2020

Understanding Performance Inefficiencies In Native And Managed Languages

Pengfei Su
William & Mary - Arts & Sciences, supengfei2015@gmail.com

Understanding Performance Inefficiencies in Native and Managed Languages

Pengfei Su

Luoyang, Henan, China

Bachelor of Network Engineering, Yunnan University, 2013
Master of Computer Science, University of Chinese Academy of Sciences, 2016

A Dissertation presented to the Graduate Faculty
of The College of William & Mary in Candidacy for the Degree of
Doctor of Philosophy

Department of Computer Science

College of William & Mary
January 2021

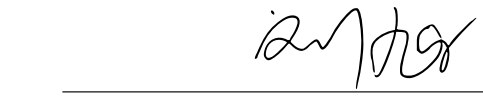
APPROVAL PAGE

This Dissertation is submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

Pengfei Su

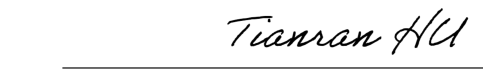
Approved by the Committee, January 2021



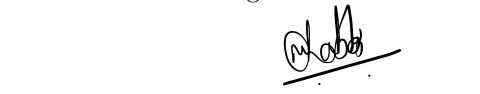
Committee Chair
Xu Liu, Assistant Professor, Computer Science
College of William & Mary

Bin Ren, Assistant Professor, Computer Science
College of William & Mary

Zhenming Liu, Assistant Professor, Computer Science
College of William & Mary



Tianran Hu, Assistant Professor, Computer Science
College of William & Mary



Milind Chabbi, Senior Researcher
Uber Technologies, Inc.

ABSTRACT

Production software packages have become increasingly complex with millions of lines of code, sophisticated control and data flow, and references to a hierarchy of external libraries. This complexity often introduces performance inefficiencies across software stacks, making it practically impossible for users to pinpoint them manually. Performance profiling tools (a.k.a. profilers) abound in the tools community to aid software developers in understanding program behavior. Classical profiling techniques focus on identifying hotspots. The hotspot analysis is indispensable; however, it can hardly diagnose whether a resource is being used in a productive manner that contributes to the overall efficiency of a program. Consequently, a significant burden is on developers to make a judgment call on whether there is scope to optimize a hotspot. Derived metrics, e.g., cache miss ratio, offer slightly better intuition into hotspots but are still not panaceas. Hence, there is a need for profilers that investigate resource wastage instead of usage. To overcome the critical missing pieces in prior work and complement existing profilers, we propose novel fine- and coarse-grained profilers to pinpoint varieties of performance inefficiencies and provide optimization guidance for a wide range of software covering benchmarks, enterprise applications, and large-scale parallel applications running on supercomputers and data centers.

Fine-grained profilers are indispensable to understand performance inefficiencies comprehensively. We propose a whole-program profiler called LoadSpy, which works on binary executables to detect and quantify wasteful memory operations in their context and scope. Our observation, which is justified by myriad case studies, is that wasteful memory operations are often an indicator of various forms of performance inefficiencies, such as suboptimal choices of algorithms or data structures, missed compiler optimizations, and developers' inattention to performance. Guided by LoadSpy, we are able to optimize a large number of well-known benchmarks and real-world applications, yielding significant speedups.

Despite deep performance insights offered by fine-grained profilers, the high overhead keeps them away from widespread adoption, particularly in production. By contrast, coarse-grained profilers introduce low overhead at the cost of poor performance insights. Hence, another research topic is how we benefit from both, that is, the combination of deep insights of fine-grained profilers and low overhead of coarse-grained ones. The first effort to do so is proposing a lightweight profiler called JXPerf. It abandons heavyweight instrumentation by combining hardware performance monitoring units and debug registers available in commodity CPUs to detect wasteful memory operations. Compared with LoadSpy, JXPerf reduces the runtime overhead from 10× to 7% on average. The lightweight nature makes it useful in production. Another effort is proposing a lightweight profiler called FVSampler, the first nonintrusive profiler to study function execution variance.

TABLE OF CONTENTS

ACKNOWLEDGMENTS

First of all, I thank my advisor, Prof. Xu Liu, from the bottom of my heart. It is extremely hard to express my gratitude for his tremendous mentoring. He strikes an excellent balance between mentoring students too much and letting them do everything. He not only spent considerable time forming my knowledge systems and helping me addressing technical challenges, but also gave me freedom over the projects. Without him, my Ph.D. life would be less fruitful and more boring.

Next, I would like to thank my collaborator, Dr. Milind Chabbi, whose carefulness, patience, creativity, and passion for research leave a deep impression on me. He offers me tremendous guidance in many aspects such as research directions, coding, paper writing, and conference presentations. I can reach out to him with ease whenever I need help.

Besides, I convey my gratitude to my defense committee members, Prof. Bin Ren, Prof. Zhenming Liu, and Prof. Tianran Hu, for helping me improving the dissertation and presentation.

I would also like to thank my labmates, classmates, and friends for their company, such as Bolun Li, Jialiang Tan, Hao Xu, Zhen Peng, Qiong Wu, Hongyuan Liu, Lishan Yang, Qi Xia, and Wei Niu.

Finally, my special thanks to my parents who have been giving me endless care, love, and support since day one.

I dedicate this dissertation to my parents Mr. Jinsheng Su and Mrs. Zhifang Wen.

LIST OF TABLES

LIST OF FIGURES

Understanding Performance Inefficiencies in Native and Managed Languages

Chapter 1

Introduction

Performance inefficiencies exist everywhere in computer systems ranging from smartphones to data centers. On the one hand, production software packages have become increasingly complex. They are comprised of a large amount of source code, sophisticated control and data flow, a hierarchy of external libraries, and growing levels of abstraction. This complexity often introduces inefficiencies across software stacks, leading to performance degradation. For example, Akamai studies [5] show that a 100-millisecond latency in website generation drops conversion rates by 7% and 53 % of visitors will leave if a mobile website fails to load within three seconds. On the other hand, the evolution of hardware outpaces the performance improvement of software, increasingly leading to resource wastage and energy dissipation in emerging architectures [17, 93]. Thus, with no careful software design and implementation, developers can easily introduce performance inefficiencies embedded deep in a large code base that are difficult to identify; even worse such inefficiencies further prevent software from enjoying full hardware capacity.

There is a long history of compiler optimizations aimed at statically analyzing and eliminating performance efficiencies by techniques such as common subexpression elimination [32], value numbering [113], constant propagation [140], to name a few. However, they have a myopic view of a program, which limits their analysis to a small scope — individual functions or files. Layers of abstractions, dynamically loaded libraries, multi-

lingual components, aggregate types, aliasing, sophisticated control flow, input-specific path-specific redundancies, and the combinatorial explosion of execution paths make it practically impossible for compilers to obtain a holistic view of a program to eliminate efficiencies comprehensively. Link-time optimization [43] can offer better visibility. However, the analysis is still conservative and may err on the side of being less exhaustive to reduce prohibitive analysis costs. Whole-program link-time optimizations [65, 129] have provided less than 5% average speedup although a lot more headroom exists as we show in this dissertation. Moreover, static compiler analysis is well-known for its inaccuracy in analyzing aliasing and pointers. Thus, despite their best efforts, compilers often fall short of eliminating runtime inefficiencies.

Orthogonal to static analysis, profiling (a form of dynamic analysis) focuses on understanding runtime inefficiencies of a program, which mostly identifies execution hotspots such as code regions suffering from excessive cache misses. The hotspot analysis can hardly diagnose whether a resource is being used in a productive manner that contributes to the overall efficiency of a program although derived metrics, e.g., Cycles-Per-Instruction (CPI), cache miss ratio, offer slightly better intuition into hotspots. Hence, there is a need for profilers that pinpoint resource wastage instead of usage.

Our observation, which is justified by myriad case studies, is that many kinds of performance inefficiencies appearing across the software stacks are usually in the form of wasteful memory operations, e.g., computations whose results may not be used [19, 117], re-computation of already computed values [144], unnecessary data movement [24, 74, 81, 87, 142], excessive synchronization [22, 133], or in the form of function execution variance, e.g., long-tail latency. Unfortunately, existing static and dynamic analysis techniques are either inefficient or incompetent in addressing them.

1.1 Thesis Statement

Profilers that leverage fine-grained code instrumentation or coarse-grained, nonintrusive sampling triggered by hardware performance monitoring units can identify and quantify wasteful memory operations and function execution variance in programs and associate them with program execution contexts and source code to offer rich insights needed for developer actions.

1.2 Contribution Highlights

To overcome the critical pieces left out in prior work and complement existing profiling techniques, we propose several novel profiling techniques aimed at pinpointing wasteful memory operation and function execution variance.

1.2.1 Wasteful Memory Operation Detection

Wasteful memory operations are those that produce/consume data to/from memory that may have been avoided. We observe that a large fraction of wasteful memory operations in the same code region often correlate with some kind of inefficiency. For example, performance inefficiencies such as suboptimal choices of algorithms or data structures, developers' inattention to performance, missed opportunities to optimize common cases, and poor compiler code transformation can show up as substantial wasteful memory operations. We address this problem with two distinct strategies: fine- and coarse-grained profiling.

Fine-grained profiling is a means to monitor execution at microscopic details: it monitors each binary instruction instance, including its operator, operands, and runtime values in registers and memory. A key advantage of microscopic program-wide monitoring is that it can identify inefficiencies irrespective of user-level program abstractions. Prior work [24, 144, 142, 100] has shown that fine-grained profiling techniques can identify many forms of software inefficiencies and offer detailed guidance to tune code. We

propose LoadSpy, a whole-program fine-grained profiler for pinpointing wasteful memory operations, which are often an indicator of resource wastage. The strength of LoadSpy exists in providing valuable guidance to developers for code tuning — calling contexts of the two parties involved in a wasteful operation, narrowed-down scopes to focus on optimization, metrics to understand the relative significance of resource wastage, and a GUI for the source code attribution. Guided by LoadSpy, we optimize several well-known benchmarks, e.g., SPEC CPU benchmarks, and real-world applications, e.g., Apache Avro, yielding significant speedups.

Despite deep insights offered by fine-grained profiling, the high overhead may keep it away from widespread adoption, particularly in production where a strict service-level agreement (SLA) needs to be complied with. By contrast, ***coarse-grained profiling*** introduces low overhead at the cost of poor program insights. Coarse-grained profilers such as Intel VTune [2], perf [78], gprof [52], OProfile [76], and CrayPAT [34] monitor code execution to identify hot code regions, idle CPU cycles, arithmetic intensity, and cache misses, etc. These tools, with low overhead, can recognize the utilization (saturation or underutilization) of hardware resources, but they cannot inform whether resources are being used efficiently. A hotspot need not mean inefficient code, and conversely, the lack of a hotspot need not mean better code. To benefit from both (i.e., the deep insights of fine-grained profiling and the low overhead of coarse-grained profiling), we propose JXPerf, a lightweight profiler for pinpointing wasteful memory operations with no instrumentation to memory accesses. 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. Two key differentiating aspects of JXPerf when compared to a large class of coarse-grained profilers are its ability to (1) filter out and show code regions that are definitely involved in some kind of inefficiency at runtime (coarse-grained profilers cannot differentiate whether or not a code region is involved in any inefficiency) and (2) pinpoint the *two parties* involved in wasteful work — the first instance of a memory access and a subsequent, unnecessary access of the same memory

— which offer actionable insights (coarse-grained profilers are limited to showing only a single party). The result is a lightweight measurement with attribution of inefficiencies to their provenance — machine and source code within full calling contexts (a.k.a. call paths). JXPerf introduces only 7% runtime and memory overhead, making it useful in production. Guided by JXPerf, we are able to optimize an array of benchmarks and real-world applications (that are the subjects of study and optimization for decades) by improving compiler code transformation and choosing superior algorithms or data structures.

1.2.2 Function Execution Variance Detection

Execution variance among different invocation instances of the same function is a common symptom of performance losses. On the one hand, Instrumentation-based tools avail themselves to function instance level metrics because the instrumentation can be placed at the entry and exit of a function; in fact, even finer-grained placement such as statements or instructions is also possible. They can count resources consumed by any invocation instance of the same code region albeit the overhead can be nontrivial. Although most instrumentation-based tools provide the selective instrumentation option to reduce overhead, selective ones have systematic blind spots and yet incur relatively high overhead. Even a 2× overhead via the selective instrumentation is problematic when profiles are needed from large-scale execution in a production setting. On the other hand, sampling-based tools insert no instrumentation and have low overhead. However, they provide statistics, e.g., total execution time, for a function without distinguishing different invocation instances of that function because they cannot synchronize samples with function boundaries (i.e., entry and exit).

Measuring the variance across function instances requires starting and stopping measurements at function entry and exit. This act of starting and stopping measurements at *every* function entry and exit is equivalent to placing instrumentation, which defeats the purpose of lightweight sampling. Hence, there is a dilemma, how can we enjoy the

low overhead of sampling and yet collect meaningful measurements at function instance boundaries so that we can compare execution variance across different invocations of the same function? We would like to emphasize that we are interested in the variance of two or more execution instances of the same function in a single execution[1]. However, we would like to collect such variance for a large number of functions that a program executes and we would like to do so in a single profiling session.

We propose FVSampler, a lightweight, sampling-based variance profiler. The strength of FVSampler exists in abandoning heavyweight code instrumentation and requiring no prior knowledge of a program for function-level variance monitoring. It employs hardware performance monitoring units in conjunction with hardware debug registers to sample and monitor whole function instances (call till return) and collect performance metrics, e.g., CPU cycles, retired instructions, cache misses, network packets, in each sampled function instance. Also, FVSampler is capable of pinpointing both intra-thread and inter-thread variance, which helps isolate performance problems in complex code bases. FVSampler typically incurs only 6% runtime overhead and negligible memory overhead, making it suitable for high-performance computing software. A thorough evaluation of several parallel applications shows that quantifying variance on per sampled function invocation opens up a new avenue for understanding performance losses; mitigating the causes of variance enhances performance. Guided by FVSampler, we are able to tune algorithms and data structures to obtain significant speedups.

1.3 Organization

The remainder of this dissertation is organized as follows. Chapter 2 offers the background knowledge necessary to understand the technical details. Chapters 3, 4, and 5 depict the methodology and implementation of LoadSpy, JXPerf, and FVSampler, as well as evaluate their accuracy, overhead, and effectiveness by applying them on a number of

[1]Comparing total samples taken by two different functions or functions from two different threads or processes is straightforward and available in almost all profilers, sampling or otherwise.

benchmarks and real-world applications. Finally, Chapter 6 presents our conclusions and overviews several possible future directions.

Chapter 2

Background

This chapter offers some background knowledge to facilitate the understanding of technical details in the subsequent chapters.

2.1 Intel Pin

Intel Pin [84] is a dynamic binary instrumentation framework. It provides rich APIs for users to build client tools (a.k.a. Pintools). A Pintool consists of two components: instrumentation code and places where the code is inserted. The former can be arbitrary C/C++ code and the latter can be arbitrary places in a binary executable. For instance, we can insert the instrumentation code at the place where a memory load instruction occurs to obtain its loaded value.

2.2 Hardware Performance Monitoring Unit

Modern CPUs expose programmable performance monitoring units (PMUs) that count various hardware events such as memory accesses, retired instructions, CPU cycles, and cache misses, to name a few. PMUs can be configured in two modes: counting and sampling. In counting mode, users can read the number of occurrences of hardware events from PMUs at any point during program execution. 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 (a.k.a. sample) and attribute the metrics collected along with the sample to the execution context. PMUs are per CPU core and virtualized by the OS for each thread.

Intel offers Precise Event-Based Sampling (PEBS) [28] in SandyBridge 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 store. This facility is often referred to as address sampling. Also, PEBS can capture the precise instruction pointer (IP) for the instruction resulting in counter overflow. AMD Instruction-Based Sampling (IBS) [38] and PowerPC Marked Events (MRK) [128] offer similar capabilities.

2.3 Hardware Debug Register

Hardware debug registers [64, 88] 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, and 8 bytes), and accessing types (trap-on-store (`W_TRAP`) and trap-on-load-or-store (`RW_TRAP`)). The number of hardware debug registers is limited; an x86 processor has four debug registers and a PowerPC processor has one debug register.

2.4 Linux perf_event

Linux offers a standard interface to program PMUs and debug registers via the `perf_event_open` system call [77] and the associated `ioctl` system call. A PMU sample is an asynchronous CPU interrupt caused when an event counter overflows, while a watchpoint exception is a synchronous CPU trap caused when an instruction accesses a monitored address. Both PMU samples and watchpoint exceptions are handled via Linux signals. The user code can mmap a circular buffer to which the kernel keeps appending the

PMU data on each sample and extract the signal context on each watchpoint exception.

2.5 Java Virtual Machine Tool Interface

Java virtual machine tool interface (JVMTI) [31] is a native programming interface of the JVM. A JVMTI client can develop a debugger/profiler (a.k.a. JVMTI agent) in C/C++ 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 JIT-compiled (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.

Chapter 3

Redundant Loads: A Software Inefficiency Indicator

3.1 Introduction

Execution profiling aims to understand the runtime behavior of a program. Coarse-grained profilers concentrate on execution hotpots and usually cannot distinguish efficient vs. inefficient code. For example, they cannot identify that repeated memory loads of the same value or result-equivalent computations waste both memory bandwidth and processor functional units. Fine-grained profilers can pinpoint inefficiencies by monitoring a subset of individual operations such as operations with symbolic equivalence [144], dead memory stores [24], and operations writing the same value to target registers or memory locations [142]. They have, however, overlooked an important category of wasteful memory operations — *temporal load redundancy* — loading the same value from the same memory location. The code on the left of Listing 3.1 shows redundant operations that are invisible in existing fine-grained profilers. In this code, assume all the scalars are in registers and vectors are in memory. Since there are no "dead store" operations (a store followed by another store to the same location without an intervening load), DeadSpy [24] does not identify any inefficiency. Since the values written in t and *delta* always change,

RedSpy [142] does not report any "silent store" operations [74]. Finally, since there is no symbolic equivalent computation, RVN [144] does not report any inefficiency. Furthermore, because the optimization involves the mathematically equivalent transformation, as shown on the right of Listing 3.1, it is difficult to optimize with other compiler techniques such as polyhedral optimization [110].

```
1 while (t < threshold) {
2   t = 0;
3   for(i = 0; i < N; i++)
4 ▶   t += A[i] + B[i] * delta;
5   delta -= 0.1 * t;
6 }
```

```
1 for (i = 0; i < N; i++) {
2   a += A[i];
3   b += B[i];
4 }
5 while (t < threshold) {
6   t = a + b * delta;
7   delta -= 0.1 * t;
8 }
```

Listing 3.1: An example code (on the left) with temporal inefficiencies that cannot be identified by existing fine-grained profilers. Because arrays `A` and `B` are immutable in the loop nest, computing on these loop invariants introduces many redundancies. One can hoist the redundant computation out of the loop (on the right) for optimization.

The code on the left of Listing 3.2 shows another kind of load redundancy, which loads the same value from *nearby* memory locations. Even though each element of array `A` is only loaded once, adjacent elements with the same value result in loading the same value and the subsequent redundant computation. We refer to this type of redundancy as *spatial load redundancy*.

```
1 int A[N] = {1, 1, 1, 15};
2 for(i = 0; i < N; i++) {
3 ▶ t += func(A[i]);
4 }
```

```
1 int A[N] = {1, 1, 1, 15};
2 a = func(A[0]);
3 for(i = 0; i < N; i++) {
4   if (A[i] != A[i-1])
5     a = func(A[i]);
6   t += a;
7 }
```

Listing 3.2: An example code (on the left) with spatial inefficiencies that cannot be identified by existing fine-grained profilers. The load redundancy happens at line 3 where the program loads the same value from nearby memory locations since some adjacent elements of array `A` have the same value. Such redundancy further results in redundant computation involved in the function `func()`. Because `func()` always returns the same value for the same input. One can compare if the adjacent elements in array `A` are equivalent to eliminate redundant computation (on the right). If they are the same, one can reuse the return value of `func()`, which is generated in the previous iteration.

Listing 3.1 and 3.2 show a tip of the iceberg of the inefficiencies we target in this work. From our observation, a variety of inefficiencies exhibit *substantial* redundant loads;

conversely, the presence of a large fraction of redundant loads in an execution is a symptom of some kind of inefficiency *in the code regions* that exhibit such redundancy. Furthermore, the subsequent operations based on redundant loads are potentially redundant.

3.1.1 Contribution Summary

We have designed and implemented a developer tool — LoadSpy — aimed at pinpointing and quantifying load redundancy in native languages, such as C, C++, and Fortran. LoadSpy highlights precise source code in its full calling contexts and the two parties involved in a redundant load. Additionally, LoadSpy narrows down the investigation scope to help developers focus on the provenance of inefficiencies. A thorough evaluation of a suite of benchmarks and real-world applications shows that looking for redundant loads in a program offers an easy avenue for performance enhancement.

In this work, we make the following contributions:

- Show that redundant loads are a common indicator of various forms of software inefficiencies. This finding serves as the foundation of LoadSpy.
- Describe the design of LoadSpy — a whole-program fine-grained profiler for pinpointing redundant loads.
- Develop strategies for analyzing a large volume of profiling data by attributing redundancy to runtime contexts, objects, and scopes.
- Enable rich visualization for profiling data coming from different threads/processes with a user-friendly GUI, which improves the usability for non-experts.
- Apply LoadSpy to pinpoint inefficiencies in well-known benchmarks and real-world applications and eliminate LoadSpy-found inefficiencies by avoiding redundant loads, which yield nontrivial speedups.

3.2 Related Work

There exist many compiler techniques and static analysis techniques [27, 32, 85, 60] to identify redundant computation. However, these static approaches suffer from limitations related to the precision of alias information, optimization scope, and insensitivity to inputs and execution contexts. To address these issues, recent approaches convert source code to specific notations for redundancy detection and removal [37], or target specific algorithms for optimization [36]. However, these approaches require substantial prior knowledge to identify whether a program suffers from redundancies that are worthy of optimization. In contrast, LoadSpy monitors execution, avoids inaccuracies associated with compile-time analysis, and needs no prior knowledge of the monitored program.

There exist many hardware-assisted approaches [80, 79, 74, 75, 91, 90, 153, 19] that optimize redundant operations. However, these approaches require hardware extension, which is unavailable in commodity processors. Instead, LoadSpy is a pure software approach and does not need any hardware changes. The remaining section reviews only other profiling techniques.

3.2.1 Value Profiling

LoadSpy is a value-aware profiler; value profiling techniques are closely related to our work. Calder *et al.* [20, 21, 42] propose probably the first value profiler on DEC Alpha processors. They instrument the program code and record top N values to pinpoint invariant or semi-invariant variables stored in registers or memory. A variant of this value profiler is proposed in later research [139]. Burrows *et al.* [18] use PMUs to sample values in Digital Continuous Profiling Infrastructure [7]. Wen *et al.* [143] combine PMUs and debug registers available in x86 CPUs to identify wasteful memory operations. These approaches do not explore whole-program load redundancy in depth. Moreover, none of them detect spatial redundancy.

Some code specialization work depends on value profiling. However, these approaches

limit themselves to only analyzing registers [95], static instructions [105], memory store operations [142], or functions [26, 67, 56]. They omit many optimization opportunities and require significant manual efforts to reason about the root causes of inefficiencies.

Unlike existing value profilers, LoadSpy has four distinct features. First, LoadSpy is the first value profiler that tracks the *history of loaded values* from individual *memory locations*, rather than the values produced by *individual instructions*. Second, LoadSpy identifies both *temporal and spatial* redundancies in load operations. Third, LoadSpy provides novel redundancy scope and metrics to guide optimization in both contexts and semantics. Fourth, LoadSpy not only identifies redundancy arising due to the same value but also identifies redundancy due to approximately equal values, which offers opportunities for *approximate computing*.

3.2.2 Value-agnostic Profiling

RVN [144] assigns symbolic values to dynamic instructions and identifies redundancy on the fly. DeadSpy [24] tracks every memory operation to pinpoint a store operation that is not loaded before a subsequent store to the same location. Travioli [108] detects redundant data structure traversals. These approaches miss out on certain opportunities that LoadSpy can detect by explicitly inspecting values generated at runtime.

Toddler [100] has to manually add loop events to instrument loops in a C code base and only identifies repetitive memory loads across loop iterations. The follow-up work LDoctor [121] reduces Toddler's overhead using a combination of ad-hoc sampling and static analysis techniques. However, LDoctor only instruments a small number of suspicious loops at compile time, which can miss redundant loads occurring in unmonitored loops. In contrast, LoadSpy works on fully optimized binaries, is independent of any compiler, and performs whole-program profiling instead of limiting itself to loop profiling.

