Pinpointing Software Inefficiencies With Profiling

Shasha Wen

William & Mary - Arts & Sciences, swen@email.wm.edu

Follow this and additional works at: https://scholarworks.wm.edu/etd

Part of the Computer Sciences Commons

Recommended Citation

http://dx.doi.org/10.21220/s2-t14d-3d56

This Dissertation is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Dissertations, Theses, and Masters Projects by an authorized administrator of W&M ScholarWorks. For more information, please contact scholarworks@wm.edu.
Pinpointing Software Inefficiencies with Profiling

Shasha Wen

Williamsburg, VA

Bachelor of Computer Science, Beihang University, China, 2010
Master of Computer Science, Beihang University, China, 2013

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 2020
This Dissertation is submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

Shasha Wen

Approved by the Committee, January 2020

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

Wei Zheng Mao

Professor Wei Zheng Mao, Computer Science
College of William & Mary

Professor Andreas Filathopoulos, Computer Science
College of William & Mary

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

Dr. Milind Chabbi
Uber Technologies
ABSTRACT

Complex code bases with several layers of abstractions have abundant inefficiencies that affect the performance. These inefficiencies arise due to various causes such as developers’ inattention to performance, inappropriate choice of algorithms and inefficient code generation among others.

To eliminate the redundancies, lots of work have been done during compiling phase. However, not all redundancies can be easily detected or eliminated with compiler optimization passes due to limited optimization scopes, and execution contexts act as severe deterrents to static program analysis. There are also profiling tools which can reveal how resources are used. However, they can hardly distinguish whether the resource are worth fully used. More profiling tools are in need to diagnose resource wastage and pinpoint inefficiencies.

We have developed three tools to pinpoint different types of inefficiencies in different granularity. We build Runtime Value Numbering (RVN), a dynamic fine-grained profiler to pinpoint and quantify redundant computations in an execution. It is based on the classical value numbering technique but works at runtime instead of compile time. We developed RedSpy—a fine-grained profiler to pinpoint and quantify value redundancies in program executions. Value redundancy may happen over time at same locations or in adjacent locations, and thus it has temporal and spatial locality. RedSpy identifies both temporal and spatial value locality. Furthermore, RedSpy is capable of identifying values that are approximately the same, enabling optimization opportunities in HPC codes that often use floating point computations. RVN and RedSpy are both instrumentation based tools. They provide comprehensive result while introducing high space and time overhead. Our lightweight framework, Witch, samples consecutive accesses to the same memory location by exploiting two ubiquitous hardware features: the performance monitoring units (PMU) and debug registers. Witch performs no instrumentation. Hence, witchcraft—tools built atop Witch—can detect a variety of software inefficiencies while introducing negligible slowdown and insignificant memory consumption and yet maintaining accuracy comparable to exhaustive instrumentation tools. Witch allowed us to scale our analysis to a large number of code bases.

All the tools work on fully optimized binary executable and provide insightful optimization guidance by apportioning redundancies to their provenance–source lines and full calling contexts. We apply RVN, RedSpy and Witch on programs that were optimization targets for decades and guided by the tools, we were able to eliminate redundancies that resulted in significant speedups.
# TABLE OF CONTENTS

