Watching for Software Inefficiencies with Witch

Shasha Wen
College of William and Mary, USA
swen@cs.wm.edu

Xu Liu
College of William and Mary, USA
xl10@cs.wm.edu

John Byrne
Hewlett Packard Labs, USA
john.l.byrne@hpe.com

Milind Chabbi
Baidu Research, USA and
Scalable Machines Research, USA
milind@ScalableMachines.org

Abstract

Inefficiencies abound in complex, layered software. A variety of inefficiencies show up as wasteful memory operations. Many existing tools instrument every load and store instruction to monitor memory, which significantly slows execution and consumes enormously extra memory. Our lightweight framework, Witch, samples consecutive accesses to the same memory location by exploiting two ubiquitous hardware features: the performance monitoring units (PMU) and debug registers. Witch performs no instrumentation. Hence, witchcraft—tools built atop Witch—can detect a variety of software inefficiencies while introducing negligible slowdown and insignificant memory consumption and yet maintaining accuracy comparable to exhaustive instrumentation tools. Witch allowed us to scale our analysis to a large number of code bases. Guided by witchcraft, we detected several performance problems in important code bases; eliminating these inefficiencies resulted in significant speedups.

CCS Concepts • **General and reference** → **Performance**; • **Software and its engineering** → **Runtime environments**; *Application specific development environments*;

Keywords Software inefficiency detection, PMU, debug registers, sampling, profiling

ACM Reference Format:
Shasha Wen, Xu Liu, John Byrne, and Milind Chabbi. 2018. Watching for Software Inefficiencies with Witch. In *ASPLOS '18: 2018 Architectural Support for Programming Languages and Operating Systems, March 24–28, 2018, Williamsburg, VA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3173162.3177159

Work done while at Hewlett Packard Labs.

1 Introduction

Large, layered, production software is complex due to a hierarchy of component libraries and sophisticated control flow. Even the high-performance computing (HPC) software achieves only 5-15% of peak performance on modern supercomputers [15, 17, 63]. Inefficiencies inherent in complex software [8, 30, 32, 35, 71, 76, 84, 85] significantly contribute to this abysmal performance. Software inefficiencies may arise during design (e.g., inappropriate choice of algorithms and data-structures), implementation (e.g., developers' inattention to performance and use of heavyweight APIs), or translation (e.g., detrimental compiler optimizations and lack of tuning for an architecture).

Inefficiencies, whatever their origin, often manifest as computations whose results may not be used [4, 69], recomputation of already computed values [83], unnecessary data movement [8, 36, 48, 50, 82], and excessive synchronization [5, 78]. Inefficiencies involving the memory subsystem are particularly egregious because of limited bandwidth shared by multiple cores and high access latencies. Repeated initialization, register spill and restore on hot paths, lack of inlining hot functions, missed optimization opportunities due to aliasing, computing and storing already computed or sparingly changing values, and contention and false sharing [22, 44, 46, 47] (in multi-threaded codes), are some of the common prodigal uses of the memory subsystem. Although compiler literature is rich with optimization to eliminate inefficiencies, in practice, layers of abstractions, dynamic libraries, multi-lingual components, aggregate types, aliasing, and combinatorial explosion of execution paths handicap optimizing compilers in delivering top application performance. Additionally, algorithmic and data structural deficiencies also appear as useless memory operations [8, 30, 35, 71, 84, 85].

Coarse-grained profilers such as VTune [27], HPCToolkit [1], gprof [21], Oracle Solaris Studio [66], Oprofile [65], Perf [40], and CrayPAT [14] identify execution "hotspots". They attribute measurements such as CPU cycles, stalls, arithmetic intensity, and cache misses, obtained from hardware performance monitoring units (PMUs) to

```
 1 void loop_regs_scan(struct loop *loop, ...){
 2            ...
 3 ▶ last_set=(rtx *) xcalloc(regs->num, sizeof (rtx));
 4   /* Scan the loop, recording register usage */
 5   for (each instruction in loop){
 6     ...
 7      if(GET_CODE (PATTERN (insn)) == SET || ...)
 8       count_one_set (...,last_set,...);
 9       ...
10       if (end of basic block)
11 ▶       memset(last_set,0,regs->num*sizeof(rtx));
12   } ... }
```

Listing 1. Dead stores in SPEC CPU2006 gcc due to an inappropriate data structure. The function iterates over the basic blocks in a loop scanning for the registers used. Line 3 allocates and zero initializes a 16K-element 132KB array representing the virtual registers. The loop body accesses only a few (<2) array elements since basic blocks are typically small. At the end of each basic block (Line 11) the code zero initializes the same array for the use in the next basic block. Line 11 is repeatedly involved in dead stores.

the source code. On the positive side, they introduce little runtime overhead and do not materially perturb execution. On the negative side, hotspots fail to distinguish efficient vs. inefficient resource usage. The SPEC CPU2006 [73] gcc code, shown in Listing 1, repeatedly zero initializes a 132KB array, most of which is already zero. None of these profilers detects this as wasted work. Ironically, a hotspot may have no further optimization scope (e.g., a highly optimized linear algebra library); and conversely, a code region acclaimed by a profiler with high arithmetic intensity (a goodness metric) may perform useless computations [83].

Fine-grained profilers such as DeadSpy [8], RedSpy [82], RVN [83], Toddler [60], and Cachetor [58], analyze dynamic instructions with specific objectives—detect useless computation or data movement. They can identify inefficiencies not detected by coarse-grained profilers. In Listing 1, they can pinpoint the source code location that re-initializes an already initialized array and quantify the wasted work. On the positive side, they offer visibility into wasted work. On the negative side, they significantly slow execution down (10-80×) and consume enormous (6-100×) extra memory.

Despite their effectiveness, the high overhead of fine-grained inefficiency detection tools has kept them away from wide adoption. They have remained isolated to small research communities or performance experts. There is a need to make such tools more available to the developer community so that inefficiency detection can be made commonplace—run with each code check-in to isolate inefficiencies at the earliest.

We developed Witch—a lightweight inefficiency-detection framework—to address this issue. Witch combines the best of both worlds—low overhead of coarse-grained profilers and inefficiency detection of fine-grained profilers. Our key observation is that an important class of inefficiency detection schemes, explored previously via fine-grained profilers [8, 48, 82], requires monitoring *consecutive accesses to the same memory location.* For example, detecting repeated initialization—a dead write [8]—requires monitoring store after store without an intervening load to the same location.

Witch samples addresses accessed by a program using hardware PMUs. Witch intercepts the subsequent access(es) to the sampled memory locations using hardware debug registers. The result is (1) the ability to observe consecutive accesses to the same memory location to detect myriad inefficiencies, and (2) no code or binary instrumentation and hence low overhead. We show the benefit of this concept by building various inefficiency-detection tools (witchcraft) atop Witch. There are various challenges in making it practical, which we detail and address in Section 4 and 5.

The idea generalizes to detect other kinds of inefficiencies—updating a location with a value already present at the location (aka silent store [36, 37]) and loading an unchanged value from memory that was previously loaded [3, 60, 72] (poor register usage). Sharing addresses sampled by one thread with another thread enables building Witch-based tools for multi-threaded programs. In this paper, we restrict ourselves to describing the Witch framework and three tools that detect inefficiencies in a thread of execution. We make the following contributions:

1. Develop a lightweight framework, Witch, suitable for a class of tools that requires observing a program's consecutive accesses to the same memory location.
2. Develop a sampling scheme to overcome hardware limitations, which works exceptionally well in practice.
3. Develop inefficiency-detection tools atop Witch, which are *at least an order of magnitude faster* than the state-of-the-art exhaustive-instrumentation tools with the same capabilities. Our tools require *negligible extra memory.*
4. Overcome practical challenges in implementing these tools and demonstrate the accuracy of our tools in comparison with the state-of-the-art.
5. Demonstrate the utility of our tools on large code bases to pinpoint inefficiencies and show up to 10× speedup.

Section 2 covers the related work and motivates our work; Section 3 offers the background; Section 4 overviews the design of Witch; Section 5 and 6, respectively, describe implementation details of Witch and its client witchcraft tools; Section 7 performs evaluation, Section 8 explores case studies, and Section 9 offers our conclusions.

2 Related Work and Motivation

There is a vast literature in detecting and eliminating software inefficiencies. We classify these techniques into *hardware* and *software* approaches. The hardware approaches [36, 37, 42, 43, 54, 55, 86] introduce new hardware components to detect and eliminate computations whose results are never used or elide memory operations that do not change the contents of their target memory cells. Our focus is on software approaches, which do not need any hardware modification.

Classic compiler optimizations such as value numbering [68], constant propagation [81], and common subexpression elimination [13] eliminate several inefficiencies.

Recently, static analysis has been used in detecting performance bugs [59, 64]. Static analysis, typically, suffers from limitations related to aliasing, optimization scope, and input and context insensitivities. A thorough literature review of static analysis is not pertinent.

The dynamic analysis addresses the limitation of static analysis. Chabbi and Mellor-Crummey [8] show that dead writes are a common symptom of myriad inefficiencies. Their tool, DeadSpy, tracks every memory operation to identify store operations that are never loaded (dead) before a subsequent store (kill) to the same location. DeadSpy associates pairs of instructions involved in a dead store (dead-kill pair) with their calling contexts and source code locations to guide manual optimizations. Using DeadSpy, the authors identify inefficiencies arising from inappropriate data structure choice, optimization inhibiting code shape, inattention to performance, and poor compiler code generation. They improve the performance of several systems by eliminating dead writes. DeadSpy's exhaustive monitoring typically introduces more than 28× slowdown and consumes more than 9× extra memory on average.

Wen et al. perform fine-grained monitoring to track symbolically equivalent [83] and result-equivalent [82] computations. Their tools detect inefficiencies in both CPU- and memory-bound operations. They detect inefficiencies arising from redundant computation, missed inlining opportunities, layers of abstractions, and redundant stores. With exhaustively monitoring, their tools incur 40-280× runtime overhead. By periodically enabling and disabling monitoring (bursty sampling [24]), they bring it down to a manageable 12× slowdown and 9× memory bloat.

Toddler [60] focuses on identifying repetitive memory load sequences across loop iterations at the cost of 10× slowdown. LDoctor [72] reduces Toddler's overhead using a combination of ad-hoc sampling and static analysis techniques. However, it only analyzes a small number of suspicious loops identified by profiling, and hence does not work for systematically detecting inefficiencies in the whole program.

Unlike these approaches, Witch, without the need of any prior knowledge of the program, monitors fully optimized native binaries and all their dynamic dependencies and typically incurs negligible runtime overhead ($< 5\%$) and memory overhead ($< 5\%$). Witch is the first lightweight measurement framework that employs PMUs and hardware debug registers to detect program inefficiencies. Neither the inefficiency detection nor the use of PMUs or debug registers is novel in itself, but their combined application is.

Tools Based on Hardware Debug Registers: Erickson et al. [18] use hardware debug registers [31, 51] to detect data races in the Windows kernel. Jiang et al. [29] extend it to the Linux. They sample memory access instructions and set watchpoints to detect conflicting accesses. They use code breakpoints to intercept random instructions and use them to monitor memory accesses for a time window. Liu et al. [45] developed DoubleTake, which uses debug registers to identify buffer overflow, use after free, and memory leaks. Pesterev et al. developed DProf [67], which combines PMU and hardware debug registers to capture the data flow across runtime objects. DProf suffers from limited debug registers; it runs a program multiple times to achieve higher coverage. These approaches focus on detecting the presence or absence of a bug; they are not concerned with quantifying the frequency of a bug or prioritizing the importance of a bug, which become necessary in performance analysis tools. Witch addresses these *quantification* and *attribution* problems necessary for performance tools.