3.3 Motivation

While there are several ways to identify inefficiencies, LoadSpy focuses on memory load operations. If two consecutive load operations performed on the same memory location load the same value, the second load operation can be deemed redundant. Thus, the second load could potentially be elided. Our study aims to pinpoint redundant loads and attribute them to the code regions that cause them. *A single instance of a redundant load is uninteresting; highly frequent redundant loads occurring in the same code location demand attention.*

It is easy to imagine how redundant loads happen: repeatedly accessing immutable data structures or algorithms. It is equally easy to imagine how inefficient code sequences show up as redundant loads: missed function inlining appears as repeatedly loading the same values in a callee, imperfect alias information shows up as loading the same value from the same location via two different pointers, redundant computations show up as the same computations being performed by loading unchanged values, algorithmic defects, e.g., frequent linear searches or hash collisions, also appear as repeatedly loading unchanged values from same locations.

Definition 1 (Temporal Load Redundancy)**.** *A memory load operation L_2, loading a value V_2 from location M, is redundant iff the previous load operation L_1, performed on M, loaded a value V_1 and $V_1 = V_2$. If $V_1 \approx V_2$, we call it approximate temporal load redundancy.*

Definition 2 (Spatial Load Redundancy)**.** *A memory load operation L_2, loading a value V_2 from location M_2, is redundant iff the previous load operation L_1, performed on location M_1, loaded a value V_1 and $V_1 = V_2$, and M_1 and M_2 belong to the address span of the same data object. If $V_1 \approx V_2$, we call it approximate spatial load redundancy.*

Definition 3 (Redundancy Fraction)**.** *We define the* redundancy fraction $\mathcal{F}$ *in an execution as the ratio of bytes redundantly loaded to the total bytes loaded in the entire execution.*

We emphasize the redundancy is defined for instruction instances, *not* static instructions. Deleting an instruction involved in an instance of a redundant load can be unsafe.

Observation 1. *A large redundancy fraction ($\mathcal{F}$) in the execution profile of a program is a symptom of some kind of software inefficiency.*

Redundant loads are neither a necessary condition nor a sufficient condition to capture all kinds of software inefficiencies. However, we show, with many illustrative case studies, that *a large fraction of redundant loads in the same code region* is often a symptom of a serious inefficiency. We notice frequent redundant loads across the board in many programs irrespective of optimization levels, raising a warning alarm of potential inefficiency. Although not all redundant loads demand optimization, in our experience, investigating the top few contributors in a profile offers a high potential to tune and optimize code. Looking for load redundancy opens up potentially an easy avenue for code optimization — manual or automatic.

We measure the redundancy fraction in a number of benchmarks — SPEC CPU2006 [124], PARSEC-2.1 [14], Rodinia-3.1 [25], and NERSC-8 [97]. We compile these benchmarks with `gcc-4.8.5 -O3`, link-time optimization (LTO), and profile-guided optimization (PGO), which is one of the highest optimization levels. In practice, most packages do not use this level of optimization. We classify the causes of redundant loads according to their provenance: input-sensitive redundant loads, suboptimal algorithms or data structures, and missed compiler optimizations. Different kinds of inefficiencies require different optimization strategies.

3.3.1 Input-sensitive Redundant Loads

In this subsection, we classify the inefficiency due to inputs. Rodinia-3.1 backprop [25], a supervised machine learning algorithm, trains the weights of connections in a neural network. The redundancy fraction of this program is 64%. It is common knowledge that as the training progresses, many weights stabilize and do not change. Hence, their

gradients become and remain zero. Listing 3.3 shows the inefficiency at line 3, where the majority of elements in arrays `delta` and `oldw` are both zeros. Computations at lines 3-5 can be bypassed when `delta[j]` and `oldw[k][j]` are zeros. Repeatedly loading the zero value from adjacent locations within arrays `delta` and `oldw` shows up as spatial load redundancy. It is easy to eliminate the input-sensitive redundant loads by predicating the subsequent computation on the values of `delta[j]` and `oldw[k][j]` being non-zero.

```
1 for (j = 1; j <= ndelta; j++) {
2   for (k = 0; k <= nly; k++) {
3 ▶   new_dw = ((ETA * delta[j] * ly[k]) + (MOMENTUM * oldw[k][j]));
4       w[k][j] += new_dw;
5       oldw[k][j] = new_dw;
6   }
7 }
```

Listing 3.3: Spatial load redundancy in Rodinia-3.1 backprop. Arrays `delta` and `oldw` are repeatedly loaded from memory whereas most array elements are zero.

3.3.2 Redundant Loads due to Suboptimal Algorithms or Data Structures

Inefficiencies of this category require semantics to identify and optimize. These inefficiencies also incur a significant number of redundant loads. We illustrate some algorithms and data structures that introduce inefficiencies in a few well-known benchmarks.

Linear search Rodinia-3.1 particlefilter [25] is used to estimate the location of a target object in signal processing and neuroscience. The redundancy fraction of this program is 99%. Listing 3.4 shows the inefficiency in function `findIndex()`, which performs a linear search (line 3) over a sorted array `CDF` to determine the location of a given particle. This linear search is invoked multiple times in a loop to become the bottleneck of the program. The symptom of this inefficiency is many redundant loads, which is caused by the repeated loads of immutable array `CDF` elements across different invocation instances of `findIndex()`. To fix the problem, one can replace the linear search with a binary search, which reduces the volume of redundant loads.

```
1 int findIndex(double *CDF, int lengthCDF, double value) {
2   for(x = 0; x < lengthCDF; x++) {
3 ▶   if (CDF[x] >= value) {
4       index = x; break;
5     }
6   }
7   ...
8   return index;
9 }
10 ...
11 for(j = 0; j < Nparticles; j++)
12   i = findIndex(CDF, Nparticles, u[j]);
```

Listing 3.4: Temporal load redundancy in Rodinia-3.1 particlefilter. A linear search loads the same values from the same memory locations.

Hash table Parsec-2.1 dedup [14] compresses data via deduplication. The redundancy fraction of this program is 75% and the inefficiency is shown in Listing 3.5. Function `hashtable_search()` is invoked in a hot loop (not shown) and in each invocation, it searches for an item in a linked list associated with a hash table entry. We notice that due to excessive hash collisions, only ∼2% of hash buckets are occupied and each occupied bucket has a long linked list. As a result, `hashtable_search()` frequently traverses the same linked list, which appears as loading the same values from the same locations (line 8). One can improve the hash function to make hash keys uniformly distributed across buckets, which will reduce the redundancy and hence the inefficiency.

```
1 struct hash_entry *hashtable_search(struct hashtable *h, void *k) {
2   struct hash_entry *e;
3   unsigned int hashvalue, index;
4   hashvalue = hash(h, k);
5   index = indexFor(h->tablelength, hashvalue);
6   e = h->table[index];
7   while (NULL != e) {
8 ▶   if ((hashvalue == e->h) && (h->eqfn(k, e->k))) return e;
9     e = e->next;
10  }
11  ...
12 }
```

Listing 3.5: Temporal load redundancy in Parsec-2.1 dedup. Excessive hash collisions in linear hashing result in long linked lists.

3.3.3 Redundant Loads due to Missed Compiler Optimizations

Inefficiencies of this category occur in small scope — loop nest or procedure call. One needs to either curate the code or manually apply transformations to eliminate these

inefficiencies. The following three examples illustrate our findings.

Missed scalar replacement Rodinia-3.1 hotspot 3D [25] is a thermal simulation program that estimates processor temperature. The redundancy fraction of this program is 95%. Listing 3.6 shows a loop nest that performs a stencil computation. At line 8, `tOut_t[c]` is updated with the values in nearby `tIn_t[]`. Typically, `w = c - 1` and `e = c + 1`. As a result, the value of `tIn_t[e]` in the current iteration equals the value of `tIn_t[c]` in the next iteration and further equals the value of `tIn_t[w]` in the iteration after next. However, the compiler does not perform register promotion of `tIn_[e]`. Hence, many *redundant loads* occur in this loop nest. To fix this inefficiency, we employ scalar replacement to eliminate inter-iteration redundant loads from memory. Specifically, we store the value of `tIn_t[e]` in a local variable in the current iteration to be reused by `tIn_t[c]` in the next iteration and by `tIn_t[w]` in the iteration after next.

```
1 for(y = 0; y < ny; y++) {
2   for(x = 0; x < nx; x++) {
3     int c, w, e, n, s, b, t;
4     c = x + y * nx + z * nx * ny;
5     w = (x == 0) ? c : c - 1;
6     e = (x == nx - 1) ? c : c + 1;
7     ...
8 ▶   tOut_t[c] = cc * tIn_t[c] + cw * tIn_t[w] + ce * tIn_t[e] + ...
9   }
10 }
```

Listing 3.6: Temporal load redundancy in Rodinia-3.1 hotspot3D. Array `tIn_t` is repeatedly loaded from memory while the values remain unchanged.

Missed constant propagation NERSC-8 msgrate [97] measures the message passing rate via the MPI interface. The redundancy fraction of this program is 97%. Listing 3.7 shows a function `cache_invalidate()`, which sets all the elements in array `cache_buf` to 1. This code adopts a suboptimal forward propagation that loads the value of `cache_buf[i-1]` and assigns it to `cache_buf[i]`. Although there is no redundant load in a single invocation, `cache_invalidate()` is invoked in a loop (not shown), resulting in excessive, redundant loads from array `cache_buf`. The compiler does not replace the assignment with a constant, possibly due to its inability to prove the safety of assigning

to a global array in the presence of concurrent threads of execution.

```
1 int *cache_buf;
2 ...
3 static void cache_invalidate(void) {
4   int i;
5   cache_buf[0] = 1;
6   for (i = 1; i < cache_size; ++i)
7 ▶   cache_buf[i] = cache_buf[i-1];
8 }
```

Listing 3.7: Temporal load redundancy in NERSC-8 msgrate. The program repeatedly loads a constant "1" from array `cache_buf`.

```
1 for (pos = 0; pos < max_pos; pos++) {
2   ...
3   if (abs_y >= 0 && abs_y <= max_height && ...) PelYline_11 = FastLine16Y_11;
4   else PelYline_11 = UMVLine16Y_11;
5   for (blky = 0; blky < 4; blky++) {
6     for (y = 0; y < 4; y++) {
7 ▶     refptr = PelYline_11(ref_pic, abs_y++, abs_x, img_height, img_width);
8       ...
9     }
10    ...
11  }
12 }
```

Listing 3.8: Temporal load redundancy in SPEC CPU2006 464.h264ref due to missed function inlining.

Missed inline substitution SPEC CPU2006 464.h264ref [124] is a reference implementation of H.264, a standard of video compression. The redundancy fraction of this program is 84%. The compiler fails to inline the frequently called function `PelYline_11()` at line 7 shown in Listing 3.8. Because it is invoked via a function pointer and the callee routines are not present in the same file. The parameters of `PelYline_11()` — `abs_x`, `img_height`, and `img_width` — are unmodified across multiple successive invocations. In each function call, the caller pushes (stores) the same parameters on the stack and then in each function return, the callee pops (loads) the same parameters from the stack, which shows up as redundant loads. To fix it, one needs to manually inline the callee into its caller [142].

Discussion We have explored other compiler flags that enable advanced optimization such as polyhedral optimization [45] in `GCC`. Unfortunately, none of them were successful in optimizing any of the aforementioned scenarios. Furthermore, we observed that using

LTO, PGO, together with the polyhedral optimization made compilation time extremely high for some cases. For example, it took over two hours to compile hotspot 3D, a 30,000× slowdown compared to simply using `-O3`. As a result, our later evaluation section does not use LTO and polyhedral optimization, but only uses `-O3` with PGO.

3.4 LoadSpy Implementation

LoadSpy employs Intel Pin to instrument every memory load operation. The instrumentation obtains the effective address M to be accessed in an instruction and the access length δ, and offers the pair to a runtime analysis routine. In the rest of this section, we discuss how LoadSpy identifies *temporal* and *spatial* load redundancies, respectively.

3.4.1 Detecting Temporal Load Redundancy

Detecting temporal load redundancy requires two pieces of information: the current value v_{new} at the target location and the last-time loaded value v_{old} from the same location. The runtime analysis routine, run just before the execution of the original program's load instruction, fetches the current value v_{new} at the memory range $[M : M + \delta)$. LoadSpy employs a shadow memory S for maintaining the last-time loaded value at the same location. $S[M]$ maintains the value last loaded by the program at location M. LoadSpy utilizes the page-table-based scheme [24] to efficiently manage its shadow memory. At runtime, the analysis routine fetches v_{old} from $S[M : M + \delta)$ and v_{new} from $[M : M + \delta)$. LoadSpy records an instance of a *redundant* load if $v_{old} = v_{new}$. All bytes must match to qualify a load as redundant. Intuitively, sub-read-size redundancy is unactionable by programmers. Note, however, that v_{old} might have been generated by multiple shorter reads, a single longer read, or more commonly a single read of the same size. If not redundant, LoadSpy updates the shadow memory with the newly loaded value. Also, LoadSpy records an instance of a *non-redundant* load if $v_{old} \neq v_{new}$.

LoadSpy provisions for approximate computing by allowing the new value generated

in a floating-point (FP) operation to *approximately* match the previously present value. If the two values are within a threshold of difference, LoadSpy considers them approximately equal and records an instance of a redundant load. The threshold is tunable; we use 1% in our experiments. Accordingly, LoadSpy decomposes the load redundancy into *precise* and *approximate*.

LoadSpy attributes each instance of redundant loads (and non-redundant loads) to two parties $\langle C_{old}, C_{new} \rangle$, where C_{old} is the calling context of the previous load operation on M and C_{new} is the calling context of the current load operation on M.

The following equations compute the fraction of temporal load redundancy in an execution:

$$\begin{aligned} \mathcal{F}_{prog}^{precise} &= \frac{\sum_i \sum_j \text{Redundant non-FP bytes loaded in } \langle C_i, C_j \rangle}{\sum_i \sum_j \text{non-FP bytes loaded in } \langle C_i, C_j \rangle} \\ \mathcal{F}_{prog}^{approx} &= \frac{\sum_i \sum_j \text{Redundant FP bytes loaded in } \langle C_i, C_j \rangle}{\sum_i \sum_j \text{FP bytes loaded in } \langle C_i, C_j \rangle} \end{aligned} \tag{3.1}$$

Load redundancy between a pair of calling contexts is given by the following equations:

$$\begin{aligned} \mathcal{F}_{\langle C_{old}, C_{new} \rangle}^{precise} &= \frac{\text{Redundant non-FP bytes loaded in } \langle C_{old}, C_{new} \rangle}{\sum_i \sum_j \text{non-FP bytes loaded in } \langle C_i, C_j \rangle} \\ \mathcal{F}_{\langle C_{old}, C_{new} \rangle}^{approx} &= \frac{\text{Redundant FP bytes loaded in } \langle C_{old}, C_{new} \rangle}{\sum_i \sum_j \text{FP bytes loaded in } \langle C_i, C_j \rangle} \end{aligned} \tag{3.2}$$

The metrics help identify code regions (pairs of calling contexts) where the highest amount of redundancy is observed.

Obtaining the Calling Context of an Instruction: Attributing runtime statistics to a flat profile (just an instruction pointer) does not offer full insights to developers. For example, attributing redundant loads to a common library function, e.g., `strcmp()`, offers little insight since `strcmp()` 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 to the full calling context: `main():line`→`A():line`→`...`→`strcmp():line`. LoadSpy requires obtaining the call-

ing context on each load operation. LoadSpy employs CCTLib [23], which efficiently maintains calling contexts as a calling context tree (CCT) [6] including complex control flows through `longjump`, tail calls, and exceptions. The calling context, which is provided as a unique 32-bit integer, is recorded (in addition to the last-time loaded value) in the shadow memory.

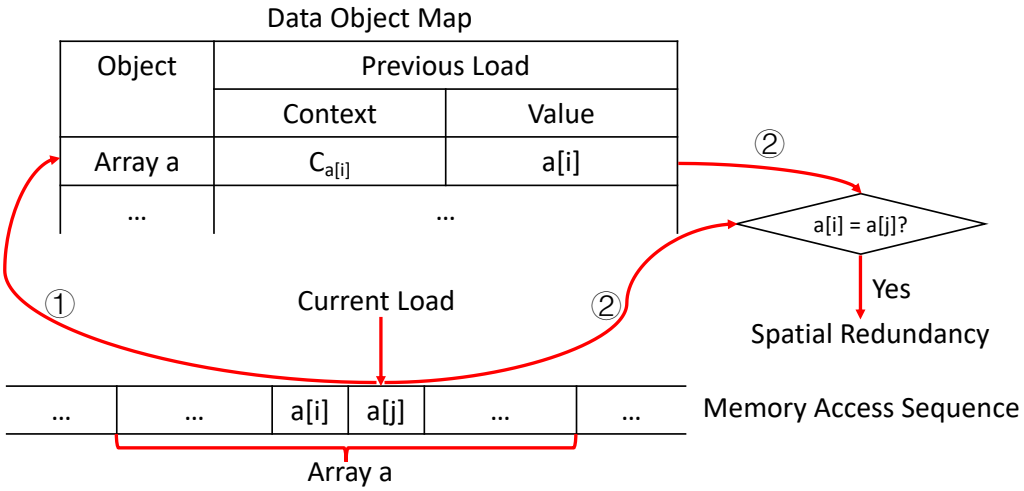


Figure 3.1: Detecting spatial load redundancy. ① LoadSpy monitors a load operation and associates its effective address with the data object. In the data object map, each data object associates itself with the value and context of the previous load belonging to this data object. ② LoadSpy compares the previous and current loaded values; if they are (approximately) the same, an instance of (approximate) spatial load redundancy is reported. ③ The value and context associated with the data object are updated with the ones from the current load.

3.4.2 Detecting Spatial Load Redundancy

For arrays and aggregate objects, LoadSpy checks whether two consecutive loads from different elements of the same object load (approximately) the same value. For example, if two consecutive loads from an array `a`, say `a[i]` and `a[j]`, load the same value, LoadSpy flags it as an instance of *spatial* load redundancy and attributes it to the same data object, as shown in Figure 3.1.

To facilitate spatial load redundancy detection, LoadSpy maintains a mapping from address ranges to active data objects in the shadow memory. Associated with each data

object $\mathcal{O}$ is two additional pieces of information: a singleton value v_{old} loaded as a result of the previous load operation performed on $\mathcal{O}$ and the calling context C_{old} associated with the previous load operation performed on $\mathcal{O}$. Upon each memory load, LoadSpy uses its effective address to look up the data object it belongs to in the map. If the value of the current load matches the one recorded with the previous load on the same object, LoadSpy records an instance of spatial load redundancy. The redundancy is hierarchically attributed first to the data object involved and then to the two calling contexts involved in the redundancy.

LoadSpy provides the similar whole-program and per-redundancy-pair metrics for spatial redundancy. Moreover, LoadSpy computes per-data-object metrics with the following equations where $\mathcal{O}$ is a data object.

$$\begin{aligned} \mathcal{F}_{\mathcal{O}}^{precise} &= \frac{\text{Redundant non-FP bytes in object } \mathcal{O}}{\sum_i \text{non-FP bytes in object i}} \\ \mathcal{F}_{\mathcal{O}}^{approx} &= \frac{\text{Redundant FP bytes in object } \mathcal{O}}{\sum_i \text{FP bytes in object i}} \end{aligned} \tag{3.3}$$

Obtaining Data-object Addresses at Runtime: LoadSpy monitors static and dynamic data objects but ignores stack objects from spatial redundancy detection. Data allocated in the `.bss` section in a load module are static objects. Each static object has a named entry in the symbol table that identifies the memory range for the object with an offset from the beginning of the load module. The lifetime of static objects begins when the enclosing load module (executable or dynamic library) is loaded into memory and ends when the load module is unloaded. LoadSpy intercepts the loading and unloading of load modules to monitor the lifetime of static data objects and establishes a mapping from an object's address range to the corresponding data object. Dynamic objects are allocated via one of malloc family of functions (`malloc`, `calloc`, `realloc`) or `mmap` [82], and reclaimed via `free` or `munmap`. LoadSpy intercepts these functions to establish a mapping from an object's address range to the corresponding data object. Querying an address at runtime obtains a handle to the corresponding static or dynamic object. The handle is a unique

identifier representing the object name for a static object or the allocation calling context for a dynamic object.

3.4.3 Identifying Redundancy Scope

When the redundancy happens in the same calling context, that is $C_{old} = C_{new}$, there is guaranteed to be a loop [1] around the redundancy location. However, in code with nested loops, it is unclear whether the redundancy occurred between iterations of an inner loop or between iterations of an outer loop or some other loop in-between. Hence, it becomes necessary to point out the syntactic scope enclosing a redundancy pair.

We illustrate the need for scope using a real-world application `MASNUM-2.2` [112] shown on the left of Listing 3.9. LoadSpy identifies 91% of memory loads are redundant and the top contributor is at line 6. It is tempting to infer that `x(iii + 1)` loaded in one iteration of the inner `do` loop (line 5) is loaded again as `x(iii)` in the next iteration. An obvious optimization is to perform scalar replacement to retain `x(iii + 1)` across iterations of the inner `do` loop (on the right of Listing 3.9). However, this optimization does not eliminate many redundant loads. Actually, the outer `do` loop (line 1) frequently searches for different items `xx`, and the inner `do` loop performs a linear search. As a result, the inner loop repeatedly loads the same set of elements across two trips of the outer loop. Thus, the load redundancy exists not only between iterations of the inner loop but also between iterations of the outer loop. The load redundancy at the outer loop highlights an algorithmic-level inefficiency — repeated linear searches. With this knowledge, we replace the linear search with a binary search to eliminate load redundancy. More details are shown in Section 3.7.2.

To assist developers to focus on the *scope* where load redundancy occurs, we have incorporated a *redundancy scope* feature in LoadSpy. We denote redundancy scope with the symbol $\mathcal{S}$. In Listing 3.9, the redundancy scope is the *outer* `do` loop. Below we detail how redundancy scope is computed.

[1] We consider natural loops [136] only.

```
1 do 500 k = 1, kl
2   ...
3   xx = x0 - deltt * (cgx + ux(ia, ic)) /
        rslat(ic) * 180 / pi
4   ...
5   do iii = ixs, ixl-1
6   ▶ if(xx >= x(iii) .and. xx <= x(iii +
        1)) then
7       ixx = iii; exit
8     endif
9   enddo
10  ...
11 500 continue
```

```
1 do 500 k = 1, kl
2   scalar = x(ixs)
3   do iii = ixs, ixl-1
4     if(xx >= scalar) then
5       scalar = x(iii + 1)
6       if (xx <= scalar) then
7         ixx = iii; exit
8       endif
9     else scalar = x(iii + 1)
10    endif
11  enddo
12  ...
13 500 continue
```

Listing 3.9: A code example (on the left) from MASNUM-2.2 [112] that requires additional information for disambiguating the scope of load redundancy. Many redundant loads occur at line 6 where the array `x` is repeatedly loaded from memory. If we only focus on the inner loop, we would be misled to believe the stencil computation, which loads `x(iii + 1)` and `x(iii)`, causes many redundant loads across iterations of the inner loop. However, performing scalar replacement (on the right) does not yield much performance improvement. Actually, an algorithmic-level redundancy happens in the outer `do` loop, which repeatedly performs linear searches over a sorted array of elements.

We first extend calling contexts to incorporate loop information. Thus, the calling context of a load operation looks as follows: $main() \rightarrow loop_1 \rightarrow f() \rightarrow ... \rightarrow loop_n \rightarrow load_{old}$. Additionally, LoadSpy maintains a 64-bit global timestamp counter $\mathcal{T}$ that is incremented when passing through each loop header and also through each load operation. Thus, the calling context snapshot may appear as follows: $C_{old} = main() \rightarrow loop_1[\mathcal{T} = 1] \rightarrow f() \rightarrow ... \rightarrow loop_n[\mathcal{T} = 9] \rightarrow load_{old}$. We extend the calling context to be a tuple, that is, $\mathcal{E}_{old} = \langle \text{pointer to old context}, \mathcal{T}_{old} \rangle = \langle C_{old}, 10 \rangle$.

```
1 main () {
2   // loop1
3   for (i = 0; i < M; i++) {
4     // loop2
5     for (k = 0; k < N; k++) {
6       // load from B[i]
7       t += B[i];
8     }
9   }
10 }
```

Listing 3.10: Redundancy in the inner loop scope.

```
1 main () {
2   // loop1
3   for (i = 0; i < M; i++) {
4     // loop2
5     for (k = 0; k < N; k++) {
6       // load from A[k]
7       t += A[k];
8     }
9   }
10 }
```

Listing 3.11: Redundancy in the outer loop scope.

