Pinpointing Performance Inefficiencies in Java

Pengfei Su
College of William & Mary, USA
psu@email.wm.edu

Qingsen Wang
College of William & Mary, USA
qwang06@email.wm.edu

Milind Chabbi
Scalable Machines Research, USA
milind@scalablemachines.org

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

ABSTRACT

Many performance inefficiencies such as inappropriate choice of algorithms or data structures, developers' inattention to performance, and missed compiler optimizations show up as *wasteful* memory operations. *Wasteful* memory operations are those that produce/consume data to/from memory that may have been avoided. We present, JXPerf, a lightweight performance analysis tool for pinpointing wasteful memory operations in Java programs. Traditional byte code instrumentation for such analysis (1) introduces prohibitive overheads and (2) misses inefficiencies in machine code generation. JXPerf overcomes both of these problems. JXPerf uses hardware performance monitoring units to sample memory locations accessed by a program and uses hardware debug registers to monitor subsequent accesses to the same memory. The result is a lightweight measurement at the machine code level with attribution of inefficiencies to their provenance — machine and source code within full calling contexts. JXPerf introduces only 7% runtime overhead and 7% memory overhead making it useful in production. Guided by JXPerf, we optimize several Java applications by improving code generation and choosing superior data structures and algorithms, which yield significant speedups.

CCS CONCEPTS

• **General and reference** → **Metrics**; **Performance**; • **Software and its engineering** → **Software maintenance tools**.

KEYWORDS

Java profiler, performance optimization, PMU, debug registers

1 INTRODUCTION

Managed languages, such as Java, have become increasingly popular in various domains, including web services, graphic interfaces, and mobile computing. Although managed languages significantly improve development velocity, they often suffer from worse performance compared with native languages. Being a step removed from the underlying hardware is one of the performance handicaps of programming in managed languages. Despite their best efforts, programmers, compilers, runtimes, and layers of libraries, can easily introduce various subtleties to find performance inefficiencies in managed program executions. Such inefficiencies can easily go unnoticed (if not carefully and periodically monitored) or remain hard to diagnose (due to layers of abstraction and detachment from the underlying code generation, libraries, and runtimes).

Performance profiles abound in the Java world to aid developers to understand their program behavior. Profiling for execution hotspots is the most popular one [8, 10, 15, 19, 24, 26]. Hotspot analysis tools identify code regions that are frequently executed disregarding whether execution is efficient or inefficient (useful or wasteful) and hence significant burden is on the developer to make a judgement call on whether there is scope to optimize a hotspot. Derived metrics such as Cycles-Per-Instruction (CPI) or cache miss ratio offer slightly better intuition into hotspots but are still not a panacea. Consider a loop repeatedly computing the exponential of the same number, which is obviously a wasteful work; the CPI metric simply acclaims such code with a low CPI value, which is considered a metric of goodness.

There is a need for tools that specifically pinpoint wasteful work and guide developers to focus on code regions where the optimizations are demanded. Our observation, which is justified by myriad case studies in this paper, is that many inefficiencies show up as wasteful operations when inspected at the machine code level, and those which involve the memory subsystem are particularly egregious. Although this is not a new observation [5, 42, 46, 47] in native languages, its application to Java code is new and the problem is particularly severe in managed languages. The following inefficiencies often show up as wasteful memory operations.

Algorithmic inefficiencies: frequently performing a linear search shows up as frequently loading the same value from the same memory location.

Data structural inefficiencies: using a dense array to store sparse data where the array is repeatedly reinitialized to store different data items shows up as frequent store-followed-by-store operations to the same memory location without an intervening load operation.

```
142  for (int bit = 0,dual = 1; bit < logn; bit++,dual *= 2) {
143    ...
144    for (int a = 1; a < dual; a++) {
145      ...
146      for (int b = 0; b < n; b += 2*dual) {
147        int i = 2*b ;
148        int j = 2*(b + dual);
149        double z1_real = data[j];
150        double z1_imag = data[j+1];
151        double wd_real = w_real*z1_real - w_imag*z1_imag;
152        double wd_imag = w_real*z1_imag + w_imag*z1_real;
153 ▶      data[j] = data[i] - wd_real;
154        data[j+1] = data[i+1] - wd_imag;
155 ▶      data[i] += wd_real;
156        data[i+1] += wd_imag;
157      }}}
```

Listing 1: Redundant memory loads in SPECjvm 2008 scimark.fft. data[i] is loaded from memory twice in a single iteration whereas it is unmodified between these two loads.

```
; data[j] = data[i] - wd_real
vmovsd 0x10(%r9,%r8,8),%xmm2
vsubsd %xmm0,%xmm2,%xmm2
...
; data[i] += wd_real ;
vaddsd 0x10(%r9,%r8,8),%xmm0,%xmm0
vmovsd %xmm0,0x10(%r9,%r8,8)
```

Figure 1: The assembly code (at&t style) of lines 153 and 155 in Listing 1.

Suboptimal code generations: missed inlining can show up as storing the same values to the same stack locations; missed scalar replacement can show up as loading the same value from the same, unmodified, memory location.

Developers' inattention to performance: recomputing the same method in successive loop iterations can show up as silent stores (consecutive writes of the same value to the same memory). For example, the Java implementation of NPB-3.0 benchmark IS [3] performs the expensive power method inside a loop and in each iteration, the power method pushes the same parameters on the same stack location. Interestingly, this inefficiency is absent in the C version of the code due to a careful implementation where the developer hoisted the power function out of the loop.

This list suffices to provide an intuition about the class of inefficiencies detectable by observing certain patterns of memory operations at runtime. Some recent Java profilers [12, 30, 31, 38, 48] identify inefficiencies of this form. However, these tools are based on exhaustive Java byte code instrumentation, which suffer from two drawbacks: (1) high (up to 200×) runtime overhead, which prevents them from being used for production software; (2) missing insights into lower-level layers e.g., inefficiencies in machine code.

1.1 A Motivating Example

Listing 1 shows a hot loop in a JIT-compiled (JITted) method (compiled with Oracle HotSpot JIT compiler) in SPECjvm2008 scimark.fft [39], a standard implementation of Fast Fourier Transforms. The JITted assembly code of the source code at lines 153 and 155 is shown in Figure 1. Notice the two loads from the memory location data[i] (0x10(%r9,%r8,8)) — once into xmm2 at line 153 and then into xmm0 at line 155. data[i] is unmodified between these two loads. Moreover, i and j differ by at least 2 and never alias to the same memory location (see lines 147 and 148). Unfortunately, the

code generation fails to exploit this aspect and trashes xmm2 at line 153, which results in reloading data[i] at line 155.

With the knowledge of never-alias, we performed scalar replacement — placed data[i] in a temporary that eliminated the redundant load, which yielded a 1.13× speedup for the entire program. Without access to the source code of the commercial Java runtime, we cannot definitively say whether the alias analysis missed the opportunity or the register allocator caused the suboptimal code generation, most likely the former. However, it suffices to highlight the fact that observing the patterns of wasteful memory operations at the machine code level at runtime, divulges inefficiencies left out at various phases of transformation and allows us to peek into what ultimately executes. A more detailed analysis of this benchmark with the optimization guided by JXPERF follows in Section 7.1.

1.2 Contribution Summary

We propose JXPERF to complement existing Java profilers; JXPERF samples hardware performance counters and employs hardware debug registers available in commodity CPUs to identify program inefficiencies that exhibit as wasteful memory operations at runtime. Two key differentiating aspects of JXPERF when compared to a large class of hotspot profilers are its ability to (1) filter out and show code regions that are definitely involved in some kind of inefficiency at runtime (hotspot profilers cannot differentiate whether or not a code region is involved in any inefficiency), and (2) pinpoint the *two parties* involved in wasteful work — e.g., the first instance of a memory access and a subsequent, unnecessary access of the same memory — which offer actionable insights (hotspot profilers are limited to showing only a single party). Via JXPERF, we make the following contributions:

- We show the design and implementation of a lightweight Java inefficiency profiler working on off-the-shelf Java virtual machine (JVM) with no byte code instrumentation to memory accesses.
- We demonstrate that JXPERF identifies inefficiencies at machine code, which can be introduced by poor code generation, inappropriate data structures, or suboptimal algorithms.
- We perform a thorough evaluation on JXPERF and show that JXPERF, with 7% runtime overhead and 7% memory overhead, is able to pinpoint inefficiencies in well-known Java benchmarks and real-world applications, which yield significant speedups after eliminating such inefficiencies.

