-strict-aliasing -O3`. We added flag `-march=zenver2` when compiling the binaries executed on the AMD Rome system.

The published AutoFDO tools [7] use the frequency of control flow transitions between consecutive basic blocks only. LBR buffer length has little impact on current FDO profile precision. The number of sampled basic blocks is more important for accuracy and is the main determinant of profiling overhead. We can control the volume of data by adjusting either the sampling frequency or the simulated LBR size. For these experiments, we simulated an LBR buffer of size 16, sampled on the branches taken event, and varied the sampling period from 500,000 to 5,000,000 taken branches.

We hooked the profiler into each SPEC benchmark at link time, using a custom wrapper for the main symbol. The wrapper starts the profiler and registers an exit handler to stop the profiling session. Our experiments are limited to the SPEC benchmarks written in `C`.

We used the reference input sets for all SPEC benchmarks. Table 1 shows the runtime overhead of each benchmark measured over 5 iterations. We observe that a few benchmarks, including `433.milc`, `470.lbm` and `429.mcf` have low overheads even at higher sampling rates, on both microarchitectures. We will look at the application characteristics that impact tracing runtime overhead in Section 3.1. For all the other benchmarks, the overhead scales almost linearly with the sampling rate, the inverse of the sampling interval, as we are expecting. With a sampling interval of 5 million taken branches, the entire suite's runtime overhead is under 2% on both systems, matching our initial target.

When we look at individual benchmarks, we notice that `400.perlbench`, `462.libquantum` and `483.xalancbmk`, have large tracing overheads, and their overheads stay above 2% even at our largest tested sampling interval. We would need to increase the sampling interval by an up to 3x factor to get to our overhead target for these benchmarks, especially on the Broadwell system. For all benchmarks, but especially for the high overhead benchmarks, the AMD system shows a smaller overhead at the same sampling frequency, which we explain in Section 3.2.

3.1 Overhead Analysis

Figure 3 shows a breakdown of the running time of each benchmark on the Broadwell system, at every sampling interval. We decompose the running time of profiled application threads into application time, cost of setting up and triggering breakpoints, instruction decoding time, and cost of setting up perf events sampling. Sampling costs are negligible in all runs. In all cases, the cost of setting up and triggering breakpoints accounts for most of the overhead. Instruction decoding cost is usually much lower. These two costs are of similar magnitude only for benchmarks `433.milc` and `470.lbm`.

Breakpoint overhead is the ratio between the cost of handling breakpoints and the cost of executing the application natively. Breakpoint cost is determined by the number of branches that must be evaluated dynamically. Branch tracing is enabled while we fill in one simulated LBR length worth of entries, and then it's disabled until we get another sample.

The number of branches that must be evaluated to fill an LBR buffer is determined by *the ratio of taken branches (t)* and *the fraction of branches that can be resolved statically (s)*. Since the branch tracing profiler records only taken branches, branches that are not taken increase the overhead. In order to fill a simulated LBR buffer of size N, the profiler needs to analyze $\frac{N}{t}$ branches, where t is the fraction of taken branches. As described in Section 2, some of the branches can be statically resolved at decode time. A branch that can be resolved statically doesn't contribute to the breakpoint cost.

Figure 4 plots the fraction of taken branches (t) and the fraction of static branches in the binary (s) for each of the SPEC benchmarks and for the entire SPEC 2006 benchmark suite. Currently, the statically resolved branches are either unconditional relative branches or direct function calls. They

Table 1. Branch tracing runtime overhead with different sampling intervals. Results averaged over 5 iterations.