Listing 3.10 shows a simplified example, where the redundancy happens in the inner

loop (scope is $loop_2$). In this setting, consider the following pair of calling context snapshot:

$$\mathcal{E}_{old} = \langle main() \rightarrow loop_1[\mathcal{T} = 1] \rightarrow loop_2[\mathcal{T} = 2] \rightarrow load_{old}, \mathcal{T}_{old} = 3 \rangle$$
$$\mathcal{E}_{new} = \langle main() \rightarrow loop_1[\mathcal{T} = 1] \rightarrow loop_2[\mathcal{T} = 4] \rightarrow load_{new}, \mathcal{T}_{new} = 5 \rangle$$

Notice that the counter associated with $loop_1$ has remained unchanged whereas the counter associated with $loop_2$ has changed. Each load maintains a *pointer* to the calling context, not the entire calling context snapshot. Hence, by the time the redundancy is detected, that is, $load_{new}$ is executed, $loop_2[\mathcal{T} = 2]$ would have gotten updated to $loop_2[\mathcal{T} = 4]$; traversing C_{old} would find $\mathcal{T}_{loop_2} = 4$. Observe that $\mathcal{T}_{old} < \mathcal{T}_{loop_2} < \mathcal{T}_{new}$. This invariant informs that $loop_2$ is the scope inside which the redundancy is happening. The same invariant does not hold for $\mathcal{T}_{loop_1}$.

Now, consider a simplified example in Listing 3.11, where redundancy happens in the outer loop (scope is $loop_1$). In this setting, consider the following pair of calling context snapshot:

$$\mathcal{E}_{old} = \langle main() \rightarrow loop_1[\mathcal{T} = 1] \rightarrow loop_2[\mathcal{T} = 2] \rightarrow load_{old}, \mathcal{T}_{old} = 3 \rangle$$
$$\mathcal{E}_{new} = \langle main() \rightarrow loop_1[\mathcal{T} = 8] \rightarrow loop_2[\mathcal{T} = 9] \rightarrow load_{new}, \mathcal{T}_{new} = 10 \rangle$$

Notice that the counter associated with both $loop_1$ and $loop_2$ have changed. Hence, by the time $load_{new}$ is executed, $loop_1[\mathcal{T} = 1]$ and $loop_2[\mathcal{T} = 2]$ would have gotten updated to $loop_1[\mathcal{T} = 8]$ and $loop_2[\mathcal{T} = 9]$, respectively; traversing C_{old} would find $\mathcal{T}_{loop_1} = 8$ and $\mathcal{T}_{loop_2} = 9$. Observe that $\mathcal{T}_{old} < \mathcal{T}_{loop_1} < \mathcal{T}_{loop_2} < \mathcal{T}_{new}$. The loop with the smallest $\mathcal{T}$ value obeying this invariant, that is $loop_1$, is the redundancy scope.

Claim 1. *Given a redundancy context pair* $\langle \langle C, \mathcal{T}_{old} \rangle, \langle C, \mathcal{T}_{new} \rangle \rangle$*, the redundancy scope* $\mathcal{S}$ *is the outermost enclosing loop* i *in* C *such that* $\mathcal{T}_{old} < \mathcal{T}_{loop_i} < \mathcal{T}_{new}$.

Proof: First, $\mathcal{T}_{loop_i}$ must be in the range of $(\mathcal{T}_{old}, \mathcal{T}_{new})$ because loop i is the redundancy scope; otherwise, loop i cannot enclose the redundant load instances. Next, assume there exists another loop j in C such that $\mathcal{T}_{old} < \mathcal{T}_{loop_j} < \mathcal{T}_{loop_i} < \mathcal{T}_{new}$ but loop j is not

the redundancy scope. Loop i and j cannot be the peer loops because they are both in the same context C. Then one loop must enclose the other. (1) If loop i encloses loop j, $\mathcal{T}_{loop_i} < \mathcal{T}_{loop_j}$ because loop j's counter is incremented at least once after loop i's counter is incremented, which contradicts the assumption that $\mathcal{T}_{loop_j} < \mathcal{T}_{loop_i}$. Hence, loop j cannot be nested inside loop i. (2) If loop j encloses loop i, then loop i is no longer the outermost loop with $\mathcal{T}_{old} < \mathcal{T}_{loop_i} < \mathcal{T}_{new}$. Hence, loop j cannot enclose loop i. Since loop i and loop j are neither peer loops, nor can they be nested within one another, the assumption is void. Thus, Claim 1 holds.

Implementing Redundancy Scope: LoadSpy combines static and dynamic analysis to compute the redundancy scope $\mathcal{S}$ for each redundancy pair. First, LoadSpy instruments each loop header in the binary (in addition to procedures) to produce calling contexts with augmented loop information. It identifies an instruction as a loop header by performing an interval analysis [55] on the binary code and integrates the information into the procedure call path. We refer to the calling context with loop information as *extended calling context*. A runtime analysis routine, run as a part of each loop header, increments the 64-bit timestamp counter $\mathcal{T}$; another runtime analysis routine, run as a part of each load instruction, also increments the counter $\mathcal{T}$. Besides, the shadow memory for each byte of the original program is extended to hold the counter $\mathcal{T}$ (in addition to the 32-bit calling context handle and the 8-bit old value).

On each detected redundant load, where $C_{old} = C_{new}$, LoadSpy searches the call path from the root (`main`) toward the leaf (`load` instruction) to look for the first loop node where the Claim 1 is found to be true. Such a loop is the redundancy scope $\mathcal{S}$ for the current instance of load redundancy. Each redundancy instance records the triplet $\langle C_{old}, C_{new}, \mathcal{S} \rangle$. If $C_{old} \neq C_{new}$, LoadSpy first finds the lowest common ancestor (LCA) function or loop enclosing C_{old} and C_{new}, and then searches their common call path from the root toward the LCA to obtain $\mathcal{S}$ based on the Claim 1.

Computing the redundancy scope for each redundancy instance introduces heavy runtime overhead. We compute the redundancy scope for a calling context pair only a thresh-

old number of times (one in our experiments), which is good enough for most programs.

3.4.4 Handling Parallelism

LoadSpy maintains per-thread data structures: calling context tree, redundancy profile, $\mathcal{T}$, among others and hence needs no concurrency control for multi-threaded programs. The runtime object map is maintained as a lock-free map allowing concurrent lookups. LoadSpy detects only intra-thread redundancy and ignores inter-thread redundancy, if any.

3.4.5 Reducing Profiling Overhead

LoadSpy can introduce relatively high runtime overhead, ∼40-150×. LoadSpy adopts a bursty sampling mechanism to control its overhead [155]. Bursty sampling involves continuous monitoring for a certain number of instructions (`WINDOW_ENABLE`) followed by not monitoring for a certain (larger) number of instructions (`WINDOW_DISABLE`) and repeating it over time. These two thresholds are tunable. In our experiments, 1% sampling rate with `WINDOW_ENABLE`=1 million and `WINDOW_DISABLE`=99 million yields a good tradeoff between overhead and accuracy.

3.5 LoadSpy Workflow

LoadSpy consists of three components: a runtime profiler (detailed previously in Section 3.4), an analyzer, and a GUI. LoadSpy accepts fully optimized binary executables and collects runtime profiles via its profiler. The analyzer and GUI, run in a postmortem fashion, consume the runtime profiles and associate them with the application source code. The rest of this section discusses the analyzer and GUI.

3.5.1 Analyzer

LoadSpy's analyzer associates the runtime profiles with source code based on the DWARF [1] information produced by compilers. As the profiler produces per-thread profiles, the analyzer needs to coalesce the profiles for the whole execution. The calling context profiles can scale the analysis of program execution to a large number of CPU cores. The coalescing procedure follows the rule: two redundancy pairs from different threads are merged iff they have the same redundant load in the same calling context with the same redundancy scope. All the metrics are also merged to compute unified ones across threads. The scheme is similar for profiles from different processes.

It is worth noting that the profile coalescing overhead grows linearly with the number of threads and processes used by the monitored program. LoadSpy leverages the reduction tree technique [131] to parallelize the merging process. Typically, LoadSpy takes less than one minute to produce the aggregate profiles in all of our case studies.

3.5.2 GUI

LoadSpy's GUI inherits the design of an existing Java-based graphical interface [3], which enables navigating the calling contexts and the corresponding source code ordered by the monitored metrics. A top-down view shows a call path C starting from function `main()` to a leaf function with the breakdown of metrics at each level. Merely attributing a metric to two independent contexts loses the association between two related contexts during post-mortem inspection. To correlate the source with the target, LoadSpy allows appending a copy of the target calling context to the source calling context. For example, if a load in context `main()`→A()→`B()` is redundant with another load in context `main()`→`C`()→`D()`, LoadSpy constructs a synthetic calling context: `main`→`A()`→`B()`→`main()`→`C()`→`D()`. The redundancy metrics will be attributed to the leaf of this call chain. These synthetic call chains make it easy to visually navigate profiles and focus on top redundancy pairs. Figure 3.4 in Section 3.7.1 shows an example of the GUI, and we postpone the explanation

of the GUI details to that section.

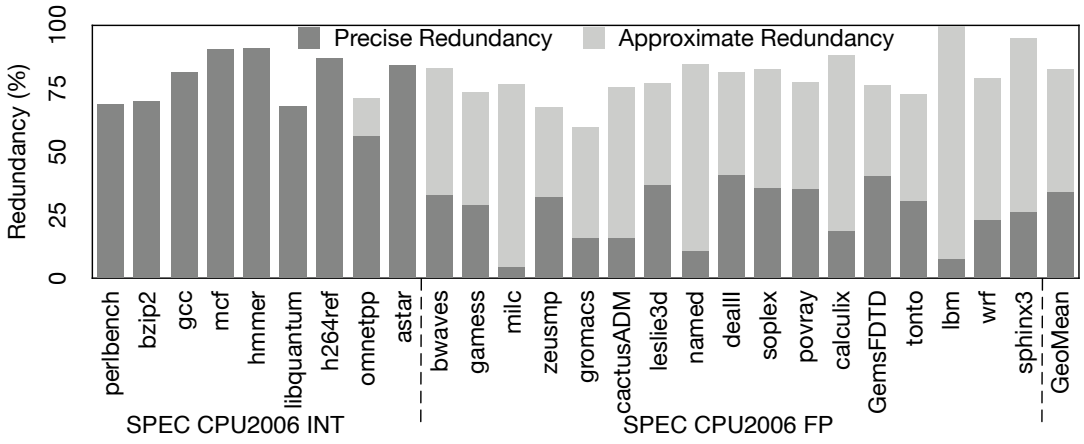
3.6 Evaluation

We evaluate LoadSpy on a 12-core Intel Xeon E5-2650 v4 CPU of 2.20GHz frequency running Linux 4.8.0. The machine has 256GB main memory. We evaluate LoadSpy with benchmarks, such as SPEC CPU2006 [124], SPEC OMP2012 [126], SPEC CPU2017 [127], Parsec-2.1 [14], Rodinia-3.1 [25], NERSC-8 [97], and Stamp-0.9.10 [92], as well as several real-world applications, such as Apache Avro-1.8.2 [8], Hoard-3.12 [13], MASNUM-2.2 [112], Shogun-6.0 [122], USQCD Chroma-3.43 [39], Stack RNN [66], Binutils-2.27 [50], and Kallisto-0.43 [89]. All the programs are compiled with `gcc-4.8.5 -O3 PGO` except Hoard-3.12 and MASNUM-2.2. For Hoard-3.12, we use `clang-5.0.0 -O3 PGO` and for MASNUM-2.2, we use `icc-17.0.4 -O3 PGO`. We apply the `ref` inputs for SPEC CPU2006, OMP2012 and CPU2017 benchmarks, the native inputs for Parsec-2.1 benchmarks, and the default inputs released with the remaining benchmarks and applications if not specified. We run all the parallel programs with four threads with simultaneous multi-threading (SMT) disabled.

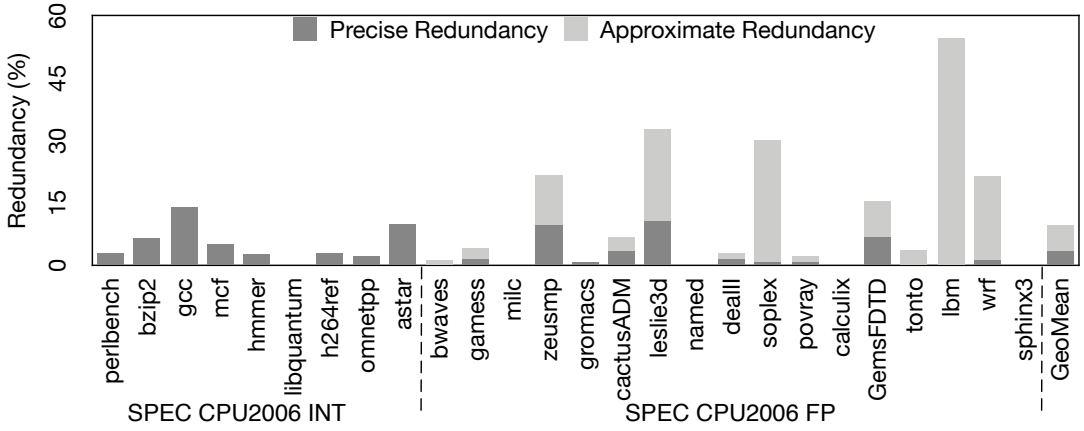
In the rest of this section, we first show the fraction of temporal and spatial redundancies on SPEC CPU2006. We then evaluate the accuracy and overhead of LoadSpy with bursty sampling enabled. We exclude three benchmarks — gobmk, sjeng, and xalancbmk — from monitoring because they have deep call recursion causing LoadSpy to run out of memory.

3.6.1 Load Redundancy in Macro Benchmarks

Figure 3.2 shows the fraction of temporal and spatial load redundancies on SPEC CPU2006 benchmarks. We can see (1) load redundancy, especially the temporal one, pervasively exists and (2) integer benchmarks show a high proportion of precise redundant loads whereas FP benchmarks show a high proportion of approximate redundant loads, as expected.



(a) Temporal redundancies.

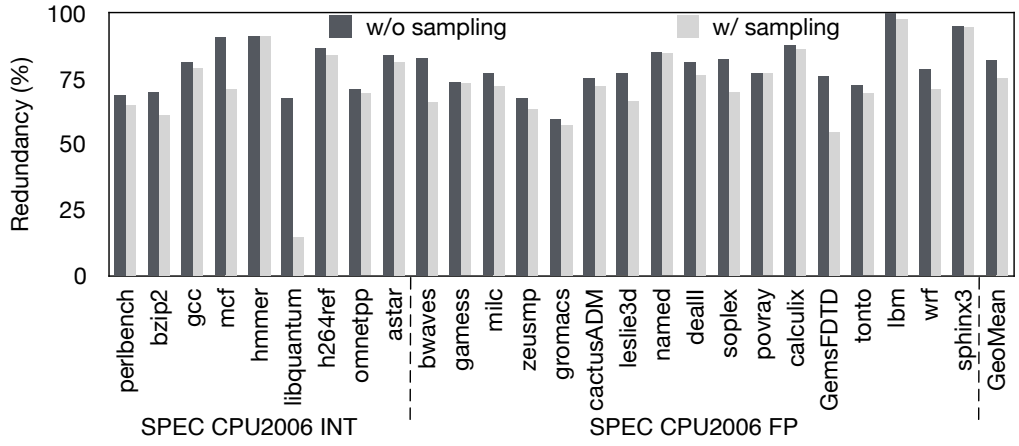


(b) Spatial redundancies.

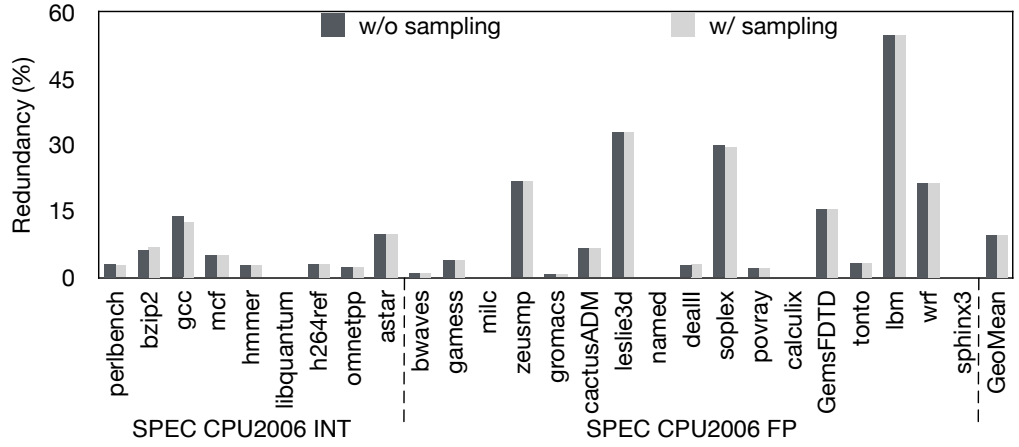
Figure 3.2: Fraction of temporal and spatial load redundancies on SPEC CPU2006.

3.6.2 Accuracy

LoadSpy offers bursty sampling as an optional feature for users willing to tradeoff measurement accuracy with performance. Figure 3.3 evaluates the accuracy of LoadSpy with bursty sampling enabled. The geometric means of spatial load redundancy fractions LoadSpy measures with sampling enabled and disabled are nearly the same — 10%. The geometric means of temporal load redundancy fractions LoadSpy measures with sampling enabled and disabled are similar — 76% and 82%. However, `libquantum` is an outlier, whose temporal redundancy fractions are 15% and 68% with sampling enabled



(a) Temporal redundancies.



(b) Spatial redundancies.

Figure 3.3: Comparing temporal and spatial load redundancies with bursty sampling disabled and enabled. The sampling rate is 1%.

and disabled. With further investigation, we find that the number of instructions executed between the source and target load operations of most redundancy pairs is more than 10 million, which is greater than the default `WINDOW_ENABLE` (= 1 million). In such a case, one can enlarge `WINDOW_ENABLE` to improve the accuracy. For instance, when we set `WINDOW_ENABLE` = 10 million and 50 million (`WINDOW_DISABLE` remains unchanged), the temporal load redundancy fraction of `libquantum` increases to 30% and 60%, respectively.

Table 3.1: LoadSpy's runtime and memory overhead on SPEC CPU2006.

Benchmark	Temporal redundancy detection		Spatial redundancy detection	
	Runtime overhead	Memory overhead	Runtime overhead	Memory overhead
perlbench	38×	11×	51×	7×
bzip2	13×	2×	13×	1.09×
gcc	19×	26×	19×	25×
mcf	6×	14×	6×	1.04×
hmmer	12×	35×	11×	20×
libquantum	12×	18×	13×	2×
h264ref	21×	20×	21×	2×
omnetpp	10×	16×	14×	25×
astar	11×	13×	11×	18×
bwaves	17×	14×	15×	1.16×
gamess	24×	25×	24×	24×
milc	4×	10×	4×	1.18×
zeusmp	8×	14×	7×	1.42×
gromacs	10×	23×	9×	15×
cactusADM	7×	10×	7×	1.36×
leslie3d	9×	10×	8×	2×
named	10×	11×	10×	9×
dealII	21×	30×	22×	19×
soplex	13×	13×	13×	2×
povray	29×	216×	28×	70×
calculix	21×	18×	20×	19×
GemsFDTD	8×	14×	8×	1.42×
tonto	22×	49×	24×	30×
lbm	4×	14×	3×	1.15×
wrf	15×	10×	16×	3×
sphinx3	13×	16×	13×	7×
Median	**12.5**×	**14**×	**13**×	**5**×
GeoMean	**13**×	**17**×	**13**×	**5**×

3.6.3 Overhead

Table 3.1 shows the runtime and memory overhead of LoadSpy on SPEC CPU2006 benchmarks. The runtime (memory) overhead is measured as the ratio of the runtime (peak memory usage) of a benchmark with LoadSpy enabled to the runtime (peak memory usage) of its native execution. The geometric means of runtime overheads for detecting temporal and spatial redundancies are both 13×, and the geometric means of memory overheads for detecting temporal and spatial redundancies are 17× and 5×, respectively. A few benchmarks such as `tonto` and `povray` show excessive memory overhead due to the following reasons: (1) `tonto` has a deep call stack, which demands excessive space to maintain its calling context tree and (2) `povray` has a small memory footprint (∼6MB), whereas some preallocated data structures in LoadSpy overshadow this baseline memory footprint.

Table 3.2: Overview of performance improvement guided by LoadSpy.

Program information			LoadSpy		Optimization	
	Program	Problematic code	Redundancy type	Inefficiency	Approach	Speedup
Macro benchmark	359.botsspar	sparselu.c:loop (191)	Temporal	Inefficient register usage	Scalar replacement	1.77×
	453.povray	csg.cpp (250)	Temporal	Missed inline substitution	Function inlining	1.05×
	464.h264ref	mv-search.c:loop (394)	Temporal	Missed inline substitution	Function inlining	1.28×
	✓470.lbm	lbm.c:LBM_performStreamCollide	Spatial	Redundant computation	Approximate computing	1.25×
	✓538.imagick_r	morphology.c:loop (2982)	Spatial	Redundant computation	Conditional computation	1.25×
	✓backprop	backprop.c:loop (322)	Spatial	Input-sensitive redundancy	Conditional computation	1.13×
	✓hotspot3D	3D.c:loop (98, 166)	Temporal	Inefficient register usage	Scalar replacement	1.13×
	✓lavaMD	kernel_cpu.c (175)	Temporal	Redundant function calls	Reusing the previous result	1.39×
	✓srad_v1	main.c:loop (256)	Temporal	Inefficient register usage	Scalar replacement	1.11×
	✓srad_v2	srad.cpp:loop (131)	Temporal	Inefficient register usage	Scalar replacement	1.12×
	✓particlefilter	ex_particle_OPENMP_seq.c:findIndex	Temporal	Linear search	Binary search	9.8×
	vacation	client.c:loop (198)	Temporal	Redundant function calls	Reusing the previous result	1.23×
	dedup	hashtable.c:hashtable_search	Temporal	Poor hashing	Reducing hash collisions	1.11×
	msgrate	msgrate.c:cache_invalidate	Temporal	Missed constant propagation	Copy propagation	3.03×
Real application	✓Apache Avro-1.8.2	Specific.hh (110, 117)	Temporal	Missed inline substitution	Function inlining	1.19×
	✓Hoard-3.12	libhoard.cpp:xxmalloc	Temporal	Redundant computation	Reusing the previous result	1.14×
	✓MASNUM-2.2	propagat.inc:loop (130, 140)	Temporal	Linear search	Locality-friendly search	1.79×
	✓USQCD Chroma-3.43	qdp_random.h (56)	Temporal	Missed inline substitution	Function inlining	1.06×
	✓Shogun-6.0	DenseFeatures.cpp (505) Distance.cpp (185)	Temporal	Missed inline substitution	Function inlining	1.06×
	✓Stack RNN	StackRNN.h:loop (350, 355, 363, 367)	Temporal Spatial	Poor choice of algorithm Redundant computation	Loop fusion Conditional computation	1.09×
	Kallisto-0.43	KmerHashTable.h (131)	Temporal	Poor hashing	Reducing hash collisions	4.1×
	Binutils-2.27	dwarf2.c:loop (2166)	Temporal	Linear search	Binary search	3.29×

✓: newfound performance bugs via LoadSpy.

3.7 Case Studies

Table 3.2 summarizes the load redundancies found in some benchmarks and real-world applications, and the speedups obtained by eliminating them. We quantify the performance improvement in execution time except for Hoard, which is in throughput. In the rest of this section, we exhaustively analyze all the newfound performance bugs.