1.3 Paper Organization

This paper is organized as follows. Section 2 describes the related work and distinguishes JXPERF. Section 3 offers the background knowledge necessary to understand JXPERF. Section 4 highlights the methodology we use to identify wasteful memory operations in Java programs. Section 5 depicts the design and implementation of JXPERF. Section 6 and 7 evaluate JXPERF and show several case studies, respectively. Section 8 discusses the threats to validity. Section 9 presents our conclusions and future work.

2 RELATED WORK

There are a number of commercial and research Java profilers, most of which fall into the two categories: hotspot profilers and inefficiency profilers.

Hotspot Profilers. Profilers such as Perf [26], Async-profiler [35], Jprofiler [15], YourKit [19], VisualVM [10], Oracle Developer Studio Performance Analyzer [8], and IBM Health Center [22] pinpoint hotspots in Java programs. Most hotspot profilers incur low overhead because they use interrupt-based sampling techniques supported by Performance Monitoring Units (PMUs) or OS timers. Hotspot profilers are able to identify code sections that account for a large number of CPU cycles, cache misses, branch mispredictions, heap memory usage, or floating point operations. While hotspot profilers are indispensable, they do not tell whether a resource is being used in a fruitful manner. For instance, they cannot report repeated memory stores of the identical value or result-equivalent computations, which squander both memory bandwidth and processor functional units.

Inefficiency Profilers. Toddler [31] detects repeated memory loads in nested loops. LDoctor [38] combines static analysis and dynamic sampling techniques to reduce Toddler's overhead. However, LDoctor detects inefficiencies in only a small number of suspicious loops instead of in the whole program. Glider [12] generates tests that expose redundant operations in Java collection traversals. MemoizeIt [11] detects redundant computations by identifying methods that repeatedly perform identical computations and output identical results. Xu et al. employ various static and dynamic analysis techniques (e.g., points-to analysis, dynamic slicing) to detect memory bloat by identifying useless data copying [49], inefficiently-used containers [51], low-utility data structures [50], reusable data structures [48] and cacheable data structures [30].

Unlike hotspot profilers, these tools can pinpoint redundant operations that lead to resource wastage. JXPᴇʀꜰ is also an inefficiency profiler. Unlike prior works, which use exhaustive byte code instrumentation, JXPᴇʀꜰ exploits features available in hardware (performance counters and debug registers) that eliminates instrumentation and dramatically reduces tool overheads. JXPᴇʀꜰ detects multiple kinds of wasteful memory access patterns. Furthermore, JXPᴇʀꜰ can be easily extended with additional memory access patterns for detecting other kinds of runtime inefficiencies. Section 6 details the comparison between JXPᴇʀꜰ and the profilers based on exhaustive byte code instrumentation.

Remix [14], similar to JXPᴇʀꜰ, also utilizes PMU; while JXPᴇʀꜰ identifies intra-thread inefficiencies, such as redundant/useless operations, Remix identifies false sharing across threads.

3 BACKGROUND

Hardware Performance Monitoring Unit (PMU). Modern CPUs expose programmable registers that count various hardware events such as memory accesses, CPU cycles, and many others. The registers can be configured in sampling mode: when a threshold number of hardware events elapse, PMUs trigger an overflow interrupt. A profiler is able to capture the interrupt as a signal (aka sample), and attribute the metrics collected along with the sample to the execution context. PMUs are per CPU core and virtualized by the operating system (OS) for each thread.

Intel offers Precise Event-Based Sampling (PEBS) [6] in Sandy-Bridge and following generations. PEBS provides the effective address (EA) at the time of sample when the sample is for a memory access instruction such as a load or a store. This facility is often

referred to as address sampling, which is a critical building block of JXPᴇʀꜰ. Also, PEBS can capture the precise instruction pointer (IP) for the instruction resulting in the counter overflow. AMD Instruction-Based Sampling (IBS) [13] and PowerPC Marked Events (MRK) [40] offer similar capabilities.

Hardware Debug Register. Hardware debug registers [23, 27] trap the CPU execution for debugging when the program counter (PC) reaches an address (breakpoint) or an instruction accesses a designated address (watchpoint). One can program debug registers to trap on various conditions: accessing addresses, accessing widths (1, 2, 4, 8 bytes), and accessing types (W_TRAP and RW_TRAP). The number of hardware debug registers is limited; modern x86 processors have only four debug registers.

Linux perf_event. Linux offers a standard interface to program PMUs and debug registers via the perf_event_open system call [25] and the associated ioctl calls. A watchpoint exception is a synchronous CPU trap caused when an instruction accesses a monitored address, while a PMU sample is a CPU interrupt caused when an event counter overflows. Both PMU samples and watchpoint exceptions are handled via Linux signals. The user code can (1) mmap a circular buffer to which the kernel keeps appending the PMU data on each sample and (2) extract the signal context on each watchpoint exception.

Java Virtual Machine Tool Interface (JVMTI). JVMTI [9] is a native programming interface of the JVM. A JVMTI client can develop a debugger/profiler (aka JVMTI agent) in any C/C++ based native language to inspect the state and control the execution of JVM-based programs. JVMTI provides a number of event callbacks to capture JVM initialization and death, thread creation and destruction, method loading and unloading, garbage collection start and end, to name a few. User-defined functions are registered in these callbacks and invoked when the associated events happen. In addition, JVMTI maintains a variety of information for queries, such as the map from the machine code of each JITted method to byte code and source code, and the call path for any given point during the execution. JVMTI is available in off-the-shelf Oracle HotSpot JVM.

4 METHODOLOGY

In the context of this paper, we define the following three kinds of wasteful memory accesses.

Definition 1 (Dead store). *S_1 and S_2 are two successive memory stores to location M (S_1 occurs before S_2). S_1 is a dead store iff there are no intervening memory loads from M between S_1 and S_2. In such a case, we call $\langle S_1, S_2 \rangle$ a dead store pair.*

Definition 2 (Slient store). *A memory store S_2, storing a value V_2 to location M, is a silent store iff the previous memory store S_1 performed on M stored a value V_1, and $V_1 = V_2$. In such a case, we call $\langle S_1, S_2 \rangle$ a silent store pair.*

Definition 3 (Silent load). *A memory load L_2, loading a value V_2 from location M is a silent load iff the previous memory load L_1 performed on M loaded a value V_1, and $V_1 = V_2$. In such a case, we call $\langle L_1, L_2 \rangle$ a silent load pair.*
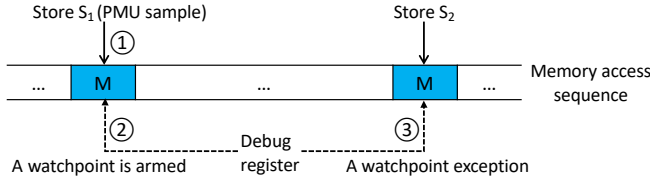
Figure 2: JXPerf's scheme for silent store detection. ① The PMU samples a memory store S_1 that touches location M. ② In the PMU sample handler, a debug register is armed to monitor subsequent access to M. ③ The debug register traps on the next store S_2 to M. ④ If S_1 and S_2 write the same values to M, JXPerf labels S_2 as a silent store and $\langle S_1, S_2 \rangle$ as a silent store pair.

Silent stores and silent loads are value-aware inefficiencies whereas dead stores are value-agnostic ones. We perform precise equality check on integer values, and approximate equality check on floating-point (FP) values within a user-specified threshold of difference (1% by default), given the fact that all FP numbers under the IEEE 754 format are approximately represented in the machine. For memory operations involved in inefficiencies, we typically use their calling contexts instead of their effective addresses to represent them, which can facilitate optimization efforts.

Figure 2 highlights the idea of JXPerf in detecting inefficiencies at runtime, exemplified with silent stores. PMU drives JXPerf by sampling precise memory stores. For a sampled store operation, JXPerf records the effective address captured by the PMU, reads the value at this address, and sets a trap-on-store watchpoint at this address via the debug register. The subsequent store to the same effective address in the execution will trap. JXPerf captures the trap and checks the value at the effective address of the trap. If the value remains unchanged between the two consecutive accesses, JXPerf reports a pair of silent stores. The watchpoint is disabled, and the execution continues as usual to detect more such instances.