Benchmark Name	Intel Xeon(R) E5-2690 - Sampling Interval 500k	1000k	2000k	4000k	5000k	AMD EPYC 7B12 - Sampling Interval 500k	1000k	2000k	4000k	5000k
400.perlbench	27.17%	15.08%	7.17%	3.64%	3.03%	21.60%	10.80%	5.82%	3.21%	3.55%
401.bzip2	15.01%	9.07%	3.42%	1.80%	1.48%	13.14%	6.60%	3.33%	1.69%	1.28%
403.gcc	20.23%	11.88%	5.64%	4.18%	3.08%	17.39%	7.47%	4.41%	1.90%	0.98%
429.mcf	6.24%	5.46%	0.84%	3.37%	0.44%	5.26%	2.48%	2.03%	1.04%	1.11%
433.milc	2.70%	2.38%	1.60%	3.36%	2.00%	3.79%	1.99%	1.47%	1.29%	0.52%
444.namd	8.69%	4.40%	2.30%	1.13%	0.96%	9.37%	4.63%	2.75%	1.39%	1.10%
445.gobmk	15.89%	8.13%	3.94%	2.23%	1.90%	13.51%	6.96%	4.17%	2.11%	1.67%
447.dealII	19.95%	10.62%	5.35%	2.90%	2.37%	15.60%	9.04%	4.22%	2.08%	1.52%
450.soplex	11.93%	5.54%	2.42%	2.61%	0.49%	10.53%	6.20%	3.62%	2.06%	1.76%
453.povray	23.48%	13.75%	8.69%	3.82%	2.90%	17.18%	8.58%	6.17%	2.26%	1.60%
456.hmmer	13.91%	5.40%	1.46%	-0.41%	-0.76%	10.24%	5.71%	2.75%	1.40%	1.38%
458.sjeng	21.13%	10.48%	5.32%	2.80%	2.38%	14.11%	7.14%	5.48%	2.35%	1.54%
462.libquantum	25.58%	15.79%	11.62%	8.38%	5.16%	20.19%	10.85%	6.42%	3.38%	2.81%
464.h264ref	16.23%	8.52%	4.75%	2.10%	1.84%	13.93%	7.46%	4.02%	2.45%	1.90%
470.lbm	2.35%	0.74%	0.61%	2.27%	0.16%	3.83%	2.02%	1.02%	0.52%	0.38%
471.omnetpp	13.66%	7.93%	4.65%	3.25%	2.39%	9.08%	6.16%	5.44%	2.94%	2.84%
473.astar	11.49%	6.33%	5.11%	1.86%	1.03%	9.04%	4.80%	3.02%	1.73%	1.11%
482.sphinx3	15.32%	8.33%	4.12%	2.81%	2.26%	16.13%	8.10%	4.48%	2.95%	1.94%
483.xalancbmk	50.09%	25.82%	13.86%	7.97%	5.71%	30.90%	15.88%	8.96%	4.78%	3.70%
Geomean	13.65%	7.45%	3.71%	2.98%	1.65%	11.78%	6.17%	3.73%	1.97%	1.49%

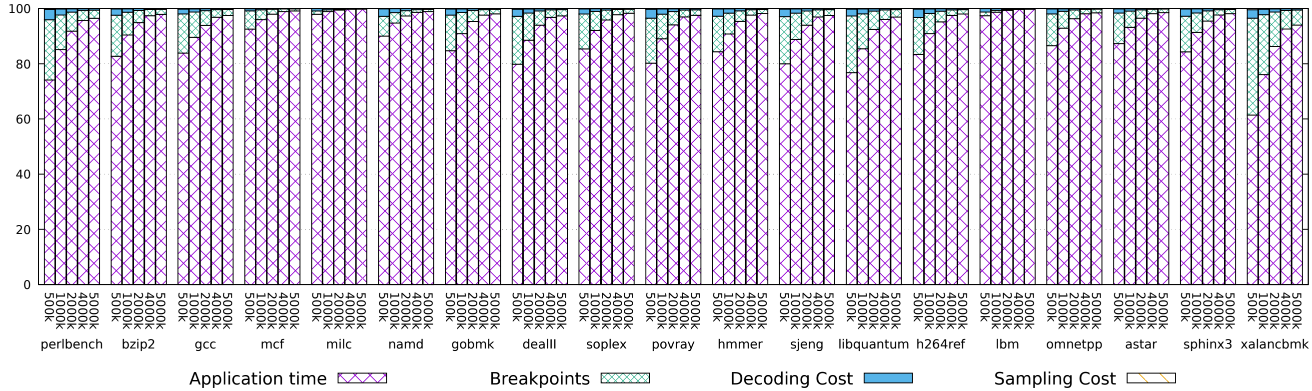


Figure 3. Breakdown of application threads running time during tracing into application time and overheads, including breakpoint handling, instruction decoding, and sampling costs. Data collected on the Intel(R) Xeon(R) CPU E5-2690 system.

are always taken. The ratio of taken branches shown in Figure 4 is computed only for branches that require dynamic evaluation. The profiler needs to evaluate $\frac{N(1-s)}{t}$ branches dynamically to fill an LBR buffer of size N.

We call $\frac{t}{1-s}$ the *breakpoint usefulness factor*. It represents the number of taken branches that can be recorded for each used breakpoint, and is shown in Figure 4 for each benchmark and for the entire benchmark suite. The higher the fraction of taken branches or the fraction of branches that can be statically resolved, the higher the breakpoint usefulness factor and the less breakpoints we need to handle.

The three metrics shown in Figure 4 are characteristics of the application binary. They are machine independent, but may be affected by compiler optimizations. They also don't change with the sampling rate. The sampling rate only impacts the absolute number of branches that we trace and record, and breakpoint cost scales linearly with it. We see its effect on the metrics shown in Table 1 and Figure 3.