3.7.1 Apache Avro-1.8.2

Avro [8] is a remote procedure call and data serialization processing system. We apply LoadSpy to evaluate the C++ version of Avro with benchmarks developed by Sorokin [123]. LoadSpy reports a temporal redundancy fraction of 79% for the entire program. Figure 3.4 shows the full calling contexts of the top redundancy pair visualized through LoadSpy's GUI. The GUI consists of three panes: the top pane shows the program source code, the bottom left pane shows the full calling contexts of each redundancy pair, and the bottom right pane shows the metrics associated with each redundancy pair. In this figure, the GUI shows two metrics: the number of redundant loads for each redundancy pair and percentage of redundant instances for each pair, which if 100%, means every instance of this pair is redundant.

We can see that the redundant loads in function `doEncodeLong()` account for 25% of the total redundant loads in the program. Moreover, all instances of this pair are redundant and the redundancy scope is the loop at lines 229-233 in the file `Specific.hh` enclosing the call site of function `encode()`. `encode()` is the caller of `doEncodeLong()`. With further analysis, we find that the epilog of `doEncodeLong()` consistently pops the same value from the same stack location to restore the register value. To eliminate redundant loads in the function epilog, we inline `doEncodeLong()` into its caller. LoadSpy also identifies another problematic function (not shown) and guides the same inlining optimization. Together, these optimizations eliminate 31% of the memory loads and 37% of the redundant memory loads, yielding a 1.19× speedup for the whole program.

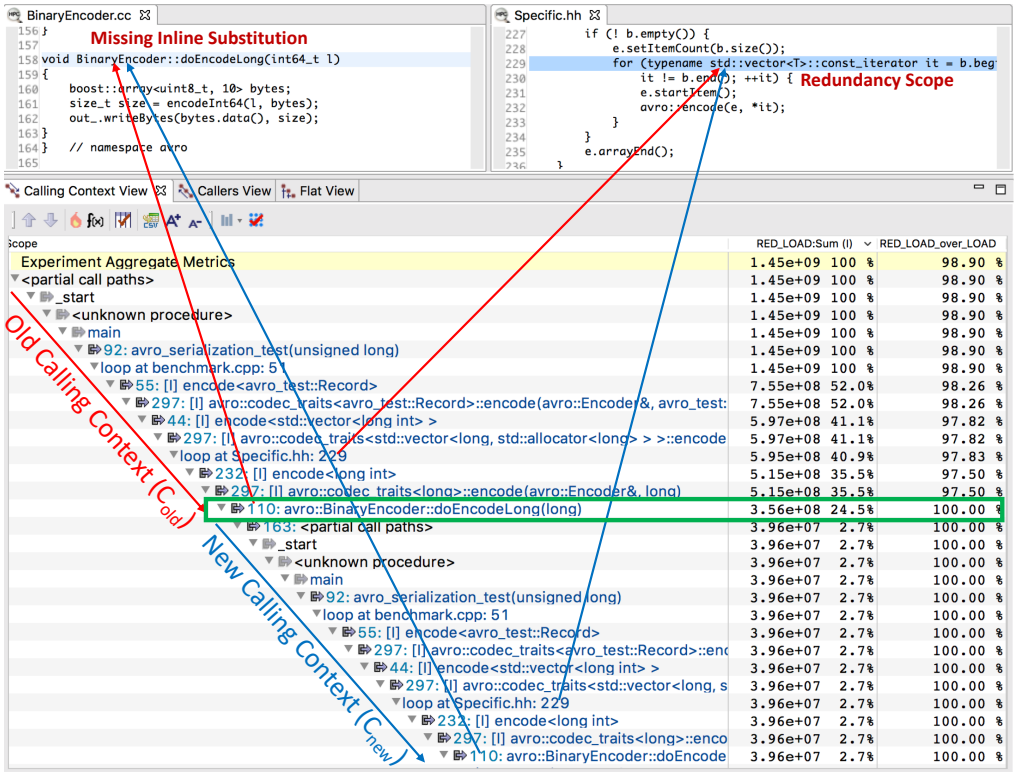


Figure 3.4: The top redundancy pair in Apache Avro-1.8.2 with full calling contexts reported by LoadSpy. Along the calling contexts shown in the bottom left pane, a procedure name following a symbol `[I]` means it is inlined. We can see that most procedures on the path are inlined except `doEncodeLong()`. Many redundant loads are from calling `doEncodeLong()`, which can be removed by function inlining.

3.7.2 MASNUM-2.2

MASNUM [112], one of the 2016 ACM Gordon Bell Prize finalists, forecasts ocean surface waves and climate change. It is written in Fortran and parallelized with MPI. LoadSpy identifies 91% of memory loads are redundant, of which 15% are attributed to array `x` at line 6 on the left of Listing 3.9. LoadSpy also pinpoints the redundancy scope as the outer loop at line 1. We find that the inner loop (line 5) performs a linear search over the non-decreasing array `x` for a given input `xx`. With multiple iterations, elements of array `x` are frequently loaded from memory for comparison, leading to the redundancy. Changing the linear search to a binary search reduces redundant loads and yields a 1.32× speedup for the entire program. It is worth noting that the binary search still incurs high load redundancy fraction because of the intensive search requests in the program. To further improve the search algorithm, we analyze the values of `xx` across iterations. We find that `xx` has good value locality, that is, the values are similar in adjacent iterations of the outer loop. Thus, we replace the binary search with a locality-friendly search. We memoize the location index `iii` when the current search finishes; in the next search, we begin at the recorded `iii` and alternate the linear search in both directions to the array start and end. This optimization eliminates 33% of the memory loads and 36% of the redundant memory loads, yielding a 1.79× speedup for the entire program.

3.7.3 Hoard-3.12

Hoard [13], a high-performance cross-platform C++ based memory allocator, has been integrated into an array of applications and programming languages such as GNU Bayonne and Cilk programming language. It has 20K lines of code and is parallelized with the `PThreads` library. LoadSpy identifies that 58% of memory loads are redundant on profiling Hoard's built-in benchmark `larson`. The top redundancy pair is associated with lines 4 and 7 shown in Listing 3.12, which accounts for 11% of the total redundant loads. The cause of such redundancy is that the program repeatedly checks whether `theTLAB` is a

```
1 static __thread TheCustomHeapType *theTLAB INITIAL_EXEC_ATTR = nullptr;
2 ...
3 bool isCustomHeapInitialized() {
4 ▶ return (theTLAB != nullptr);
5 }
6 TheCustomHeapType *getCustomHeap() {
7 ▶ auto tlab = theTLAB;
8   if (tlab == nullptr) {
9     tlab = initializeCustomHeap();
10    theTLAB = tlab;
11  }
12  return tlab;
13 }
14 void *xxmalloc (size_t sz) {
15  if (isCustomHeapInitialized()) {
16   void *ptr = getCustomHeap()->malloc(sz);
17   ...
18  }
19 }
```

Listing 3.12: Temporal load redundancy in Hoard-3.12. The program repeatedly checks whether the pointer variable `theTLAB` is null.

null pointer. More specifically, function `isCustomHeapInitialized()` invoked at line 15 and function `getCustomHeap()` invoked at line 16 both include the code to check whether `theTLAB` is equal to `nullptr`. Hence, the second check at lines 8-11 is redundant.

To eliminate such redundant loads, we inline these two functions into their caller `xxmalloc()` and remove the redundant check. This optimization eliminates 3% of the memory loads and 2% of the redundant memory loads, which improves the throughput (i.e., the number of memory operations per second) of Hoard by 1.14×.

3.7.4 USQCD Chroma-3.43

Chroma [39] is a complex toolbox for performing quantum chromodynamics lattice computations, which has more than 200K lines of code. We evaluate it using the built-in benchmark `t_mesplq`. LoadSpy reports a temporal redundancy fraction of 61%. The top redundancy pair is attributed to the function `sranf()` at line 3 shown in Listing 3.13. With further investigation, we notice that Chroma has a similar performance bug to the one in Apache Avro: the epilog of `sranf()` repeatedly pops the same values from the same stack locations to restore register values.

To eliminate such redundant loads, we manually inline sranf() into its caller. This

```
1 template<class T1, class T2>
2 inline void fill_random(float &d, T1 &seed, T2 &skewed_seed, const T1 &seed_mult) {
3 ▶ d = float(RNG::sranf(seed, skewed_seed, seed_mult));
4 }
```

Listing 3.13: Temporal load redundancy in USQCD Chroma-3.43. The epilog of function `sranf()` often pops the same values from the same stack locations to restore register values.

optimization eliminates 6% of the memory loads and 7% of the redundant memory loads, yielding a 1.06× speedup for the whole program.

3.7.5 Shogun-6.0

Shogun [122] is an efficient machine learning toolbox. LoadSpy reports a temporal redundancy fraction of 71% on profiling its built-in benchmark `kernel_matrix_sum_benchmark`. Listing 3.14 shows one of the top redundancy pairs at line 6. The cause of such redundancy is similar to Apache Avro: the epilog of function `get_feature_vector()` repeatedly pops the same values from the same stack locations to restore register values. We manually inline the callee into its caller to eliminate the redundant loads. Additionally, We perform the same optimization for other function invocations that have the same performance issue. These optimizations eliminate 7% of the memory loads and 2% of the redundant memory loads, yielding a 1.06× speedup for the whole program.

```
1 template<class ST> float64_t CDenseFeatures<ST>::dot(int32_t vec_idx1, CDotFeatures
      *df, int32_t vec_idx2) {
2   ...
3   CDenseFeatures<ST> *sf = (CDenseFeatures<ST> *)df;
4   int32_t len1, len2;
5   bool free1, free2;
6 ▶ ST *vec1 = get_feature_vector(vec_idx1, len1, free1);
7   ...
8 }
```

Listing 3.14: Temporal load redundancy in Shogun-6.0. The epilog of function `get_feature_vector()` often pops the same values from the same stack locations to restore register values.

```
1 for (my_int i = _TOP_OF_STACK; i < _TOP_OF_STACK + _STACK_SIZE - 1; i++) {
2 ▶ _pred_err_stack[s][i+1] += _err_stack[s][i] * _act[s][itm][pop];
3 }
4 for (my_int i = _TOP_OF_STACK; i < _TOP_OF_STACK + _STACK_SIZE - 1; i++) {
5 ▶ _err_act[s][pop] += _err_stack[s][i] * _stack[s][old_it][i+1];
6 }
7 _err_act[s][pop] += _err_stack[s][_TOP_OF_STACK + _STACK_SIZE - 1] *
      EMPTY_STACK_VALUE;
8 for (my_int i = _TOP_OF_STACK + 1; i < _TOP_OF_STACK + _STACK_SIZE; i++) {
9 ▶ _pred_err_stack[s][i-1] += _err_stack[s][i] * _act[s][itm][push];
10 }
11 for (my_int i = _TOP_OF_STACK + 1; i < _TOP_OF_STACK + _STACK_SIZE; i++) {
12 ▶ _err_act[s][push] += _err_stack[s][i] * _stack[s][old_it][i-1];
13 }
```

Listing 3.15: Temporal and spatial load redundancies in Stack RNN. Array `_err_stack` is loaded from memory by each of the four loops, resulting in temporal load redundancy. Besides, most elements of array `_err_stack` equal zero, resulting in spatial load redundancy.

3.7.6 Stack RNN

Stack RNN [66] is a C++ based project originating from Facebook AI research, which applies the memory stack to optimize and extend a recurrent neural network. We evaluate Stack RNN by profiling its built-in application `train_add` with LoadSpy. LoadSpy quantifies a redundancy fraction of 81%, and pinpoints that the top temporal and spatial load redundancy pairs are associated with four loops shown in Listing 3.15.

The cause of the temporal load redundancy is that each of the four loops accesses array `_err_stack`. However, the compiler cannot keep all elements of array `_err_stack` in CPU registers across these loops. Thus, the elements of array `_err_stack` are repeatedly loaded from memory into registers. We eliminate the temporal redundant loads by loop fusion, which fuses the four loops into one such that array `_err_stack` is only loaded once.

The cause of the spatial load redundancy is that most elements of array `_err_stack` are zeros, resulting in identity computation at lines 2, 5, 9, and 12 shown in Listing 3.15. We employ a conditional check to avoid the computation on identities. These two optimizations together eliminate 10% of the memory loads and 15% of the redundant memory loads, yielding a 1.09× speedup for the whole program.

```
1 for (i = 0; i < Nr; i++) {
2   iN[i] = i-1;
3   iS[i] = i+1;
4 }
5 ...
6 iN[0] = 0;
7 iS[Nr-1] = Nr-1;
8 ...
9 for (j = 0; j < Nc; j++) {
10   for (i = 0; i < Nr; i++) {
11     k = i + Nr*j;
12 ▶    Jc = image[k];
13 ▶    dN[k] = image[iN[i] + Nr*j] - Jc;
14 ▶    dS[k] = image[iS[i] + Nr*j] - Jc;
15   }
16 }
```

Listing 3.16: Temporal load redundancy in Rodinia-3.1 srad_v1. Array `image` is repeatedly loaded from memory while the values remain unchanged.

3.7.7 Rodinia-3.1 Srad

Srad [25] applies partial differential equations to filter noise in images, which is widely used in ultrasonic and radar imaging applications. We profile the OpenMP version of srad_v1. LoadSpy reports a temporal redundancy fraction of 99%, of which 8% is attributed to array `image` at lines 12-14 shown in Listing 3.16. We notice when 0 < `i` < `Nr` - 1, the value of `image[iS[i] + Nr * j]` in one iteration equals the value of `image[k]` in the next iteration and further equals the value of `image[iN[i] + Nr * j]` in the iteration after next.

To fix this problem, we adopt scalar replacement to avoid redundant loads across iterations, which eliminates 33% of the memory loads and yields a 1.11× speedup for the whole program. It is worth noting that the indirect accesses in this inefficient code snippet introduce challenges in compiler's static analysis and optimization.

Additionally, LoadSpy also identifies the same inefficiency occurring in srad_v2. With the same optimization, srad_v2 achieves a 1.12× speedup.

3.7.8 Rodinia-3.1 LavaMD

LavaMD [25] calculates particle potential and relocation among particles. We apply LoadSpy to evaluate its OpenMP version. LoadSpy reports that 87% of memory loads

are redundant, and the top contributor is the `glibc` function `exp()` at line 7 shown in Listing 3.17. We notice that the value of `u2` often remains unchanged across iterations. As a result, a number of redundant loads and computations occur inside `exp()` due to redundant function calls. With further analysis, we find that `a2` is a loop invariant, and `u2` is derived from `a2` and `r2`. Thus, we infer that `r2` often has the same value across iterations.

To optimize this inefficiency, we introduce a conditional check on `r2` such that the program can reuse the return value of `exp()` from the previous iteration if the value of `r2` has not changed. This optimization eliminates 76% of the memory loads and 93% of the redundant memory loads, yielding a 1.39× speedup for the entire program.

```
1 for (k = 0; k < (1 + box[l].nn); k++) {
2   ...
3   for (i = 0; i < NUMBER_PAR_PER_BOX; i = i + 1) {
4     for (j = 0; j < NUMBER_PAR_PER_BOX; j = j + 1) {
5       r2 = rA[i].v + rB[j].v - DOT(rA[i], rB[j]);
6       u2 = a2 * r2;
7 ▶     vij = exp(-u2);
8       fs = 2. * vij;
9       ...
10    }
11  }
12 }
```

Listing 3.17: Temporal load redundancy in Rodinia-3.1 lavaMD due to redundant function calls.

3.7.9 SPEC CPU2017 538.imagick_r

538.imagick_r [127] is applied to create, edit, compose, or convert bitmap images. LoadSpy reports that spatial redundant loads account for 13% of the total memory loads, of which 24% are attributed to the variable `k` at lines 6-9 shown in Listing 3.18. `k` is a pointer to the FP array `values` at line 2 and decremented by one in each loop iteration. We find most array elements are zeros, causing `*k` to equal zero in most loop iterations.

To remove the identity computation on `*k`, we introduce a conditional check to filter out zero values. With this optimization, the memory loads and redundant memory loads are reduced by 19% and 51% respectively, and the whole program gains a 1.25× speedup.

```
1 register const double *restrict k;
2 k = &kernel->values[kernel->width * kernel->height-1]
3 ...
4 for (u = 0; u < (ssize_t) kernel->width; u++, k--) {
5   if (IsNaN(*k)) continue;
6 ▶   result.red += (*k) * k_pixels[u].red;
7 ▶   result.green += (*k) * k_pixels[u].green;
8 ▶   result.blue += (*k) * k_pixels[u].blue;
9 ▶   result.opacity += (*k) * k_pixels[u].opacity;
10    ...
11 }
```

Listing 3.18: Spatial load redundancy in SPEC CPU2017 538.imagick_r. Array `values` is frequently loaded from memory whereas most array elements equal zero.

```
1 #define SWEEP_START(x1, y1, z1, x2, y2, z2) \
2   for(i = CALC_INDEX(x1, y1, z1, 0); \
3   i < CALC_INDEX(x2, y2, z2, 0); \
4   i += N_CELL_ENTRIES) {
5 #define SWEEP_END }
6 ...
7 static double srcGrid[SIZE_Z * SIZE_Y * SIZE_X * N_CELL_ENTRIES];
8 ...
9 SWEEP_START(0, 0, 0, 0, 0, SIZE_Z) // loop entry
10 ...
11 ▶ rho = + SRC_C(srcGrid) + SRC_N(srcGrid)
12 ▶ + SRC_S(srcGrid) + SRC_E(srcGrid)
13 ▶ + SRC_W(srcGrid) + SRC_T(srcGrid)
14 ▶ + SRC_B(srcGrid) + SRC_NE(srcGrid)
15 ▶ + SRC_NW(srcGrid) + ...
16 ▶ ux = + SRC_E(srcGrid) - SRC_W(srcGrid)
17 ▶ + SRC_NE(srcGrid) - SRC_NW(srcGrid)
18 ▶ + SRC_SE(srcGrid) - SRC_SW(srcGrid)
19 ▶ + SRC_ET(srcGrid) + SRC_EB(srcGrid)
20 ▶ -SRC_WT(srcGrid) - SRC_WB(srcGrid);
21 ▶ uy = + SRC_N(srcGrid) - SRC_S(srcGrid)
22 ▶ + SRC_NE(srcGrid) + SRC_NW(srcGrid)
23 ▶ - SRC_SE(srcGrid) - SRC_SW(srcGrid)
24 ▶ + SRC_NT(srcGrid) + SRC_NB(srcGrid)
25 ▶ - SRC_ST(srcGrid) - SRC_SB(srcGrid);
26 ▶ uz = + SRC_T(srcGrid) - SRC_B(srcGrid)
27 ▶ + SRC_NT(srcGrid) - SRC_NB(srcGrid)
28 ▶ + SRC_ST(srcGrid) - SRC_SB(srcGrid)
29 ▶ + SRC_ET(srcGrid) - SRC_EB(srcGrid)
30 ▶ + SRC_WT(srcGrid) - SRC_WB(srcGrid);
31 ...
32 ▶ DST_WT(dstGrid) = (1.0 - OMEGA) * SRC_WT(srcGrid) + ...
33 ▶ DST_WB(dstGrid) = (1.0 - OMEGA) * SRC_WB(srcGrid) + ...
34 ...
35 SWEEP_END // loop exit
```

Listing 3.19: Spatial load redundancy in SPEC CPU2006 470.lbm. Array `srcGrid` is frequently loaded from memory while most array elements have same values.

3.7.10 SPEC CPU2006 470.lbm

470.lbm [124] employs the Lattice Boltzmann Method (LBM) to simulate incompressible fluids in three-dimensional space. LoadSpy reports that spatial redundant loads account

for 55% of the memory loads, of which more than 30% are attributed to array `srcGrid` at lines 11-33 shown in Listing 3.19. With code study, we find array `srcGrid` is traversed across loop iterations and most of its elements are identical, resulting in many redundant loads.

To optimize this inefficiency, we apply loop perforation [120] to reduce the number of iterations at the cost of accuracy. With this optimization, the memory loads and redundant memory loads are reduced by 26% and 60% respectively, and the whole program gains a 1.25× with trivial accuracy loss (7.7e-5%).

3.8 Summary

In this chapter, we present a study of identifying program inefficiencies by focusing on whole-program load redundancy. We demonstrate that redundant loads are often a symptom of various inefficiencies arising from inputs, suboptimal algorithms or data structures, and missed compiler optimizations. To pinpoint these inefficiencies in complex software code bases, we have developed LoadSpy, a fine-grained profiler that profiles load redundancy. LoadSpy works on fully optimized binary executables, adopts various optimization techniques (e.g., bursty sampling, lock-free data structures, reduction trees) to reduce the online profiling and offline data coalescing overhead, and provides a rich GUI, which make it a complete developer tool. We evaluate LoadSpy using benchmarks and real-world applications. Guided by LoadSpy, we are able to optimize prior-known and newfound inefficiencies in these programs, yielding nontrivial speedups.

Chapter 4

Pinpointing Performance Inefficiencies in Java

4.1 Introduction

Managed languages, such as Java, have become increasingly popular in various domains, including web services, graphical 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 profiling abounds in the Java world to aid developers to understand their program behavior. Profiling for execution hotspots is the most popular one [78, 76, 41, 49, 30, 29]. Hotspot analysis tools identify code regions that are frequently executed disregarding whether the execution is efficient or inefficient (useful or wasteful) and hence

a significant burden is on developers to make a judgment call on whether there is scope to optimize a hotspot.

There is a need for tools that specifically pinpoint wasteful work and guide developers to focus on code regions where the optimizations are demanded. Based on numerous case studies investigated in this chapter, we find 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. 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.

Suboptimal code generation: missed inlining can show up as storing the same value to the same stack location; 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 [12] performs the expensive power method inside a loop and in each iteration, the power method pushes the same parameters on the same stack locations. 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 [146, 99, 35, 100, 121] identify inefficiencies of this form. However, these tools are based on exhaustive Java byte code instrumentation, which suffers from two drawbacks: (1) high (up to 200×) runtime overhead, which prevents them from being used in production and (2) missing insights into lower-level layers, e.g., inefficiencies in machine code.

```
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     }
158   }
159 }
```

Listing 4.1: Redundant memory loads in SPECjvm2008 scimark.fft. `data[i]` is loaded from memory twice in a single iteration whereas it is unmodified between these two loads.

4.1.1 Motivating Example

Listing 4.1 shows a hot loop in a JITted method (compiled with Oracle HotSpot JIT compiler) in SPECjvm2008 scimark.fft [125], a standard implementation of Fast Fourier Transforms (FFT). The JITted assembly code of the source code at lines 153 and 155 is shown in Figure 4.1. Notice the two loads from the memory location `data[i]` (`0x10(%r9,%r8,8)`) — once into register `xmm2` at line 153 and then into register `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 loads, yielding a 1.13× speedup for the en-

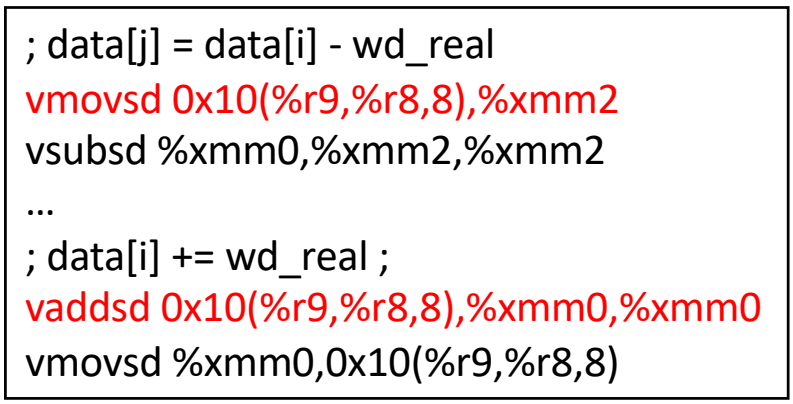


Figure 4.1: The assembly code (at&t style) of lines 153 and 155 in Listing 4.1.

tire program. Without access to the source code of the commercial Java runtime, we cannot definitively know 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 the 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 4.6.1.

4.1.2 Contribution Summary

We propose JXPerf to complement existing Java profilers; JXPerf samples PMUs and employs debug registers available in commodity CPUs to identify program inefficiencies that exhibit as wasteful memory operations at runtime. In this work, we make the following contributions:

- 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.

- Demonstrate that JXPerf identifies inefficiencies at machine code, which can be introduced by poor code generation, inappropriate data structures, or suboptimal

algorithms.

- Perform a thorough evaluation of 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, yielding significant speedups after eliminating such inefficiencies.

4.2 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.