5 DESIGN AND IMPLEMENTATION

Figure 3 overviews JXPerf in the entire system stack. JXPerf requires no modification to hardware (x86), OS (Linux), JVM (Oracle HotSpot), and monitored Java applications. In this section, we first describe the implementation details of JXPerf in identifying wasteful memory operations, then show how JXPerf addresses the challenges, and finally depict how JXPerf provides extra information to guide code optimization. JXPerf produces per-thread profiles and merges them to provide an aggregate view as the output.

5.1 Lightweight Inefficiency Detection

Silent Store Detection.

(1) JXPerf subscribes to the precise PMU store event at the JVM initialization callback and sets up PMUs and debug registers for each thread via perf_event API at the thread creation callback.
(2) When a PMU counter overflows during the execution, it triggers an interrupt. JXPerf captures the interrupt, constructs the calling context C_1 of the interrupt, and extracts the effective address M and the value V_1 stored at M.
(3) JXPerf sets a W_TRAP (trap-on-store) watchpoint at M and resumes program execution.
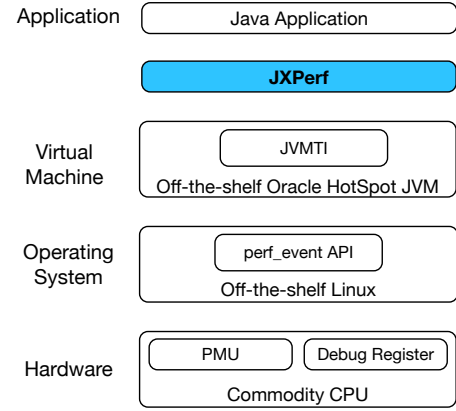

Figure 3: Overview of JXPerf in the system stack.

(4) A subsequent store to M triggers a trap. JXPerf handles the trap signal, constructs the calling context C_2 of the trap, and inspects the value V_2 stored at M.
(5) JXPerf compares V_1 and V_2. If $V_1 = V_2$, a silent store is detected, and JXPerf labels the context pair $\langle C_1, C_2 \rangle$ as an instance of silent store pair.
(6) JXPerf disarms the debug register and resumes execution.

Dead Store Detection. JXPerf subscribes to the precise PMU store event for dead store detection. When a PMU counter overflows, JXPerf constructs the calling context C_1 of the interrupt, extracts the effective address M, sets a RW_TRAP (load and store) watchpoint at M, and resumes program execution. When the subsequent access traps, JXPerf examines the access type (store or load). If it is a store, JXPerf constructs the calling context C_2 of the trap and records the pair $\langle C_1, C_2 \rangle$ as an instance of dead store pair. Otherwise, it is not a dead store.

Silent Load Detection. The detection is similar to the silent store detection, except that JXPerf subscribes to the precise PMU load event and sets a RW_TRAP watchpoint [1] to trap the subsequent access to the same memory address. If the watchpoint triggers on a load that reads the same value as the previous load from the same location, JXPerf reports an instance of silent load pair.

The following metrics compute the fraction of wasteful memory operations in an execution:

$$
\begin{aligned}
\mathcal{F}_{prog}^{DeadStore} &= \frac{\sum_i \sum_j \text{Dead bytes stored in} \langle C_i, C_j \rangle}{\sum_i \sum_j \text{Bytes stored in} \langle C_i, C_j \rangle} \\
\mathcal{F}_{prog}^{SilentStore} &= \frac{\sum_i \sum_j \text{Silent bytes stored in} \langle C_i, C_j \rangle}{\sum_i \sum_j \text{Bytes stored in} \langle C_i, C_j \rangle} \\
\mathcal{F}_{prog}^{SilentLoad} &= \frac{\sum_i \sum_j \text{Silent bytes loaded from} \langle C_i, C_j \rangle}{\sum_i \sum_j \text{Bytes loaded from} \langle C_i, C_j \rangle}
\end{aligned}
\tag{1}
$$

Fraction of wasteful memory operations in a calling context pair is given as follows:

$$
\begin{aligned}
\mathcal{F}_{\langle C_{watch}, C_{trap} \rangle}^{DeadStore} &= \frac{\text{Dead bytes stored in} \langle C_{watch}, C_{trap} \rangle}{\sum_i \sum_j \text{Bytes stored in} \langle C_i, C_j \rangle} \\
\mathcal{F}_{\langle C_{watch}, C_{trap} \rangle}^{SilentStore} &= \frac{\text{Silent bytes stored in} \langle C_{watch}, C_{trap} \rangle}{\sum_i \sum_j \text{Bytes stored in} \langle C_i, C_j \rangle} \\
\mathcal{F}_{\langle C_{watch}, C_{trap} \rangle}^{SilentLoad} &= \frac{\text{Silent bytes loaded from} \langle C_{watch}, C_{trap} \rangle}{\sum_i \sum_j \text{Bytes loaded from} \langle C_i, C_j \rangle}
\end{aligned}
\tag{2}
$$

---

[1] x86 debug registers do not offer trap-only-on-load facility.

5.2 Limited Number of Debug Registers

Hardware offers a small number of debug registers, which becomes a limitation if the PMU delivers a new sample before a previously set watchpoint traps. To better understand the problem, consider the silent load example in Listing 2. Assume the loop indices i and j, and the scalars sum1 and sum2 are in registers. Further assume the PMU is configured to deliver a sample every 1K memory loads and the number of debug register is only one. The first sample occurs in loop i when accessing array[1K], which results in setting a watchpoint to monitor the address of array[1K]. The second sample occurs when accessing array[2K]. Since the watchpoint armed at array[1K] is still active, we should either forgo monitoring it in favor of array[2K] or ignore the new sample. The former choice allows us to potentially detect a pair of silent loads separated by many intervening loads, and the latter choice allows us to detect a pair of silent loads separated by only a few intervening loads. The option is not obvious without looking into the future. A naive "*replace the oldest policy*" is futile as it will not detect a single silent load in the above example. Even a slightly smart *exponential decay* strategy will not work because the survival probability of an old watchpoint will be minuscule over many samples.

JXPERF employs reservoir sampling [44, 45, 47], which uniformly chooses between old and new samples with no temporal bias. The first sampled address M_1, occupies the debug register with 1.0 probability. The second sampled address M_2, occupies the previously armed watchpoint with $1/2$ probability and retains M_1 with $1/2$ probability. The third sampled address M_3, either occupies the previously armed watchpoint with $1/3$ probability or retains it (M_1 or M_2) with $2/3$ probability. The ith sampled address M_i since the last time a debug register was available, replaces the previously armed watchpoint with $1/i$ probability. The probability P_k of monitoring any sampled address M_k, $1 \le k \le i$, is the same ($1/i$), ensuring uniform sampling over time. When a watchpoint exception occurs, JXPERF disarms that watchpoint and resets its reservoir (replacement) probability to 1.0. Obviously, with this scheme JXPERF does not miss any sample if every watchpoint traps before being replaced.

The scheme trivially extends to more number of debug registers, say $N \ge 1$. JXPERF maintains an independent reservoir probability P_α for each debug register α, $(1 \le \alpha \le N)$. On a PMU sample, if there is an available debug register, JXPERF arms it and decrements the reservoir probability of other already-armed debug registers; otherwise JXPERF visits each debug register α and attempts to replace it with the probability P_α. The process may succeed or fail in arming a debug register for a new sample, but it gives a new sample N chances to remain in a system with N watchpoints. Whether success or failure, P_α of each in-use debug register is updated after a sample. The order of visiting the debug registers is randomized for each sample to ensure fairness. Notice that this scheme maintains only a count of previous samples (not an access log), which consumes $O(1)$ memory.