If we sample on the taken branches event, a small ratio of taken branches reduces the frequency of the tracing by the same factor the breakpoint usefulness metric is reduced. This effectively cancels the effect of this metric on the overhead. However, we cannot always sample on the taken branches

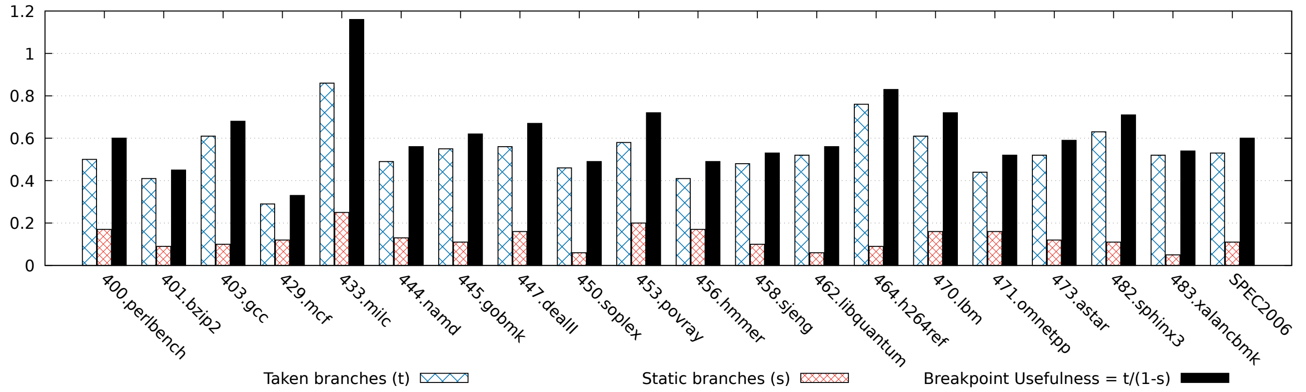


Figure 4. Fractions of taken branches (t) and statically resolved branches (s), and the breakpoint usefulness factor ($\frac{t}{1-s}$) for each benchmark.

event. Some microarchitectures don't provide such a performance monitoring event.

Compiler optimizations that reduce the number of taken branches still impact the shape of the data collected even if the profiling overhead is not impacted. A high fraction of taken branches results in many small basic blocks, while a small fraction of taken branches results in fewer larger basic blocks being collected. Here, we use basic block to mean a sequence of fall-through code between two taken branches.

The application execution cost, which is the denominator in the breakpoint overhead formula, is impacted by the fraction of non-branch instructions in the application or the *average basic block size*, and by the *application IPC rate*. Long basic blocks and low application IPC rates reduce the execution frequency of control transfer instructions, and therefore the observed slowdown during profiling.

Based on the metrics in Figure 4, the SPEC 2006 benchmark suite has a breakpoint usefulness factor of 0.6. Looking at individual benchmarks, `433.milc` is the only benchmark with a breakpoint usefulness factor above 1.0. This means that `433.milc` can record 16 consecutive taken branches using only 13.8 breakpoints on average. Benchmark `433.milc` has both a high ratio of taken branches, 86%, and a relatively high fraction of branches that can be resolved statically, at 25%, and these metrics explain in part why it has a low runtime overhead with branch tracing even at higher sampling rates. At the other end of the spectrum is benchmark `429.mcf`, which has the lowest breakpoint usefulness factor at 0.33. It needs to set three breakpoints in order to record one taken branch. While `429.mcf` doesn't have a particularly large overhead in absolute terms, see Table 1, breakpoint costs are a higher fraction of its total runtime overhead than for any other benchmark except `483.xalancbmk`. The explanation for `429.mcf`'s low runtime overhead under branch tracing is that it has a low rate of branches retired per second due to a low IPC [21].

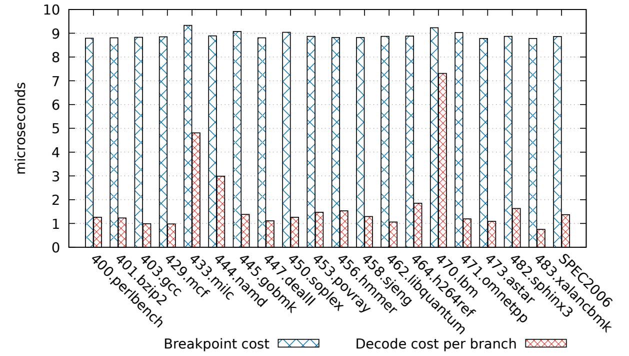


Figure 5. Cost per breakpoint and instruction decode cost per basic block, in microseconds, on the Broadwell system.

Figure 5 shows two normalized runtime costs for each benchmark: (1) the average cost of setting up and triggering a breakpoint, and (2) the instruction decoding cost per traced branch, including both taken and not taken branches. Both metrics were collected on the Intel Broadwell system and are shown in microseconds. Breakpoint cost is more a measure of system performance. We expect the breakpoint costs to be similar for every application, and Figure 5 confirms this expectation. On our Broadwell system, it takes about $9\mu s$ to set up and trigger one breakpoint. Our measurement calipers for breakpoint cost include the time from setting up a breakpoint and until we receive the signal handler. This interval also includes the time taken by the processor to execute the instructions up to the breakpoint address, and this time is larger for applications with larger basic blocks and a higher fraction of static branches, which explains the slightly larger values measured for benchmarks `433.milc` and `470.lbm`.

The normalized instruction decode cost is a measure of average basic block size for each benchmark. The larger the

block size, the more time we spend in instruction decoding before we find a control transfer instruction. We notice that benchmarks `470.lbm` and `433.milc` have significantly larger basic blocks, which contributes to their low runtime overhead under branch tracing.