4.2.1 Hotspot Profilers

Profilers such as Async-profiler [109], Jprofiler [41], YourKit [49], VisualVM [30], Oracle Developer Studio Performance Analyzer [29], and IBM Health Center [61] pinpoint hotspots in Java programs. Most hotspot profilers incur low overhead because they use interrupt-based sampling techniques supported by PMUs or OS timers. Hotspot profilers can identify code sections that account for a large number of resources such as CPU cycles, cache misses, heap memory usage, and floating-point operations. However, they cannot tell whether the resources are used efficiently.

4.2.2 Inefficiency Profilers

Glider [35] generates tests that expose redundant operations in Java collection traversals. MemoizeIt [33] 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 [147], inefficiently-used containers [149], low-utility data structures [148], reusable data structures [146], and cacheable data structures [99]. Unlike hotspot ones, these profilers can pinpoint redundant operations that

lead to resource wastage.

JXPerf is also an inefficiency profiler. Unlike prior work that uses exhaustive byte code instrumentation, JXPerf exploits features available in hardware (PMUs and debug registers) that eliminates instrumentation and dramatically reduces overhead. Section 4.5 details the comparison between JXPerf and the profilers based on exhaustive byte code instrumentation.

Remix [40], similar to JXPerf, also utilizes PMUs; while JXPerf identifies intra-thread inefficiencies, e.g., wasteful operations, Remix identifies false sharing across threads.

4.3 Methodology

We define the following three kinds of wasteful memory access patterns.

Definition 4 (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 is no intervening memory load 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 5 (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 6 (Silent load[1])**.** *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.*

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 FP values within a user-specified threshold of difference (1% by default). For memory operations involved in inefficiencies, we use their calling contexts instead of their effective addresses to represent them, which can facilitate optimization efforts.

[1] In this dissertation, "redundant load" and "silent load" are interchangeable terms.

Figure 4.2 highlights the idea of JXPerf in detecting inefficiencies at runtime, exemplified with silent stores. PMUs drive JXPerf by sampling precise memory stores. On a sampled memory store, JXPerf records the effective address captured by the PMU, reads the value stored at this address, and sets a `W_TRAP` watchpoint at this address via a debug register. The subsequent store to the same address traps the execution. JXPerf captures the trap and checks the value stored 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.

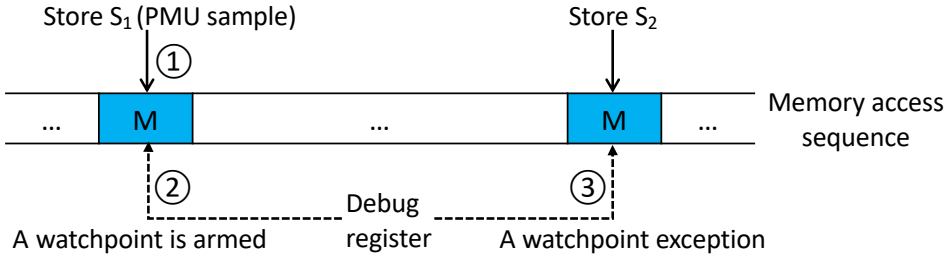


Figure 4.2: JXPerf's scheme for silent store detection. ① The PMU samples a memory store S_1 that touches location M. ② In the PMU sample handler, a debug register is armed to monitor the subsequent access to M. ③ The debug register traps on the next store S_2 to M. ④ If S_1 and S_2 write the same value to M, JXPerf labels S_2 as a silent store and $\langle S_1, S_2 \rangle$ as a silent store pair.

4.4 Design and Implementation

JXPerf, similar to LoadSpy, also produces per-thread profiles at runtime and merges them to into a single profile in a postmortem fashion to minimize thread synchronization overhead. Figure 4.3 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.

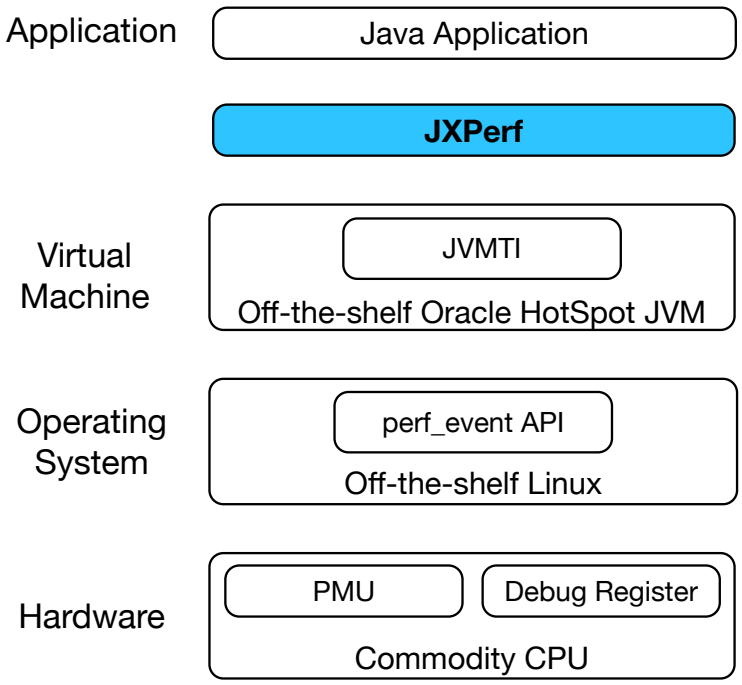


Figure 4.3: Overview of JXPerf in the system stack.

4.4.1 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 the `perf_event` API at the thread creation callback.

2. When a PMU counter overflows during program 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` watchpoint at M via a debug register and resumes the execution.

4. A subsequent store to M triggers a watchpoint 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 a silent store.

6. JXPerf disarms the debug register and resumes the 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` watchpoint at M, and resumes program execution. When the subsequent access to M 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 a dead store. 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 [2] 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 a silent load.

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

$$\begin{aligned}
\mathcal{F}_{prog}^{DeadStore} &= \frac{\sum_i \sum_j \text{Dead bytes stored in } \langle C_i, C_j \rangle}{\sum_i \sum_j \text{ Bytes stored in } \langle C_i, C_j \rangle} \\
\mathcal{F}_{prog}^{SilentStore} &= \frac{\sum_i \sum_j \text{Silent bytes stored in } \langle C_i, C_j \rangle}{\sum_i \sum_j \text{ Bytes stored in } \langle C_i, C_j \rangle} \\
\mathcal{F}_{prog}^{SilentLoad} &= \frac{\sum_i \sum_j \text{Silent bytes loaded from } \langle C_i, C_j \rangle}{\sum_i \sum_j \text{Bytes loaded from } \langle C_i, C_j \rangle}
\end{aligned} \tag{4.1}$$

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

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

$$\begin{aligned}
\mathcal{F}^{DeadStore}_{\langle C_{watch}, C_{trap}\rangle} &= \frac{\text{Dead bytes stored in } \langle C_{watch}, C_{trap}\rangle}{\sum_i \sum_j \text{Bytes stored in } \langle C_i, C_j\rangle} \\
\mathcal{F}^{SilentStore}_{\langle C_{watch}, C_{trap}\rangle} &= \frac{\text{Silent bytes stored in } \langle C_{watch}, C_{trap}\rangle}{\sum_i \sum_j \text{Bytes stored in } \langle C_i, C_j\rangle} \\
\mathcal{F}^{SilentLoad}_{\langle C_{watch}, C_{trap}\rangle} &= \frac{\text{Silent bytes loaded from } \langle C_{watch}, C_{trap}\rangle}{\sum_i \sum_j \text{Bytes loaded from } \langle C_i, C_j\rangle}
\end{aligned} \tag{4.2}$$

```
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 4.2: Long-distance silent loads. All four watchpoints are armed in the first four samples taken in loop `i` at the sampling period of 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 reservoir sampling to ensure each sample equal probability to survive.

4.4.2 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 4.2. 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 registers 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 detect a pair of silent loads separated by only a few intervening loads, and the latter choice allows us to potentially detect a pair of silent loads separated by many 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 [137, 143, 138], 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 i^{th} sampled address M_i since the last time a debug register was available, occupies the previously armed watchpoint with $1/i$ probability. The probability P_k of monitoring any sampled address M_k, $1 \leq k \leq i$, 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 $N \geq 1$. JXPerf maintains an independent reservoir probability P_α for each debug register α, ($1 \leq \alpha \leq N$). On a PMU sample, if there is an available debug register, JXPerf arms it and decreases 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 $\mathcal{O}(1)$ memory.

4.4.3 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 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 in that thread. Notice that GC threads are 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 [15].

4.4.4 Attributing Measurement to Binary

JXPerf uses Intel XED library [62] 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.

4.4.5 Attributing Measurement to Calling Context

Oracle JDK offers users two APIs to obtain the calling context of an instruction: 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 [96, 57]. To avoid the bias, JXPerf employs `AsyncGetCallTrace()` to facilitate non-safepoint collection of calling contexts [102]. `AsyncGetCallTrace()` accepts `u_context` obtained from a PMU interrupt or watchpoint trap as the input, and returns the method ID and byte code index (BCI) for each stack frame in the calling context of this interrupt or trap. 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.

4.5 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 [15], Dacapo-9.12-MR1-bach [15], and ScalaBench [118] as well as on two real-world performance bug datasets [4, 94]. All the programs are built with Oracle JDK11 except DaCapo 2006 bloat, Dacapo-9.12-MR1-bach batik and eclipse, and ScalaBench actors, which are built 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. The parallel programs, excluding threads used for the JIT compilation and GC, are run with four threads if 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 [47] to use a confidence interval for the mean to report results. The confidence interval for the mean is computed by 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., $z = 1.96$).

$$\overline{x} \pm z \times \frac{\sigma}{\sqrt{n}} \tag{4.3}$$

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 numbers of debug registers. We then evaluate the overhead of JXPerf on them. 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

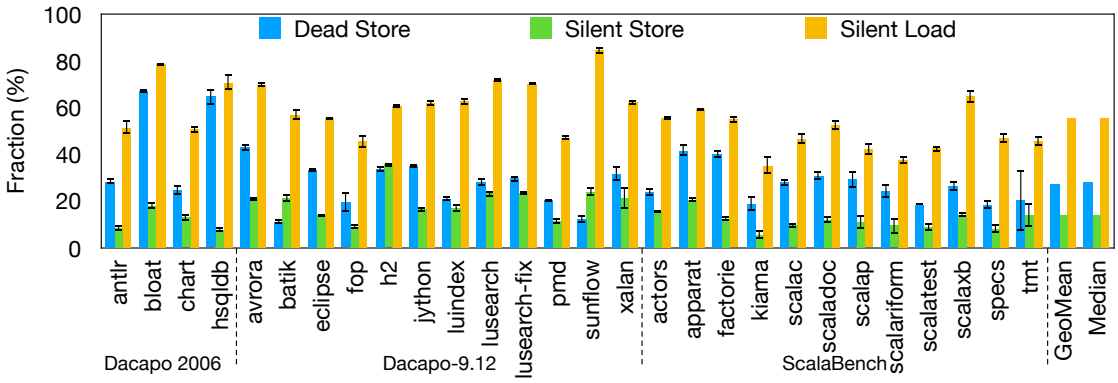


Figure 4.4: Fraction of wasteful memory operations on DaCapo 2006, Dacapo-9.12-MR1-bach, and ScalaBench benchmark suites at the sampling periods of 500K, 1M, 5M, and 10M. The error bars are for different sampling periods.

existing tools.

4.5.1 Fraction of Wasteful Memory Operations

Figure 4.4 shows the fraction of dead stores, silent stores, and silent loads on DaCapo 2006, Dacapo-9.12-MR1-bach, and ScalaBench benchmark suites at the sampling periods of 500K, 1M, 5M, and 10M. The following two takeaways are obvious:

- The inefficiencies, i.e., 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 4.5. 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 reservoir sampling. We check the top five inefficiency pairs and their percentage contributions and find negligible variance across different sampling periods and different numbers of debug registers.

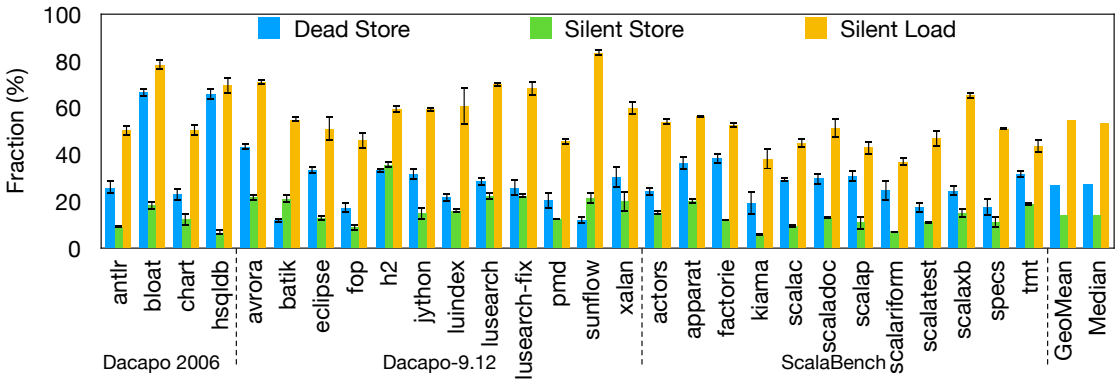


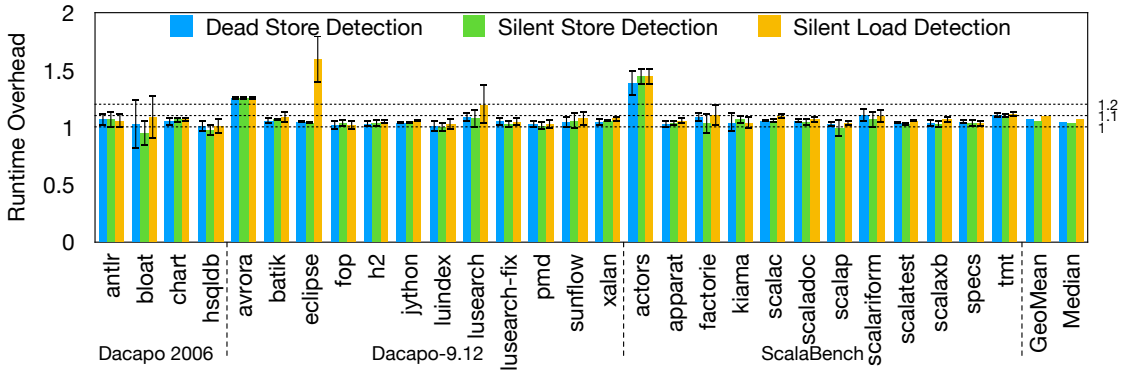
Figure 4.5: Fraction of wasteful memory operations on DaCapo 2006, Dacapo-9.12-MR1-bach, and ScalaBench benchmark suites by using different numbers of debug registers at the 5M sampling period. The error bars are for different number of debug registers.

Table 4.1: Geometric mean and median of runtime and memory overhead (×) of JXPerf 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).

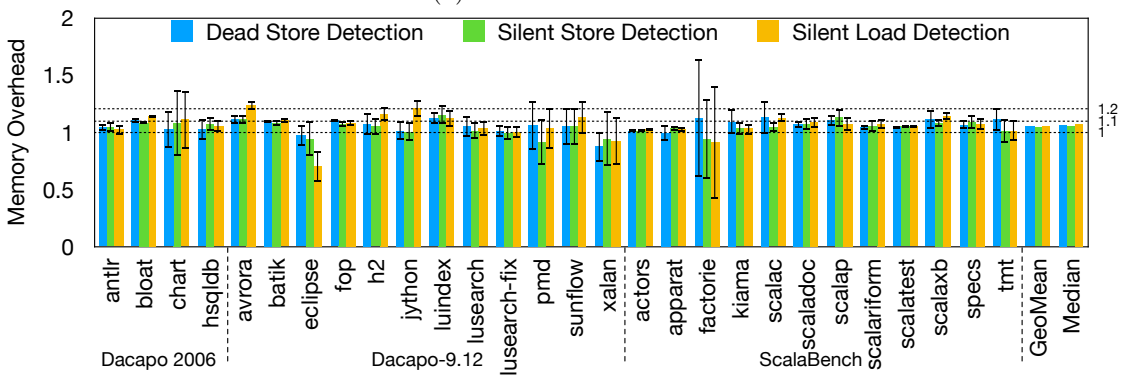
GeoMean/Median		Sampling period			
		500K	1M	5M	10M
DS detection	Runtime overhead	1.18/1.18	1.11/1.1	1.07/1.05	1.04/1.03
	Memory overhead	1.06/1.08	1.06/1.08	1.05/1.06	1.04/1.05
SS detection	Runtime overhead	1.16/1.14	1.1/1.1	1.06/1.04	1.05/1.04
	Memory overhead	1.06/1.07	1.06/1.06	1.04/1.05	1.05/1.05
SL detection	Runtime overhead	1.35/1.34	1.24/1.21	1.1/1.07	1.07/1.05
	Memory overhead	1.19/1.17	1.11/1.1	1.05/1.07	1.06/1.06

4.5.2 Overhead

Runtime (memory) overhead is measured as the ratio of the runtime (peak memory usage) of a benchmark with JXPerf enabled to the runtime (peak memory usage) of its native execution. Table 4.1 shows the geometric mean and median of runtime and memory overhead at different sampling periods. As the sampling period increases (i.e., the sampling rate decreases), the overhead drops as expected. We empirically find that the 5M sampling period yields a good tradeoff between overhead and accuracy, which typically incurs 7%



(a) Runtime overhead.



(b) Memory overhead.

Figure 4.6: JXPerf's runtime and memory overhead (×) at the 5M sampling period on DaCapo 2006, Dacapo-9.12-MR1-bach and ScalaBench benchmark suites.

runtime overhead and 7% memory overhead.

Figure 4.6 further quantifies the overhead of JXPerf 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 program execution. Moreover, JXPerf 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 the others because it executes more load operations and has more methods of large size that require JXPerf to take more efforts to correct the off-by-one error at each watchpoint trap. Due to the non-deterministic behavior of GC, the peak memory usage for a couple of benchmarks with JXPerf enabled is less than the native run (e.g., `eclipse`, `xalan`) or varies significantly

among different runs (e.g., `factorie`).

4.5.3 Effectiveness

We investigate the performance bugs reported by several state-of-the-art tools such as Toddler [100], Clarity [106], Glider [35], and LDoctor [121]. Among them, the developers of Toddler and Glider share their bug datasets and test cases that expose the bugs online [4, 94]. Therefore, we validate the effectiveness of JXPerf by checking whether the bugs reported by Toddler and Glider can also be identified by JXPerf. Toddler and Glider are both built atop Soot [135] 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 4.2: Effectiveness of JXPerf. Toddler and Glider report 33 and 46 performance bugs from eight real-world applications, among which JXPerf 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**

Table 4.2 shows the comparison results. Toddler reports 33 bugs (we exclude the bugs whose source code or test cases are no longer available), among which JXPerf misses only two bugs: `Apache Ant#53637` and `Apache Collections#409`. Glider reports 46 bugs, among which JXPerf misses only two bugs: `Apache Ant#53637` and `JFreeChart` (unknown bug ID). Take Apache Collections#588, one of the reported bugs, as an example

```
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 4.3: 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.

to illustrate how JXPerf identifies it. Listing 4.3 shows the 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 `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 $\mathcal{O}(1)$ 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 load 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.

Table 4.3: Overview of performance improvement guided by JXPerf.

	Program	Inefficiency			Optimization	
		Code	Type	Root cause	Approach	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 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 JXPerf.
SS: silent store, DS: dead store, SL: silent load.

4.6 Case Studies

In addition to confirming the performance bugs reported by the existing tools, we apply JXPerf on more benchmark suites — DaCapo 2006 [15], SPECjvm2008 [125], NPB-3.0 [12] and Grande-2.0 [103], and real-world applications — SableCC-3.7 [44], FindBugs-3.0.1 [111], and JFreeChart-1.0.19 [48] to identify varieties of inefficiencies.

Table 4.3 summarizes the newly found performance bugs via JXPerf as well as previously found ones but with different insights provided by JXPerf. All the programs are built with Oracle JDK11 except Dacapo 2006 bloat and FindBugs-3.0.1, which are built with Oracle JDK8. We quantify the performance improvement in execution time except for 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 4.3.

4.6.1 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

```
--------------------------------------------------------------------------------
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 4.7: A silent load pair with full calling contexts reported by JXPerf in SPECjvm2008 scimark.fft.

in Listing 4.1, which account for 27% of the total silent loads. They both suffer from the same performance issue: poor code generation detailed in Section 4.1.1. We take lines 153 and 155 as an example to illustrate our optimization, of which the culprit calling contexts are shown in Figure 4.7. 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 a single iteration. We also eliminate the silent loads between lines 154 and 156 using the same approach. They together eliminate 15% of the memory loads and yield a $(1.13 \pm 0.02)\times$ speedup for the entire program.

```
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 4.4: 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 4.8: The assembly code (at&t style) of lines 5, 7 and 12 in Listing 4.4.

4.6.2 Grande-2.0 Euler: Dead Stores

Euler [103] employs a structured mesh to solve the time-dependent Euler equations. JXPerf identifies 46% of memory stores are dead. One of the top dead store pairs is associated with the variable `temp2.a` at lines 5 and 12 in Listing 4.4, which appears in a loop nest (not shown). By inspecting the JITted assembly code shown in Figure 4.8, 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 [152].

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

4.6.3 SableCC-3.7: Silent Loads

SableCC [44] is a lexer and parser framework for compilers and interpreters. JXPerf profiles the latest stable version of SableCC using the JDK7 grammar file as the input. JXPerf identifies that silent loads account for 94% of the 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 4.9. The silent loads occur at line 568 in `TreeMap.java`, whose source code is shown in Listing 4.5. `TreeMap` is a red-black-tree-based map where a `put` operation requires $\mathcal{O}(\log n)$ 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 re-loaded to compare with new elements being inserted in different invocation instances of `put()`, which shows up as silent loads.

By consulting the SableCC developers, we choose an alternative data structure. We substitute `LinkedHashMap` for `TreeMap` because (1) the linked list preserves ordering from one execution to another and (2) the hash table offers $\mathcal{O}(1)$ time complexity and significantly reduces the number of loads as well as the fraction of silent loads. We employ this transformation in five classes: `LR0ItemSet`, `LR1ItemSet`, `LR0Collection`,

```
-----------------------------------------------------------------------
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 4.9: A silent load pair reported by JXPerf in SableCC-3.7.

`LR1Collection`, and `Grammar`. This optimization reduces the memory loads by 43% and delivers a (3.08±0.32)× speedup to the entire program.

```
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 4.5: Method put() of the JDK TreeMap class. A put operation requires $\mathcal{O}(\log n)$ comparisons to insert an element.

4.6.4 NPB-3.0 IS: Silent Stores

IS [12] sorts integers using the bucket sort. With the class B input and four threads, JXPerf pinpoints that 70% of memory stores are silent, of which more than 50% are associated with method `pow()` at lines 3-6 in Listing 4.6. We notice method `randlc()` is invoked in a hot loop (not shown) and the arguments passed to `pow()` are loop invariants. Across loop iterations, `pow()` pushes the same parameters on the same stack locations, which shows up as silent stores.

To eliminate such wasteful operations, we hoist the four calls to `pow()` out of `randlc()`

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

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

4.6.5 Dacapo 2006 Bloat: Dead Stores

Bloat [15] is a toolkit for analyzing and optimizing Java byte code. With the large input, JXPerf reports 78% dead stores. More than 30% of the dead stores are attributed to the call site of method `addAll()` at lines 4 and 5 in Listing 4.7, 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 that the root cause of such dead stores is related to the field `current` of the JDK `HashMap` class, as shown in Listing 4.8. Method `addAll()` frequently invokes the method `nextNode()` of the `HashMap` class in a loop (not shown). In each iteration, the field `current` is overwritten with a 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 the memory stores and delivers a (1.35±0.05)× speedup to the entire program.

Yang *et al.* [151] also identify the same optimization opportunity via the high-level container usage analysis, which is different from JXPerf's binary-level inefficiency analysis.

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

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

4.6.6 FindBugs-3.0.1: Dead Stores

FindBugs [111] is a static analysis tool for detecting security and performance bugs. We profile it using the JDK rt.jar as the input. JXPerf reports 47% dead stores. One of the top dead store pairs is attributed to the instance variable `ArrayList slotList` at lines 9 and 11 in Listing 4.9. With an investigation into the implementation of the JDK `ArrayList` class, we find that the method `clear()` assigns the `null` value to all elements in `slotList` and sets its size to zero instead of reclaiming the occupied space. When an element is inserted into `slotList` later by invoking the method `add()`, the `null` value

at the given location of `slotList`, without any usage, is overwritten, which shows up as dead stores.