Kasikci et al. [34] describe a *spatially* unbiased sampling scheme to trace cold code for code coverage. In contrast, Witch develops a *temporally* unbiased sampling scheme to monitor memory locations. Kasikci et al. dynamically rewrite the first instruction of every basic block with the `int 3` breakpoint instruction, which causes a trap; there is no hardware limit on how many blocks they can monitor. Breakpoints set in hot code regions drive their sampling, and they throttle too frequently trapping breakpoints. In contrast, Witch does not modify the binary (not even at runtime), it uses the PMU as its sampling engine, but it has to workaround the limited number of debug registers.

3 Background and Terminology

In this section, we present the background necessary to understand Witch. Expert readers may skip this section.

Hardware Performance Monitoring Units (PMU): CPU's PMUs offer a programmable way to count hardware events such as loads, stores, CPU cycles, etc. A PMU can trigger an overflow interrupt once a threshold number of events accumulate. A profiler, running in the address space of the monitored program, can handle the interrupt and attribute the measurement "appropriately". We refer to a PMU counter overflow as a "sample".

Intel SandyBridge and its successors support Precise Event-Based Sampling (PEBS) [25]. A PMU captures a snapshot of the user-visible register state including the program counter (PC) and the effective address (EA) accessed by the instruction on an event overflow. AMD Instruction-Based Sampling (IBS) [16] and PowerPC Marked Event Sampling (MRK) [75] offer commensurate capabilities.

Hardware Debug Registers: Hardware debug registers [31, 51] enable trapping the CPU execution for debugging when the PC reaches an address (breakpoint) or an instruction accesses a designated address (watchpoint). One can program debug registers with different addresses, widths, and conditions that will cause the CPU to trap on reaching the programmed conditions. Today's x86 processors have four debug registers. If used for the break-on-data-access (store, or load-or-store), on x86 processors, the trap occurs *after* the instruction execution. Hence, if a store instruction results in a trap, the contents of the target memory will contain the results of the store operation.

Linux Perf_events: Linux offers a standard interface to program and sample PMUs using the `perf_event_open` system call [39] and the associated `ioctl` calls. The Linux kernel can deliver a signal to the thread whose PMU event overflows. The user code can `mmap` a circular buffer into which the kernel keeps appending the PMU data on each sample. Linux `2.6.33` and its successors incorporate the debug registers in the `perf_event` interface, however, the support has several limitations, which we discuss and fix in our work. We implement Witch on Intel processors with the PEBS facility. It is straightforward to extend Witch to work on AMD with IBS and PowerPC with MRK.

Call Path Profiling: Call path profiling [23] is a profiling technique where runtime events (e.g., cache misses) are attributed to the full call path seen at the time of the event. Call path profiling offers insightful details in complex applications with deep call chains. The *calling context* of an event is a set of active procedure frames when the event happens. A calling context begins at a process or thread entry function such as `main` and ends at the instruction pointer (IP) of the instruction that triggers the event. The alternative, flat profiling, merely attributes events to the leaf function involved in the event, which introduces ambiguities when the same leaf function (e.g., `memset`) can be invoked from multiple contexts.

Terminology: A *watchpoint* is a software abstraction of a debug register to monitor a data access. An address is *monitored* if we set a watchpoint at that address. A watchpoint can be set to trap on write (`W_TRAP`) or trap on read-or-write (`RW_TRAP`). A *watchpoint exception* (aka trigger) is a synchronous CPU trap caused when an instruction accesses a monitored address. A *PMU sample* is a CPU *interrupt* caused when an event counter overflows. Both PMU samples and watchpoint exceptions are handled via the Linux signals.

4 Methodology and Design

We want to answer the following questions: 1) Do consecutive store operations to a memory location have an intervening load? 2) Do consecutive stores to a memory location store the same value? 3) Do consecutive loads from a memory location load the same value? 4) Is a cacheline accessed by one thread immediately accessed by another thread?

Summary: PMU samples that include the effective address accessed in a sample provide the knowledge of the addresses accessed in an execution. Given this effective address, a hardware debug register allows us to keep an eye on (watch) a location and recognize what the program subsequently does to such location. Since the hardware can monitor a small number of locations at a time, reservoir sampling [80] allows monitoring a subset of previously seen addresses without any temporal bias. Finally, we scale the measurements taken for a few monitored samples in a calling context to other unmonitored samples in the same calling context; the scaling is based on the observation that the code behavior in a calling context typically remains the same.

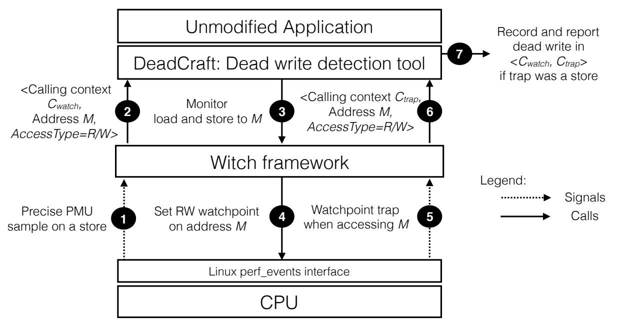

Unmodified Application
DeadCraft: Dead write detection tool
7
Record and report dead write in <C_{watch}, C_{trap}> if trap was a store
<Calling context C_{watch}, Address M, AccessType=R/W>
2
Monitor load and store to M
3
<Calling context C_{trap}, Address M, AccessType=R/W>
6
Witch framework
Legend:
Signals
Calls
Precise PMU sample on a store
1
Set RW watchpoint on address M
4
Watchpoint trap when accessing M
5
Linux perf_events interface
CPU

Figure 1. Detecting dead writes using Witch. The client, `DeadCraft`, subscribes to the precise PMU store event with a desired sample period. ① PMU counter overflows triggering an interrupt. ② Witch handles the signal, extracts the calling context (C_{watch}) of the interrupt and the address accessed (M), and offers the triplet $\langle C_{watch}, M, AccessType\rangle$ to `DeadCraft`. ③ `DeadCraft` asks Witch to monitor subsequent load or store to M. ④ Witch sets a watchpoint to monitor M, and the execution continues ⑤ Program accesses M, which causes a CPU trap. ⑥ Witch handles the trap signal, extracts the calling context (C_{trap}), and offers the triplet $\langle C_{trap}, M, AccessType\rangle$ to `DeadCraft`. ⑦ If the AccessType is a store, `DeadCraft` infers a dead write and attributes it to $\langle C_{watch}, C_{trap}\rangle$.

Details: Precise PMU samples drive Witch. Client tools subscribe to PMU events of their choice. On each PMU sample, the client obtains the memory address M accessed in the sample. Clients subscribe to a watchpoint at the sampled address in the signal handler and continue their execution.[1] When the program accesses M next time, a CPU trap happens. Witch handles the watchpoint exception, captures information associated with the trap, associates any information given by the client at the watchpoint subscribe time, and gives control to the client tool for appropriate actions.

We use our dead store detection tool—`DeadCraft`, shown in Figure 1—as a running example to illustrate our methodology. The ideas generalize to any tool built atop Witch. A store followed by another store to the same address is an instance of a dead store. A store followed by a load to the same address is not a dead store. A software instrumentation tool such as DeadSpy [8] maintains a large shadow memory where it stores the last operation performed on each byte of the original program. A write→write transition in a shadow byte indicates an instance of a dead write.

[1]A client may set a watchpoint at an address derived from the sampled address or any other address instead of the sampled address itself.

`DeadCraft` mimics the behavior of DeadSpy but on a subset of addresses seen in PMU samples. `DeadCraft` samples the PMU *store* events at a chosen frequency. Let the address accessed in a PMU sample be M and let the calling context where the sample happens be C_{watch}. In the PMU overflow handler, Witch offers the triplet $\langle C_{watch}, M, AccessType\rangle$ to `DeadCraft`. `DeadCraft` memorizes the tuple and in-turn asks Witch to set a `RW_TRAP` watchpoint W at M. The normal execution continues. W traps when the program accesses M next time; we defer discussing another sample happening before the trap to Section 4.1. Let the address accessed in the trap be M and let its calling context be C_{trap}. Witch handles the trap and offers the triplet $\langle C_{trap}, M, AccessType\rangle$ to `DeadCraft`. If a *load* causes a trap, `DeadCraft` treats it as a useful operation and disables the watchpoint. If a *store* causes a trap, however, `DeadCraft` infers the store seen in the context C_{watch} as a dead store. It attributes a "unit" of dead store to the calling context pair $\langle C_{watch}, C_{trap}\rangle$.

Since dead stores can happen only on store instructions, and since every store instruction is sampled at a frequency proportional to its occurrence, transitively, we would detect dead writes at a frequency proportional to their occurrence, if we had infinite debug registers.

4.1 Challenge with Samples Intervening Accesses

Hardware can monitor only a small number of addresses at a time since they have only a handful of debug registers. The scenario of two accesses to the same memory separated by a large distance, where many PMU samples occur in the intervening time, complicates matters.

Consider the dead store example in Listing 2. Assume the loop index variables `i` and `j` are in registers, the sampling period is 10K stores, and the number of debug register is one. The first sample happens in the `i` loop when accessing `array[10K]`. `DeadCraft` sets a watchpoint to monitor `&array[10K]` since a debug register is available. The second sample happens when accessing `array[20K]`. Since the watchpoint armed for address `&array[10K]` is still active, there is no room to monitor `&array[20K]`.

A naive "replace the oldest watchpoint" scheme cannot detect any dead stores in this code. In such scheme, when the `j` loop begins, the only active watchpoint would be the last sampled address `&array[100K]` in the `i` loop. The PMU continues delivering samples in the `j` loop. At `j=10K`, the scheme replaces the last watchpoint on `&array[100K]` with `&array[10K]`, which would not be accessed again. At the end of the `j` loop not a single watchpoint would have triggered, and hence no dead store detected. The same problem exists for more than one debug register. A slightly smarter strategy is to flip a coin to decide whether or not to set a watchpoint on a sample. This strategy fails because the survival probability of an older sample becomes minuscule if a large number of samples happen between consecutive accesses to the same location.

```
for(int i = 1; i <= 100K; i++){
  array[i] = 0;
}
for(int j = 1; j <= 100K; j++){
  array[j] = j;
}
```

Listing 2. Long distance inefficiencies: All (say 4) watchpoints will be armed when sampling at 10K store in the first four samples taken in the `i` loop. A naive replacement will not trigger a single watchpoint due to many samples taken in the `i` loop before reaching the `j` loop. Witch ensures each sample equal probability to survive.

Monitoring a new sample may help detect a new, previously unseen problem whereas continuing to monitor an old, already-armed address may help detect a problem separated by many intervening operations. We should detect both. But, we do not know when in the future a watchpoint may trap, if at all. Our solution strikes a balance between new vs. old by being unbiased in choosing among the previously accessed addresses (reservoir sampling [80]), and we rely on multiple such unbiased samples taken over a repetitive execution to capture both scenarios. We first show our approach for a single debug register and then generalize it for an arbitrary but finite number of debug registers.