Benchmark `483.xalancbmk` has short basic blocks as seen in Figure 5, a small fraction of branches that can be resolved statically as seen in Figure 4, and a 52% fraction of taken branches. Unlike `429.mcf`, it has a high branch retirement rate, the highest of all the benchmarks, and all these factors contribute to it having the highest runtime overhead under branch trace profiling, on both test machines.

3.2 Breakpoint Performance Considerations

Breakpoint costs dominate the profiler's runtime overhead. In this section, we try to understand how these costs change and scale on different architectures. For this, we wrote a microbenchmark that repeatedly sets program breakpoints and triggers them, using a variable number of program threads. Results are aggregated in Figure 6.

The two charts show the cost of setting up a breakpoint, which includes updating the breakpoint address and enabling the breakpoint, and the total cost of using a breakpoint, which includes the cost of triggering a breakpoint in addition to the previous costs. All costs are shown in microseconds and measured using CPU thread timers and all axes are shown on a logarithmic scale. The cost of triggering breakpoints was measured using an empty signal handler.

Table 2. Experimental platforms for breakpoint cost results.

Microarchitecture	ISA	sockets	hardware threads
Intel Broadwell	x86-64	2	56
Intel Cascade Lake	x86-64	2	112
AMD Rome	x86-64	1	128
AMD Rome	x86-64	2	256
Intel Skylake	x86-64	2	112
Marvell Thunder X2	ARM64	2	256

We measured breakpoint costs on six platforms based on five different microarchitectures with different numbers of hardware threads, as seen in Table 2.

Figure 6a shows that the cost of setting up a breakpoint goes up slightly with the number of threads. In the single threaded case, the breakpoint setting cost ranges from 0.9μs on the single socket AMD Rome machine up to 3.3μs on the Thunder X2. With 400 threads, this cost goes up to 1.3μs on the two AMD Rome machines and 7.1μs on the Thunder X2.

The total breakpoint cost, shown in Figure 6b, which includes the cost of triggering the breakpoint, shows a very different behavior under contention. For one to four threads, total breakpoint cost goes up only slightly. After that, as we increase the number of threads, the CPU time of setting and triggering a breakpoint goes up linearly with the number of threads up to the total number of hardware threads on the machine, after which it plateaus. The linear cost increase with the number of threads, suggest the presence of lock contention in the kernel code dealing with signal delivery.

Table 3. Profiler overhead for different sampling periods with datacenter application with 400+ threads.

Sampling period	Application time	Breakpoints	Decoding cost	Sampling
500k	53.77%	44.64%	1.46%	0.12%
1000k	87.62%	11.00%	1.27%	0.11%
2000k	97.06%	2.22%	0.66%	0.06%
5000k	98.88%	0.81%	0.28%	0.03%

The total breakpoint cost is relevant for the branch tracing profiler, as the profiler needs to both setup and trigger a breakpoint on each branch evaluation. We observed the effect of breakpoint contention on a datacenter application with several hundred threads, where the profiler's runtime overhead scales super linearly with the sampling rate. The data in Table 3 was measured on a dual socket Intel Skylake server with 112 hardware threads.

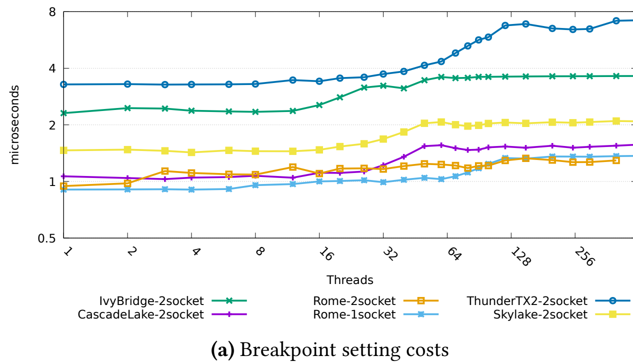
Figure 6b shows that in a low threading environment, the two socket Rome system has lower total breakpoint cost than the Broadwell system, which explains the lower branch tracing profiling overheads observed on the Rome system with the single threaded SPEC benchmarks.

3.3 Profiler Optimizations

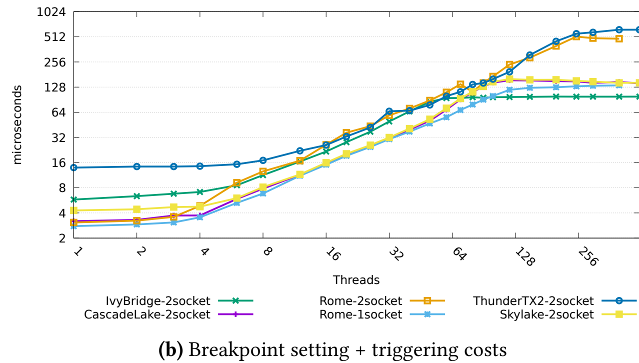
The profiler achieves a runtime overhead of $< 2\%$ on the SPEC 2006 benchmark suite as a whole when we sample every 5 million taken branches. These results were achieved with a rather naive implementation. Several high level optimizations are possible that can drastically reduce the profiler overhead in many cases.