We redesign the code to eliminate the dead stores, as shown in Listing 4.10. We first compare the size of ArrayList `slotList`, say a, with the size of `ArrayList other.slotList`, say b, to obtain the size of the smaller, say min. We then replace the first min elements in `slotList` with the first min elements in `other.slotList` (line 6) by invoking method `set()`. Finally, if $a > min$, we invoke `clear()` to clear only the remaining elements in `slotList` (line 7); otherwise, we invoke `add()` to append the remaining elements in `other.slotList` to `slotList` (line 10). With this optimization, the memory stores are reduced by 6% and the entire program gains a (1.02±0.01)× speedup.

```
1 private final ArrayList<ValueType> slotList;
2 ...
3 public void copyFrom(Frame<ValueType> other) {
4   int size = slotList.size();
5   if (size == other.slotList.size()) {
6     for (int i = 0; i < size; i++)
7       slotList.set(i, other.slotList.get(i));
8   } else {
9 ▶   slotList.clear();
10    for (ValueType v: other.slotList)
11 ▶    slotList.add(v);
12  }
13  ...
14 }
```

Listing 4.9: Dead stores in FindBugs-3.0.1. Inefficiently-used `ArrayList` leads to dead stores.

```
1 public void copyFrom(Frame<ValueType> other) {
2   int a = slotList.size();
3   int b = other.slotList.size();
4   int min = a > b ? b : a;
5   for (int i = 0; i < min; i++)
6 ▶   slotList.set(i, other.slotList.get(i));
7 ▶ if (a > min) slotList.subList(b,a).clear();
8   else
9     for (int i = a; i < b; i++)
10 ▶    slotList.add(other.slotList.get(i));
11 }
```

Listing 4.10: Optimizing the code in Listing 4.9 to eliminate dead stores.

4.6.7 JFreeChart-1.0.19: Silent Loads

JFreeChart [48] is a chart library. JXPerf reports 90% of memory loads are silent on profiling the built-in test case `SegmentedTimelineTest` and 30% of silent loads are attributed to method `getExceptionSegmentCount()`, as shown in Listing 4.11. It performs a linear search (line 7) over `ArrayList exceptionSegments` to count the number of segments that intersect a given segment [`fromMillisecond`, `toMillisecond`]. This linear search is called many times in a loop to become the performance bottleneck. The symptom of such inefficiency is silent loads, which is caused by the repeated loads of immutable `ArrayList` elements in different invocation instances of `getExceptionSegmentCount()`.

We notice that segments in `exceptionSegments` are stored in ascending order, that is, the end point of the segment `exceptionSegments.get(i)` $<$ the start point of the segment `exceptionSegments.get(j)` *iff* i $<$ j. Therefore, there is no need to traverse the remaining segments in `exceptionSegments` if the start point of the current segment is already greater than `toMillisecond`. With this optimization, we reduce the memory loads by 23% and the entire program achieves a (1.64±0.04)× speedup.

Nistor *et al.* [100] also identify the same performance issue with Toddler. However, their optimization [101] guided by Toddler benefits the program only in two extreme situations: `toMillisecond` $<$ the start point of the first segment in `exceptionSegments` or `fromMillisecond` $>$ the end point of the last segment in `exceptionSegments`.

```
1 private List exceptionSegments = new ArrayList();
2 ...
3 public long getExceptionSegmentCount(long fromMillisecond, long toMillisecond) {
4   int n = 0;
5   for (Iterator iter = this.exceptionSegments.iterator(); iter.hasNext();) {
6     Segment segment = (Segment)iter.next();
7 ▶   Segment intersection = segment.intersect(fromMillisecond, toMillisecond);
8     if (intersection != null) {
9       n += intersection.getSegmentCount();
10    }
11  }
12  return (n);
13 }
```

Listing 4.11: Silent loads in JFreeChart-1.0.19. Immutable `ArrayList` elements are repeatedly loaded from memory across invocation instances of method `getExceptionSegmentCount()`.

4.7 Summary

In this chapter, we present JXPerf, a Java profiler that pinpoints performance inefficiencies arising from wasteful memory operations. JXPerf samples PMUs for addresses accessed by a program and uses debug registers to monitor these addresses. 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. JXPerf runs on off-the-shelf JVM, OS, and CPU, works on unmodified Java applications, and introduces only 7% runtime and memory overhead. Guided by JXPerf, we are able to optimize several benchmarks and real-world applications, yielding significant speedups.

Chapter 5

Pinpointing Performance Inefficiencies via Lightweight Variance Profiling

5.1 Introduction

Software developers primarily think of code in terms of functions (a.k.a. procedures) which form mental boundaries of functionality. It is natural that when developers investigate performance problems, they often want to see execution metrics at function level granularity. Almost all performance tools facilitate function level attribution; in fact, most tools offer finer-grained attribution such as loops or statements with call path attribution. A recent line of work has investigated procedure instance level variance as a major cause for performance problems such as long-tail latency [54, 53, 63, 58, 73] in enterprise cloud systems.

This chapter targets the procedure instance level execution variance in the high-performance computing (HPC) domain. Variance is a concern in the HPC domain as well, as we show with a motivating example in Section 5.1.1 and several case studies in our evaluation section. A prerequisite of profiling for variance among procedure instances

is the ability to place monitoring calipers around procedure entry and exit. This allows comparing the metrics from two or more invocation instances of the same procedure within the same execution.

Instrumentation-based tools [84, 59, 46, 16, 98, 104, 119] can insert calipers around functions to identify execution variance; however, they introduce high overhead. Sampling-based tools [78, 52, 3, 119, 34, 2, 150], on the other hand, use the interrupt-based mechanism supported by PMUs or OS timers, attribute samples to code regions, and highlight hotspots based on the number of samples taken in the same code region. Expecting a PMU sample to be delivered precisely at the entry of a procedure instance and the immediate next sample to be delivered precisely at the exit of that procedure instance is wishful thinking but impractical for almost any PMU- or timer-based sampling tool. Each sample is a point in time, one sample cannot be compared with another sample quantitatively. Hence, identifying execution variance of the same procedure across different invocation instances is seemingly impossible[1]. There is little variance between two samples since each one is delivered after the same number of preconfigured events. Furthermore, variance among procedure instances is not statistically significant in a sampling-based profiler if the sampling interval is larger than the execution time of the procedure itself. *In summary, sampling-based tools, until now, have not been able to synchronize samples with procedure boundaries.*

We address the aforementioned problem in FVSampler, a lightweight sampling-based variance profiler with the ability to show procedure instance level execution variance. FVSampler employs PMUs to sample function call (entry) and then uses debugs registers to intercept the return (exit) from the same function invocation and measures metrics between these two points. The metrics can be any of the supported PMU events, e.g., CPU cycles, cache misses, energy consumption, to name a few. The key differentiating

[1]One may be able to approximately infer procedure boundaries by looking at consecutive samples taken in the same procedure, however, this method is inaccurate for a small procedure called in a loop when consecutive samples across multiple invocations of the same procedure are not interleaved by a sample in another procedure.

aspect of FVSampler when compared to prior work is its ability to intercept function call and return with no instrumentation (source or binary) and prior knowledge of a program, which makes it useful in production. We evaluate FVSampler with several parallel applications and demonstrate its effectiveness in pinpointing execution variance.

In the rest of this section, we first describe a motivating example, showing that identifying execution variance in HPC code bases yields unique optimization opportunities. We then summarize the contributions of this work.

5.1.1 Motivating Example

NERSC-8 GTC [97], a particle-in-cell code, is used for Gyrokinetic Particle Simulation of Turbulent Transport in Burning Plasmas. A previous study [86] on GTC shows that cache misses in different invocation instances of the procedure that accesses an array of particles in sequential order varies significantly and increases as the execution progresses. With the source code analysis, we notice that at the program start, all particles are stored in cell order, which exactly matches access order, as shown in Figure 5.1a. However, as the program continues, particles move from one cell to another, resulting in the mismatch between access order and storage order (loss of data locality), as shown in Figure 5.1b and 5.1c. Periodically sorting particles in cell order can avoid the loss of data locality and improve the program performance by more than 20%. However, no existing sampling-based tools can identify procedure instance execution variance since they cannot distinguish whether two samples from the same procedure belong to the same instance of that procedure. As a result, they offer little help in optimizing this problematic procedure in GTC. Instrumentation-based tools can show procedure instance execution variance by instrumenting function call and return albeit the overhead is quite high. For example, when we employ Intel Pin to capture each procedure instance in GTC by instrumenting call and return instructions, a 5× runtime overhead is introduced and even worse an 8× runtime overhead is introduced with call path collection enabled. Compile-time instrumentation can result in lower overhead but will not be able to instrument the library code.

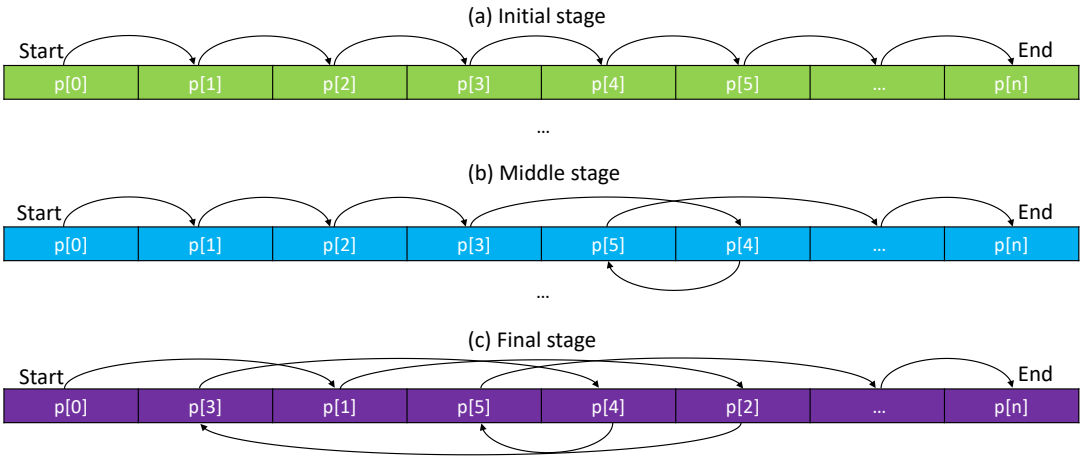


Figure 5.1: Access order and storage order of an array of particles ($p[]$) in GTC. (a) At the program start, particles are stored in cell order, which exactly matches access order. (b) and (c) As the execution progresses, particles move from one cell to another, resulting in the mismatch between access order and storage order.

5.1.2 Contribution Summary

In this work, we make the following contributions:

- Develop a technique to overcome a critical missing piece in sampling-based tools — synchronize samples with procedure boundaries to monitor whole procedure instances.
- Develop a lightweight sampling-based variance profiler — FVSampler — that combines PMUs and debug registers available in commodity CPUs to quantify variance across different invocations of the same function without requiring code instrumentation.
- Address the challenges raising due to combining the usage of PMUs and the limited number of debug registers.
- Show that FVSampler monitors fully optimized, unmodified binary executables and provides rich information to guide code optimization, such as calling contexts, variance metrics and their distributions, and source code attribution.
- Demonstrate the effectiveness of FVSampler by optimizing several parallel applications under the guidance of FVSampler, yielding significant speedups.

5.2 Related Work

5.2.1 Tracing Tools

HPCToolkit [3], perf [78], gprof [52], CrayPAT [34], Oracle Solaris Studio [107], OpenSpeedShop [116], and PGPROF [134] use interrupt-based sampling techniques supported by PMUs or OS timers to sample performance events and present them in chronological order. Unlike FVSampler, these tools do not capture function entry and exit to pinpoint function execution variance. TAU [119], Scalasca [46], DynamoRio [16], Valgrind [98], and Dyninst [104] show function execution variance via exhaustive or selective code instrumentation. Compared to the exhaustive instrumentation, FVSampler incurs much lower overhead in both runtime and memory; compared to the selective instrumentation, which needs to know the interesting functions for study, FVSampler does not require any prior knowledge of the monitored program.

5.2.2 Variance Diagnosis Tools

X-Ray [11] pinpoints performance inefficiencies by employing dynamic binary instrumentation to identify basic block level performance variance. Spectroscope [115] diagnoses performance changes in distributed systems by comparing request flows between two time periods (the period before the change and the period after the change). Yoon *et al.* [154] combine outlier detection and causality analysis to detect performance anomalies on individual transactions in online transaction processing systems. VarianceFinder [114] identifies the performance variance of requests under the same call path. Unlike these approaches, FVSampler focuses on identifying function-level variance.

Szebenyi *et al.* [130] use instrumentation to intercept MPI routines and use sampling to profile the remaining code during program execution. Unlike it, FVSampler only uses sampling to profile function invocation instances and does not distinguish libraries from the main executable. Any function called via a call instruction is a potential candidate to be monitored. VProfiler [59], an instrumentation-based tool, also identifies function-

level variance. However, users have to manually annotate code regions of interest before applying VProfiler to the target program. Moreover, VProfiler only identifies latency variance. In contrast, FVSampler is able to identify the variance of any PMU event, such as CPU cycles and cache misses.

To the best of our knowledge, FVSampler is the first nonintrusive sampling-based tool to study the function level variance of HPC workloads.

5.2.3 Software-based Return Address Interception

Kasikci *et al.* [69] trace cold code by dynamically rewriting the first instruction of every basic block with the int 3 breakpoint instruction, which causes a trap. This approach can be extended to rewrite return instructions to intercept function exits. However, such binary rewriting does not offer per-thread breakpoints and maintaining local breakpoints with code caches incurs high overhead. Arnold and Sweeney [10] perform call stack unwinding by replacing the function return address with a trampoline (the address of a handcrafted code snippet). When the modified function returns, the control is first transferred to the trampoline and then transferred back to the program. This software approach is also able to intercept the return from the same function invocation. Unlike these approaches, FVSampler uses hardware debug registers to intercept the return from the same function invocation and targets a completely different problem — variance profiling.

5.3 Methodology

PMUs provide precise events to sample call and return instructions, however, that is not sufficient — PMU samples cannot be configured to deliver one sample at the function entry and another at the return from the same function instance. Our solution is to use PMUs to sample only the call instructions and use debug registers to intercept the returns from the functions matching the sampled call instructions.

The point where the PMU delivers an interrupt is at the function entry, that is, right

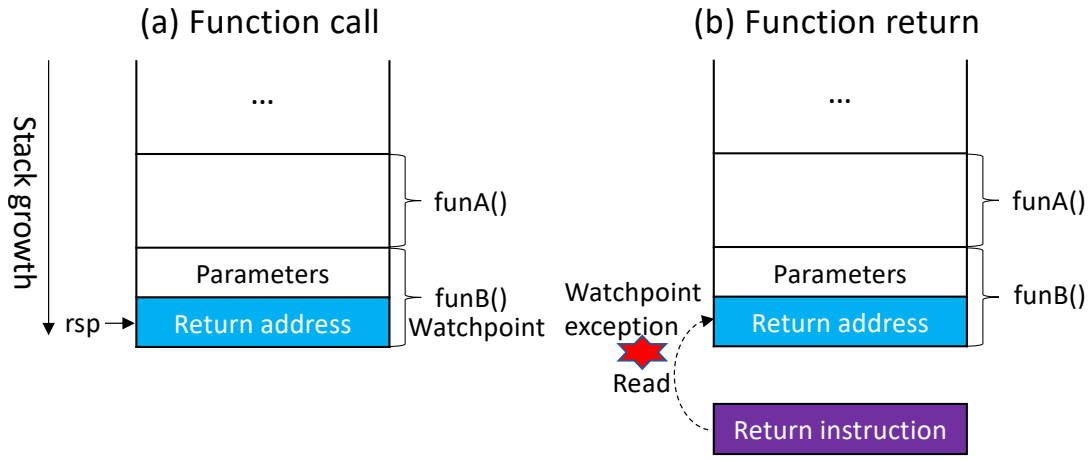


Figure 5.2: Actions on function call and return. (a) The call instruction in `funA()` (the caller) pushes the parameters and return address of `funB()` (the callee) on the stack; after the call instruction execution, the return address is on the top of the stack. We set a watchpoint at the stack location (marked in blue) that holds the return address. (b) The return instruction in `funB()` fetches the return address from the stack, which triggers a watchpoint trap.

after the call instruction execution in the caller. At this point, the stack pointer (register `rsp` in x86) points to the top of the stack (M[`rsp`]), which holds the return address for the caller to continue (Figure 5.2a). The callee accesses this return address stored on the stack just when it is about to return. We can intercept the return from the callee by protecting the access to this memory location (M[`rsp`]). We use debug registers to protect the subsequent access to M[`rsp`]. When the callee fetches the return address from M[`rsp`], it triggers a watchpoint trap (`RW_TRAP`), as shown in Figure 5.2b. Furthermore, the signal handlers invoked at these two points (PMU sample at a call and watchpoint trap at the return) allow us to record the metrics of interest and the difference in metrics between these two points can be attributed to the function invocation instance. In summary, we can now synchronize samples with function entry and exit and since we rely on PMU samples, we have not introduced any source or binary instrumentation; statistically significant functions (i.e., functions with a high invocation frequency) appear in our samples with a high probability.

Before arriving at the final design, we explored two other strategies in capturing function exits. These two approaches used debug registers as breakpoints (trapping on instruction execution) instead of watchpoints (trapping on memory accesses). In the first approach, we used debug registers to directly monitor *return instructions*, e.g., `retq` in the body of the callee; the return instructions were obtained via an on-the-fly binary analysis technique. However, it is common that a function has many return instructions but this approach could only monitor four return instructions in a function body with the four available debug registers. This approach would fail to capture the exit of a function if the unmonitored return instruction is executed. In the second approach, we used debug registers to monitor the *return address* — the address of the instruction in the caller that is executed right after the callee returns. This approach, however, failed for recursive functions because different invocation instances of a recursively called function all share the same return address. Consequently, we arrived at the final, correct approach of monitoring the stack location holding the return address of a function invocation.

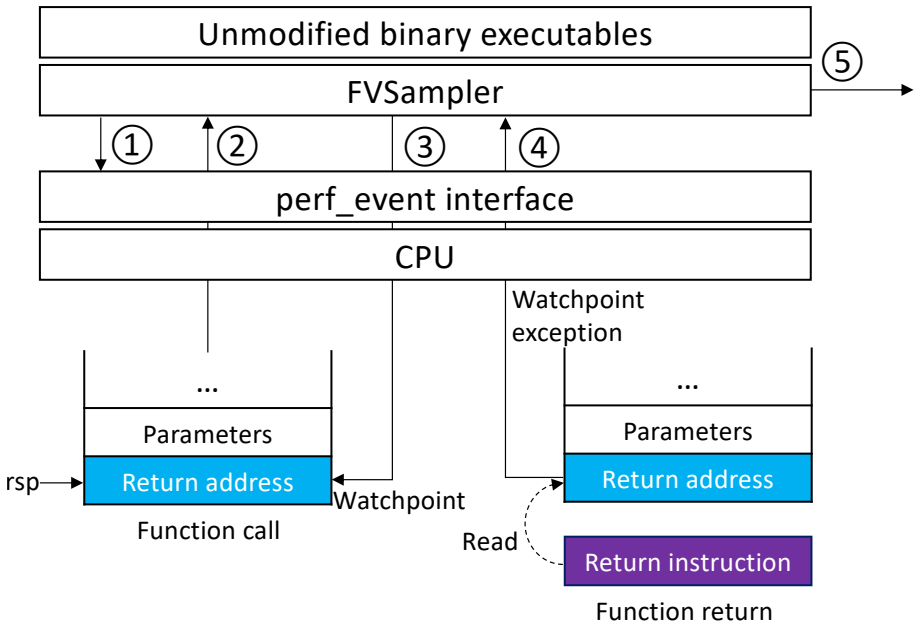


Figure 5.3: FVSampler's actions in steps to collect variance metrics.

5.4 Implementation

Figure 5.3 shows the implementation of how FVSampler uses PMUs to sample function call and uses debug registers to intercept the return from the same function instance. ① FVSampler subscribes to the precise PMU call event in sampling mode and configures debug registers as watchpoints for each thread via the `perf_event` API. FVSampler also configures other PMUs in counting mode to monitor user-specified performance events (e.g., CPU cycles, cache misses) as for variance metrics. ② When the PMU counter overflows on sampling function calls, it triggers an interrupt. FVSampler handles the interrupt signal, constructs the calling context at the interrupt via unwinding the execution call stack, and reads the user-specified PMU counters to obtain their current values (V_{call}). ③ FVSampler obtains the stack address $M[\texttt{rsp}]$ recorded in register `rsp`, sets a `RW_TRAP` watchpoint at $M[\texttt{rsp}]$, and resumes the program execution. ④ When the return instruction reads the return address from $M[\texttt{rsp}]$ [2], it triggers a watchpoint trap. FVSampler handles the trap signal and reads the user-specified PMU counters to obtain their current values (V_{ret}). ⑤ FVSampler records the difference between V_{ret} and V_{call}, which is the count of the user-specified performance events occurring in the current function instance. FVSampler disarms the watchpoint and resumes the program execution until the next PMU overflow. When the signal handler code is executing, we stop all PMU counters so that FVSampler's overhead is not counted towards the metrics collected for the function under investigation.

This scheme assumes flat function calls — we capture each function instance's call and return before monitoring the next. In reality, however, we need to handle the code with deep call chains.

[2] In a function's execution, only return instructions read the return address from $M[\texttt{rsp}]$ and no instructions write values to $M[\texttt{rsp}]$. We do not consider buffer overflows in the security domain.

5.4.1 Addressing Deep Call Chains

As we mentioned in Section 4.4.2, hardware exposes only a very small number of debug registers, which limits the number of PMU samples that can be monitored simultaneously. To better illustrate the problem, consider a call chain consisting of three functions: `main()`→`funA()`→`funB()`. Assume the PMU is able to sample both `funA()` and `funB()`, and there is only one available debug register. The first sample occurs when `funA()` is being called by `main()`, which results in setting a watchpoint at the stack address holding the return address of `funA()`. The second sample occurs when `funA()` is calling `funB()`. However, there is no room to monitor the stack address holding the return address of `funB()` since the previously set watchpoint is still active. With this scheme, in a system with N debug registers, at most N function instances can be monitored simultaneously.

FVSampler addresses this problem based on an observation: a callee always returns before its caller returns [3]. Thus, FVSampler maintains a stack $\mathcal{S}$ to save active stack addresses being monitored by watchpoints. We use the same call chain as an example to illustrate our idea, as shown in Figure 5.4. Upon the sample that captures the call instruction to `funA()`, FVSampler sets a watchpoint at the stack address holding the return address of `funA()` since a debug register is available, as shown in Figure 5.4a. Upon the next sample that captures the call instruction to `funB()`, FVSampler disarms the watchpoint, pushes the address that the watchpoint is monitoring for `funA()` on $\mathcal{S}$, and reconfigures the debug register to monitor the stack address holding the return address of `funB()`, as shown in Figure 5.4b. When `funB()` returns later, it triggers a watchpoint trap. FVSampler handles the trap as normal for variance metrics, disarms the watchpoint, pops the stack address holding the return address of `funA()` from $\mathcal{S}$, and reconfigures the watchpoint to monitor this stack address, as shown in Figure 5.4c.

With this scheme, only one debug register is needed to handle the deep call chain, which makes our technique widely applicable to both x86 and PowerPC architectures.

[3] `longjmp()` is an exception, which is discussed in Section 5.4.5.

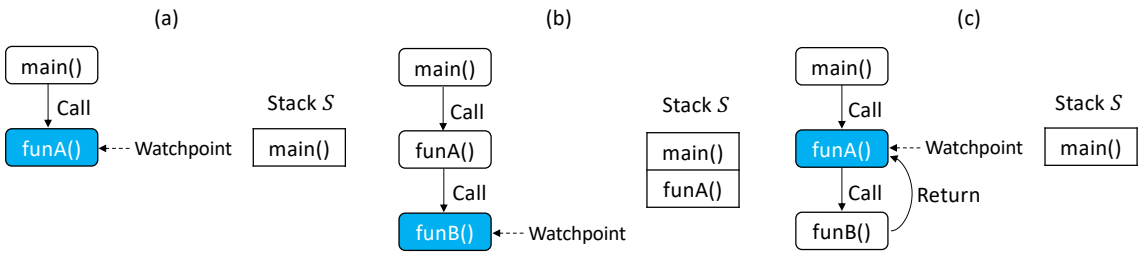


Figure 5.4: Using one debug register to monitor all sampled function instances. FVSampler maintains a stack $\mathcal{S}$ to save active stack addresses being monitored by watchpoints. (a) When `main()` is calling `funA()`, FVSampler sets a watchpoint at the stack address holding the return address of `funA()`. (b) When `funA()` is calling `funB()`, FVSampler pushes the address the watchpoint is monitoring for `funA()` on $\mathcal{S}$, disarms the watchpoint, and sets it at the stack address holding the return address of `funB()`. (c) When `funB()` is returning to `funA()`, FVSampler pops the address holding the return address of `funA()` from $\mathcal{S}$ and resets the watchpoint at it.