Acknowledgments vi

Dedication vii

List of Tables viii

List of Figures ix

1 Introduction 2

Thesis Statement 3

Thesis Contributions 3

Thesis Organization 4

2 RVN: Pinpointing Redundant Computations 5

2.1 Introduction 5

2.2 Background and Motivation 6

A practical example of computation redundancies 8

2.3 Related Work 9

Static analysis of computation redundancies 9

Dynamic analysis of redundancies 10

2.4 Basic Runtime Value Numbering Methodology 10

2.4.1 Implementation Details 11

Efficiently maintaining value numbers 11

Handling operand aliases 12
4.2 Related Work and Motivation .................................................. 48
  Tools Based on Hardware Debug Registers: ......................... 49
4.3 Background and Terminology .................................................. 50
  Hardware Performance Monitoring Units (PMU): ..................... 50
  Hardware Debug Registers: ............................................. 50
  Linux Perf_events: ..................................................... 50
  Call Path Profiling: ..................................................... 50
  Terminology: ............................................................. 51
4.4 Methodology and Design ..................................................... 51
  4.4.1 Challenge with Samples Intervening Accesses .................. 52
  Adversary Sample: ...................................................... 54
  4.4.2 Challenges with Proportional Attribution ...................... 54
  4.4.3 Limitations ........................................................ 56
4.5 Design and Implementation ................................................... 57
  PMU Sampling: .......................................................... 57
  Watchpoint Registration: .............................................. 57
  Precise PC of a Watchpoint: .......................................... 57
  Fast Watchpoint Replacement: ........................................ 58
  Stack Addresses: ........................................................ 58
4.6 Witchcraft: Client Tools of WITCH ........................................ 59
  4.6.1 SilentCraft: Silent Store Detection ............................. 59
  4.6.2 LoadCraft: Load-after-load Detection ......................... 60
  4.6.3 Witchcrafts on Multi-threading ................................. 60
  4.6.4 Discussion ........................................................ 61
  4.6.5 Presentation ...................................................... 61
4.7 Evaluation ........................................................................... 61
4.8 Case Studies ........................................................................ 64
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.8.1 NWChem-6.3</td>
<td>65</td>
</tr>
<tr>
<td>4.8.2 Caffe-1.0</td>
<td>66</td>
</tr>
<tr>
<td>4.8.3 GNU Binutils-2.27</td>
<td>67</td>
</tr>
<tr>
<td>4.8.4 SPEC OMP2012 367.imagick</td>
<td>67</td>
</tr>
<tr>
<td>4.8.5 Discussion on Other Optimizations</td>
<td>68</td>
</tr>
<tr>
<td>5 Conclusion</td>
<td>69</td>
</tr>
<tr>
<td>Thesis Confirmation</td>
<td>70</td>
</tr>
<tr>
<td>5.1 Innovation Highlights</td>
<td>70</td>
</tr>
<tr>
<td>Instrument with Sliding Window</td>
<td>70</td>
</tr>
<tr>
<td>Temporal and Spatial Value Locality</td>
<td>70</td>
</tr>
<tr>
<td>Approximation Checking for Floating Points</td>
<td>70</td>
</tr>
<tr>
<td>Monitoring Consecutive Accesses with Sampling</td>
<td>70</td>
</tr>
<tr>
<td>New Linux Kernel Patch to Better Support Debug Registers</td>
<td>71</td>
</tr>
<tr>
<td>5.2 Research Highlights</td>
<td>71</td>
</tr>
</tbody>
</table>

Bibliography                                                             71
ACKNOWLEDGMENTS

My thanks go to my advisor, Dr. Xu Liu, without whom this PhD program would not be successful. Dr Liu is a good mentor who gave very good suggestions whenever I struggled. Dr Liu is also a good friend who cares not only the progress of the projects but also the student’s life and careers. I also want to thank Milind Chabbi, one of my collaborators. His wealth of knowledge and thoughtful thinking impressed and helped me a lot. I have also met good friends, Han Li, Yubao Zhang, Du Shen among others who enriched my PhD live with lots of joy.
This dissertation is dedicated to my parents who support me without hesitate, my husband Zhang Xu who encourages me in various ways, and also my cute baby girl Luciana Xu.
LIST OF TABLES

2.1 Performance improvement after eliminating redundancies in the stencil code. ........................................... 9
2.2 Redundant Fraction for SPEC Benchmarks ................................. 17
2.3 Overhead of RVN with sampling ........................................... 17
2.4 Performance Improvement ................................................. 18

3.1 Machine configurations. .................................................. 37
3.2 Breakdown of temporal redundant bytes and redundant instructions in different benchmark suites. ............................. 38
3.3 Comparing overhead and redundancy with sampling enabled and disabled. The sampling covers 1% instructions. ......................... 39
3.4 RedSpy’s space and time overheads in the unit of times (×) on SPEC CPU2006 benchmarks. .............................. 40
3.5 Overview of performance improvement guided by RedSpy on different platforms. ........................................... 40
3.6 RedSpy vs. other tools: whether value redundancies identified by RedSpy can be identified by other tools. ......................... 45

4.1 Runtime slowdown (×) and memory bloat (×) over native execution: Witch (DeadCraft, SilentCraft, LoadCraft) vs. exhaustive monitoring tools (DeadSpy, RedSpy, LoadSpy). .......................... 64
4.2 Geomean and median of slowdown and memory bloat of Witch tools at different sampling rates on SPEC CPU2006. ...................... 65
4.3 Performance improvement guided by Witch. ............................. 65
LIST OF FIGURES