On the first sample, S_1, if the debug register is unarmed, Witch sets the watchpoint with 1.0 probability. The second sample, S_2, replaces the previously armed watchpoint (sample S_1) with $1/2$ probability and installs itself. Thus, at the end of S_2, both S_1 and S_2 have equal ($1/2$) probability of being monitored. The third sample, S_3, replaces the previously armed watchpoint with $1/3$ probability to install itself. Since the previously armed watchpoint is S_1 or S_2 with $1/2$ probability each, they each survive with $1/3$ probability. The k^{th} sample S_k since the last time a debug register was empty, replaces the previously armed watchpoint with $1/k$ probability. The previously armed watchpoint could be any one of $\{S_1, S_2, \ldots, S_{k-1}\}$ with $1/k-1$ probability each. At the end of k^{th} sample, the probability of monitoring any sampled address S_i, $1 \le i \le (k-1)$ of the prior $(k-1)$ samples is:

$$Pr[\text{monitoring } S_i] = Pr[S_i \text{ survived in } S_{k-1}] \times Pr[\text{ not retaining } S_k]$$
$$= \frac{1}{k-1} \times \frac{k-1}{k} = \frac{1}{k} = Pr[\text{monitoring } S_k]$$

Any time a watchpoint traps and the client chooses to disarm the watchpoint, and the probability is reset to 1.0, which ensures that the immediately next sample is monitored. Naturally, if every watchpoint triggers before the next sample, we will monitor every address seen in every sample.

In a system with N debug registers, on a new sample, we populate any unused debug register as long as we find one. If no debug register is freed up in a window of N consecutive samples, there will be no room for the $(N+1)^{th}$ sample. We install the sample S_{N+1} with $N/N+1$ probability. If the choice is to install S_{N+1}, we randomly choose one of the N debug registers and replace it with S_{N+1}. It follows that at the end of S_{N+1}, the probability of monitoring any sample S_i, $1 \le i \le N+1$, is $N/N+1$.

The sample S_k, $k > N$, since the last time a debug register was empty, replaces one of the surviving N samples with N/k probability. It follows that at the end of S_k, every sample has the same N/k probability of being monitored. Anytime when a watchpoint traps and the client chooses to disarm the watchpoint, the probability resets to 1.0. Our technique maintains only a count of previous samples—not a log of all previous samples—which needs $O(1)$ memory.

Adversary Sample: If a "never-again-to-be-accessed" address α finds a place in a watchpoint, it can affect the subsequent samples. If no watchpoint has triggered for H samples when α is sampled, the expected number of samples before α will be replaced is $1.7H$, which follows from the sum of harmonic series. The number of debug registers does not influence α.

The number of consecutive PMU samples that are not monitored form a "blindspot" window; the longer the window is, the larger the probability of missing bugs. In our experience, many software in practice often have very short windows. For example, in the SPEC CPU2006 [73] reference benchmarks, on an Intel Haswell machine, we found the largest blind-spot to be, typically, extremely small ($< 0.02\%$ of the total samples in a program), and the worst case was 0.5% of the total samples in the `mcf` benchmark.

4.2 Challenges with Proportional Attribution

Consider the code in Listing 3. For brevity, line numbers represent contexts. 25% PMU samples will be attributed to each of Line 3, 7, 8, and 11. If the outer loop executes 1K times and if the sampling period is 10K store operations, each of these lines will get approximately 10K PMU samples. The number of sampled dead writes should be 10K for each line pair $\langle 3, 11\rangle$, $\langle 11, 3\rangle$, $\langle 7, 8\rangle$, and $\langle 8, 7\rangle$. That is, 25% each. This expectation in quantification is not preserved with our sampling scheme because of a mixture of sparse monitoring (lines 3 and 11) and dense monitoring (lines 7 and 8). As soon as a watchpoint traps on Line 7, a debug register frees up; every subsequent PMU sample in the k loop will find a free debug register. Hence, there will be a disproportionately large number of dead writes recorded for the line pairs $\langle 7, 8\rangle$ and $\langle 8, 7\rangle$ compared to rest.

We solve this problem with a context-sensitive approximation. The code behavior is typically same in a calling context; hence, an observation made by monitoring an address accessed in a calling context can approximately represent other unmonitored samples occurring in the same calling context. If in a sequence of N samples occurring in a calling context C, only one sample is monitored through a debug register, we scale the observation made for the monitored sample by N to approximate the behavior of the remaining $N - 1$ unmonitored samples taken at C. In this scheme, in a sequence of ten PMU samples taken at line 3, only one is monitored through a debug register, and that address leads to a dead

```
 1 for( ... many iterations ...){
 2   for(int i = 1; i <= 100K; i++){
 3     array[i] = 0;
 4   }
 5   // p and q alias to the same location
 6   for(int k = 1; k <= 100K; k++){
 7     *p = 0; // dead write
 8     *q = 0;
 9   }
10   for(int j = 1; j <= 100K; j++){
11     array[j] = 0;
12   }
13 }
```

Listing 3. 100K stores in the `i` loop are dead by the overwriting `j` loop, but only a few watchpoints survive between these two loops. 100K writes to *p are also dead but trigger many more watchpoints at *q. Witch applies a proportional attribution heuristic by accounting the samples taken in a context.

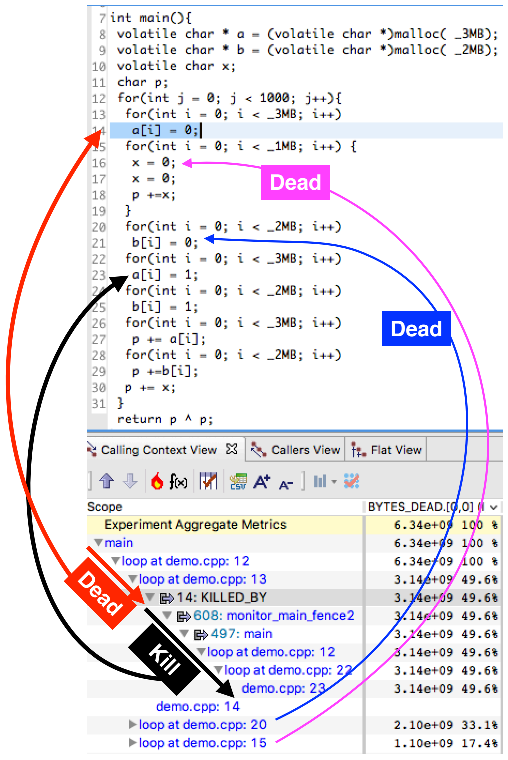
write with line 11, we scale and record number of dead writes between lines $\langle 3, 11\rangle$ as ten.

Implementation: Every PMU sample increments a metric $\mu(C)$ in the calling context C where it happens. Another metric $\eta(C)$ catches up with $\mu(C)$ each time a watchpoint set in C traps. Both metrics are initially zero. Assume we set a watchpoint W in calling context C_{watch}, and it traps in a calling context, say C_{trap}; C_{trap} can be C_{watch}. $\big(\mu(C) - \eta(C)\big) \geq 1$ is the number of samples that W is representing. Assume the sampling period (threshold) is P. If the trapping instruction is a store with M-bytes of overlap over the monitored address range set in W, we approximate and attribute $\big(\mu(C) - \eta(C)\big) \times P \times M$ bytes of "waste" to the ordered pair $\langle C_{watch}, C_{trap}\rangle$. Conversely, if the trapping instruction is a load with M-bytes of overlap over the monitored address range set in C_{watch}, we approximate and attribute $\big(\mu(C) - \eta(C)\big) \times P \times M$ bytes of "use" to the ordered pair $\langle C_{watch}, C_{trap}\rangle$. In either case, we update $\eta(C) = \mu(C)$. Both *use* and *waste* metrics are additive—they accumulate overtime for the attributions happening in the same calling context pairs. Thus, the total inefficiency (dead-writes) is:

$$\widehat{D} = \frac{\sum_i \sum_j \text{waste in } \langle C_i, C_j\rangle}{\sum_i \sum_j \text{waste in } \langle C_i, C_j\rangle + \sum_i \sum_j \text{use in } \langle C_i, C_j\rangle} \qquad (1)$$

The metric is similar to the "deadness" D metric described in [6]; instead of deriving the metric by measuring every load and store, we are approximating. Equation 1 is an optional feature available for the clients of Witch; not all clients need this kind of proportional attribution.

In Listing 3, when a watchpoint traps for the first time on Line 11($= C_{trap}$), and if there were 10 PMU samples accumulated at the source Line 3 ($= C_{watch}$), we attribute $10 \times 10K \times 4$ bytes $= 400K$ bytes of dead writes to the line pair $\langle 3, 11\rangle$. This scheme allows the dead writes metric to catchup with the PMU samples, resulting in proportional attribution. Thus, even though we have very few watchpoints, we use PMU samples in a context to approximate the dead writes in that context. If multiple watchpoints were simultaneously set from the same calling context at different addresses, we proportionally distribute the samples among them.


```
 7 int main(){
 8  volatile char * a = (volatile char *)malloc( _3MB);
 9  volatile char * b = (volatile char *)malloc( _2MB);
10  volatile char x;
11  char p;
12  for(int j = 0; j < 1000; j++){
13   for(int i = 0; i < _3MB; i++)
14    a[i] = 0;
15   for(int i = 0; i < _1MB; i++) {
16    x = 0;
17    x = 0;
18    p +=x;
19   }
20   for(int i = 0; i < _2MB; i++)
21    b[i] = 0;
22   for(int i = 0; i < _3MB; i++)
23    a[i] = 1;
24   for(int i = 0; i < _2MB; i++)
25    b[i] = 1;
26   for(int i = 0; i < _3MB; i++)
27    p += a[i];
28   for(int i = 0; i < _2MB; i++)
29    p +=b[i];
30   p += x;
31  }
    return p ^ p;
```

```
Scope                                    BYTES_DEAD.[0,0]
  Experiment Aggregate Metrics           6.34e+09 100 %
▼main                                    6.34e+09 100 %
  ▼loop at demo.cpp: 12                  6.34e+09 100 %
    ▼loop at demo.cpp: 13                3.14e+09 49.6%
      ▼ 14: KILLED_BY                    3.14e+09 49.6%
        ▼ 608: monitor_main_fence2       3.14e+09 49.6%
          ▼ 497: main                    3.14e+09 49.6%
            ▼loop at demo.cpp: 12        3.14e+09 49.6%
              ▼loop at demo.cpp: 22      3.14e+09 49.6%
                 demo.cpp: 23            3.14e+09 49.6%
        demo.cpp: 14
    ▶loop at demo.cpp: 20                2.10e+09 33.1%
    ▶loop at demo.cpp: 15                1.10e+09 17.4%
```

Figure 2. a[] and b[] and x are involved in dead writes in 3:2:1 ratio (50%:33%:17%), respectively. The sampling interval is 50K stores. Our proportional, context-sensitive scheme apportions dead writes in near perfect ratio.

Figure 2 shows Witch's attribution of dead writes in a more complex scenario, which perfectly matches our expectation of 50%:33%:17% dead writes to a:b:x. Without proportional attribution, we noticed a biased attribution of 5%:2%:93%. With random sampling, rather than our equal probability sampling, 100% samples get attributed to the line pair ⟨16, 17⟩.

4.3 Limitations

Witch employs Monte-Carlo experiments to approximately model real-world observations and suffers from the limitations of any sampling system. Insufficient samples can result in overestimation or underestimation. Witch cannot monitor register-to-register operations. Witch cannot hide the deficiencies of the underlying PMU used to drive its sampling: on some Intel architectures, sporadically, the shadow sampling effect [11, 38, 61] may hide a short latency store behind a long latency store. This behavior can bias the samples to favor long latency stores.

Witch can simultaneously monitor only as many memory locations as the number of debug registers. This physical constraint often is not a problem in practice as we show in our evaluation. However, an adversary may be able to construct a program where the effects of limited registers can be more pronounced.

Witch's context-sensitive attribution is an optional feature available for its tools. It approximates the behavior of one monitored sample in a context to many samples taken in the same context. If very few monitored samples in a context are used to approximate the behavior of a large number of samples with different traits in that context, it can result in noticeable overestimation or underestimation.