5.3 Interference of the Garbage Collector

Garbage collection (GC) can move live objects from one memory location to another. Without paying heed to GC events, JXPERF can introduce two kinds of errors: (1) it may erroneously attribute an instance of inefficiency (e.g., dead store) to a location that is in

```
1 for (int i = 1; i <= 10K; i++) sum1 += array[i];
2 for (int j = 1; j <= 10K; j++) sum2 += array[j]; // silent loads
```

Listing 2: Long-distance silent loads. All four watchpoints are armed in the first four samples taken in loop i when the sampling period is 1K memory loads. Naively replacing the oldest watchpoint will not trigger a single watchpoint owing to many samples taken in loop i before reaching loop j. JXPERF employs the reservoir sampling to ensure each sample equal probability to survive.

reality occupied by two different objects between two consecutive accesses by the same thread; (2) it may miss attributing an inefficiency metric to an object because it was moved from one memory location to another between two consecutive accesses by the same thread.

If JXPERF were able to query the garbage collector for moved objects or addresses, it could have avoided such errors, however, no such facility exists to the best of our knowledge in commercial JVMs. JXPERF's solution is to monitor accesses only between GC epochs. JXPERF captures the start and end points of GC by registering the JVMTI callbacks GarbageCollectionStart and GarbageCollectionFinish to demarcate epochs. Watchpoints armed in an older epoch are not carried over to a new epoch: the first PMU sample or watchpoint trap that happens in a thread in a new epoch disarms all active watchpoints in that thread and begins afresh with a reservoir sampling probability of 1.0 for all debug registers for that thread. Note that the GC thread is never monitored. Typically, two consecutive accesses separated by a GC is infrequent; for example, the ratio of $\frac{\text{\# of GCs}}{\text{\# of PMU samples}}$ is 4.4e-5 in Dacapo-9.12-MR1-bach eclipse [4].

5.4 Attributing Measurement to Binary

JXPERF uses Intel XED library [7] for on-the-fly disassembly of JITed methods. JXPERF retains the disassembly for post-mortem inspection if desired. It also uses XED to determine whether a watchpoint trap was caused by a load or a store instruction.

A subtle implementation issue is in extracting the instruction that causes the watchpoint trap. JXPERF uses the perf_event API to register a HW_BREAKPOINT perf event (watchpoint event) for a monitored address. Although the watchpoint causes a trap immediately after the instruction execution, the instruction pointer (IP) seen in the signal handler context (contextIP) is one ahead of the actual IP (trapIP) that triggers the trap. In the x86 variable-length instruction set, it is nontrivial to derive the trapIP, even though it is just one instruction before the contextIP. The HW_BREAKPOINT event in perf_event is not a PMU event; hence, the Intel PEBS support, which otherwise provides the precise register state, is unavailable for a watchpoint. JXPERF disassembles every instruction from the method beginning till it reaches the IP just before the contextIP. The expensive disassembly is amortized by caching results for subsequent traps that often happen at the same IP. The caching is particularly important in methods with a large body; for example, when detecting silent loads in Dacapo-9.12-MR1-bach eclipse, without caching JXPERF introduces 4× runtime overhead.

5.5 Attributing Measurement to Source Code

Attributing runtime statistics to a flat profile (just an instruction) does not provide the full details needed for developer

action. For example, attributing inefficiencies to a common JDK method, e.g., `string.equals()`, offers little insight since `string.equals()` can be invoked from several places in a large code base; some invocations may not even be obvious to the user code. A detailed attribution demands associating profiles with the full calling context: `packageA.classB.methodC:line#.-> ...->java.lang.String.equals():line#`. Thus, JXPerf requires obtaining the calling context where a PMU sample occurs and the calling context where a watchpoint traps.

Obtaining Calling Contexts without Safepoint Bias. Oracle JDK offers users two APIs to obtain calling contexts: officially documented `GetStackTrace()` and undocumented `AsyncGetCallTrace()`. Profilers that use `GetStackTrace()` suffer from the safepoint bias since JVM requires the program to reach a safepoint before collecting any calling context [21, 29]. To avoid the bias, JXPerf employs `AsyncGetCallTrace()` to facilitate non-safepoint collection of calling contexts [32]. `AsyncGetCallTrace()` accepts `u_context` obtained from the PMU interrupts or debug register traps as the input, and returns the method ID and byte code index (BCI) for each stack frame in the calling context. Method ID uniquely identifies distinct methods and distinct JITted instances of the same method (a single method may be JITted multiple times). With the method ID, JXPerf is able to obtain the associated class name and method name by querying JVM via JVMTI. To obtain the line number, JXPerf maintains a "`BCI->line number`" mapping table for each method instance by querying JVM via JVMTI API `GetLineNumberTable()`. As a result, for any given BCI, JXPerf returns its line number by looking up the mapping table.

5.6 Post-mortem Analysis

JXPerf produces per-thread profiles to minimize thread synchronization overhead during program execution. JXPerf coalesces these per-thread profiles into a single profile in a post-mortem fashion. The coalescing procedure follows the rule: two silent load (silent store or dead store) pairs from different threads are coalesced *iff* they have the same loads (stores) in the same calling contexts. All metrics are also aggregated across threads. Typically, it takes less than one minute to merge all profiles according to our experiments.

6 EVALUATION

We evaluate JXPerf on an 18-core Intel Xeon E5-2699 v3 CPU of 2.30GHz frequency running Linux 4.8.0. The machine has 128GB main memory. JXPerf is built with Oracle JDK11 and complied with `gcc-5.4.1 -O3`. The Oracle HotSpot JVM is run in the server mode. JXPerf samples the PMU event `MEM_UOPS_RETIRED:ALL_STORES` to detect dead stores and silent stores, and `MEM_UOPS_RETIRED:ALL_LOADS` to detect silent loads.

We evaluate JXPerf on three well-known benchmark suites — DaCapo 2006 [4], Dacapo-9.12-MR1-bach [4] and ScalaBench [37]. Additionally, we use two real-world performance bug datasets [1, 28]. All programs are built with Oracle JDK11 except DaCapo 2006 bloat, Dacapo-9.12-MR1-bach batik and eclipse, and ScalaBench actors with Oracle JDK8 due to the incompatibility. We apply the large input for DaCapo 2006, Dacapo-9.12-MR1-bach and ScalaBench, and the default inputs released with the remaining programs if not specified. Parallel programs, excluding threads used for the JIT
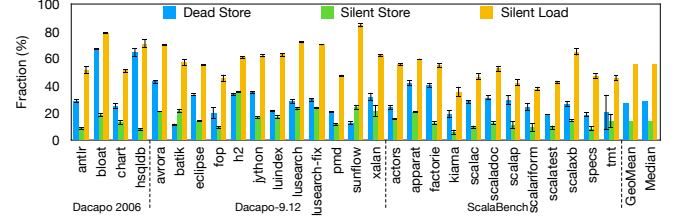

Figure 4: Fraction of wasteful memory operations on DaCapo 2006, Dacapo-9.12-MR1-bach and ScalaBench benchmark suites at the sampling periods of 500K, 1M, 5M and 10M. The error bars are for different sampling periods.
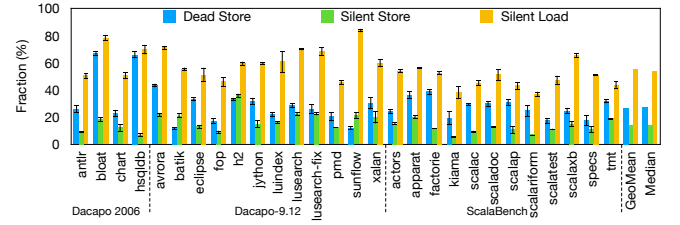

Figure 5: Fraction of wasteful memory operations on DaCapo 2006, Dacapo-9.12-MR1-bach and ScalaBench benchmark suites by using different numbers of debug registers at the 5M sampling period. The error bars are for different number of debug registers.

compilation and GC, are run with four threads if users are allowed to specify the number of threads.

To deal with the impact of the non-deterministic execution (e.g., non-deterministic GC) of Java programs on experimental results, we refer to Georges et al.'s work [17] to use a confidence interval for the mean to report results. The confidence interval for the mean is computed based on the following formula where n is the number of samples, $\overline{x}$ is the mean, σ is the standard deviation and z is a statistic determined by the confidence interval. In our experiments, we run each benchmark 30 times (i.e., $n = 30$) and use a 95% confidence interval (i.e., $z = 1.96$).

$$\overline{x} \pm z \times \frac{\sigma}{\sqrt{n}} \tag{3}$$

In the rest of this section, we first show the fraction of wasteful memory operations — dead stores, silent stores and silents loads on DaCapo 2006, Dacapo-9.12-MR1-bach, and ScalaBench benchmark suites at different sampling periods and different number of debug registers. We then evaluate the overhead of JXPerf on these benchmarks. We exclude three benchmarks — Dacapo-9.12-MR1-bach `tradesoap`, `tradebeans`, and `tomcat` — from monitoring because of the huge variance in execution time of the native run (`tradesoap` and `tradebeans`) or runtime errors of the native run (`tomcat`). Finally, we evaluate the effectiveness of JXPerf on the known performance bug datasets reported by existing tools.