2.1 An example of value numbering. ................................. 7
2.2 Hashing < operator, VN(\(\hat{S}[0]\)), VN(\(\hat{S}[1]\)), \ldots > to a 64-bit integer. 13
2.3 A redundancy pair reported in bwaves. ....................... 19

3.1 Breakdown of redundant bytes written in different benchmark suites. 37
3.2 A redundancy pair reported in h264ref. .......................... 41
3.3 A redundancy pair reported in NWChem. ....................... 42

4.1 Detecting dead writes using Witch. The client, DeadCraft, subscribes to the precise PMU store event with a desired sample period. ① PMU counter overflows triggering an interrupt. ② Witch handles the signal, extracts the calling context (\(C_{\text{watch}}\)) of the interrupt and the address accessed (\(M\)), and offers the triplet \(\langle C_{\text{watch}}, M, \text{AccessType} \rangle\) to DeadCraft. ③ DeadCraft asks Witch to monitor subsequent load or store to \(M\). ④ Witch sets a watchpoint to monitor \(M\), and the execution continues ⑤ Program accesses \(M\), which causes a CPU trap. ⑥ Witch handles the trap signal, extracts the calling context (\(C_{\text{trap}}\)), and offers the triplet \(\langle C_{\text{trap}}, M, \text{AccessType} \rangle\) to DeadCraft. ⑦ If the AccessType is a store, DeadCraft infers a dead write and attributes it to \(\langle C_{\text{watch}}, C_{\text{trap}} \rangle\) .......................................................... 51

4.2 \(a[]\) and \(b[]\) and \(x\) are involved in dead writes in 3:2:1 ratio (50%:33%:17%), respectively. The sampling interval is 50K stores. Our proportional, context-sensitive scheme apportions dead writes in near perfect ratio. .......................... 56

4.3 (a) A PMU sample happens in a deeper call stack when \(B()\) is accessing address \(M\); signal handler sets a watchpoint to monitor the address \(M\). (b) A shallower application call stack, function \(A()\), triggers another PMU sample, the signal handler is established in a location that overwrites \(M\), triggering a spurious watchpoint. (c) An alternate signal stack for PMU signal handler and watchdog signal handler solves the problem. .......................................................... 59

4.4 Witch tools vs. instrumentation tools on SPEC CPU2006. Error bars capture different sampling rates. Ground truth instrumentation data is unavailable for gobmk, sjeng, and Xalan since they ran out of memory. The benchmarks with multiple inputs (e.g., bzip2) appear multiple times with different numerical suffixes. .......................... 63

4.5 Comparison of dead writes with different number of debug registers. Error bars are for different (100K - 100M) sampling intervals. .................. 64
4.6 The pair of dead and kill stores with full contexts reported by Witch’s
dead store client. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
Pinpointing Software Inefficiencies with Profiling
Chapter 1

Introduction

High tech now is everywhere in our lives, we have voice service enabling us to “talk” with the devices, we have auto drive to save us from the traffic, we have smart phones to manage almost everything in our lives. All these high technologies are supported on top of massive computations. To meet the rapidly increasing computing needs, as we can see, the hardware in recent decades has developed quite fast. More cores are integrated on chip for high thread-level parallelism. Deeper memory hierarchy with multiple layers of caches is applied to shorten the data access latency. To take fully use of the highly developed hardwares, production softwares, including HPC applications need to be seriously designed and maintained which is nontrivial to achieve. On the other hand, sophisticated flow of control and deep hierarchy of component libraries increase the complexity of the software systems resulting it more challenging to keep the efficiency. Various inefficiencies hide in the execution of the softwares. These inefficiencies can be treat as two categories according to how they occur. Some ones exist in the logic of the program and are irrelevant with the hardware, which we refer as “bare-software” inefficiencies. Others happen when we map the execution to the hardwares and are hardware relevant, which we refer as “software-hardware interaction” inefficiencies. “Bare-software” inefficiencies can arise from causes such as developers’ attention to functionality instead of performance, suboptimal implementation choice, suboptimal code generation among others. “Software-hardware interaction” inefficiencies are related with resource allocation and resource contention like the data placement in heterogeneous memory systems, cache false sharing issue in multi-threading programs, bandwidth contention when manipulating large amounts of data.