Like any profiler, our tools detect only dynamic instances of inefficiencies. False positives or false negatives can happen based on the kind of tools built atop Witch. A dead write detection tool has false negatives (can miss dead writes in an execution) but it has no false positives (all reported dead writes are dead writes). The performance benefit of using debug registers overweighs the downside of a small number of potential false negatives. Developer investigation or post-processing is necessary to make optimization choices—not all reported inefficiencies need be eliminated. Only high-frequency inefficiency spots are interesting; eliminating a long tail of insignificant inefficiencies that do not add up to a significant fraction is impractical and probably ineffective. Our investigation shows that only a few calling contexts contribute to most of the measured inefficiencies; for example, in SPEC CPU2006 benchmarks, fewer than five contexts, typically, contributed to over 90% of dead writes.

5 Design and Implementation

We implement Witch in the open-source HPCToolkit [1] performance analysis tools suite. HPCToolkit works on multilingual, multi-threaded, and multi-process, fully optimized applications on multiple programming models such as MPI and OpenMP. On a PMU sample, HPCToolkit's profiler, hpcrun, walks the sampled thread's call stack using an on-the-fly binary analysis technique and attributes the measurements to the sampled call path. hpcrun introduces negligible runtime overhead (∼3%) and consumes only a few megabytes of memory space for its metrics data when sampling at ∼200 samples/second/thread [77].

PMU Sampling: Although the clients of Witch can sample any precise PMU event to set a watchpoint, on Intel processors, typically, we use MEM_UOPS_RETIRED:ALL_STORES and MEM_UOPS_RETIRED:ALL_LOADS to drive PMU sampling. These events offer the address accessed in a sample.

Watchpoint Registration: Witch automatically discovers the number of hardware debug registers supported on the platform. When a client wants to monitor an address, Witch uses the Linux perf_event interface to register a watchpoint event. The event is a HW_BREAKPOINT perf event (a PERF_TYPE_SOFTWARE event category). Witch registers a signal handler to capture watchpoint exceptions that the Linux perf_event interface raises when the event overflows. Witch sets the sample_period to 1 for its HW_BREAKPOINT events, which ensures that the trap signal is delivered immediately after accessing the monitored address.

Precise PC of a Watchpoint: Some clients need the precise instruction pointer of the instruction triggering the watchpoint, for example, to distinguish a load from a store when a `RW_TRAP` watchpoint triggers. The `BREAKPOINT` event in Linux `perf_event` is not a PMU event and hence the Intel PEBS support, which otherwise provides the precise register state, is unavailable for a watchpoint. Although the watchpoint causes a trap immediately after the instruction execution, the PC seen in the signal handler context (`contextPC`) is one ahead of the actual PC (`precisePC`) that causes the trap. In the x86 variable instruction set ISA, it is non-trivial to derive the `precisePC`, even though it is just one instruction before the `contextPC`. A software solution is to find the function enclosing `contextPC` and disassemble every instruction till we reach the `contextPC`. This solution may fail with linear disassembly due to 1) data embedded in instruction and 2) missing function bounds [77]. Furthermore, it can be time-consuming if the function body is large.

Our solution depends on the Last Branch Record (LBR) facility [25] provided by Intel Nehalem and its successors, which is exposed through the Linux `perf_event` interface. LBR tracks taken branches throughout CPU execution and continuously records the `<from:to>` pairs of instruction pointers in a fixed-size in-CPU circular buffer. Witch exploits the LBR facility by modifying the `perf_event` implementation inside the Linux kernel. Linux `perf_event` already has the facility to construct the precise PC by disassembling the instructions starting from the "to" field of the last entry in the LBR until the disassembly reaches the `contextPC`. Disassembling a basic block is "feather light" compared to full function disassembly. We reuse this component with `PERF_TYPE_SOFTWARE` to construct the `precisePC` when a watchpoint trap event happens. The kernel makes the `precisePC` available to Witch's watchpoint exception handler in the ring buffer associated with the event on each watchpoint trap. This reduces ~5% runtime overhead.

Fast Watchpoint Replacement: Witch requires frequently disabling a watchpoint, closing all the kernel resources (`perf_event` file descriptor and an `mmaped` ring buffer) associated with the watchpoint, and recreating the same for another watchpoint. We enhance the kernel `perf_event ioctl` interface with an additional flag `PERF_EVENT_IOC_MODIFY_ATTRIBUTES`. This flag allows `perf_event` users to update the address and the access length associated with an already installed watchpoint. As a result, the user code can continue to reuse all the kernel resources associated with the previous `perf_event` file descriptor. Although Witch is functionally correct without this support, we found it useful to optimize this use case (~5% overhead reduction). This change is being contributed to the Linux kernel as of this writing.

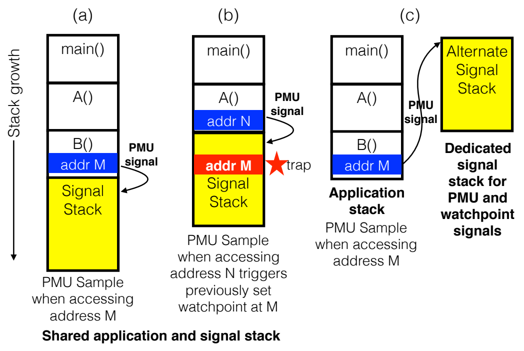

(a)
(b)
(c)
Stack growth
main()
A()
B()
addr M
Signal Stack
PMU signal
PMU Sample when accessing address M
addr N
addr M
trap
PMU Sample when accessing address N triggers previously set watchpoint at M
Shared application and signal stack
Application stack
PMU Sample when accessing address M
Alternate Signal Stack
Dedicated signal stack for PMU and watchpoint signals

Figure 3. (a) A PMU sample happens in a deeper call stack when `B()` is accessing address M; signal handler sets a watchpoint to monitor the address M. (b) A shallower application call stack, function `A()`, triggers another PMU sample, the signal handler is established in a location that overwrites M, triggering a spurious watchpoint. (c) An alternate signal stack for PMU signal handler and watchpoint signal handler solves the problem.

Stack Addresses: Clients of Witch may set a watchpoint on the stack in one function that returns, and another function invocation may overwrite the previous stack frame. Such situation will cause the watchpoint to trap, and Witch has no problem for such normal call-return sequence. If there is a redundancy in a callee, e.g., write to a variable in a callee that is frequent not read before returning to the caller, Witch can easily detect it.

Setting a watchpoint on the application stack address has a corner case. On a PMU sample, the profiler's overflow signal handler, by default, shares the same stack as the application thread. In Figure 3(a), assume M is the sampled stack address. Assume we set a watchpoint at M. If the next PMU sample is taken with a shallower stack (Figure 3(b)), and the signal stack frame overwrites M; it spuriously triggers the watchpoint. Similarly, one watchpoint exception handler stack frame may trap another watchpoint.

We avoid this problem by establishing a separate signal-handler stack frame for both PMU signal handler and watchpoint exception handler using the Linux `sigaltstack` facility [41]. The `sigaltstack` facility allows each thread in a process to define an alternate signal stack in a user-designated memory region. We use alternate stack to handle PMU and watchpoint signals as shown in Figure 3(c). All other signals continue to use the default stack unless specified otherwise by the application.

6 Witchcraft: Client Tools of Witch

We have already discussed the dead store detection client in the previous sections as a running example. In this section, we elaborate two more clients that use the Witch framework to pinpoint different kinds of inefficiencies.

6.1 SilentCraft: Silent Store Detection

Updating a location with a value already present at the location is a silent store. Silent stores are useless since they do not change system state. Our prior work, RedSpy [82], shows that useless computations that store their results into memory often show up as silent stores. Here, we devise SilentCraft, a silent store detection client that mimics RedSpy.

SilentCraft samples PMU *store* events. On each PMU sample, SilentCraft remembers the contents (value) of the memory location accessed in the sampled address. SilentCraft, then, arms a W_TRAP watchpoint W. SilentCraft disregards the loads that may intervene between two store operations. Hence loads do not trigger a watchpoint trap. SilentCraft also associates the calling context C_{watch} of the sample point with the watchpoint W.

The next store operation (say in context C_{trap}), overlapping the same memory address, triggers a watchpoint exception. SilentCraft obtains the precise PC and the address accessed in the watchpoint from Witch and compares the current contents of the memory location with the previously recorded value. The comparison is limited to the bytes that overlap between a) the sampled address and its access length and b) and trapped address and its access length. If all overlapping bytes are same, SilentCraft marks the calling context pair $\langle C_{watch}, C_{trap} \rangle$ with proportional units of silent stores. Proportionality computation follows the previously discussed proportional attribution heuristic. To identify opportunities for approximate computation, for the floating-point operations, SilentCraft performs approximate equality check within a user-specified precision level. SilentCraft infers that a datum is a floating-point value by disassembling the instruction accessing the address.

SilentCraft quantifies the store redundancy $\widehat{R}$ in an execution analogous to DeadCraft (Equation 1); two consecutive stores with unchanged values (approximately the same for floating point values) contribute to the "waste" and contribute to the "use" otherwise.

6.2 LoadCraft: Load-after-load Detection

We developed a new tool—LoadCraft—that detects a load followed by another load from the same location where the value remains unchanged between the two loads. It ignores intervening stores to the same address that may change the value and revert it to the original value before a load. Not all load-load redundancies can be eliminated. Since machines have a small number of registers, they often spill values to memory to be read back later. Unfavorable algorithms and data structures often show up as load-load redundancies that shed light on domain-specific optimization opportunities.

LoadCraft samples PMU *load* events. The rest of the functionality is similar to that of SilentCraft, except that it requests a watchpoint for load access on the monitored location. Witch uses RW_TRAP because x86 machines do not offer a trap-on-load watchpoint. If a watchpoint triggers on a store operation, Witch merely drops it. LoadCraft quantifies the load redundancy $\widehat{L}$ in an execution analogous to DeadCraft (Equation 1), where two consecutive loads with (approximately) unchanged values contribute to the "waste" and different values contribute to the "use".

6.3 Witchcrafts on Multi-threading

Debug registers and PMUs are per CPU core and virtualized for each software thread. All the previously discussed Witch tools work on multi-threaded codes; they, however, track intra-thread inefficiencies only. If a thread T_1 configures a watchpoint at address M, a trap occurs only in T_1; other threads remain unaffected whether they access M or not. Sharing addresses accessed by one thread with another thread allows building several tools for multi-threaded applications. Atop Witch, we have developed Feather [9]—a tool to detect false sharing in parallel programs.

6.4 Discussion

Developers can only reason about inefficiencies at instruction, source line, or data-type granularities. Hence, in all tools we discussed, if a dynamic instruction writes M bytes, either all M bytes contribute to the inefficiency metric or none. In the three tools we developed, we made the following implementation decision: if the monitored element of a SIMD instruction instance is found to be wasteful (useful), we approximate that all elements in the SIMD instruction instance as wasteful (useful). Other tools are free to make a different choice.

Currently, Witch is implemented to work on the native code such as C/C++/Fortran applications. The basic idea extends to a managed language but requires runtime support to map JIT-generated instruction to the source code.

6.5 Presentation

HPCToolkit maintains all sampled call paths in a compact calling context tree (CCT) format [2]. HPCViewer, the graphical interface, enables navigating the CCT and the corresponding source code ordered by the monitored metrics. A top-down view shows a call path C starting from main to a leaf function with the breakdown of metrics at each level. Witch tools discussed here need to attribute metrics to calling context *pairs* $\langle C_{watch}, C_{trap} \rangle$. Merely attributing a metric to two independent contexts loses the association between two related contexts during postmortem inspection. To maintain a correlation between a *source* context (e.g., dead) and target (killing) context, Witch appending a copy of the *target* calling context to a source calling context. For example, if a store in context main->A->B is overwritten by another store in context main->C->D, DeadCraft constructs a synthetic calling context: main->A->B->KILLED_BY->main->C->D. The dead write metrics will be attributed to the leaf of this call chain. These synthetic call chains make it easy to visually navigate the CCT and focus on top redundancy pairs. Figure 2 in Section 4 depicts this scheme.