Fraction of Wasteful Memory Operations. Figure 4 shows the fraction of dead stores, silent stores, and silent loads on DaCapo 2006, Dacapo-9.12-MR1-bach, and ScalaBench benchmark suites at
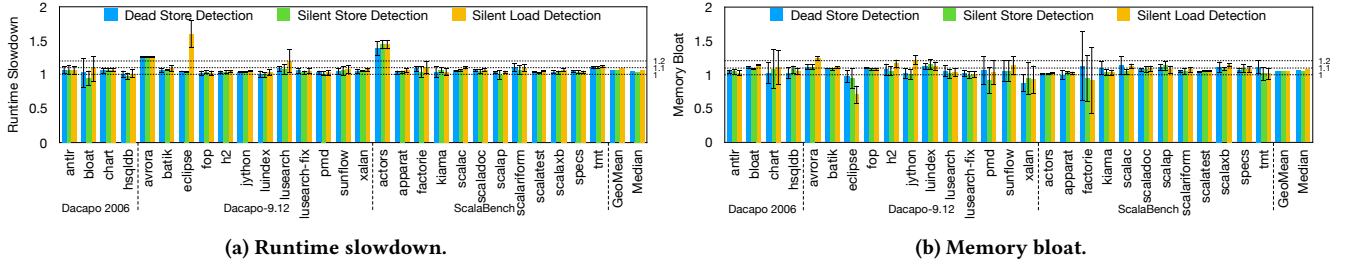
(a) Runtime slowdown.

(b) Memory bloat.

Figure 6: Runtime slowdown (×) and memory bloat (×) of JXPᴇʀꜰ at the 5M sampling period on DaCapo 2006, Dacapo-9.12-MR1-bach and ScalaBench benchmark suites.

Table 1: Geometric mean and median of runtime slowdown (×) and memory bloat (×) of JXPᴇʀꜰ at different sampling periods on DaCapo 2006, Dacapo-9.12-MR1-bach and ScalaBench benchmark suites (DS: dead store, SS: silent store, SL: silent load).

GeoMean/Median		Sampling Period			
		500K	1M	5M	10M
DS Detection	Slowdown	1.18/1.18	1.11/1.1	1.07/1.05	1.04/1.03
	Memory bloat	1.06/1.08	1.06/1.08	1.05/1.06	1.04/1.05
SS Detection	Slowdown	1.16/1.14	1.1/1.1	1.06/1.04	1.05/1.04
	Memory bloat	1.06/1.07	1.06/1.06	1.04/1.05	1.05/1.05
SL Detection	Slowdown	1.35/1.34	1.24/1.21	1.1/1.07	1.07/1.05
	Memory bloat	1.19/1.17	1.11/1.1	1.05/1.07	1.06/1.06

Table 2: Effectiveness of JXPᴇʀꜰ. Toddler and Glider report 33 and 46 performance bugs from eight real-world applications, among which JXPᴇʀꜰ succeeds in reproducing 31 and 44 bugs, respectively.

Application	# of bugs reported by Toddler/Glider	# of bugs reproduced by JXPerf
Apache Ant	5/6	4/5
Apache Collections	21/16	20/16
Apache Groovy	1/6	1/6
Apache Lucene	0/1	0/1
Google Guava	4/9	4/9
JFreeChart	1/3	1/2
JDK	1/0	1/0
PDFBox	0/5	0/5
Sum	**33/46**	**31/44**

the sampling periods of 500K, 1M, 5M, and 10M. The following two takeaways are obvious:

- The inefficiencies, such as dead stores, silent stores, and silent loads, pervasively exist in Java programs.
- The sampling period does not significantly impact the fraction of inefficiencies in Java programs.

We further vary the number of debug registers from one to four to observe the variance in results at the same sampling period — 5M, as shown in Figure 5. We find the number of debug registers has minuscule impacts on the results except for a couple of short-running (e.g., < 2s) benchmarks such as luindex and kiama, which validates the strength of the reservoir sampling. We check the top five inefficiency pairs and their percentage contributions and find negligible variance across different sampling periods and different number of debug registers.

Overhead. Runtime slowdown (memory bloat) is measured as the ratio of the runtime (peak memory usage) of a benchmark with

JXPᴇʀꜰ enabled to the runtime (peak memory usage) of its native execution. Table 1 shows the geometric mean and median of runtime slowdown and memory bloat at different sampling periods. When the sampling period increases (i.e., sampling rate decreases), the overhead drops as expected. We emperically find the 5M sampling period yields a good tradeoff between overhead and accuracy, which typically incurs 7% runtime slowdown and 7% memory bloat.

Figure 6 quantifies the overhead of JXPᴇʀꜰ on each benchmark at the 5M sampling period. Silent load detection typically has a higher overhead than the other two because loads are more common than stores in a program execution. Moreover, JXPᴇʀꜰ sets the RW_TRAP (trap-only-on-load watchpoints are unavailable in x86 processors), which triggers an exception on both stores (ignored) and loads. From the program perspective, silent load detection for eclipse incurs higher runtime overhead than others because it executes more load operations and has more methods of large size that require JXPᴇʀꜰ to take more efforts to correct the off-by-one error at each watchpoint trap. Furthermore, due to the non-deterministic behavior of GC, the peak memory usage for a couple of benchmarks with JXPᴇʀꜰ enabled is less than the native run (e.g., eclipse, xalan) or varies significantly among different runs (e.g., factorie).

Effectiveness. We investigate the performance bugs reported by several state-of-the-art tools such as Toddler [31], Clarity [34], Glider [12], and LDoctor [38]. Among them, the developers of Toddler and Glider share their bug datasets and test cases that expose the bugs online [1, 28]. Therefore, we validate the effectiveness of JXPᴇʀꜰ by checking whether the bugs reported by Toddler and Glider can also be identified by JXPᴇʀꜰ. Toddler and Glider are both built atop Soot [43] to identify a restricted class of performance issues: redundant operations involved in Java collection traversals, of which the symptom is silent loads. It is worth noting that the runtime overheads of Toddler and Glider are ~16× and ~150×, respectively.

Table 2 shows the comparison results. Toddler reports 33 bugs (we exclude the bugs whose source codes or test cases are no longer available), among which JXPᴇʀꜰ misses only two bugs: Apache Ant#53637 and Apache Collections#409. Glider reports 46 bugs, among which JXPᴇʀꜰ misses only two bugs: Apache Ant#53637 and JFreeChart (unknown bug ID). Take Apache Collections#588, one of the reported bugs, as an example to illustrate how JXPᴇʀꜰ identifies it. Listing 3 shows the inefficient implementation of method retainAll() in Apache Collections#588. JXPᴇʀꜰ reports that 49% of silent loads are associated with method contains() at

```
1 public boolean retainAll(final Collection<?> coll) {
2   if (coll != null) {
3     boolean modified = false;
4     final Iterator<E> e = iterator();
5     while (e.hasNext()) {
6 ▶     if (!coll.contains(e.next())) {
7         e.remove();
8         modified = true;
9       }
10    }
11    return modified;
12  } else return decorated().retainAll(null);
13 }
```

Listing 3: Inefficient implementation of method `retainAll()` in Apache Collections#588. JXPerf reports that 49% of silent loads are associated with method `contains()` at line 6 when the parameter `coll` is of type list.

line 6 when the parameter `Collection coll` is of type list. For each element in `Iterator e`, `contains()` performs a linear search over `coll` to check whether `coll` contains this element. Consequently, elements in `coll` are repeatedly traversed whereas their values remain unchanged, which shows up as silent loads. Converting `coll` to a hash set is a superior choice of data structure that enables $O(1)$ search algorithm and dramatically reduces the number of loads and also the fraction of silent loads.