5.4.2 Obtaining the Calling Context of a Function Instance

To provide rich insights for developer actions, FVSampler need record the calling context where a function call occurs. Since a PMU interrupt happens immediately after the function call, the calling context of the interrupt is at the function entry. At the function return, FVSampler need not determine the calling context as it is the same as the one obtained at the function call. FVSampler constructs calling contexts with an on-the-fly binary analysis technique [132], which efficiently maintains calling contexts as a compact calling context tree [6] by merging common prefixes.

5.4.3 Obtaining Variance Metrics

FVSampler provides two options to present variance metrics. One is to plot the metrics collected from all the sampled instances for a given function in a given calling context. This plot provides the most straightforward view of variance and is able to expose variance patterns (e.g., the increase of cache misses in GTC described in Section 5.1.1) for optimization actions. The other is to provide a compact view, which computes the mean, standard deviation, and coefficient of variation across the metrics collected from the sam-

pled instances for any function in any calling context. Such a compact view can help users quickly locate the problematic functions for further investigation. To avoid recording metrics of every sampled function instance, we leverage Welford's online algorithm [145]. In this section, we briefly describe this algorithm; details about the rigorous proofs can be found in the related paper [141].

When the i^{th} sample of a function occurs, the mean ($\overline{V}_{\{1,...,i\}}$), standard deviation ($SD_{V_{\{1,...,i\}}}$), and coefficient of variation ($CV_{V_{\{1,...,i\}}}$) of the variance metric ($V_{\{1,...,i\}}$) across the first i samples are calculated by the following equations:

$$\begin{aligned} \overline{V}_{\{1,...,i\}} &= \frac{(i-1)\overline{V}_{\{1,...,i-1\}} + V_i}{i} \\ SD_{V_{\{1,...,i\}}} &= \sqrt{\frac{(V_i - \overline{V}_{\{1,...,i-1\}})(V_i - \overline{V}_{\{1,...,i\}}) + (i-2)SD^2_{V_{\{1,...,i-1\}}}}{i-1}} \\ CV_{V_{\{1,...,i\}}} &= \frac{SD_{V_{\{1,...,i\}}}}{\overline{V}_{\{1,...,i\}}} \end{aligned} \tag{5.1}$$

From these equations, we can see that computation on these metrics enjoys an incremental fashion, with no need to record all samples. In this work, we employ the coefficient of variation metric to quantify procedure instance execution variance.

5.4.4 Handling Parallelism

FVSampler works for MPI programs as it monitors each MPI process independently and also works for multi-threaded programs as PMUs and debug registers are virtualized by the OS for each thread. FVSampler does not handle user-level threading where a function call and its corresponding return are executed on two different OS threads. A solution to the user-level threading would require minimal support from the runtime — the user-level thread switching should save and restore the debug register state.

Table 5.1: Optimization decisions based on execution time and variance. Our optimization efforts are focused on functions with both high execution time and variance.

Execution time	Variance	Guidance
High	High	Actions should be taken to reduce variance for performance
High	Low	Performance is unrelated to variance
Low	High	Reducing variance yields little benefit to the entire program
Low	Low	No action on variance optimization

5.4.5 Handling `longjmp()`

`setjmp()`/`longjmp()` provide inter-procedure jumps, which deviates from the typical calling conventions. FVSampler intercepts them by overloading their calls. When the `longjmp()` executes, FVSampler disarms the active watchpoint and clears the watchpoint stack $\mathcal{S}$ because we do not know which stack frame the `longjmp()` will jump to.

5.4.6 Optimization Guidance

Our optimization decision on a function is based on its execution time and variance (i.e., coefficient of variation metrics), as shown in Table 5.1. Only functions with both high execution time and variance are worthy of efforts for further performance analysis. In all of our case studies, we investigate a function *iff* it accounts for more than 10% CPU cycles over the entire program and has larger than 20% *intra-thread variance* or 10% inter-thread variance. Once FVSampler pinpoints a problematic function, it plots the metrics collected from all its sampled instances in the timeline. The variance pattern can effectively guide unique code optimization.

5.4.7 Understanding the Limitation of Sampling

Like any sampling-based tool, FVSampler captures statistically significant functions (i.e., high invocation frequency) and misses out on some insignificant ones. It satisfies the needs

for studying variance because variance is meaningful only on functions with high invocation frequency. Seldom called functions (e.g., `main()`) are less interesting. Also, FVSampler misses out on functions that are not invoked via a call instruction, e.g., functions that are inlined or called via a tail call.

5.5 Overhead Evaluation

We evaluate FVSampler on a machine with two 18-core Intel Xeon E5-2699 v3 CPUs of 2.30GHz frequency running Linux 4.8.0. The machine has 128GB main memory. FVSampler subscribes to the precise PMU event `BR_INST_RETIRED.NEAR_CALL` to sample call instructions. Runtime overhead is measured as the ratio of the runtime of a benchmark monitored with FVSampler to the runtime of its native execution. Table 5.2 shows the runtime overhead of FVSampler on two HPC benchmark suites — NERSC-8 [97] and CORAL-2 [72] as well as five HPC benchmarks — LULESH-2 [68], Sweep3D [70], MASNUM-2.2 [112], Sequoia AMG2006 [71], and PARSEC-2.1 dedup [14]. All the programs are compiled with `gcc-5.4.1 -O3` except MASNUM-2.2 that is compiled with `icc-18.0.2 -O3`. The MPI programs are run with 36 processes and OpenMP programs are run with 36 threads, which are pinned to cores. We tune the sampling period to ensure that at least 30 samples are collected per second per thread. We use the PMU event `PERF_COUNT_HW_INSTRUCTIONS` in counting mode to count the number of instructions executed by each sampled function instance. We run each program five times and report the average runtime overhead. In Table 5.2, we can see that FVSampler typically incurs 6% runtime overhead. FVSampler can incur more overhead when profiling short-running programs due to the fixed overhead of setting up PMUs and debug registers. Table 5.3 shows per-sample overhead and per-watchpoint-trap overhead, respectively. We can see that FVSampler typically incurs 44 microseconds overhead per sample and 11 microseconds overhead per watchpoint trap. In addition, FVSampler incurs average 7MB memory overhead per thread in all these programs. Such low overhead makes

FVSampler appropriate for production runs.

Table 5.2: FVSampler's runtime overhead.

Benchmark		Language	Programming model	Native runtime (sec)	Overhead
NERSC-8	AMG	C	MPI + OpenMP	42.14	1.07 X
	GTC	Fortran	MPI + OpenMP	50.29	1.04 X
	MILC	C	MPI + OpenMP	81.63	1.05 X
	MiniFE	C++	MPI + OpenMP	53.17	1.08 X
	PSNAP	C	MPI	49.66	1.05 X
	SMB	C	MPI	113.43	1 X
	SNAP	Fortran	MPI + OpenMP	43.91	1.06 X
	STREAM	C/Fortran	MPI + OpenMP	35.33	1.06 X
CORAL-2	CLOMP	C	MPI + OpenMP	21.21	1.06 X
	MDTest	C	MPI	55.49	1.04 X
	PENNANT	C++	MPI + OpenMP	24.76	1.09 X
	Quicksilver	C++	MPI + OpenMP	27.61	1.08 X
LULESH-2		C++	MPI + OpenMP	36.59	1.08 X
Sweep3D		Fortran	MPI	34.78	1.06 X
MASNUM-2.2		Fortran	MPI	67.12	1.12 X
Sequoia AMG2006		C	MPI + OpenMP	44.13	1.05 X
PARSEC-2.1 dedup		C	Pthreads	20.42	1.11 X
Median		–	–	–	**1.06 X**
GeoMean		–	–	–	**1.06 X**

5.6 Case Studies

Table 5.4 summarizes the performance inefficiencies found by FVSampler via function-level execution variance analysis. All the programs are compiled with `gcc-5.4.1 -O3` and run with 36 threads except MASNUM-2.2 that is compiled with `icc-18.0.2 -O3` and run with 36 MPI processes. We quantify the performance improvement in execution time. It is worth noting that the inefficiencies found in MASNUM-2.2 and PARSEC-2.1 dedup by FVSampler are also found by LoadSpy() via redundancy analysis (see Sections 3.3.2

Table 5.3: FVSampler's per-sample and per-watchpoint-trap runtime overhead.

Benchmark		Overhead (microsecond)	
		Per sample	Per watchpoint trap
NERSC-8	AMG	24	8
	GTC	50	9
	MILC	44	11
	MiniFE	16	7
	PSNAP	13	7
	SMB	54	9
	SNAP	52	14
	STREAM	43	12
CORAL-2	CLOMP	154	27
	MDTest	71	12
	PENNANT	60	13
	Quicksilver	26	9
LULESH-2		59	11
Sweep3D		102	11
MASNUM-2.2		13	8
Sequoia AMG2006		24	13
PARSEC-2.1 dedup		14	8
Median		**44**	**11**
GeoMean		**45**	**11**

Table 5.4: Overview of performance improvement guided by FVSampler.

Program	Inefficiency		Optimization	
	Call site of the problematic function	Root cause	Approach	Speedup
MASNUM-2.2	propagat.inc (96, 103)	Linear search	Locality-friendly search	1.74×
Sequoia AMG2006	par_relax.c (1654, 1658)	Load imbalance	Reducing the granularity of parallel work	1.08×
NERSC-8 MiniFE	SparseMatrix_functions.hpp (465, 474)	Poor data structure	Replacing C++ `set` with `unordered_set`	1.96×
PARSEC-2.1 dedup	encoder.c (120, 226, 840, 891, 1003)	Poor hashing algorithm	Reducing hash collisions	1.08×

and 3.7.2). Thus, we only analyze the problematic functions detected in Sequoia AMG2006 and NERSC-8 MiniFE in this section.

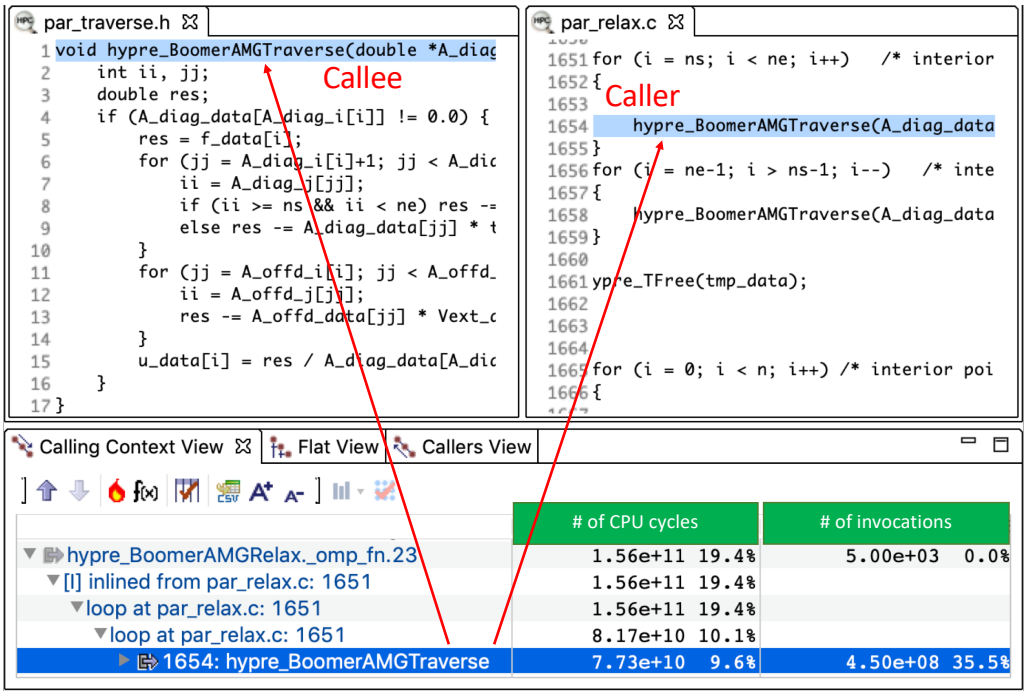


Figure 5.5: Inter-thread variance in Sequoia AMG2006. The number of instructions executed in `hypre_BoomerAMGTraverse()` varies significantly on different threads. `hypre_BoomerAMGTraverse()` takes different branches, resulting in execution variance.

5.6.1 Sequoia AMG2006

Sequoia AMG2006 [71] is a parallel algebraic multigrid solver for linear systems arising from problems on unstructured grids. We study an optimized version from Liu and Mellor-Crummey [83]. The code is written in C and parallelized with MPI and OpenMP. We run AMG2006 on a $30 \times 30 \times 30$ grid.

FVSampler reports that function `hypre_BoomerAMGTraverse()` consumes 10% of the total CPU cycles and accounts for 36% of the total function invocations, as shown in Figure 5.5. FVSampler further identifies that the number of instructions executed in `hypre_BoomerAMGTraverse()` varies significantly on different threads, as shown in Figure 5.6. With the source code study, we find that `hypre_BoomerAMGTraverse()` is invoked in a loop nest and the outer loop is a statically scheduled OpenMP loop, which divides the iterations into equal-sized chunks and assigns them to each thread. It appears that

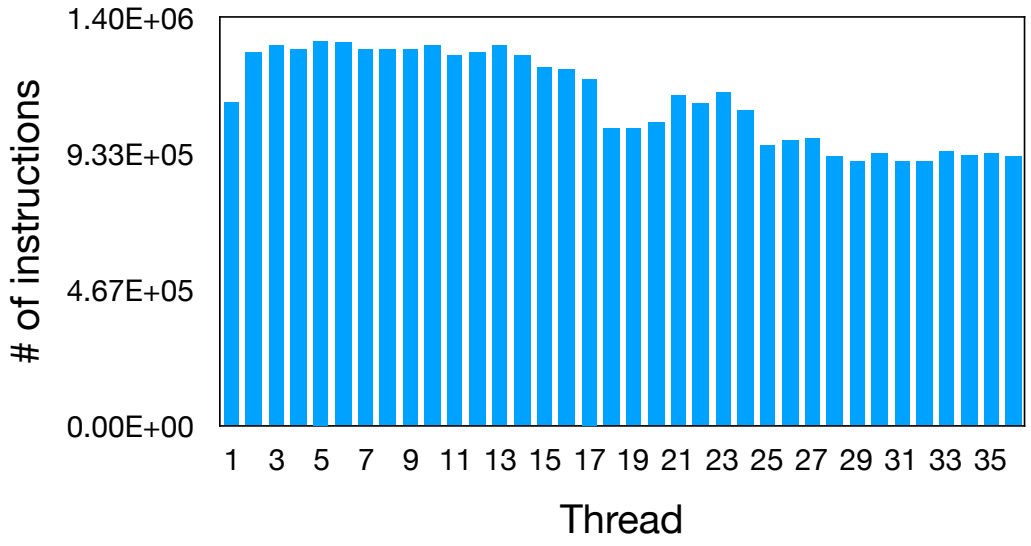


Figure 5.6: The number of instructions executed in `hypre_BoomerAMGTraverse()` on each thread.

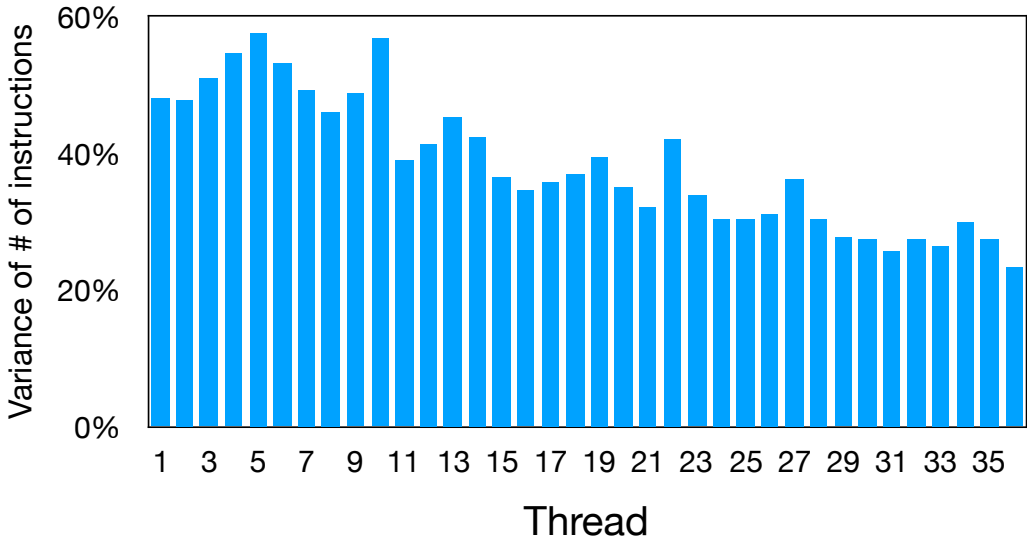


Figure 5.7: Variance of the number of instructions executed in different invocation instances of `hypre_BoomerAMGTraverse()` on each thread.

each thread has an equal amount of work because each chunk consists of an equal number of iterations.

When investigating the function body, we find `hypre_BoomerAMGTraverse()` employs branches, which results in execution variance depending on the taken branch. This exe-

cution variance across threads is a symptom of load imbalance. The most straightforward optimization is to redistribute the iterations to different threads. However, as Figure 5.7 shows, the execution variance of `hypre_BoomerAMGTraverse()` inside each thread is also high. Thus, it is difficult to assess the workload of each iteration and achieve load balance via static scheduling.

To mitigate the load imbalance, we reduce the chunk size to $\frac{1}{5}$ of the original chunk size and employ dynamic scheduling to balance the work across threads. With this optimization, the variance of the work (number of instructions) assigned to each thread is reduced from 14% to 3%. FVSampler also identifies other functions with the similar issue and guides the similar optimization. Finally, the entire program gains a 1.08× speedup.

```
1 void impose_dirichlet(..., const std::set<typename MatrixType::GlobalOrdinalType> &
      bc_rows) {
2   ...
3   for(size_t i = 0; i < A.rows.size(); ++i) {
4     ...
5     A.get_row_pointers(row, row_length, cols, coefs);
6     Scalar sum = 0;
7     for(size_t j = 0; j < row_length; ++j) {
8 ▶     if (bc_rows.find(cols[j]) != bc_rows.end()) {
9         sum += coefs[j];
10        coefs[j] = 0;
11      }
12    }
13  }
14 }
```

Listing 5.1: Call site of `std::set::find()` in NERSC-8 MiniFE, which accounts for 22% of the total CPU cycles.

5.6.2 NERSC-8 MiniFE

NERSC-8 MiniFE [97] employs the implicit finite-element method (FEM) to solve problems of engineering and mathematical physics. The code is written in C++ and parallelized with MPI and OpenMP. We apply FVSampler to evaluate it with the default input. Listing 5.1 highlights a hot function — `std::set::find()` at line 8, which accounts for 22% of the total CPU cycles and is executed only on the master thread. FVSampler further reports the number of instructions executed in different invocation instances of

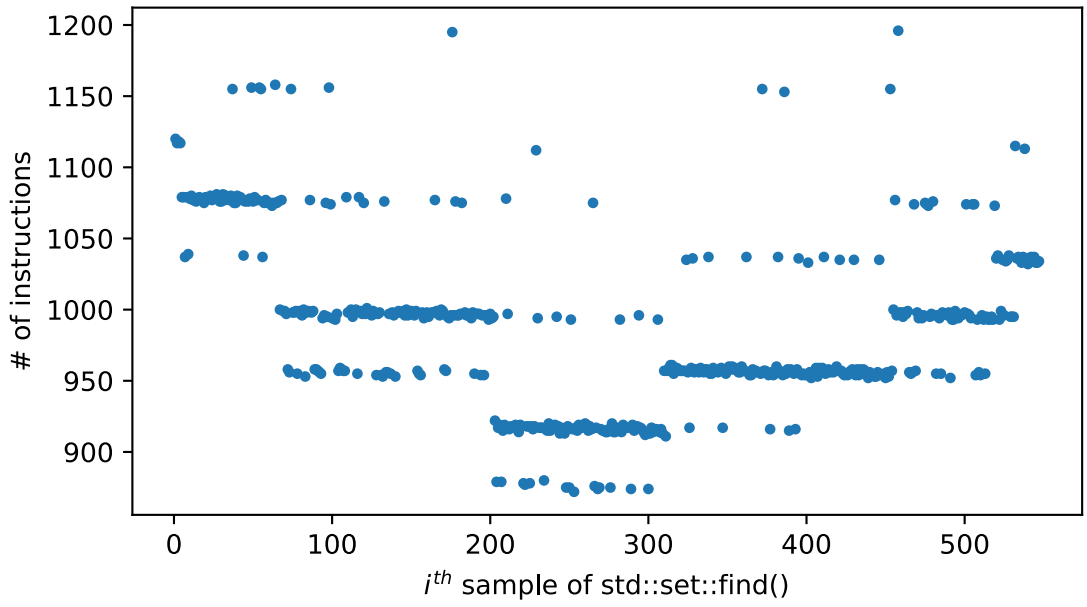


Figure 5.8: Variance of the number of instructions executed in different invocation instances of `std::set::find()` in NERSC-8 MiniFE. `std::set` is implemented as a red-black tree where a lookup operation requires one comparison in the best case and $\mathcal{O}(\log n)$ comparisons in the worst case. Consequently, the number of instructions executed in different invocation instances of `std::set::find()` varies from one to $\mathcal{O}(\log n)$.

`std::set::find()`, as shown in Figure 5.8. We can see that the work (number of instructions) performed by different instances varies significantly. The underlying implementation of `std::set` in C++ is a red-black tree where a lookup operation requires one comparison in the best case and $\mathcal{O}(\log n)$ comparisons in the worst case. Hence, the number of comparisons involved in `std::set::find()` varies from one to $\mathcal{O}(\log n)$, which shows up as the large execution variance.

To improve the lookup operation, we replace `std::set` with `std::unordered_set`. The latter uses a hash table to store elements, which requires expected $\mathcal{O}(1)$ comparisons to look up an element. With this optimization, the execution variance reduces significantly, yielding a 1.96× speedup for the entire program.

5.7 Summary

In this chapter, we present FVSampler, a lightweight variance profiler for HPC applications. FVSampler adopts PMUs to sample function call and uses debug registers to intercept the return from the sample function invocation instance to synchronize samples with function boundaries, which abandons heavyweight code instrumentation for variance analysis. FVSampler further collects the performance events, e.g., CPU cycles, instruction instances, cache misses, occurring in each sampled function instance and computes the variance metrics across different instances of the same function. FVSampler incurs low runtime and memory overhead, which makes it attractive for production. Guided by FVSampler, we are able to optimize several parallel applications, yielding up to a 1.96× speedup.

Chapter 6

Conclusions and Future Directions

This chapter concludes the dissertation and overviews future directions.

6.1 Conclusions

In the course of software development, wasteful memory operations and function execution variance are common indicators of performance inefficiencies arising from user inputs, suboptimal algorithms or data structures, and missed compiler optimizations. This dissertation demonstrates that one can identify such inefficiencies and obtain insightful optimization guidance by leveraging fine-grained code instrumentation or coarse-grained, hardware-assisted sampling.

In this dissertation, we draw the following conclusions:

6.1.1 Profiling for Wasteful Memory Operations

This dissertation defines three kinds of wasteful memory operations: dead stores, silent stores, and silent loads (a.k.a. redundant loads). A memory store is dead if it is followed by another store to the same memory location without an intervening load. A memory store is silent if the previous store performed on the same memory location stores the same value. A memory load is silent if the previous load performed on the same memory location loads the same value.

The microscopic observation of whole execution at a fine-granularity level (instructions and operands) breaks abstractions and helps recognize resource wastage that masquerades in complex code bases. We propose LoadSpy, a fine-grained profiler that studies wasteful memory operations in native languages. It automates important use cases to help developers investigate performance inefficiencies, opens up a new avenue for tuning software for high performance, and receives broad interests in industry (e.g., Uber) and national labs (e.g., Jefferson Lab). This work got accepted to ICSE'19 and won the ACM SIGSOFT Distinguished Paper Award.