7 Evaluation

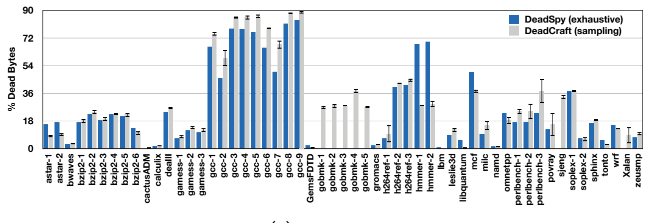
We evaluate Witch on a 2-socket, 18-core Intel Xeon E5-2699 v3 (Haswell) CPU clocked at 2.30GHz running Linux `4.8.0`. The machine has 128GB DDR3 RAM. Simultaneous multi-threading (SMT) facility is not used in our experiments. All experiments use GCC `v5.4.1` tools with `-O3` and profile-guided optimization (PGO) to ensure the highest level of optimization. `DeadCraft` and `SilentCraft` use the PMU event `MEM_UOPS_RETIRED:ALL_STORES` whereas `LoadCraft` uses `MEM_UOPS_RETIRED:ALL_LOADS`. In our experiments, we use the nearest prime number for the shown sampling intervals, which is the recommended method in PMU sampling. The raw data from our experiments are available online [7].

Two aspects are critically important in evaluating Witch: *accuracy* and *overhead* compared to the exhaustive instrumentation techniques. We use SPEC CPU2006 reference benchmarks for this aspect of evaluation.

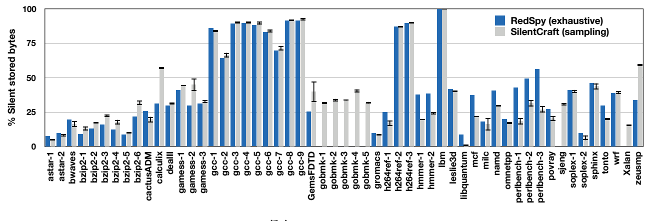
Accuracy: For accuracy, we need to answer three questions: (1) how accurate are the results compared to exhaustive monitoring, (2) how does the accuracy vary with sampling rates, and (3) how stable are the sampled results from one run to another.

The quantitative metric of dead writes is the percent of dead stores $\widehat{D}$ (bytes overwritten without reading) as described in Equation 1, which we compare against the ground-truth dead stores D from DeadSpy [8, 10]. We compare the percent of silent stores $\widehat{R}$ from `SilentCraft` against the ground-truth exhaustive monitoring metric R from RedSpy [10, 82]. No prior tool exists to compare against `LoadCraft`; hence we implemented an exhaustive load-load value redundancy detection tool called LoadSpy. We compare the percent of silent loads $\widehat{L}$ from `LoadCraft` against the ground-truth exhaustive monitoring metric L from LoadSpy. RedSpy also performs redundancy detection in registers, which we disabled for our evaluation. To assess the accuracy of our sampling clients against the ground-truth, we disable the bursty sampling used by RedSpy. `SilentCraft`, `LoadCraft`, RedSpy, and LoadSpy use 1% precision when comparing floating point values.

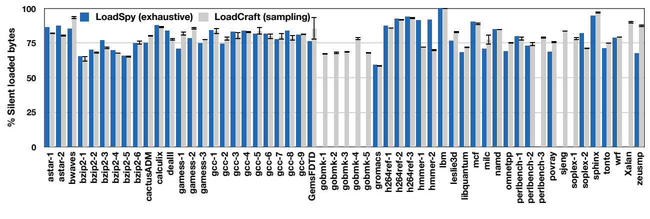
Figure 4 compares the total redundancies found by different sampling vs. exhaustive monitoring tools. The error bars represent the metric values at different sampling rates for Witch tools, i.e., 100K (high), 500K, 1M, 5M, 10M, and 100M (low) events per PMU interrupt. Clearly, the sampling rate, when chosen with some care, does not significantly affect the results. The sampling tools are highly accurate in almost all cases. There are, however, some exceptions. `DeadCraft` and `SilentCraft` on `hmmer` and `calculix` suffer from shadow sampling effects [11, 38, 61], where high



(a) Dead Store



(b) Silent Store



(c) Silent Load

Figure 4. Witch tools vs. instrumentation tools on SPEC CPU2006. Error bars capture different sampling rates. Ground truth instrumentation data is unavailable for `gobmk`, `sjeng`, and `Xalan` since they ran out of memory. The benchmarks with multiple inputs (e.g., `bzip2`) appear multiple times with different numerical suffixes.

latency stores hide low latency stores. `GemsFDTD`, `perlbench`, and `zeusmp` have many small inefficiencies scattered all over the code, leading to inaccuracies in `SilentCraft`. We ran each benchmark 10 times at 5M sampling rate (not shown) and the maximum standard deviations were 2.27%, 1.89%, and 0.77% for `DeadCraft`, `SilentCraft`, and `LoadCraft` respectively, which proves the run-to-run sampling stability.

`lbm` has ∼100% silent stores and silent loads, but it has negligible dead stores. `lbm` is a floating point code, which simulates incompressible fluids in 3D. One iteration updates the values in an array that are loaded in the next iteration. The difference between the values produced in adjacent iterations is less than our predefined 1% threshold. Hence, `LoadCraft` treats these loads as redundant ones. Similarly, `SilentCraft` treats the stores to be approximately the same.

To assess the effectiveness of reservoir sampling, we vary the number of debug registers from one to four and compare the redundancy metrics against the ground truth. Figure 5 shows that the number of debug registers has little practical influence in `DeadCraft` on the quality of results except `h264ref`, which shows better results with four debug registers. The online compendium [7] corroborates this observation on `SilentCraft` and `LoadCraft`.

To assess the effectiveness of our proportional attribution based on samples taken in a context, we compared the accuracy with and without this feature at different sampling

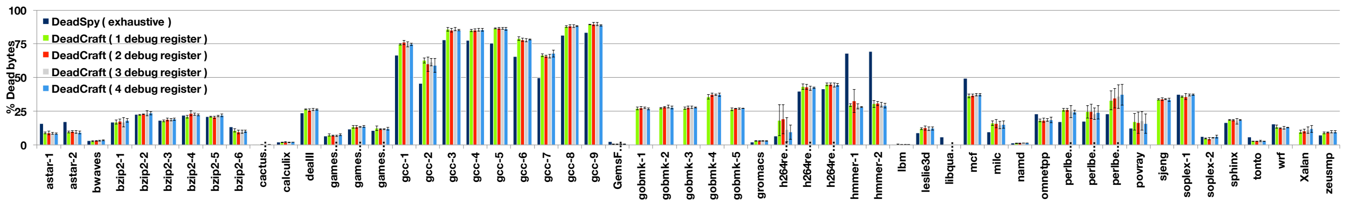


Figure 5. Comparison of dead writes with different number of debug registers. Error bars are for different (100K - 100M) sampling intervals.

rates and also with different number of debug registers with all three tools (not shown); we also compared it against the ground truth. In general, the feature did not make significantly positive or negative impact. `GemsFDTD` and `perl` were exceptions, where having the feature improved the accuracy.

To further understand the accuracy, we compared the rank ordering and percentage contribution of the top N redundancy pairs between DeadSpy and `DeadCraft`; we chose N to add up to 90% of redundancy observed in execution. No single metric suffices to compare this type of complex data. We used edit distance and set difference of the top N contexts and also compared weights at each position. Our measurements [7] show that only a handful of context pairs account for the majority of redundancies and their rank ordering and individual weights match the exhaustive monitoring.

Overhead: Table 1 shows the runtime slowdown and memory bloat of sampling vs. exhaustive monitoring. Slowdown (memory bloat) is the ratio of the runtime (peak memory usage) under monitoring to the runtime (peak memory usage) of the corresponding native execution. We show the average values for the same benchmark with multiple inputs. We used the sampling period of one in 5M stores and one in 10M loads (since loads are more common), which we found to be highly effective. Two critical things to observe about the sampling tools are 1) their overheads are at least an order of magnitude less than the exhaustive instrumentation tools, and 2) they introduce negligible overhead. Deep recursive codes such as `xalanbmk`, `sjeng`, and `gombk` incur higher space and time overheads; and their instrumentation counterparts do not run to completion. Recursive codes with inefficiencies (e.g., `SilentCraft` on `gobmk` and `LoadCraft` on `xalanbmk`) exacerbate memory bloat due to large calling context trees. Codes with a very small memory footprint (e.g., `povray`) show higher memory bloat because of some basic pre-allocated data structures used in our tools.

`LoadCraft` has higher overhead compared to the other two tools since 1) loads are more common than stores, 2) a high fraction of loading the same value leads to more watchpoint traps and inefficiency reporting cost, 3) most PMU samples find a free debug register and incur the cost of arming it, and finally 4) `LoadCraft` sets the `RW_TRAP` watchpoint (x86 does not support break on load watchpoint), which triggers a spurious exception on a store. Table 2 shows the geometric mean and median of the slowdown and memory bloat at different sampling periods in SPEC CPU2006.

8 Case Studies

The lightweight nature of Witch tools allowed us to apply it on an array of benchmark suites—SPEC CPU2006 [73], SPEC OMP2012 [74], NERSC Trinity [57], Rodinia [62], and STAMP [56] and full applications—NWChem [79], Caffe [28], GNU Binutils [20], and Kallisto RNA sequencing [52]. Table 3 summarizes the new performance bugs found by our tools (denoted by ✓ prefix) and confirms previously found performance issues [8, 82]. In this section, we describe four case studies covering the analyses by the three Witch tools.

8.1 NWChem-6.3

NWChem [79] is a production computational chemistry package, which implements several quantum mechanics and molecular mechanics methods. NWChem consists of six million lines of code written primarily in Fortran and C and parallelized with MPI [53]. We use the `QM-CC aug-cc-pvdz` input and eight MPI processes in our studies.

`DeadCraft` reports that more than 60% of memory stores are dead. Figure 6 shows the full calling contexts of the top (94% contribution to total dead writes) dead and killing store pair in the call of function `dfill`, which zeroes the array `work2`. With the given input, calls to `dfill` repeat more than 200K times, resulting in writing 500GB data that are never used. With further analysis, we identified that the size of `work2` was larger than necessary, and the zero initialization was unnecessary, leading to the dead and killing writes in the same location. We eliminate this unnecessary initialization, yielding a 1.43× speedup. This bug, which was hiding in the large code base, is now fixed. Witch incurs only 6% runtime overhead whereas the fine-grained profiler, DeadSpy, incurs $> 10\times$ slowdown identifying the same problem.

8.2 Caffe-1.0

We apply `SilentCraft` on the deep learning framework Caffe [28]. We study the OpenMP C++ CPU version, which uses Intel MKL [26] to parallelize its computation kernels. We use the CIFAR-10 dataset to train the CIFAR network with 0.9 momentum, 4e-3 weight decay, 1e-3 learning rate, 128 batch size. We run Caffe with eight threads.