All the missed performance bugs fall into the same category: inefficiency observed in adjacent memory locations rather than the same memory location. We take `Apache Ant#53637` as an example to illustrate why JXPerf misses it. The method "`A.addAll(int index, Collection B)`" in Ant#53637 requires inserting elements of Collection A one by one into the location "index" of Collection B. In each insertion, elements at and behind the location "index" of B have to be shifted. Consequently, elements in B suffer from the repeated shifts. The symptom of such inefficiency is that the same value is repeatedly loaded from adjacent memory locations. JXPerf only identifies silent loads that repeatedly read the same value from the same memory location. JXPerf can be extended with a heuristic to record values at adjacent locations at the sample point and compare them at the watchpoint trap. It is worth noting that inefficiencies identified by Toddler, Clarity, Glider, and LDoctor are mostly related to load operations, whereas JXPerf also identifies significant store-related inefficiencies.

7 CASE STUDIES

In addition to confirming the performance bugs reported by existing tools, we apply JXPerf on more benchmark suites — DaCapo 2006 [4], SPECjvm2008 [39], NPB-3.0 [3] and Grande-2.0 [33], and real-world applications — SableCC-3.7 [16], FindBugs-3.0.1 [36], JFreeChart-1.0.19 [18] to identify varieties of inefficiencies.

Table 3 summarizes the newly found performance bugs via JXPerf as well as previously found ones but with different insights provided by JXPerf. All programs are built with Oracle JDK11 except Dacapo 2006 bloat and FindBugs-3.0.1, which are built with Oracle JDK8. We measure the performance of all programs in execution time except SPECjvm2008 scimark.fft, which is in throughput. We run each program 30 times and use a 95% confidence interval for the mean speedup to report the performance improvement. In the rest of this section, we study each program shown in Table 3 except FindBugs-3.0.1 and JFreeChart-1.0.19, which are posted on arXiv [41].

```
--------------------------------------------------------------------------------
spec.harness.BenchmarkThread.run(BenchmarkThread.java:59)
 spec.harness.BenchmarkThread.executeIteration(BenchmarkThread.java:82)
  spec.harness.BenchmarkThread.runLoop(BenchmarkThread.java:170)
   spec.benchmarks.scimark.fft.Main.harnessMain(Main.java:36)
    spec.benchmarks.scimark.fft.Main.runBenchmark(Main.java:27)
     spec.benchmarks.scimark.fft.FFT.main(FFT.java:89)
      spec.benchmarks.scimark.fft.FFT.run(FFT.java:246)
       spec.benchmarks.scimark.fft.FFT.measureFFT(FFT.java:231)
        spec.benchmarks.scimark.fft.FFT.test(FFT.java:70)
         spec.benchmarks.scimark.fft.FFT.inverse(FFT.java:52)
          vmovsd 0x10(%r9,%r8,8),%xmm2:...transform_internal(FFT.java:153)
*************************REDUNDANT WITH*********************************
spec.harness.BenchmarkThread.run(BenchmarkThread.java:59)
 spec.harness.BenchmarkThread.executeIteration(BenchmarkThread.java:82)
  spec.harness.BenchmarkThread.runLoop(BenchmarkThread.java:170)
   spec.benchmarks.scimark.fft.Main.harnessMain(Main.java:36)
    spec.benchmarks.scimark.fft.Main.runBenchmark(Main.java:27)
     spec.benchmarks.scimark.fft.FFT.main(FFT.java:89)
      spec.benchmarks.scimark.fft.FFT.run(FFT.java:246)
       spec.benchmarks.scimark.fft.FFT.measureFFT(FFT.java:231)
        spec.benchmarks.scimark.fft.FFT.test(FFT.java:70)
         spec.benchmarks.scimark.fft.FFT.inverse(FFT.java:52)
          vaddsd 0x10(%r9,%r8,8),%xmm0,%xmm0:...transform_internal(FFT.java:155)
--------------------------------------------------------------------------------
```

Figure 7: A silent load pair with full calling contexts reported by JXPerf in SPECjvm2008 scimark.fft.

7.1 SPECjvm2008 Scimark.fft: Silent Loads

With the large input and four threads, JXPerf reports 33% of memory loads are silent. The top two silent load pairs are attributed to lines 153 and 155, and lines 154 and 156 in Listing 1, which account for 27% of the total silent loads. They both suffer from the same performance issue: poor code generation detailed in Section 1.1. We take lines 153 and 155 as an example to illustrate our optimization, of which the culprit calling contexts are shown in Figure 7. We employ scalar replacement to eliminate such intra-iteration silent loads. In each iteration, we store the value of `data[i]` in a temporary before performing line 153, which enables `data[i]` to be loaded only once in each loop iteration. We also eliminate the silent loads between lines 154 and 156 using the same approach. They together eliminate 15% of the total memory loads and yield a $(1.13\pm0.02)\times$ speedup for the entire program.

7.2 Grande-2.0 Euler: Dead Stores

Euler [33] employs a structured mesh to solve the time-dependent Euler equations. JXPerf identifies 46% stores are dead. One of the top dead store pairs is associated with the variable `temp2.a` at lines 5 and 12 in Listing 4, which appears in a loop nest (not shown). By inspecting the JITted assembly code shown in Figure 8, we find the value of `temp2.a` computed at line 5 is held in a register, which is reused at line 7. However, the memory store to `temp2.a` at line 5 is not eliminated. As a result, the memory store to `temp2.a` at line 12 overwrites the previous memory store to `temp2.a` at line 5. Although CPUs buffer stores, workloads with many store operations, such as Euler, can cause CPU stalls due to store buffers filling up [54].

To eliminate the dead stores, we use a temporary to replace `temp2.a` at lines 5, 7, and 12 to avoid using `temp2.a`. JXPerf also identifies other dead store pairs with the same issue and guides the same optimization. Our optimization eliminates 59% of total memory stores and yields a $(1.1\pm0.02)\times$ speedup. Our optimization is safe because `temp2` is a local object defined in the method `calculateDamping` (line 2) to store the intermediate results; the object it refers to is never referenced by any other variable.

Table 3: Overview of performance improvement guided by JXPᴇʀꜰ.

Program		Inefficiency			Optimization	
		Code	Type	Root cause	Patch	Speedup (×)
Macro benchmark	✓SPECjvm2008 scimark.fft	FFT.java:loop (153-156)	SL	Poor machine code generation	Scalar replacement	1.13±0.02
	✓NPB-3.0 IS	Random.java: randlc	SS	Redundant method invocations	Reusing the previous result	1.89±0.04
	✓Grande-2.0 Euler	Tunnel.java:calculateR Tunnel.java:calculateDamping	DS	Poor machine code generation	Scalar replacement	1.1±0.02
Real-world application	✓SableCC-3.7	Grammar.java (15,16,64,65) LR0Collection.java (16,57,82,112) LR1Collection.java (16,17,27,28,33,34) LR0ItemSet.java (15,20,26) LR1ItemSet.java (15,20,26,124)	SL	Poor data structure	Replacing TreeMap with LinkedHashMap	3.08±0.32
	✓FindBugs-3.0.1	Frame.java:copyFrom	DS	Inefficiently-used ArrayList	Improving ArrayList usage	1.02±0.01
	✓Dacapo 2006 bloat	RegisterAllocator.java:loop (283)	DS	Useless value assignment in JDK	Removing the overpopulated containers	1.35±0.05
	JFreeChart-1.0.19	SegmentedTimeline.java:loop (1026)	SL	Poor linear search	Linear search with a break check	1.64±0.04

✓: newfound performance bugs via JXPᴇʀꜰ.
SS: silent store, DS: dead store, SL: silent load.

```
1  private void calculateDamping(double localpg[][], Statevector
       localug[][]) {
2    Statevector temp2 = new Statevector();
3    if (j > 1 && j < jmax-1) {
4      temp = localug[i][j+2].svect(localug[i][j-1]);
5 ▶    temp2.a = 3.0*(localug[i][j].a-localug[i][j+1].a);
6      ...
7      scrap4.a = tempdouble*(temp.a+temp2.a);
8    }
9    ...
10   if (j > 1 && j < jmax-1) {
11     temp = localug[i][j+1].svect(localug[i][j-2]);
12 ▶   temp2.a = 3.0*(localug[i][j-1].a-localug[i][j].a);
13     ...
14   }
15   ...
16 }
```

Listing 4: Dead stores in Grande-2.0 Euler. Successive memory stores to `temp2.a` without an intervening memory load.

```
; temp2.a = 3.0*(localug[i][j].a-localug[i][j+1].a)
vsubsd %xmm1,%xmm0,%xmm0
vmulsd -0x1a76(%rip),%xmm0,%xmm0
vmovsd %xmm0,0x10(%r8)
…
; scrap4.a = tempdouble*(temp.a+temp2.a)
vaddsd %xmm0,%xmm5,%xmm5
vmulsd %xmm4,%xmm5,%xmm5
vmovsd %xmm5,0x10(%r9)
…
; temp2.a = 3.0*(localug[i][j-1].a-localug[i][j].a)
vsubsd %xmm1,%xmm0,%xmm0
vmulsd -0x2077(%rip),%xmm0,%xmm0
vmovsd %xmm0,0x10(%r8)
```

Figure 8: The assembly code (at&t style) of lines 5, 7 and 12 in Listing 4.