The analysis in Section 3.1 shows that most of the runtime overhead is caused by the use of breakpoints to resolve the direction and the target of control transfer instructions, while the instruction decode cost is one order of magnitude lower on average. To lessen the frequency at which we need to set breakpoints, we can employ data flow analysis to predict the direction and target of several branches at a time, using the CPU state and memory content available inside the breakpoint handler. Such improvements can be implemented incrementally. The profiler can fall back to dynamic evaluation via breakpoints when the evaluation requires unsafe assumptions. This optimization would help especially with applications that have tight, frequently executed loops, where the loop exit condition can be reasoned about using data flow analysis.

Second, the profiler currently doesn't cache the results of instruction decoding. It decodes every instruction each



(a) Breakpoint setting costs



(b) Breakpoint setting + triggering costs

Figure 6. Breakpoint costs under contention on different microarchitectures

time it executes a basic block, even when these instructions are in a tight loop. An instruction cache needs to save only the machine independent representation of branches and the start addresses of their basic blocks. The call to *decoder*.`FindNextBranch(pc)` at line 34 in Algorithm 1 will first query the cache for a basic block that includes the *pc* address and return the machine independent representation of the branch from the cache on a hit.

The impact from caching on overhead would be small with the current implementation where breakpoints account for the bulk of the overhead. However, with the first optimization in place, caching the decoded instructions and the results of the data flow analysis would become much more important. We believe that these two high level optimizations have the potential to lower the profiler's runtime overhead by integer factors.

4 Profile Similarity

To evaluate the representativeness of the LBR data collected by our software based branch tracer and to understand its robustness with respect to the sampling interval, we used the published AutoFDO tools [7] to generate FDO profiles using the data collected at each sampling interval. We compare the resulting profiles against a profile generated from data collected using the perf tool and the hardware LBR available on the Intel system. We used sampling on the taken branches event and a sampling interval of 1 million taken branches to collect the Intel LBR data with perf.

We used the `profile_diff` tool at [7] to compare the software based profiles collected at each sampling rate against the hardware based profile for each benchmark. `profile_diff` computes the overlap between two profiles taking into account the relative contribution of each symbol in the profile, with scores in the range [0, 1]. Results for the Intel system are aggregated in Table 4 and results for the AMD system are shown in Table 5.

First, we notice the profile similarity scores are insensitive to the sampling period. This bodes well for our approach.

Table 4. AutoFDO profile similarity between Intel LBR and software tracing with different sampling intervals on Intel.

Benchmark	500k	1000k	2000k	4000k	5000k
400.perlbench	85.91%	85.90%	86.33%	85.77%	86.32%
401.bzip2	99.31%	99.47%	99.55%	99.38%	99.55%
403.gcc	99.50%	99.49%	99.44%	99.45%	99.52%
429.mcf	96.30%	96.33%	96.48%	96.78%	96.17%
433.milc	98.49%	98.41%	98.46%	98.65%	98.78%
444.namd	99.29%	98.93%	99.12%	98.96%	99.24%
445.gobmk	95.56%	95.41%	95.25%	95.22%	95.25%
447.dealII	97.00%	97.44%	97.11%	97.14%	96.89%
450.soplex	99.05%	99.16%	99.15%	98.89%	98.48%
453.povray	92.84%	92.82%	92.83%	92.84%	92.88%
456.hmmer	99.73%	99.76%	99.73%	99.76%	99.77%
458.sjeng	95.36%	95.38%	95.34%	95.30%	95.44%
462.libquantum	99.65%	99.72%	99.72%	99.67%	99.66%
464.h264ref	97.23%	97.09%	97.20%	97.20%	97.29%
470.lbm	99.88%	99.88%	99.88%	99.80%	
471.omnetpp	91.11%	91.05%	91.01%	90.98%	91.00%
473.astar	97.84%	97.79%	97.75%	97.56%	97.62%
482.sphinx3	98.01%	98.00%	98.02%	98.18%	97.52%
483.xalancbmk	95.03%	95.20%	95.18%	95.10%	95.17%
Geomean	96.62%	96.63%	96.65%	96.60%	96.41%

It means that lowering the sampling rate by a factor of 10 doesn't lower the representativeness of the profile. The lack of sensitivity may be due to the large amount of data available. For each profile, we used the sampling data from five iterations of the `ref` input size. This volume of data is not difficult to collect for a datacenter application running on a production fleet, or for an application running on a fleet of mobile devices, even at much lower sampling rates.

Second, similarity scores with the hardware based profiles are high for both systems, with a geometric mean of 96.6% for the software based profiles collected on the Intel system, and a geometric mean of 91.2% for the software based profiles collected in the AMD system.