`SilentCraft` attributes 25% of total memory stores as redundant in a loop nest belonging to a major computation kernel in pooling and normalization layers (Listing 4). The memory stores to the array `bottom_diff` (Line 8) account for 17% of total silent stores. A large portion of elements in

Benchmark			astar	bwaves	bzip2	cactusADM	calculix	dealII	gamess	gcc	GemsFDTD	gobmk	gromacs	h264ref	hmmer	lbm	leslie3d	libquantum	mcf	milc	namd	omnetpp	perlbench	povray	sjeng	soplex	sphinx3	tonto	wrf	xalancbmk	zeusmp	GeoMean	Median
Original Time (second)			139	303	64	371	635	246	50	24	297	71	317	138	160	342	215	173	221	458	318	182	65	101	367	86	423	408	312	158	360		
Original Memory Usage (MB)			232	875	562	664	118	795	22	459	831	30	16	38	16	411	125	95	1677	681	48	171	400	7	176	279	44	36	695	421	512		
Dead store	Slowdown (times)	DeadSpy	22.65	31.70	32.32	29.93	33.40	54.70	40.00	43.57	26.36	-	26.95	59.67	52.01	28.86	31.70	27.87	21.69	14.61	22.18	37.80	71.91	71.13	-	26.45	26.16	34.85	24.60	-	19.95	32.48	30.82
		DeadCraft	1.01	1.00	1.00	1.04	1.01	1.01	1.07	1.02	1.03	1.00	1.02	1.01	1.02	1.00	1.01	1.00	1.00	1.01	1.00	1.01	1.03	1.02	1.00	1.00	1.01	1.02	1.04	1.19	1.00	1.02	1.01
	Memory bloat (times)	DeadSpy	6.47	6.40	6.27	5.84	8.53	14.70	8.20	32.26	6.12	-	9.80	11.93	20.65	6.06	6.25	9.49	6.00	6.26	6.55	6.84	45.75	38.66	-	17.54	7.48	16.76	6.70	-	6.03	9.87	7.16
		DeadCraft	1.03	1.01	1.02	1.00	1.07	1.01	1.30	1.03	1.00	2.06	1.38	1.30	1.62	1.01	1.07	1.11	1.00	1.01	1.14	1.04	1.02	2.56	1.09	1.05	1.15	1.64	1.01	5.36	1.02	1.23	1.05
Silent store	Slowdown (times)	RedSpy	16.33	17.62	23.75	45.64	26.17	23.60	33.00	200.07	41.24	-	25.60	101.66	26.66	14.53	39.43	23.67	10.76	10.94	16.94	27.97	59.71	67.50	-	23.60	16.46	29.32	33.00	-	27.66	29.10	26.42
		SilentCraft	1.01	1.01	1.00	1.05	1.01	1.00	1.05	1.03	1.02	1.00	1.02	1.02	1.03	1.00	1.01	1.03	1.00	1.00	1.02	1.01	1.04	1.03	1.00	1.02	1.00	1.01	1.04	1.13	1.00	1.02	1.01
	Memory bloat (times)	RedSpy	5.35	5.42	5.26	5.06	8.24	12.50	9.80	34.84	5.22	-	10.08	9.95	15.57	5.15	5.46	6.35	5.08	5.31	5.97	6.42	67.76	51.87	-	3.73	6.53	17.33	5.90	-	5.20	8.58	6.16
		SilentCraft	1.03	1.01	1.02	1.00	1.07	1.01	1.31	1.03	1.01	2.05	1.55	1.33	1.64	1.01	1.07	1.11	1.00	1.01	1.14	1.04	1.02	2.78	1.09	1.04	1.15	1.64	1.01	5.17	1.01	1.24	1.04
Silent load	Slowdown (times)	LoadSpy	30.00	87.70	53.00	123.00	75.30	81.30	100.00	51.80	69.60	-	39.80	185.00	95.30	15.10	98.60	36.80	26.90	26.90	46.10	36.00	82.00	156.00	-	31.20	60.10	54.10	81.90	-	51.00	58.66	57.10
		LoadCraft	1.04	1.00	2.16	1.69	1.09	1.00	1.12	1.04	1.08	1.00	1.06	1.04	1.04	1.00	1.09	1.01	1.00	1.02	1.58	1.00	1.51	1.05	1.00	1.00	1.01	1.05	1.10	1.86	1.08	1.13	1.04
	Memory bloat (times)	LoadSpy	6.80	6.50	4.00	5.90	8.49	14.40	12.00	50.10	6.30	-	12.60	18.40	23.90	6.20	4.90	50.20	6.09	6.40	6.80	8.20	184.00	1051.00	-	13.20	9.30	36.70	5.20	-	6.30	13.52	8.35
		LoadCraft	1.03	1.01	1.03	1.00	1.08	1.01	1.36	1.03	1.01	2.01	1.43	1.36	1.64	1.01	1.07	1.11	1.00	1.01	1.13	1.04	1.02	3.55	1.14	1.05	1.19	1.87	1.02	24.93	1.02	1.33	1.05

Table 1. Runtime slowdown (×) and memory bloat (×) over native execution: Witch (DeadCraft, SilentCraft, LoadCraft) vs. exhaustive monitoring tools (DeadSpy, RedSpy, LoadSpy).

GeoMean /Median	DeadCraft		SilentCraft		LoadCraft	
	Slowdown (times)	Memory bloat (times)	Slowdown (times)	Memory bloat (times)	Slowdown (times)	Memory bloat (times)
100M	1.00/1.00	1.12/1.03	1.01/1.00	1.12/1.04	1.04/1.00	1.17/1.05
10M	1.01/1.01	1.19/1.05	1.01/1.00	1.19/1.04	1.13/1.04	1.33/1.05
5M	1.02/1.01	1.23/1.05	1.02/1.01	1.24/1.04	1.20/1.06	1.42/1.06
1M	1.05/1.03	1.40/1.05	1.06/1.03	1.39/1.04	1.48/1.27	1.66/1.07
500K	1.09/1.03	1.48/1.06	1.09/1.04	1.47/1.05	1.92/1.53	1.74/1.07

Table 2. Geomean and median of slowdown and memory bloat of Witch tools at different sampling rates on SPEC CPU2006.

Benchmark Information		Witch	WS*
program	problem code	Inefficiencies (Client)	
gcc [73]	cselib.c:cselib_init	Poor data structure (DS)	1.33×
bzip2 [73]	blocksort.c:mainGtU_init	Poor code generation# (DS)	1.07×
hmmer [73]	fast_algorithms.c:loop(119)	No-vectorization (DS/SS)	1.28×
h264ref [73]	mv-search.c:loop(394)	Missed inlining (SL)	1.27×
✓povray [73]	csg.cpp:loop(248)	Missed inlining (DS)	1.08×
✓Chombo [12]	PolytropicPhysicsF.ChF:(434)	Inattention to perf. (DS)	1.07×
✓botsspar [74]	sparselu.c:fwd	Redundant computation (SL)	1.15×
✓imagick [74]	magick_effect.c:loop(1482)	Redundant computation (SL)	1.6×
✓SMB [57]	msgrate.c:cache_invalidate	Redundant computation (SL)	1.47×
backprop [62]	bpnn_adjust_weights	Redundant computation (SS)	1.20×
lavaMD [62]	kernel_cpu.c:loop(117)	Redundant computation (SL)	1.66×
✓vacation [56]	client.c:loop(198)	Redundant computation (SL)	1.31×
NWChem-6.3 [79]	tce_mo2e_trans.F(240)	Useless initialization (DS/SS)	1.43×
✓Caffe-1.0 [28]	pooling_layer.cpp(289)	Redundant computation (SS)	1.06×
✓Binutils-2.27 [20]	dwarf2.c(1561)	Linear search algorithm (SL)	10×
✓Kallisto-0.43 [52]	KmerHashTable.h(131)	Poor hashing (SL)	4.1×

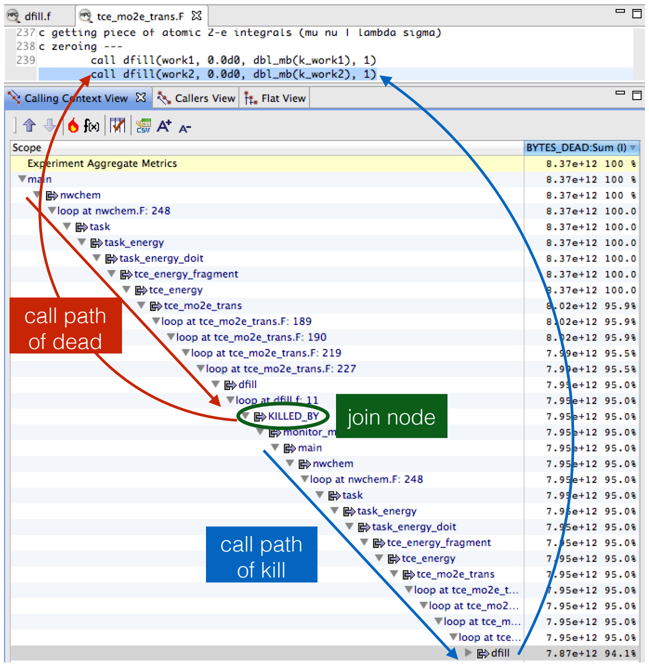
*WS means whole-program speedup after problem elimination.

+DS means dead store, SS means silent store, SL means silent load.

✓ newly found issues via Witch   # used gcc-4.1.2

Table 3. Performance improvement guided by Witch.

`top_diff` are zeroes; hence the same values overwrite the existing values in the same memory location of `bottom_diff`. The iteration over all the elements of `bottom_diff` in the four-level nested loop amplifies the fraction of silent stores. We optimize this code by introducing a check for the value in `top_diff`. If it is a zero, we bypass a division, an addition, and a memory store. This optimization speeds up the pooling layer by 1.16× the normalization layers by 1.34×. We observe



```
237 c getting piece of atomic 2-e integrals (mu nu | lambda sigma)
238 c zeroing ---
239         call dfill(work1, 0.0d0, dbl_mb(k_work1), 1)
            call dfill(work2, 0.0d0, dbl_mb(k_work2), 1)
```

```
Experiment Aggregate Metrics                          8.37e+12 100 %
main                                                  8.37e+12 100 %
 nwchem                                               8.37e+12 100 %
  loop at nwchem.F: 248                               8.37e+12 100.0
   task                                               8.37e+12 100.0
    task_energy                                       8.37e+12 100.0
     task_energy_doit                                 8.37e+12 100.0
      tce_energy_fragment                             8.37e+12 100.0
       tce_energy                                     8.37e+12 100.0
        tce_mo2e_trans                                8.02e+12 95.9%
         loop at tce_mo2e_trans.F: 189                8.02e+12 95.9%
          loop at tce_mo2e_trans.F: 190               8.02e+12 95.9%
           loop at tce_mo2e_trans.F: 219              7.99e+12 95.5%
            loop at tce_mo2e_trans.F: 227             7.99e+12 95.5%
             dfill                                    7.95e+12 95.0%
              loop at dfill.f: 11                     7.95e+12 95.0%
               KILLED_BY                              7.95e+12 95.0%
                monitor_m                             7.95e+12 95.0%
                 main                                 7.95e+12 95.0%
                  nwchem                              7.95e+12 95.0%
                   loop at nwchem.F: 248              7.95e+12 95.0%
                    task                              7.95e+12 95.0%
                     task_energy                      7.95e+12 95.0%
                      task_energy_doit                7.95e+12 95.0%
                       tce_energy_fragment            7.95e+12 95.0%
                        tce_energy                    7.95e+12 95.0%
                         tce_mo2e_trans               7.95e+12 95.0%
                          loop at tce_mo2e_t...       7.95e+12 95.0%
                           loop at tce_mo2...         7.95e+12 95.0%
                            loop at tce_m...          7.95e+12 95.0%
                             loop at tce...           7.95e+12 95.0%
                              dfill                   7.87e+12 94.1%
```

Figure 6. The pair of dead and kill stores with full contexts reported by Witch's dead store client.

```
for (int n = 0; n < top[0]->num(); ++n) {
  for (int c = 0; c < channels_; ++c) {
    for (int ph = 0; ph < pooled_height_; ++ph) {
      for (int pw = 0; pw < pooled_width_; ++pw) {
        ...
        for (int h = hstart; h < hend; ++h) {
          for (int w = wstart; w < wend; ++w) {
▶           bottom_diff[h * width_ + w] +=
                top_diff[ph * pooled_width_ + pw] /
                    pool_size;
    }}}}
    ...}}
```

Listing 4. Silent stores to array `bottim_diff` in Caffe.