7.3 SableCC-3.7: Silent Loads

SableCC [16] is a lexer and parser framework for compilers and interpreters. JXPᴇʀꜰ profiles the latest stable version of SableCC by using the JDK7 grammar file as the input. JXPᴇʀꜰ identifies that silent loads account for 94% of the total memory loads and more than 80% of silent loads are associated with method put() of the JDK TreeMap class. One of such top inefficiency pairs with calling contexts is shown in Figure 9. The silent loads occur at line 568 in TreeMap.java, whose source code is shown in Listing 5. TreeMap is a Red-Black tree-based map where a put operation requires $O(\log n)$ comparisons to insert an element. put() is frequently invoked to update the TreeMap during program execution. Consequently, previously loaded elements in the TreeMap are often

```
org.sablecc.sablecc.SableCC.main(SableCC.java:136)
  org.sablecc.sablecc.SableCC.processGrammar(SableCC.java:170)
            ...
    mov 0x20(%rbp),%r10d: java.util.TreeMap.put(TreeMap.java:568)
*************************REDUNDANT WITH**********************
org.sablecc.sablecc.SableCC.main(SableCC.java:136)
  org.sablecc.sablecc.SableCC.processGrammar(SableCC.java:170)
            ...
    mov 0x20(%rbp),%r10d: java.util.TreeMap.put(TreeMap.java:568)
```

Figure 9: A silent load pair reported by JXPᴇʀꜰ in SableCC-3.7.

```
561 public V put(K key, V value) {
562   Entry<K,V> t = root;
563   ...
564   do {
565     parent = t;
566     cmp = k.compareTo(t.key);
567     if (cmp < 0)
568 ▶     t = t.left;
569     else if (cmp > 0)
570       t = t.right;
571     ...
572   } while (t != null);
573   ...
574 }
```

Listing 5: Method put() of the JDK TreeMap class. A put operation requires $O(\log n)$ comparisons to insert an element.

re-loaded to compare with new elements being inserted in different instances of put(), which shows up as silent loads.

By consulting the SableCC developers, we choose an alternative data structure. We replace TreeMap with LinkedHashMap because (1) the linked list preserves ordering from one execution to another and (2) the hash table offers $O(1)$ time complexity and obviously reduces the number of loads as well as the fraction of silent loads. We employ this transformation in five classes: LR0ItemSet, LR1ItemSet, LR0Collection, LR1Collection, and Grammar. This optimization reduces the memory loads by 43% and delivers a (3.08±0.32)× speedup to the entire program.

7.4 NPB-3.0 IS: Silent Stores

IS [3] sorts integers using the bucket sort. With the class B input and four threads, JXPᴇʀꜰ pinpoints that 70% of memory stores are silent, of which more than 50% are associated with method pow() at lines 3-6 in Listing 6. We notice method randlc() is invoked in a hot loop (not shown) and the arguments passed to pow() are loop invariant. Across loop iterations, pow() pushes the same parameters on the same stack location, which shows up as silent stores.

To eliminate such redundant operations, we hoist the four calls to pow() outside of randlc() and memoize their return values in

```
1 public double randlc(double a) {
2   double y[],r23,r46,t23,t46, ...;
3 ▶ r23 = Math.pow(0.5,23);
4 ▶ r46 = Math.pow(r23, 2);
5 ▶ t23 = Math.pow(2.0,23);
6 ▶ t46 = Math.pow(t23, 2);
7   ...
8 }
```

Listing 6: Silent stores in NPB-3.0 IS. Method `pow()` repeatedly pushes the same parameters on the same stack across loop iterations.

```
1 union = new HashSet();
2 for (int i = 1; i < copies.size(); i++) {
3   ...
4 ▶ union.addAll(ig.succs(copy[0]));
5 ▶ union.addAll(ig.succs(copy[1]));
6   weight /= union.size();
7   ...
8 }
```

Listing 7: Dead stores in Dacapo 2006 bloat. Useless value assignment in the JDK `HashMap` class leads to dead stores.

private class variables. JXPERF further identifies other code snippets having the same issue and guides the same optimization. These optimizations eliminate 96% of memory stores and yield a (1.89±0.04)× speedup for the entire program.

7.5 Dacapo 2006 Bloat: Dead Stores

Bloat [4] is a toolkit for analyzing and optimizing Java byte code. With the large input, JXPERF reports 78% dead stores. More than 30% of dead stores are attributed to the call site of method `addAll()` at lines 4 and 5 in Listing 7, where the program computes the union of `HashSet ig.succs(copy[0])` and `HashSet ig.succs(copy[1])`, and stores the result in `HashSet` "union". Guided by the culprit calling contexts, we notice the root cause of such dead stores is related to the field `current` of the JDK HashMap class, as shown in Listing 8. Method `addAll()` invokes the method `nextNode()` of the `HashMap` class in a loop (not shown). In each iteration, the field `current` is overwritten with the newly inserted value, but never gets used during the execution, which shows up as dead stores.

With further code investigation, we find that `HashSet` "union" is created for only computing the size of the union of `ig.succs(copy[0])` and `ig.succs(copy[1])`, and elements in "union" are never used. Therefore, we can eliminate the dead stores by avoiding creating "union". We declare a counter variable to record the size of the union of `ig.succs(copy[0])` and `ig.succs(copy[1])`. The counter is initialized to the size of the larger one in `ig.succs(copy[0])` and `ig.succs(copy[1])`. Then we visit each element of the smaller one and check whether that element is already in the larger one. If not, the counter increments by 1. This optimization reduces 32% of memory stores and delivers a (1.35±0.05)× speedup to the entire program.

Yang et al. [52] also identify the same optimization opportunity via the high-level container usage analysis, which is different from JXPERF's binary-level inefficiency analysis.

8 THREATS TO VALIDITY

JXPERF works on multi-threaded programs since PMUs and debug registers are virtualized by the OS for each program thread. JXPERF

```
1 final Node<K,V> nextNode() {
2   Node<K,V>[] t;
3   Node<K,V> e = next;
4   ...
5 ▶ if ((next=(current=e).next)==null&&(t=table)!=null) {
6     do {} while (index<t.length&&(next=t[index++])==null);
7   }
8   return e;
9 }
```

Listing 8: Method `nextNode()` of the JDK `HashMap` class.

detects only intra-thread wasteful memory accesses and ignores inter-thread ones because a watchpoint traps *iff* the same thread accesses the memory address the watchpoint is monitoring. Inter-thread access pattern detection is feasible but unwarranted for the class of inefficiencies pertinent to our current work.

Due to the intrinsic feature of sampling, JXPERF captures statistically significant memory accesses, but may miss some insignificant ones. Usually, optimizing such insignificant memory accesses yields trivial speedups.

Furthermore, not all reported wasteful memory operations need be eliminated. For example, compiler-savvy developers introduce a small number of silent loads or stores to make programs more regular for the purpose of vectorization, and security-savvy developers introduce a small number of dead stores to clear confidential data [53]. Developers' investigation or post-mortem analysis is necessary to make optimization choices. Only high-frequency inefficiencies are interesting; eliminating a long tail of insignificant inefficiencies that do not add up to a significant fraction is impractical and probably ineffective.