The “software-hardware interaction” inefficiencies are more popular among researchers. Different methodologies, online or offline, are introduced to extract the access patterns of memory traces to conclude better data allocation strategies – on local memory or on remote memory in NUMA architectures, on DRAM or on NVM on heterogeneous memory systems. Various simulations are built to estimate whether memory contention happens. Machine learning techniques are also applied in predicting whether contention happens and even diagnosing how it happens. In comparison, less works are trying to solve the “bare-software” inefficiencies which include unnecessary computation, unnecessary data manipulation, and excessive synchronization, to name a few. This thesis focuses on reducing “bare-software” inefficiencies and we use inefficiencies for short in later descriptions.
Traditionally, optimizing compilers are adept at eliminating redundant operations by techniques such as common sub-expression elimination [29], value numbering [98], and constant propagation [115] among others. However, compilers' myopic program view limits their analysis to a small scope—individual functions or files. Inefficiencies related with aliasing, input sensitivity, work-flow sensitivity among some of others can hardly be captured at compile time. Link-time optimizations [36, 56] can gain more details and offer better visibility; however, the analysis is still conservative. Layers of abstractions, dynamically loaded libraries, multi-lingual components, aggregate types, aliasing, sophisticated flows of control, and combinatorial explosion of execution paths make it practically impossible to obtain a holistic view of an application to apply all available compiler optimizations. On top of compiling phase based optimization methodologies, monitoring runtime executions is in need to explore more improvement opportunities. State-of-the-art performance profiling techniques such as HPCToolkit [5], VTune [49], gprof [39], OProfile [95], and CrayPAT [30] monitor code execution to identify hot code regions, idle CPU cycles, arithmetic intensity, and cache misses, among others. These tools can recognize the utilization (saturation or underutilization) of hardware resources, but they cannot inform whether a resource is being used in a fruitful manner that contributes to the overall efficiency of a program. For example, none of the aforementioned profilers can identify if computing the exponential of a loop invariant number inside a loop is a wasteful use of the floating-point unit. They may, in fact, mislead us by acclaiming such loop with a high IPC (instructions executed per cycle) metric. When profiling performance with these tools, significant manual efforts are in need to root cause where the inefficiencies happen.

**Thesis Statement**  
Runtime wastage profiling can monitor the real execution of programs running on modern CPU machines to pinpoint whether any hardware resources are unfaithfully used, the severity of the issue and provide valuable insights to understand program and improve the code quality.

**Thesis Contributions**  
To detect unnecessary operations, one solution is fine-grained program monitoring. Fine-grained analysis profilers microscopically monitor each dynamic instruction, its operands, memory accesses, and runtime values. A key advantage of microscopic program-wide monitoring is that it can identify redundancies irrespective of user-level program abstractions. Runtime tracking different forms of redundancies offers visibility into program inefficiencies and hence offers new avenues to tune codes. Another advantage of fine-grained program monitoring is that it can give a comprehensive analysis about the redundancies by monitoring all the instructions of interest to the developers.

We have build two different fine-grained monitoring profilers to detect different kinds of unnecessary operations from different point of views. The first profiler, Runtime Value Numbering (RVN), targets to analyze instructions running on the CPU and detect if there are any CPU cycles wastage. In RVN, we implement value numbering technique during runtime, capture the operands and operator for all the symbolic computations and explore if the same computation is repeatedly executed by the CPU. When the same computation is conducted multiple times, one may find a way to reuse the previous calculated result and save the CPU cycles for more valuable work. The second profiler, RedSpy, targets value related redundancy. We verified the existence and significance of value locality, the
same (similar) value has a probability to be rewrite to the same location and the same (similar) value has a probability to be write to adjacent locations. Writing the same data to the same location, also known as silent write, doesn’t change the status of the system and is a symptom of some kinds of redundancies in the code. RedSpy explores write value locality happening in both memory and registers, two typical storage locations in a CPU system. Redundancies are reported when the same (similar) data are updated to the same location. RedSpy exposed unnecessary operations from the data’s point of view.