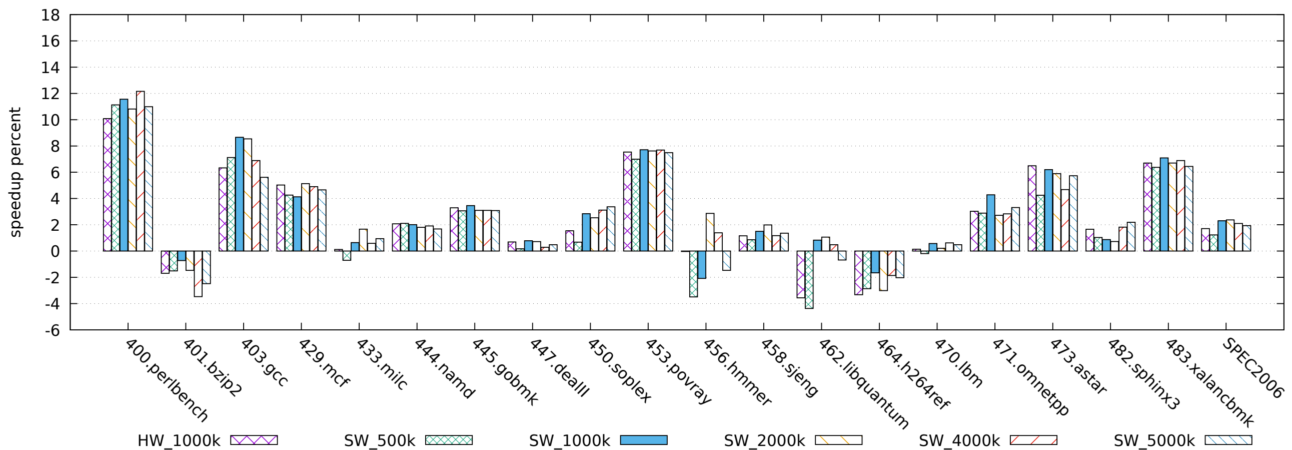


Figure 7. Speedup of FDO compiled binaries on the Intel Broadwell system. Results averaged over 5 iterations.

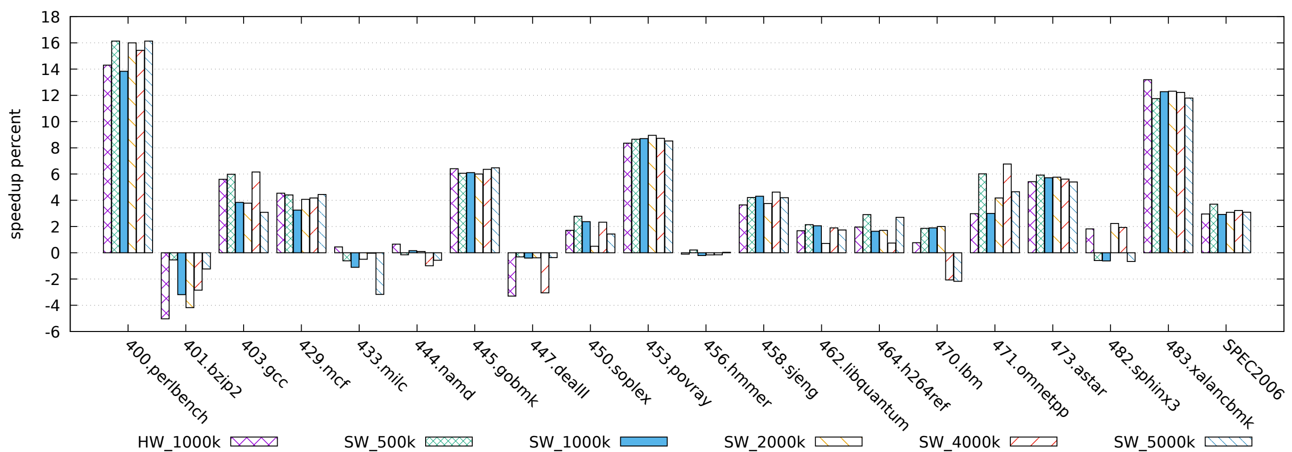


Figure 8. Speedup of FDO compiled binaries on the AMD Rome system. Results averaged over 5 iterations.

Third, similarity scores between the hardware LBR profiles and the software LBR profiles collected on the AMD system are lower than the similarity scores computed for the software based profiles collected on the Intel system. This shows that there are some runtime differences between the executions on the two systems, even for a benchmark suite like SPEC CPU. Out of all the profiles, we notice that the software based profiles for `453.povray` and `471.omnetpp` collected on the AMD system have the lowest similarity with the hardware LBR profiles collected on the Intel system.

These profile differences may not necessarily translate into performance differences. We know that the relative ranking of symbols and their contribution being over or under a certain threshold is more important to compiler decisions than the precise number of sample or the exact fraction of samples attributed to a symbol. As a result, an important validation test is to compare the speedups achieved by FDO compilation using the software based profiles relative to the speedups obtained using the hardware based profiles.

Figure 7 shows the average speedups of the FDO compiled benchmarks on the Intel system. Figure 8 shows the corresponding speedups observed on the AMD Rome system. For each system, we compare the hardware based profile from the Intel system and the software based profiles collected on the same system where the speedups are measured. Results are averaged over 5 iterations with the `ref` input size.