9 CONCLUSIONS AND FUTURE WORK

Wasteful memory operations are often the symptoms of inappropriate choice of algorithms or data structures, developers' inattention to performance, and inefficient machine code generation. This paper presents JXPERF, a Java profiler that pinpoints performance inefficiencies arising from wasteful memory operations. JXPERF samples CPU performance counters for addresses accessed by a program and uses hardware debug registers to monitor those addresses in an intelligent way. This hardware-assisted profiling avoids exhaustive byte code instrumentation and delivers a lightweight, effective tool, which does not compromise its ability to detect performance bugs. In fact, JXPERF's ability to operate at the machine code level allows it to detect low-level code generation problems that are not apparent via byte code instrumentation. JXPERF runs on off-the-shelf JVM, OS, and CPU, works on unmodified Java applications, and introduces only 7% runtime overhead and 7% memory overhead. Guided by JXPERF, we are able to optimize several benchmarks and real-world applications, yielding significant speedups.

As one of our future plans, we will extend JXPERF to other popular managed languages, such as Python and JavaScript, which recently employ JITters — PyPy [2] for Python and V8 [20] for JavaScript.

ACKNOWLEDGMENTS

REFERENCES

[1] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. 2013. Toddler: Detecting Performance Problems via Similar Memory-Access Patterns. http://www.cs.fsu.edu/~nistor/toddler.

[2] Armin Rigo, Maciej Fijalkowski, Carl Friedrich Bolz, Antonio Cuni, Benjamin Peterson, Alex Gaynor, Holger Krekel, and Samuele Pedroni. 2018. A fast, compliant alternative implementation of the Python language. https://pypy.org.

[3] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. 1991. The NAS Parallel Benchmarks&Mdash;Summary and Preliminary Results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (Supercomputing '91)*. ACM, New York, NY, USA, 158–165.

[4] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. ACM, New York, NY, USA, 169–190.

[5] Milind Chabbi and John Mellor-Crummey. 2012. DeadSpy: A Tool to Pinpoint Program Inefficiencies. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO '12)*. ACM, New York, NY, USA, 124–134.

[6] Intel Corp. 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.

[7] Intel Corp. 2015. Intel X86 Encoder Decoder Software Library. https://software.intel.com/en-us/articles/xed-x86-encoder-decoder-software-library.

[8] Oracle Corp. 2017. Oracle Developer Studio Performance Analyzer. https://www.oracle.com/technetwork/server-storage/solarisstudio/documentation/o11-151-perf-analyzer-brief-1405338.pdf.

[9] Oracle Corp. 2018. JVMTM Tool Interface. https://docs.oracle.com/en/java/javase/11/docs/specs/jvmti.html.

[10] Oracle Corporation. 2018. All-in-One Java Troubleshooting Tool. https://visualvm.github.io.

[11] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. 2015. Performance Problems You Can Fix: A Dynamic Analysis of Memoization Opportunities. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 607–622.

[12] Monika Dhok and Murali Krishna Ramanathan. 2016. Directed Test Generation to Detect Loop Inefficiencies. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 895–907.

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

[14] Ariel Eizenberg, Shiliang Hu, Gilles Pokam, and Joseph Devietti. 2016. Remix: Online Detection and Repair of Cache Contention for the JVM. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 251–265.

[15] ej-technologies GmbH. 2018. THE AWARD-WINNING ALL-IN-ONE JAVA PROFILER. https://www.ej-technologies.com/products/jprofiler/overview.html.

[16] Etienne Gagnon. 2018. The Sable Research Group's Compiler Compiler. http://sablecc.org. May 2018.

[17] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically Rigorous Java Performance Evaluation. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA '07)*. ACM, New York, NY, USA, 57–76.

[18] David Gilbert. 2017. Welcome To JFree.org. http://www.jfree.org. November 2017.

[19] YourKit GmbH. 2018. The Industry Leader in .NET & Java Profiling. https://www.yourkit.com.

[20] Google Corp. 2018. Google V8 JavaScript Engine. https://v8.dev.

[21] Peter Hofer and Hanspeter Mössenböck. 2014. Fast Java Profiling with Scheduling-aware Stack Fragment Sampling and Asynchronous Analysis. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '14)*. ACM, New York, NY, USA, 145–156.

[22] IBM Corp. 2018. Monitoring and Post Mortem. https://developer.ibm.com/javasdk/tools.

[23] 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.

[24] John Levon *et al.* 2017. OProfile. http://oprofile.sourceforge.net.

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

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

[27] 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.

[28] Monika Dhok and Murali Krishna Ramanathan. 2016. Artifact: Directed Test Generation to Detect Loop Inefficiencies. https://drona.csa.iisc.ac.in/~sss/tools/glider.

[29] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2010. Evaluating the Accuracy of Java Profilers. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. ACM, New York, NY, USA, 187–197.

[30] 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.

[31] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. 2013. Toddler: Detecting Performance Problems via Similar Memory-access Patterns. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 562–571.

[32] Nitsan Wakart. 2016. The Pros and Cons of AsyncGetCallTrace Profilers. http://psy-lob-saw.blogspot.com/2016/06/the-pros-and-cons-of-agct.html.

[33] The University of Edinburgh. 2018. JAVA Grande Benchmark Suite. https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/java-grande-benchmark-suite. October 2018.

[34] 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.

[35] Andrei Pangin. 2018. Async-profiler. https://github.com/jvm-profiling-tools/async-profiler.

[36] Bill Pugh and David Hovemeyer. 2015. Find Bugs in Java Programs. http://findbugs.sourceforge.net. March 2015.

[37] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. 2011. Da Capo Con Scala: Design and Analysis of a Scala Benchmark Suite for the Java Virtual Machine. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '11)*. ACM, New York, NY, USA, 657–676.

[38] Linhai Song and Shan Lu. 2017. Performance Diagnosis for Inefficient Loops. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 370–380.

[39] SPEC Corporation. 2015. SPEC JVM2008 Benchmark Suite. https://www.spec.org/jvm2008. November 2015.

[40] 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.

[41] Pengfei Su, Qingsen Wang, Milind Chabbi, and Xu Liu. 2019. Pinpointing Performance Inefficiencies in Java. *ArXiv e-prints* (2019). arXiv:1906.12066

[42] Pengfei Su, Shasha Wen, Hailong Yang, Milind Chabbi, and Xu Liu. 2019. Redundant Loads: A Software Inefficiency Indicator. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 982–993.

[43] The Sable Research Group. 2018. A framework for analyzing and transforming Java and Android applications. https://sable.github.io/soot.

[44] Jeffrey S. Vitter. 1985. Random Sampling with a Reservoir. *ACM Trans. Math. Softw.* 11, 1 (March 1985), 37–57.

[45] Qingsen Wang, Xu Liu, and Milind Chabbi. 2019. Featherlight Reuse-Distance Measurement. In *Proceedings of The 25th IEEE International Symposium on High-Performance Computer Architecture*. 440–453.

[46] 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.

[47] Shasha Wen, Xu Liu, John Byrne, and Milind Chabbi. 2018. Watching for Software Inefficiencies with Witch. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 332–347.

[48] Guoqing Xu. 2013. Resurrector: A Tunable Object Lifetime Profiling Technique for Optimizing Real-world Programs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '13)*. ACM, New York, NY, USA, 111–130.

[49] Guoqing Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, and Gary Sevitsky. 2009. Go with the Flow: Profiling Copies to Find Runtime Bloat. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 419–430.

[50] 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.

[51] Guoqing Xu and Atanas Rountev. 2010. Detecting Inefficiently-used Containers to Avoid Bloat. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. ACM, New York, NY, USA, 160–173.

[52] Shengqian Yang, Dacong Yan, Guoqing Xu, and Atanas Rountev. 2012. Dynamic Analysis of Inefficiently-used Containers. In *Proceedings of the Ninth International Workshop on Dynamic Analysis (WODA 2012)*. ACM, New York, NY, USA, 30–35.

[53] Zhaomo Yang, Brian Johannesmeyer, Anders Trier Olesen, Sorin Lerner, and Kirill Levchenko. 2017. Dead Store Elimination (Still) Considered Harmful. In *26th USENIX Security Symposium*. USENIX Association, Berkeley, CA, USA, 1025–1040.

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