While providing higher visibility, fine-grained monitoring suffers from high overhead and massive memory usage. The high overhead of fine-grained inefficiency detection tools has kept them away from wide adoption. There is a need to make such tools more available to the developer community so that inefficiency detection can be made commonplace-run with each code check-in to isolate inefficiencies at the earliest. We then developed Witch, a lightweight inefficiency detection framework, to address this issue. Witch combines the best of both worlds, low overhead of coarse-grained profilers and inefficiency detection of fine-grained profilers. Our key observation is that an important class of inefficiency detection schemes, explored previously via fine-grained profilers, requires monitoring consecutive accesses to the same memory location. For example, detecting repeated initialization, a dead write [20], requires monitoring store after store without an intervening load to the same location. To achieve the goal, Witch applies two hardware resources, performance monitoring unit (PMU) and debug registers to monitor consecutive accesses and explore unnecessary data movements.

**Thesis Organization** The rest of this dissertation is structured as follows. Chapter 2 discusses more details about the methodology and implementation of RVN profiler. Chapter 3 shows instead of the traditional instruction-based analysis, how RedSpy can expose inefficiencies from data’s point of view. Chapter 4 proposes a much lightweight framework, Witch, to pinpoint unnecessary memory data manipulating with negligible overhead. 5 concludes the work.
Chapter 2

RVN: Pinpointing Redundant Computations

2.1 Introduction

Software systems are increasing in their complexity since they employ a hierarchy of libraries. Library abstractions provide reusability, but they introduce redundancies that add additional overheads. Furthermore, unlike previous generation microprocessors that consisted of a few heavyweight superscalar cores, emerging microprocessor architectures consist of many lightweight cores. In the light of these trends, efficiently using CPU cycles is becoming increasingly important.

Computation redundancy is a common kind of inefficiency in programs. Classical compile-time optimizations such as global value numbering [98], constant propagation [115], and common sub-expression elimination [29], fall short of expectations due to aliasing, limited scopes of optimization, and redundancies that are specific to some inputs, execution paths, or runtime execution contexts. Our experiments show that more than 20% instructions are redundant in many SPEC CPU2006 reference benchmarks, even when the benchmarks were fully optimized using profile-guided optimization.

Performance tools, such as HPCToolkit [5], VTune [49], gprof [39], OProfile [95], and CrayPAT [30], among others can efficiently identify code sections executing excessive amount of CPU cycles via well-known metrics such as floating point operations per second (FLOPS) or cycles per instruction (CPI). While these tools may identify hot code regions where a program spends a lot of time, they do not provide insights on whether the resources were well utilized. A low CPI (or high FLOPS) ensures that the application is not stalled for resources; however, it does not ensure that the application is making good use of its resources.

While a vast amount of literature has focused on profiling for hot paths and memory access latencies, little has been done to profile redundant computations. To overcome the limitations of state-of-the-art optimization methods and profiling tools, we design,
implement, and evaluate a profiler for pinpointing computation redundancies. Our profiler performs runtime value numbering (RVN) to identify computation redundancies in fully optimized binary code. RVN identifies code regions that have prodigal CPU resource consumption. Then, RVN attributes such resource wastage to source lines in their full contexts and quantifies the wastage with a redundancy metric. Finally, RVN informs top calling contexts where redundancy is high, which enables developers to focus on code regions that demonstrate opportunities for non-trivial improvement.

RVN complements the static compiler technique by identifying more computation redundancies for optimization, giving opportunities for feedback-directed optimization, code specialization, and workload-based performance tuning. Designing an effective RVN profiler faces two challenges:

1) **Overhead:** both time and space overheads of runtime value numbering is comparable to that of instruction-level trace collection. Such overhead is prohibitively high, making it impractical to profile long-running programs.

2) **Result interpretation:** because profiling is applied to the binary code, the measurement provides redundancy information at the assembly level that is difficult to interpret by programmers.

This work addresses these challenges and makes RVN practical for real programs. To evaluate RVN, we select four sequential programs from the SPEC CPU2000 and CPU2006 benchmark suites and one parallel HPC benchmark from the DOE national laboratory.

Guided by our profiler, we identify significant computation redundancies and obtain up to more than 20% speedups for these benchmarks compiled by different compilers with optimization options. In summary, our work has the following three contributions:

- We implement runtime value numbering (RVN) to profiler computation redundancies. Our profiler works on unmodified, fully optimized binaries. RVN outperforms the static analysis by identifying more redundancies in programs.