1.03× speedup for the entire program. We further relax the check for the absolute value in `top_diff` with a small delta 1e-7 rather than 0. If it is smaller, we bypass the computation for approximate results. This optimization, with less than

```
1    bfd_boolean lookup_address_in_function_table (struct
         comp_unit *unit,  bfd_vma addr, ...) {
2     ...
3    for (each_func = unit->function_table; ...) {
4     for (arange = &each_func->arange;  ...) {
5 ▶   if (addr >= arange->low && addr < arange->high){
6      if (!best_fit  || ... ) {
7       best_fit = each_func;
8       best_fit_len = arange->high - arange->low;
9      }}}}
10    . . .
11   }
```

Listing 5. Redundant loads in `binutils-2.27` dwarf2.c file. Linear searches load same the values from same locations.

2% accuracy loss, yields 1.16× and 2.23× speedups for the pooling and normalization layers, respectively. The entire program obtains a 1.06× speedup.

8.3 GNU Binutils-2.27

GNU `Binutils` [20] is a collection of binary tools used by many binary analysis tools such as Pin [49] and command-line tools such as `objdump` [19]. Disassembling an object file containing many functions using `objdump` with `-d -S -l` flags (map assembly to symbol and source lines) is unusually slow. We profile `objdump` in `binutils-2.27` with `LoadCraft` by disassembling the LULESH-2.0 [33] binary, which contains many functions. `LoadCraft` identifies 96% of the loads in the program as loading the same value from the same location. The top contributor is the Line 5 (Listing 5) in the function `lookup_address_in_function_table` with 70% redundant loads attributed to it. The function performs a linear scan over the addresses covered by each line of each function, maintained as a linked list, looking for the best match for a given address range.

When repeatedly called for different addresses in an object file containing many functions linear search is a poor choice of algorithm. We replace the linked list with a sorted array and perform a binary search over it. This solution speeds up the execution by 10×. This problem is fixed in the latest binutils. Pinpointing that the code always loads the same values from the same location raised a red flag, clearly indicating an algorithmic deficiency.

8.4 SPEC OMP2012 367.imagick

SPEC OMP2012 367.imagick [74] is an OpenMP software to manipulate bitmap images. With the `ref` input and eight threads, `LoadCraft` reports that more than 99% of total memory loads are redundant and 85% of the redundant loads are associated with the loop nests shown in Listing 6.

The loop body has six memory loads for different fields of `pixel` and `kernel_pixels`. Each of the loads is often redundant with a load in a prior iteration. We find that the fields `red`, `green`, and `blue` of `kernel_pixels[u]` are mostly zeros. For optimization, we introduce a conditional check on `kernel_pixels[u]`. If it is zero, we skip the computation, which saves a memory load from address `k`, a multiplication, and a memory load to the field of `pixel`. This optimization yields a 1.6× speedup.

```
for (y=0; y < (ssize_t) image->rows; y++) {
 for (x=0; x < (ssize_t) image->columns; x++) {
  for (v=0; v < (ssize_t) width; v++) {
   for (u=0; u < (ssize_t) width; u++) {
▶    pixel.red+=(*k)*kernel_pixels[u].red;
▶    pixel.green+=(*k)*kernel_pixels[u].green;
▶    pixel.blue+=(*k)*kernel_pixels[u].blue;
     k++;
}}}}
```

Listing 6. Redundant loads to different fields of structure `pixel` and array `kernel_pixels` in 367.imagick

8.5 Discussion on Other Optimizations

Many algorithmic deficiencies show up as useless loads and stores. While hotspots may indicate where a large fraction of time is spent, they do not indicate the usefulness of the work. Such defects stand out when profiled with our tools.

We presented a subset of programs where we found inefficiencies using witchcraft. `Kallisto-0.43` [52] is an important RNA-sequencing software where `LoadCraft` found more than 98% redundant loads. The problem was a large, linear-probing hash-table with excessive hash collisions. We fixed `Kallisto` by reducing the load factor on the hash table and gained 4.1× speedup. `Vacation` is a STAMP [56] transactional memory benchmark, where we found unnecessary calls to a hash-table lookup of an item that was already found in the previous line of the code. Memoizing the result of the previous lookup resulted in 1.3× speedup. The results from our tools showed us that SPEC CPU2006 `lbm` is an excellent candidate for approximate computing; we applied loop perforation [70] to `lbm` and obtained 1.25x speedup with insignificant (7.7e-5%) accuracy loss.

9 Conclusions

Fine-grained execution monitoring via binary instrumentation introduces heavy slowdown and memory bloat. Witch, open sourced at [7], is a lightweight monitoring framework, which employs PMU sampling in conjunction with hardware debug registers to monitor sampled memory addresses. We overcome the problem of limited hardware debug registers with a novel sampling technique. Witch-based tools show high measurement accuracy. Low overhead combined with rich contextual attribution makes our tools attractive to developers in pinpointing inefficiencies in large, complex, production software. We demonstrate the effectiveness of our tools by identifying inefficiencies in several complex, parallel software projects that were the subject of optimization for decades, and we tune them with the guidance from Witch to gain significant speedups.

Acknowledgments

We thank the anonymous reviewers and our shepherd Shan Lu for the detailed comments on the paper. We appreciate the Ph.D. student Probir Roy from College of William and Mary for helping study Caffe 1.0. We also thank Dr. Aarthi Muthusamy for sketching the title Witch icon.

References

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. 2010. HPCToolkit: Tools for Performance Analysis of Optimized Parallel Programs. *Concurrency Computation : Practice Expererience* 22, 6 (April 2010), 685–701.

[2] Glenn Ammons, Thomas Ball, and James R. Larus. 1997. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *SIGPLAN Conference on Programming Language Design and Implementation*. ACM, NY, NY, USA, 85–96.

[3] R. Barik and V. Sarkar. 2009. Interprocedural Load Elimination for Dynamic Optimization of Parallel Programs. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*. 41–52. https://doi.org/10.1109/PACT.2009.32

[4] J. Adam Butts and Guri Sohi. 2002. Dynamic Dead-instruction Detection and Elimination. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*. 199–210. https://doi.org/10.1145/605397.605419

[5] Milind Chabbi, Wim Lavrijsen, Wibe de Jong, Koushik Sen, John Mellor-Crummey, and Costin Iancu. 2015. Barrier Elision for Production Parallel Programs. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2015)*. ACM, New York, NY, USA, 109–119. https://doi.org/10.1145/2688500.2688502

[6] Milind Chabbi, Xu Liu, and John Mellor-Crummey. 2014. Call Paths for Pin Tools. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. Article 76, 11 pages. https://doi.org/10.1145/2544137.2544164

[7] Milind Chabbi, Xu Liu, and Shasha Wen. 2017. WitchTools. https://github.com/WitchTools/Witch.git. (2017).

[8] Milind Chabbi and John Mellor-Crummey. 2012. DeadSpy: a tool to pinpoint program inefficiencies. In *Proceedings of the 10th International Symposium on Code Generation and Optimization*. 124–134.

[9] Milind Chabbi, Shasha Wen, and Xu Liu. 2018. Featherlight On-the-fly False Sharing Detection. In *Proceedings of the 23th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2018)*. ACM, New York, NY, USA. https://doi.org/10.1145/3178487.3178499

[10] Milind Chabbi, Shasha Wen, Xu Liu, et al. 2014. CCTLib. https://github.com/CCTLib/CCTLib. (2014).

[11] 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 (CGO '10)*. ACM, New York, NY, USA, 42–52. https://doi.org/10.1145/1772954.1772963

[12] P Colella, DT Graves, ND Keen, TJ Ligocki, DF Martin, PW Mc-Corquodale, D Modiano, PO Schwartz, TD Sternberg, and B Van Straalen. 2013. Chombo Software Package for AMR Applications - Design Document. *Lawrence Berkeley National Laboratory Technical Report LBNL-6616E* (2013).

[13] Steven J Deitz, Bradford L Chamberlain, and Lawrence Snyder. 2001. Eliminating redundancies in sum-of-product array computations. In *Proceedings of the 15th International Conference on Supercomputing*. 65–77.

[14] Luiz DeRose, Bill Homer, Dean Johnson, Steve Kaufmann, and Heidi Poxon. 2008. Cray Performance Analysis Tools. In *Tools for High Performance Computing*. Springer Berlin Heidelberg, 191–199. https://doi.org/10.1007/978-3-540-68564-7_12

[15] Jack Dongarra and Michael A Heroux. 2013. Toward a new metric for ranking high performance computing systems. *Sandia Report, SAND2013-4744* 312 (2013), 150.

[16] Paul J. Drongowski. 2007. Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors. https://pdfs.semanticscholar.org/5219/4b43b8385ce39b2b08ecd409c753e0efafe5.pdf. (November 2007).

[17] Ryusuke Egawa, Kazuhiko Komatsu, Shintaro Momose, Yoko Isobe, Akihiro Musa, Hiroyuki Takizawa, and Hiroaki Kobayashi. 2017. Potential of a modern vector supercomputer for practical applications: performance evaluation of SX-ACE. *The Journal of Supercomputing* (07 Mar 2017). https://doi.org/10.1007/s11227-017-1993-y

[18] 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 (OSDI'10)*. USENIX Association, Berkeley, CA, USA, 151–162. http://dl.acm.org/citation.cfm?id=1924943.1924954

[19] Free Software Foundation. 2017. objdump. https://sourceware.org/binutils/docs/binutils/objdump.html. (2017).

[20] GNU. 2014. GNU Binutils. https://www.gnu.org/software/binutils/. (2014). September 2014.

[21] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. 1982. Gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*. ACM Press, New York, NY, USA, 120–126.

[22] Stephan M. Günther and Josef Weidendorfer. 2009. Assessing cache false sharing effects by dynamic binary instrumentation. In *WBIA '09: Proceedings of the Workshop on Binary Instrumentation and Applications*. ACM, New York, NY, USA, 26–33. https://doi.org/10.1145/1791194.1791198

[23] Robert J. Hall. 1992. Call Path Profiling. In *Proceedings of the 14th International Conference on Software Engineering (ICSE '92)*. ACM, New York, NY, USA, 296–306. https://doi.org/10.1145/143062.143147

[24] M. Hirzel and T. Chilimbi. 2001. Bursty tracing: A framework for low overhead temporal profiling. In *ACM Workshop on Feedback-Directed and Dynamic Optimization*. ACM, 117–226.

[25] Intel. 2010. Intel Microarchitecture Codename Nehalem Performance Monitoring Unit Programming Guide. https://software.intel.com/sites/default/files/m/5/2/c/f/1/30320-Nehalem-PMU-Programming-Guide-Core.pdf. (2010).

[26] Intel. 2015. Intel Math Kernel Library (MKL). https://software.intel.com/en-us/intel-mkl. (2015).

[27] Intel. 2017. Intel VTune. https://software.intel.com/en-us/intel-vtune-amplifier-xe. (2017).

[28] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093* (2014).