In contrast to rich insights and high overhead of fine-grained profiling, coarse-grained profiling introduces low overhead by sacrificing performance insights it can offer. To make the best of both, we propose JXPerf, a profiler that studies wasteful memory operations in managed languages. It abandons exhaustive byte code instrumentation by combining PMUs with debug registers to sample and monitor memory accesses, which offers insightful optimization guidance as well as enjoys negligible overhead. JXPerf's ability to operate at the machine code level allows it to detect low-level code generation inefficiencies that are not apparent via byte code instrumentation. This work got accepted to ESEC/FSE'19.

6.1.2 Profiling for Function Execution Variance

Variance profiling is a vital means to identify performance anomalies, especially in latency-sensitive applications such as long-tail latency in cloud services. We propose FVSampler, the first nonintrusive profiler that studies function execution variance. It advances the state-of-the-art in sampling-based profilers by employing PMUs in conjunction with debug registers to deliver profiling samples precisely at function boundaries. Our case studies show that investigating function execution variance via FVSampler can easily bubble up an erratic function hidden due to the good average performance of its different invocation instances. This work got accepted to SC'19.

6.2 Future Directions

Performance profiling beyond x86 The Arm architecture prevails in the mobile market and has been making significant progress in the PC market. For instance, Apple Inc. is migrating from Intel processors to ARM processors for MacBook laptops. LoadSpy is built atop Intel Pin, which only works for the x86 architecture. We will extend LoadSpy to the ARM architecture by leveraging DynamoRio [16], which is a cross-platform (e.g., x86, AMD, ARM) binary instrumentation framework.

Performance profiling beyond Java One future direction is to extend JXPerf to other popular managed languages, such as Python and JavaScript, which recently employ JITters — PyPy [9] for Python and V8 [51] for JavaScript.

Variance profiling beyond functions In the future, we will investigate variance at a finer granularity (e.g., basic blocks) to pinpoint low-level code transformation variance and also at a coarser granularity (e.g., a series of functions for a semantic interval) to pinpoint high-level algorithmic or data structural variance.

Bibliography

[1] The DWARF Debugging Standard. http://www.dwarfstd.org, 2012.

[2] Intel VTune. http://software.intel.com/en-us/intel-vtune-amplifier-xe, 2018.

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

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

[5] Akamai Technologies, Inc. Akamai online retail performance report: Milliseconds are critical. http://www.akamai.com/uk/en/about/news/press/2017-press/akamai-releases-spring-2017-state-of-online-retail-performance-report.jsp, 2017.

[6] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI '97, pages 85–96, New York, NY, USA, 1997. ACM.

[7] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous Profiling: Where Have All the Cycles Gone? *ACM Trans. Comput. Syst.*, 15(4):357–390, November 1997.

[8] Apache Software Foundation. Apache avro. http://avro.apache.org, 2017. 21 February 2018.

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

[10] M. Arnold and P. F. Sweeney. Approximating the calling context tree via sampling. Technical Report, IBM, 1999.

[11] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 307–320, Hollywood, CA, 2012. USENIX.

[12] 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. The nas parallel benchmarks—summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, page 158–165, New York, NY, USA, 1991. Association for Computing Machinery.

[13] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the Ninth International Conference on Architectural Support*

for Programming Languages and Operating Systems, ASPLOS IX, pages 117–128, New York, NY, USA, 2000. ACM.

[14] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[15] 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. 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, pages 169–190, New York, NY, USA, 2006. ACM.

[16] Derek L. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Cambridge, MA, USA, 2004. AAI0807735.

[17] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Addison-Wesley Publishing Company, USA, 2nd edition, 2010.

[18] M. Burrows, Ú Erlingsson, S-T. A. Leung, M. T. Vandevoorde, C. A. Waldspurger, K. Walker, and W. E. Weihl. Efficient and Flexible Value Sampling. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, pages 160–167, New York, NY, USA, 2000. ACM.

[19] J. Adam Butts and Guri Sohi. Dynamic Dead-instruction Detection and Elimination. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 199–210, 2002.

[20] Brad Calder, Peter Feller, and Alan Eustace. Value profiling. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 30, pages 259–269, Washington, DC, USA, 1997. IEEE Computer Society.

[21] Brad Calder, Peter Feller, and Alan Eustace. Value Profiling and Optimization. *Journal of Instruction Level Parallelism*, 1, 1999.

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

[23] Milind Chabbi, Xu Liu, and John Mellor-Crummey. Call paths for pin tools. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 76:76–76:86, New York, NY, USA, 2014. ACM.

[24] Milind Chabbi and John Mellor-Crummey. Deadspy: A tool to pinpoint program inefficiencies. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 124–134, New York, NY, USA, 2012. ACM.

[25] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, Oct 2009.

[26] Eui-Young Chung, Luca Benini, and Giovanni De Micheli. Energy Efficient Source Code Transformation based on Value Profiling. In *PROC. INTERNATIONAL WORKSHOP ON COMPILERS AND OPERATING SYSTEMS FOR LOW POWER*, 2000.

[27] Keith Cooper, Jason Eckhardt, and Ken Kennedy. Redundancy elimination revisited. In *Proceedings of the 17th International Conference on Parallel architectures and compilation techniques*, pages 12–21, 2008.

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

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

[30] Oracle Corp. All-in-one java troubleshooting tool. http://visualvm.github.io, 2018.

[31] Oracle Corp. Jvmtm tool interface. http://docs.oracle.com/en/java/javase/11/docs/specs/jvmti.html, 2018.

[32] Steven J. Deitz, Bradford L. Chamberlain, and Lawrence Snyder. Eliminating Redundancies in Sum-of-product Array Computations. In *Proceedings of the 15th International Conference on Supercomputing*, ICS '01, pages 65–77, New York, NY, USA, 2001. ACM.

[33] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. 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, pages 607–622, New York, NY, USA, 2015. ACM.

[34] Luiz DeRose, Bill Homer, Dean Johnson, Steve Kaufmann, and Heidi Poxon. Cray performance analysis tools. In *Tools for High Performance Computing*, pages 191–199. Springer Berlin Heidelberg, 2008.

[35] Monika Dhok and Murali Krishna Ramanathan. Directed test generation to detect loop inefficiencies. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 895–907, New York, NY, USA, 2016. ACM.

[36] Yufei Ding, Lin Ning, Hui Guan, and Xipeng Shen. Generalizations of the theory and deployment of triangular inequality for compiler-based strength reduction. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 33–48, New York, NY, USA, 2017. ACM.

[37] Yufei Ding and Xipeng Shen. Glore: Generalized loop redundancy elimination upon ler-notation. *Proc. ACM Program. Lang.*, 1(OOPSLA):74:1–74:28, October 2017.

[38] Paul J. Drongowski. Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors. http://developer.amd.com/wordpress/media/2012/10/AMD_IBS_paper_EN.pdf, November 2007.

[39] Robert G. Edwards and Balint Joo. The chroma software system for lattice qcd. *Nucl. Phys. Proc. Suppl.*, 140:832, 2005.

[40] Ariel Eizenberg, Shiliang Hu, Gilles Pokam, and Joseph Devietti. 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, pages 251–265, New York, NY, USA, 2016. ACM.

[41] ej-technologies GmbH. The award-winning all-in-one java profiler. http://www.ej-technologies.com/products/jprofiler/overview.html, 2018.

[42] Peter T. Feller. *Value Profiling for Instructions and Memory Locations*. Master dissertation, 1998.

[43] Mary F. Fernández. Simple and Effective Link-time Optimization of Modula-3 Programs. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, pages 103–115, New York, NY, USA, 1995. ACM.

[44] Etienne Gagnon. The sable research group's compiler compiler. http://sablecc.org, 2018. May 2018.

[45] GCC Wiki. Graphite: Gimple Represented as Polyhedra. http://gcc.gnu.org/wiki/Graphite, 2015.

[46] Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The scalasca performance toolset architecture. *Concurr. Comput. : Pract. Exper.*, 22(6):702–719, April 2010.

[47] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 57–76, New York, NY, USA, 2007. ACM.

[48] David Gilbert. Welcome to jfree.org. http://www.jfree.org, 2017. November 2017.

[49] YourKit GmbH. The industry leader in .net & java profiling. http://www.yourkit.com, 2018.

[50] GNU. GNU Binutils. http://www.gnu.org/software/binutils, 2014. September 2014.

[51] Google Corp. Google v8 javascript engine. http://v8.dev, 2018.

[52] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A Call Graph Execution Profiler. In *Proceedings of the 1982 SIGPLAN Symposium*

on Compiler Construction, SIGPLAN '82, pages 120–126, New York, NY, USA, 1982. ACM.

[53] Md E. Haque, Yong hun Eom, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, and Kathryn S. McKinley. Few-to-many: Incremental parallelism for reducing tail latency in interactive services. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 161–175, New York, NY, USA, 2015. ACM.

[54] Md E. Haque, Yuxiong He, Sameh Elnikety, Thu D. Nguyen, Ricardo Bianchini, and Kathryn S. McKinley. Exploiting heterogeneity for tail latency and energy efficiency. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pages 625–638, New York, NY, USA, 2017. ACM.

[55] Paul Havlak. Nesting of reducible and irreducible loops. *ACM TOPLAS*, 19(4):557–567, 1997.

[56] Sylvain Henry, Hugo Bolloré, and Emmanuel Oseret. Towards the Generalization of Value Profiling for High-Performance Application Optimization. http://hsyl20.fr/home/files/papers/shenry_2015_vprof.pdf, 2015.

[57] Peter Hofer and Hanspeter Mössenböck. 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, pages 145–156, New York, NY, USA, 2014. ACM.

[58] Chang-Hong Hsu, Yunqi Zhang, Michael A. Laurenzano, David Meisner, Thomas Wenisch, Ronald G. Dreslinski, Jason Mars, and Lingjia Tang. Reining in long tails in warehouse-scale computers with quick voltage boosting using adrenaline. *ACM Trans. Comput. Syst.*, 35(1):2:1–2:33, March 2017.

[59] Jiamin Huang, Barzan Mozafari, and Thomas F. Wenisch. Statistical analysis of latency through semantic profiling. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 64–79, New York, NY, USA, 2017. ACM.

[60] Robert Hundt, Easwaran Raman, Martin Thuresson, and Neil Vachharajani. MAO – An Extensible Micro-architectural Optimizer. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 1–10, Washington, DC, USA, 2011. IEEE Computer Society.

[61] IBM Corp. Monitoring and Post Mortem. `http://developer.ibm.com/javasdk/tools`, 2018.

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

[63] Myeongjae Jeon, Yuxiong He, Hwanju Kim, Sameh Elnikety, Scott Rixner, and Alan L. Cox. Tpc: Target-driven parallelism combining prediction and correction to reduce tail latency in interactive services. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 129–141, New York, NY, USA, 2016. ACM.

[64] Mark Scott Johnson. 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, pages 140–148, New York, NY, USA, 1982. ACM.

[65] Teresa Johnson, Mehdi Amini, and Xinliang David Li. Thinlto: Scalable and incremental lto. In *Proceedings of the 2017 International Symposium on Code*

Generation and Optimization, CGO '17, pages 111–121, Piscataway, NJ, USA, 2017. IEEE Press.

[66] A. Joulin and T. Mikolov. Inferring Algorithmic Patterns with Stack-Augmented Recurrent Nets. *ArXiv e-prints*, March 2015.

[67] Takahiro Kamio and Hidehiko Masahura. A Value Profiler for Assisting Object-Oriented Program Specialization. In *Proceedings of Workshop on New Approaches to Software Construction*, 2004.

[68] I. Karlin, A. Bhatele, J. Keasler, B. L. Chamberlain, J. Cohen, Z. Devito, R. Haque, D. Laney, E. Luke, F. Wang, D. Richards, M. Schulz, and C. H. Still. Exploring traditional and emerging parallel programming models using a proxy application. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 919–932, May 2013.

[69] Baris Kasikci, Thomas Ball, George Candea, John Erickson, and Madanlal Musuvathi. Efficient tracing of cold code via bias-free sampling. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 243–254, Berkeley, CA, USA, 2014. USENIX Association.

[70] Lawrence Livermore National Laboratory. Sweep3D Benchmark Code. `http://www.llnl.gov/asci_benchmarks/asci/limited/sweep3d/asci_sweep3d.html`, 1995.

[71] Lawrence Livermore National Laboratory. ASC Sequoia Benchmark Codes. `http://asc.llnl.gov/sequoia/benchmarks`, 2013.

[72] Lawrence Livermore National Laboratory. CORAL-2 Benchmarks. `http://asc.llnl.gov/coral-2-benchmarks`, 2018.

[73] Z. Lai, Y. Cui, M. Li, Z. Li, N. Dai, and Y. Chen. Tailcutter: Wisely cutting tail latency in cloud cdn under cost constraints. In *IEEE INFOCOM 2016 - The*

35th Annual IEEE International Conference on Computer Communications, pages 1–9, April 2016.

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

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

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

[77] Linux. perf_event_open - Linux man page. http://linux.die.net/man/2/perf_event_open, 2012.

[78] Linux. Linux perf tool. http://perf.wiki.kernel.org/index.php/Main_Page, 2015.

[79] Mikko H. Lipasti and John Paul Shen. Exceeding the Dataflow Limit via Value Prediction. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 29, pages 226–237, Washington, DC, USA, 1996. IEEE Computer Society.

[80] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. Value Locality and Load Value Prediction. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, pages 138–147, New York, NY, USA, 1996. ACM.

[81] X. Liu and J. Mellor-Crummey. Pinpointing data locality bottlenecks with low overhead. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 183–193, April 2013.

[82] Xu Liu and John Mellor-Crummey. A data-centric profiler for parallel programs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 28:1–28:12, New York, NY, USA, 2013. ACM.

[83] Xu Liu and John Mellor-Crummey. A tool to analyze the performance of multithreaded programs on numa architectures. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 259–272, New York, NY, USA, 2014. ACM.

[84] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 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, pages 190–200, New York, NY, USA, 2005. ACM.

[85] YuLong Luo and GuangMing Tan. Optimizing Stencil Code via Locality of Computation. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, pages 477–478, 2014.

[86] G Marin, G Jin, and J Mellor-Crummey. Managing locality in grand challenge applications: a case study of the gyrokinetic toroidal code. *Journal of Physics: Conference Series*, 125:012087, jul 2008.

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

[88] R. E. McLear, D. M. Scheibelhut, and E. Tammaru. Guidelines for Creating a Debuggable Processor. In *Proceedings of the First International Symposium on*

Architectural Support for Programming Languages and Operating Systems, ASPLOS I, pages 100–106, New York, NY, USA, 1982. ACM.

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

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

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

[92] Chi Cao Minh, JaeWoong Chung, C. Kozyrakis, and K. Olukotun. Stamp: Stanford transactional applications for multi-processing. In *2008 IEEE International Symposium on Workload Characterization*, pages 35–46, Sept 2008.

[93] Ian Molyneaux. *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*. O'Reilly Media, Inc., 1st edition, 2009.

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

[95] Robert Muth, Scott A. Watterson, and Saumya K. Debray. Code Specialization Based on Value Profiles. In *Proceedings of the 7th International Symposium on Static Analysis*, SAS '00, pages 340–359, London, UK, UK, 2000. Springer-Verlag.

[96] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Evaluating the accuracy of java profilers. In *Proceedings of the 31st*

ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '10, pages 187–197, New York, NY, USA, 2010. ACM.

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

[98] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavy-weight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[99] Khanh Nguyen and Guoqing Xu. Cachetor: Detecting cacheable data to remove bloat. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 268–278, New York, NY, USA, 2013. ACM.

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

[101] Adrian Nistor. fast return for segmentedtimeline.getexceptionsegmentcount(). http://sourceforge.net/p/jfreechart/patches/300, 2012. November 2012.

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

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

[104] University of Maryland and University of Wisconsin. Putting the performance in high performance computing. http://www.dyninst.org, 2017.

[105] Taewook Oh, Hanjun Kim, Nick P. Johnson, Jae W. Lee, and David I. August. Practical Automatic Loop Specialization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 419–430, New York, NY, USA, 2013. ACM.

[106] Oswaldo Olivo, Isil Dillig, and Calvin Lin. 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, pages 369–378, New York, NY, USA, 2015. ACM.

[107] Oracle Corp. Oracle Solaris Studio. `http://www.oracle.com/technetwork/server-storage/solarisstudio/overview/index.html`, 2017.

[108] Rohan Padhye and Koushik Sen. Travioli: A dynamic analysis for detecting data-structure traversals. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 473–483, Piscataway, NJ, USA, 2017. IEEE Press.

[109] Andrei Pangin. Async-profiler. `http://github.com/jvm-profiling-tools/async-profiler`, 2018.

[110] Sebastian Pop, Albert Cohen, Cédric Bastoul, Sylvain Girbal, Georges-André Silber, and Nicolas Vasilache. Graphite: Polyhedral analyses and optimizations for GCC. In *Proceedings of the 2006 GCC Developers Summit*, page 2006, 2006.

[111] Bill Pugh and David Hovemeyer. Find bugs in java programs. `http://findbugs.sourceforge.net`, 2015. March 2015.

[112] Fangli Qiao, Wei Zhao, Xunqiang Yin, Xiaomeng Huang, Xin Liu, Qi Shu, Guansuo Wang, Zhenya Song, Xinfang Li, Haixing Liu, Guangwen Yang, and Yeli Yuan. A highly effective global surface wave numerical simulation with

ultra-high resolution. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 5:1–5:11, Piscataway, NJ, USA, 2016. IEEE Press.

[113] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global Value Numbers and Redundant Computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 12–27, 1988.

[114] Raja R. Sambasivan and Gregory R. Ganger. Automated diagnosis without predictability is a recipe for failure. In *Presented as part of the*. USENIX, Submitted.

[115] Raja R. Sambasivan, Alice X. Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R. Ganger. Diagnosing performance changes by comparing request flows. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 43–56, Berkeley, CA, USA, 2011. USENIX Association.

[116] Martin Schulz, Jim Galarowicz, Don Maghrak, William Hachfeld, David Montoya, and Scott Cranford. Openspeedshop: An open source infrastructure for parallel performance analysis. *Sci. Program.*, 16(2-3):105–121, April 2008.

[117] John S. Seng and Dean M. Tullsen. Architecture-level power optimization—what are the limits? *J. Instruction-Level Parallelism*, 7, 2005.

[118] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. 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, pages 657–676, New York, NY, USA, 2011. ACM.

[119] Sameer S. Shende and Allen D. Malony. The tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, May 2006.

[120] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. 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, pages 124–134, New York, NY, USA, 2011. ACM.

[121] Linhai Song and Shan Lu. Performance diagnosis for inefficient loops. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 370–380, Piscataway, NJ, USA, 2017. IEEE Press.

[122] Soeren Sonnenburg, Heiko Strathmann, Sergey Lisitsyn, Viktor Gal, Fernando J. Iglesias García, Wu Lin, Soumyajit De, Chiyuan Zhang, frx, tklein23, Evgeniy Andreev, JonasBehr, sploving, Parijat Mazumdar, Christian Widmer, Pan Deng / Zora, Saurabh Mahindre, Abhijeet Kislay, Kevin Hughes, Roman Votyakov, khalednasr, Sanuj Sharma, Alesis Novik, Abinash Panda, Evangelos Anagnostopoulos, Liang Pang, Alex Binder, serialhex, Esben Sørig, and Björn Esser. shogun-toolbox/shogun: Shogun 6.0.0 - Baba Nobuharu, April 2017.

[123] Konstantin Sorokin. Benchmark comparing various data serialization libraries (thrift, protobuf etc.) for C++. http://github.com/thekvs/cpp-serializers, 2014.

[124] SPEC Corp. SPEC CPU2006 benchmark suite. http://www.spec.org/cpu2006, 2007. November 2007.

[125] SPEC Corp. SPEC JVM2008 benchmark suite. http://www.spec.org/jvm2008, 2015. November 2015.

[126] SPEC Corp. SPEC OMP2012 benchmark suite. http://www.spec.org/omp2012, 2015. May 2015.

[127] SPEC Corp. SPEC CPU2017 benchmark suite. http://www.spec.org/cpu2017, 2017. November 2017.

[128] 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. IBM POWER7 performance modeling, verification, and evaluation. *IBM JRD*, 55(3):4:1–4:19, May-June 2011.

[129] Amitabh Srivastava and David W. Wall. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages*, 1(1):1–18, December 1992.

[130] Z. Szebenyi, T. Gamblin, M. Schulz, B. R. d. Supinski, F. Wolf, and B. J. N. Wylie. Reconciling sampling and direct instrumentation for unintrusive call-path profiling of mpi programs. In *2011 IEEE International Parallel Distributed Processing Symposium*, pages 640–651, May 2011.

[131] Nathan R. Tallent, Laksono Adhianto, and John M. Mellor-Crummey. Scalable identification of load imbalance in parallel executions using call path profiles. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[132] Nathan R. Tallent, John M. Mellor-Crummey, and Michael W. Fagan. Binary analysis for measurement and attribution of program performance. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 441–452, New York, NY, USA, 2009. ACM.

[133] Nathan R. Tallent, John M. Mellor-Crummey, and Allan Porterfield. Analyzing Lock Contention in Multithreaded Applications. *SIGPLAN Not.*, 45(5):269–280, January 2010.

[134] The Portland Group. PGPROF profiler guide parallel profiling for scientists and engineers. http://www.pgroup.com/doc/pgprofug.pdf, 2011.

[135] The Sable Research Group. A framework for analyzing and transforming Java and Android applications. `http://github.com/soot-oss/soot`, 2018.

[136] Linda Torczon and Keith Cooper. *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2011.

[137] Jeffrey S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, March 1985.

[138] Q. Wang, X. Liu, and M. Chabbi. Featherlight reuse-distance measurement. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 440–453, Los Alamitos, CA, USA, feb 2019. IEEE Computer Society.

[139] Scott A. Watterson and Saumya K. Debray. Goal-Directed Value Profiling. In *Proceedings of the 10th International Conference on Compiler Construction*, CC '01, pages 319–333, London, UK, UK, 2001. Springer-Verlag.

[140] Mark N. Wegman and F. Kenneth Zadeck. Constant Propagation with Conditional Branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, Apr 1991.

[141] B. P. Welford. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):419–420, 1962.

[142] Shasha Wen, Milind Chabbi, and Xu Liu. 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, pages 47–61, New York, NY, USA, 2017. ACM.

[143] Shasha Wen, Xu Liu, John Byrne, and Milind Chabbi. 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, pages 332–347, New York, NY, USA, 2018. ACM.

[144] Shasha Wen, Xu Liu, and Milind Chabbi. 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, pages 254–265, Washington, DC, USA, 2015. IEEE Computer Society.

[145] Wikipedia. Algorithms for calculating variance. http://en.wikipedia.org/wiki/Algorithms_for_calculating_variance, 2019.

[146] Guoqing Xu. 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, pages 111–130, New York, NY, USA, 2013. ACM.

[147] Guoqing Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, and Gary Sevitsky. 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, pages 419–430, New York, NY, USA, 2009. ACM.

[148] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. Finding low-utility data structures. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 174–186, New York, NY, USA, 2010. ACM.

[149] Guoqing Xu and Atanas Rountev. Detecting inefficiently-used containers to avoid bloat. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 160–173, New York, NY, USA, 2010. ACM.

[150] Hao Xu, Qingsen Wang, Shuang Song, Lizy Kurian John, and Xu Liu. Can we trust profiling results?: Understanding and fixing the inaccuracy in modern profilers. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '19, pages 284–295, New York, NY, USA, 2019. ACM.

[151] Shengqian Yang, Dacong Yan, Guoqing Xu, and Atanas Rountev. Dynamic analysis of inefficiently-used containers. In *Proceedings of the Ninth International Workshop on Dynamic Analysis*, WODA 2012, pages 30–35, New York, NY, USA, 2012. ACM.

[152] A. Yasin. A top-down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 35–44, March 2014.

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

[154] Dong Young Yoon, Ning Niu, and Barzan Mozafari. Dbsherlock: A performance diagnostic tool for transactional databases. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1599–1614, New York, NY, USA, 2016. ACM.

[155] Yutao Zhong and Wentao Chang. Sampling-based program locality approximation. In *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, pages 91–100, New York, NY, USA, 2008. ACM.