Not all benchmarks benefit from FDO compilation. FDO helps the most to reduce front-end stalls, but not all the benchmarks have a significant front-end bottleneck. Benchmarks that benefit the most are `400.perlbench`, `453.povray` and `483.xalancbmk`. Other benchmarks, `403.gcc`, `429.mcf`, `445.gobmk`, `471.omnetpp`, `473.astar`, also benefit to some degree from FDO compilation. The remaining benchmarks show more runtime noise, and some, like `401.bzip2` seem to be negatively impacted by FDO.

Table 5. AutoFDO profile similarity between Intel LBR and software tracing with different sampling intervals on AMD.

Benchmark	500k	1000k	2000k	4000k	5000k
400.perlbench	87.71%	87.67%	87.70%	87.32%	86.98%
401.bzip2	97.91%	97.98%	97.66%	97.98%	97.98%
403.gcc	98.90%	98.84%	98.96%	98.97%	99.13%
429.mcf	90.60%	90.43%	90.07%	91.55%	90.55%
433.milc	98.72%	98.57%	98.49%	98.61%	98.64%
444.namd	95.91%	95.64%	95.71%	95.68%	95.78%
445.gobmk	85.68%	86.04%	85.62%	85.86%	85.26%
447.dealII	92.27%	92.40%	92.57%	92.88%	92.56%
450.soplex	98.32%	98.12%	98.24%	98.27%	97.79%
453.povray	74.00%	74.30%	74.46%	73.65%	74.19%
456.hmmer	99.84%	99.82%	99.81%	99.80%	99.84%
458.sjeng	88.67%	88.84%	88.45%	88.77%	88.76%
462.libquantum	98.03%	98.05%	98.09%	98.12%	98.04%
464.h264ref	90.57%	90.54%	90.57%	90.51%	90.38%
470.lbm	99.74%	99.76%	99.70%	99.77%	99.55%
471.omnetpp	73.22%	73.54%	72.73%	72.95%	73.05%
473.astar	95.84%	95.69%	96.17%	96.22%	96.10%
482.sphinx3	92.53%	92.74%	92.48%	92.09%	92.52%
483.xalancbmk	81.85%	82.00%	82.06%	82.08%	81.99%
Geomean	91.22%	91.27%	91.18%	91.26%	91.16%

We notice that the software based profiles achieve a similar speedup as the hardware based profile across the entire benchmark suite, but there is some variability when we look at individual benchmarks, especially the ones that don't benefit from FDO that much.

The two benchmarks with the lowest similarity scores on the AMD system, `453.povray` and `471.omnetpp`, achieve marginally better speedups with the software based profiles, but the difference is very small and may very well be in the noise. For all intents and purposes, the software based profiles seem no worse than the hardware based ones for the SPEC CPU 2006 benchmarks.

5 Related Work

There is a long history of control flow profiling tools in the research literature. Ball and Larus [1] propose an efficient instrumentation based path profiler that minimizes runtime overhead by carefully placing counters on select CFG edges. Despite the optimal counter placement, instrumentation based methods still have a high runtime overhead, 31% for path profiling and 16% for edge profiling. Traub et al. [18] use sampling and program instrumentation hooks that they can use to turn instrumentation on and off, to capture a small and fixed number of branch executions with relatively low overhead. Their approach is not that dissimilar to ours, but we avoid instrumentation and code rewriting at runtime, by using instruction decoding and breakpoints instead, which provide more reliability, better platform portability, and generally a lower overhead.

Program sampling and debug registers have been previously used to implement low overhead correctness and profiling tools with good results. Pesterev et al. [16] use debug registers to report code locations responsible for cache misses on the same data, Erickson et al. [6] use sampling and debug registers to find data races with low overhead. DoubleTake [12] uses debug registers to watch for accesses to memory locations that were found to have been unexpectedly modified, for precise error reporting. WITCH [19] uses sampling and debug registers to find software inefficiencies such as dead writes and redundant loads. The authors use similar techniques to detect false sharing [3] with low overhead. Pengfei et al. [17] use sampling on function calls and breakpoints on return addresses for precise monitoring and variance profiling at procedure boundaries.

6 Conclusions

Sampling-based FDO methods are commonly used in datacenter applications to attain significant performance savings. These methods rely on hardware facilities like Intel's LBR to profile applications' control flows with low overhead in production environments. Sampling FDO can also benefit large mobile applications. However, LBR based profiling is not yet available on popular mobile CPUs, which limits the use of sampling-based FDO methods on these systems.

In this paper, we present a software based branch tracing profiler that collects LBR-like data from unmodified application binaries. The profiler uses hardware counters based sampling to select random starting addresses for tracing in application threads, uses look-ahead instruction decoding to identify upcoming control transfer instructions, and uses program breakpoints to stop application threads at branches that require dynamic evaluation of their directions or targets. We've performed a detailed analysis of the profiler's runtime overhead with the SPEC CPU 2006 benchmarks. The profiler has zero overhead when it is off. When enabled, we've shown that we can reduce its runtime overhead to arbitrarily low values by varying the sampling rate, achieving a $< 2\%$ overhead across the entire SPEC benchmark suite. Most of the overhead can be attributed to the use of breakpoints to stop application threads on branches that require dynamic evaluation. We have identified additional high level optimizations that can lower the profiler overhead further, by reducing the frequency with which breakpoints are used.