[29] Yunyun Jiang, Yi Yang, Tian Xiao, Tianwei Sheng, and Wenguang Chen. 2016. DRDDR: A Lightweight Method to Detect Data Races in Linux Kernel. *The Journal of Supercomputing* 72, 4 (April 2016), 1645–1659. https://doi.org/10.1007/s11227-016-1691-1

[30] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and Detecting Real-world Performance Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 77–88. https://doi.org/10.1145/2254064.2254075

[31] Mark Scott Johnson. 1982. Some Requirements for Architectural Support of Software Debugging. In *Proceedings of the First International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS I)*. ACM, New York, NY, USA, 140–148. https://doi.org/10.1145/800050.801837

[32] Milan Jovic, Andrea Adamoli, and Matthias Hauswirth. 2011. Catch Me if You Can: Performance Bug Detection in the Wild. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '11)*. ACM, New York, NY, USA, 155–170. https://doi.org/10.1145/2048066.2048081

[33] Ian Karlin, Abhinav Bhatele, Jeff Keasler, Bradford L. Chamberlain, Jonathan Cohen, Zachary DeVito, Riyaz Haque, Dan Laney, Edward Luke, Felix Wang, David Richards, Martin Schulz, and Charles Still. 2013. Exploring Traditional and Emerging Parallel Programming

Models using a Proxy Application. In *27th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2013)*. Boston, USA.

[34] Baris Kasikci, Thomas Ball, George Candea, John Erickson, and Madanlal Musuvathi. 2014. Efficient Tracing of Cold Code via Bias-free Sampling. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*. USENIX Association, Berkeley, CA, USA, 243–254. http://dl.acm.org/citation.cfm?id=2643634.2643660

[35] Charles Killian, Karthik Nagaraj, Salman Pervez, Ryan Braud, James W. Anderson, and Ranjit Jhala. 2010. Finding Latent Performance Bugs in Systems Implementations. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10)*. ACM, New York, NY, USA, 17–26. https://doi.org/10.1145/1882291.1882297

[36] K. M. Lepak and M. H. Lipasti. 2000. On the Value Locality of Store Instructions. In *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)*. 182–191.

[37] Kevin M. Lepak and Mikko H. Lipasti. 2000. Silent Stores for Free. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 33)*. ACM, New York, NY, USA, 22–31.

[38] Levinthal, David. 2009. Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors, Version 1.0. https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf. (2009).

[39] Linux. 2012. perf_event_open - Linux man page. https://linux.die.net/man/2/perf_event_open. (2012).

[40] Linux. 2015. Linux Perf Tool. https://perf.wiki.kernel.org/index.php/Main_Page. (2015).

[41] Linux. 2017. SIGALTSTACK. http://man7.org/linux/man-pages/man2/sigaltstack.2.html. (2017).

[42] Mikko H. Lipasti and John Paul Shen. 1996. Exceeding the Dataflow Limit via Value Prediction. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 29)*. IEEE Computer Society, Washington, DC, USA, 226–237. http://dl.acm.org/citation.cfm?id=243846.243889

[43] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. 1996. Value Locality and Load Value Prediction. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*. ACM, New York, NY, USA, 138–147. https://doi.org/10.1145/237090.237173

[44] Chien-Lung Liu. 2009. *False Sharing Analysis for Multithreaded Programs*. Master's thesis. National Chung Cheng University.

[45] 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 (ICSE '16)*. ACM, New York, NY, USA, 911–922. https://doi.org/10.1145/2884781.2884784

[46] Tongping Liu and Xu Liu. 2016. Cheetah: Detecting False Sharing Efficiently and Effectively. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO '16)*. ACM, New York, NY, USA, 1–11. https://doi.org/10.1145/2854038.2854039

[47] Tongping Liu, Chen Tian, Hu Ziang, and Emery D. Berger. 2014. Predator: Predictive False Sharing Detection. In *Proceedings of 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'14)*. ACM, New York, NY, USA.

[48] X. Liu and J. Mellor-Crummey. 2013. Pinpointing data locality bottlenecks with low overhead. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 183–193. https://doi.org/10.1109/ISPASS.2013.6557169

[49] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 190–200. https://doi.org/10.1145/1065010.1065034

[50] Gabriel Marin and John Mellor-Crummey. 2008. Pinpointing and Exploiting Opportunities for Enhancing Data Reuse. In *IEEE Intl. Symposium on Performance Analysis of Systems and Software (ISPASS '08)*. IEEE Computer Society, Washington, DC, USA, 115–126.

[51] R. E. McLear, D. M. Scheibelhut, and E. Tammaru. 1982. Guidelines for Creating a Debuggable Processor. In *Proceedings of the First International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS I)*. ACM, New York, NY, USA, 100–106. https://doi.org/10.1145/800050.801833

[52] Pall Melsted, Harold Pimentel, and Lior Pachter. 2014. Near-optimal RNA-Seq quantification. https://github.com/makaho/kallisto. (2014).

[53] Message Passing Interface Forum 1997. *MPI-2: Extensions to the Message Passing Interface Standard*. Message Passing Interface Forum.

[54] Joshua San Miguel, Jorge Albericio, Andreas Moshovos, and Natalie Enright Jerger. 2015. Doppelganger: A Cache for Approximate Computing. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 50–61.

[55] Joshua San Miguel, Mario Badr, and Natalie Enright Jerger. 2014. Load Value Approximation. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. IEEE Computer Society, Washington, DC, USA, 127–139.

[56] Takuya Nakaike, Rei Odaira, Matthew Gaudet, Maged M. Michael, and Hisanobu Tomari. 2015. Quantitative Comparison of Hardware Transactional Memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 144–157. https://doi.org/10.1145/2749469.2750403

[57] NERSC. 2016. NERSC-8 / Trinity Benchmarks. http://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks/. (2016).

[58] Khanh Nguyen and Guoqing Xu. 2013. Cachetor: Detecting Cacheable Data to Remove Bloat. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, New York, NY, USA, 268–278. https://doi.org/10.1145/2491411.2491416

[59] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. 2015. Caramel: Detecting and Fixing Performance Problems that Have Non-intrusive Fixes. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 902–912. http://dl.acm.org/citation.cfm?id=2818754.2818863

[60] A. Nistor, L. Song, D. Marinov, and S. Lu. 2013. Toddler: Detecting performance problems via similar memory-access patterns. In *2013 35th International Conference on Software Engineering (ICSE)*. 562–571. https://doi.org/10.1109/ICSE.2013.6606602

[61] Andrzej Nowak, Ahmad Yasin, Avi Mendelson, and Willy Zwaenepoel. 2015. Establishing a Base of Trust with Performance Counters for Enterprise Workloads. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '15)*. USENIX Association, Berkeley, CA, USA, 541–548. http://dl.acm.org/citation.cfm?id=2813767.2813808

[62] University of Virginia. 2015. Rodinia benchmark suite. http://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php/Rodinia:Accelerating_Compute-Intensive_Applications_with_Accelerators. (2015).

[63] Leonid Oliker, Andrew Canning, Jonathan Carter, John Shalf, and Stéphane Ethier. 2006. Scientific Application Performance on Leading Scalar and Vector Supercomputing Platforms. *International Journal of High Performance Computing Applications* (2006).

[64] Oswaldo Olivo, Isil Dillig, and Calvin Lin. 2015. Static Detection of Asymptotic Performance Bugs in Collection Traversals. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 369–378. https://doi.org/10.1145/2737924.2737966

[65] OProfile development team. 2008. OProfile. http://oprofile.sourceforge.net. (2008).

[66] Oracle. 2017. Oracle Solaris Studio. http://www.oracle.com/technetwork/server-storage/solarisstudio/overview/index.html. (2017).

[67] 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 (EuroSys '10)*. ACM, New York, NY, USA, 335–348. https://doi.org/10.1145/1755913.1755947

[68] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1988. Global Value Numbers and Redundant Computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 12–27. https://doi.org/10.1145/73560.73562

[69] John S. Seng and Dean M. Tullsen. 2005. Architecture-Level Power Optimization—What Are the Limits? *J. Instruction-Level Parallelism* 7 (2005). http://www.jilp.org/vol7/v7paper4.pdf

[70] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. 2011. Managing Performance vs. Accuracy Trade-offs with Loop Perforation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. ACM, New York, NY, USA, 124–134. https://doi.org/10.1145/2025113.2025133

[71] Connie U. Smith and Lloyd G. Williams. 2000. Software Performance Antipatterns. In *Proceedings of the 2Nd International Workshop on Software and Performance (WOSP '00)*. ACM, New York, NY, USA, 127–136. https://doi.org/10.1145/350391.350420

[72] L. Song and S. Lu. 2017. Performance Diagnosis for Inefficient Loops. In *2017 39th International Conference on Software Engineering (ICSE)*.

[73] SPEC Corporation. 2007. SPEC CPU2006 Benchmark Suite. http://www.spec.org/cpu2006. (2007). 3 November 2007.

[74] SPEC Corporation. 2015. SPEC OMP2012 Benchmark Suite. https://www.spec.org/omp2012/. (2015). May 2015.

[75] M. Srinivas, B. Sinharoy, R. J. Eickemeyer, R. Raghavan, S. Kunkel, T. Chen, W. Maron, D. Flemming, A. Blanchard, P. Seshadri, J. W. Kellington, A. Mericas, A. E. Petruski, V. R. Indukuru, and S. Reyes. 2011. IBM POWER7 performance modeling, verification, and evaluation. *IBM JRD* 55, 3 (May-June 2011), 4:1–4:19.

[76] C. Stewart, K. Shen, A. Iyengar, and J. Yin. 2010. EntomoModel: Understanding and Avoiding Performance Anomaly Manifestations. In *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. 3–13. https://doi.org/10.1109/MASCOTS.2010.10

[77] Nathan R. Tallent, John Mellor-Crummey, and Michael W. Fagan. 2009. Binary Analysis for Measurement and Attribution of Program Performance. In *Proceedings of the 2009 ACM PLDI*. ACM, NY, NY, USA, 441–452.

[78] Nathan R. Tallent, John M. Mellor-Crummey, and Allan Porterfield. 2010. Analyzing Lock Contention in Multithreaded Applications. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '10)*. ACM, New York, NY, USA, 269–280. https://doi.org/10.1145/1693453.1693489

[79] M. Valiev, E.J. Bylaska, N. Govind, K. Kowalski, T.P. Straatsma, H.J.J. Van Dam, D. Wang, J. Nieplocha, E. Apra, T.L. Windus, and W.A. de Jong. 2010. NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications* 181, 9 (2010), 1477 – 1489.

[80] Jeffrey S. Vitter. 1985. Random Sampling with a Reservoir. *ACM Trans. Math. Softw.* 11, 1 (March 1985), 37–57. https://doi.org/10.1145/3147.3165

[81] Mark N. Wegman and F. Kenneth Zadeck. 1991. Constant Propagation with Conditional Branches. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 2 (April 1991), 181–210. https://doi.org/10.1145/103135.103136

[82] Shasha Wen, Milind Chabbi, and Xu Liu. 2017. RedSpy: Exploring Value Locality in Software. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 47–61. https://doi.org/10.1145/3037697.3037729

[83] Shasha Wen, Xu Liu, and Milind Chabbi. 2015. Runtime Value Numbering: A Profiling Technique to Pinpoint Redundant Computations. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT) (PACT '15)*. IEEE Computer Society, Washington, DC, USA, 254–265. https://doi.org/10.1109/PACT.2015.29

[84] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. 2010. Finding Low-utility Data Structures. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. ACM, New York, NY, USA, 174–186. https://doi.org/10.1145/1806596.1806617

[85] Cong Yan, Alvin Cheung, Junwen Yang, and Shan Lu. 2017. Understanding Database Performance Inefficiencies in Real-world Web Applications. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management (CIKM '17)*. ACM, New York, NY, USA, 1299–1308. https://doi.org/10.1145/3132847.3132954

[86] Amir Yazdanbakhsh, Gennady Pekhimenko, Bradley Thwaites, Hadi Esmaeilzadeh, Onur Mutlu, and Todd C Mowry. 2016. RFVP: Rollback-free Value Prediction with Safe-to-approximate Loads. *ACM Transactions on Architecture and Code Optimization (TACO)* 12, 4 (2016), 62.