We've shown that software based FDO profiles perform equally well to FDO profiles generated from hardware LBR data when optimizing the SPEC benchmarks on two different systems. This work enables the collection of sampling-based FDO profiles in live production environments from fleets of non-Intel devices, such as mobile devices.

Finally, software branch tracing is not bound to a fixed LBR hardware size. In future work, we will explore if longer traces can enable additional FDO optimizations.

References

[1] Thomas Ball and James R. Larus. 1996. Efficient Path Profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture* (Paris, France) *(MICRO 29)*. IEEE Computer Society, USA, 46–57.

[2] D. Bruening, T. Garnett, and S. Amarasinghe. 2003. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* 265–275.

[3] Milind Chabbi, Shasha Wen, and Xu Liu. 2018. Featherlight On-the-Fly False-Sharing Detection. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Vienna, Austria) *(PPoPP '18)*. Association for Computing Machinery, New York, NY, USA, 152–167. https://doi.org/10.1145/3178487.3178499

[4] Dehao Chen, Tipp Moseley, and David Xinliang Li. 2016. AutoFDO: Automatic feedback-directed optimization for warehouse-scale applications. In *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 12–23.

[5] Dehao Chen, Neil Vachharajani, Robert Hundt, Shih-wei Liao, Vinodha Ramasamy, Paul Yuan, Wenguang Chen, and Weimin Zheng. 2010. Taming Hardware Event Samples for FDO Compilation. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Toronto, Ontario, Canada) *(CGO '10)*. Association for Computing Machinery, New York, NY, USA, 42–52. https://doi.org/10.1145/1772954.1772963

[6] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. 2010. Effective Data-Race Detection for the Kernel. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada) *(OSDI'10)*. USENIX Association, USA, 151–162.

[7] Google. [n.d.]. AutoFDO. ([n. d.]). https://github.com/google/autofdo [5 March 2021].

[8] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News* 34, 4 (Sept. 2006), 1–17. https://doi.org/10.1145/1186736.1186737

[9] Intel. 2016. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3B.* Intel Corporation.

[10] Intel Corporation. [n.d.]. XED. ([n. d.]). https://intelxed.github.io/ [5 March 2021].

[11] S. Kanev, J. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G. Wei, and D. Brooks. 2016. Profiling a Warehouse-Scale Computer. *IEEE Micro* 36, 03 (may 2016), 54–59. https://doi.org/10.1109/MM.2016.38

[12] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. 2016. DoubleTake: Fast and Precise Error Detection via Evidence-Based Dynamic Analysis. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 911–922. https://doi.org/10.1145/2884781.2884784

[13] LLVM. [n.d.]. Profiling with Instrumentation. ([n. d.]). https://clang.llvm.org/docs/UsersManual.html#profiling-with-instrumentation [5 March 2021].

[14] M. Panchenko, R. Auler, B. Nell, and G. Ottoni. 2019. BOLT: A Practical Binary Optimizer for Data Centers and Beyond. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14.

[15] Paul Turner and Andrew Hunter. [n.d.]. *LPC - PerCpu Atomics.* Linux Plumbers Conference. http://www.linuxplumbersconf.net/2013/ocw//system/presentations/1695/original/LPC-PerCpuAtomics.pdf

[16] Aleksey Pesterev, Nickolai Zeldovich, and Robert T. Morris. 2010. Locating Cache Performance Bottlenecks Using Data Profiling. In *Proceedings of the 5th European Conference on Computer Systems* (Paris, France) *(EuroSys '10)*. Association for Computing Machinery, New York, NY, USA, 335–348. https://doi.org/10.1145/1755913.1755947

[17] Pengfei Su, Shuyin Jiao, Milind Chabbi, and Xu Liu. 2019. Pinpointing Performance Inefficiencies via Lightweight Variance Profiling. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 19, 19 pages. https://doi.org/10.1145/3295500.3356167

[18] Omri Traub, S. Schechter, and M. Smith. 2000. *Ephemeral Instrumentation for Lightweight Program Profiling.* Technical Report. Department of Electrical Engineering and Computer Science, Hardward University, Cambridge, Massachusetts.

[19] Shasha Wen, Xu Liu, John Byrne, and Milind Chabbi. 2018. Watching for Software Inefficiencies with Witch. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 332–347. https://doi.org/10.1145/3173162.3177159

[20] A. Yasin. 2014. A Top-Down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 35–44.

[21] D. Ye, J. Ray, C. Harle, and D. Kaeli. 2006. Performance Characterization of SPEC CPU2006 Integer Benchmarks on x86-64 Architecture. In *2006 IEEE International Symposium on Workload Characterization.* 120–127.