- We develop a variety of optimization techniques in RVN to minimize its runtime and space overheads, and make RVN applicable to real sequential and MPI programs.

- We collect rich information about the computation redundancies identified by RVN, such as fraction of redundancy and attribute it to the code in its full calling contexts. Such information provides insights for programmers to refactor their code to eliminate redundancies that can obtain non-trivial performance gains.

### 2.2 Background and Motivation

Value numbering (VN for short) [98] is a well-known data-flow analysis technique for determining the equivalence of two computations in a program and eliminating one of them with the results of the other. VN assigns symbolic values to computations in such a way that two computations are assigned the same symbolic value only if they are equivalent. However, not all equivalent computation need be assigned the same symbolic value. An optimistic VN algorithm assigns the same value to all expressions unless proved otherwise, whereas a pessimistic VN algorithm assigns different values to different expression unless
they are proved equal. VN can be performed per basic block, over a region of blocks or on the entire function via dominance information. VN may discover more redundancies than other compiler techniques do, such as constant propagation [115], partial redundant elimination [24], common subexpression elimination [29], and code motion [26]. VN, however, may also miss redundancies that these techniques may discover.

Figure 2.1 shows a concrete example of how VN identifies computation redundancies within a basic block. The VN processes each instruction statically. It obtains the previously computed symbolic value of each operand on the RHS, assigning a unique number if the operand is newly encountered. Then, it hashes the symbolic values assigned to operands together with the operator to obtain a symbolic value for the computation. If the computed symbolic value for a computation is already present in the table of previously computed values, then the current computation is redundant. In this basic block, instructions on Line 4 and 5 are redundant since the computations on the RHS are already computed by instruction on Line 1 and 2.

The aforementioned classical value numbering is employed by modern compilers to eliminate redundancies in scalar variables. Cooper et al. [28] have proposed applying the VN to subscript array variables. The compile-time VN does not eliminate all redundancies because of the following reasons:

1. Static analysis cannot accurately identify redundancies if pointers, aliases, and memory accesses are involved. For example, compilers cannot detect the redundant expression \((\ast c + \ast d)\) in Line 5 in Listing 2.1.

2. Static analysis has limited analysis scopes. VN can’t be applied across different procedures or compilation units. For example, the mod computation, \(a \% b\), on Line 3 in the function \texttt{CalleeModule} in Listing 2.3 called from the function \texttt{CallerModule} makes the mod computation on Line 11 redundant. However, since \texttt{CalleeModule} and \texttt{CallerModule} are independent compilation units, the redundancy is not detected.

3. Static analysis does not take program inputs into consideration, omitting input-dependent redundancies. Line 4 in Listing 2.4 computes the same value as Line 3, when the input arrays A and B have same values. But neither \texttt{icc}, nor \texttt{gcc} detected this.

4. Static analysis is always conservative in nature. It does not optimize context- or path-sensitive redundancies. In Listing 2.2, Line 7 has a redundant computation along the BB1→BB2→BB3 path, which is not redundant along the BB1→BB3 path.

**Figure 2.1:** An example of value numbering.
/* a and c alias each other */
/* b and d alias each other */

```c
int AliasRedundancy(int *a, int *b, int *c, int *d) {
    int v1 = *a + *b;
    int v2 = *c + *d;
    return v1 + v2;
}
```

Listing 2.1: Redundancy due to aliasing.

```c
void PathSensitiveRedundancy() {
    BB1: v1 = a + b;
    if (cont) {
        BB2: a = c;
        b = d;
    }
    BB3: v2 = c + d;
}
```

Listing 2.2: Path-sensitive redundancy.

Finally, static analysis does not quantify the benefit from redundancy elimination. Large optimization efforts may lead little performance gains.

A practical example of computation redundancies Listing 2.5 shows a stencil computation that is widely used in scientific applications [44]. The computation updates a value of an element in the matrix, by adding the values of its neighbors in all four directions. However, when we compute two adjacent elements, there are many redundant addition operations on the neighborhood elements in each direction. We partially eliminate the redundancies by reusing the addition result computed in the j direction. Table 2.1 shows the reduction of instructions and CPU cycles for the optimized stencil code with \( M = N = 10000 \) and \( T = 10 \). The significant performance improvement (35.8\%) shows the importance of redundancy elimination. It is worth noting that such redundancies are difficult to optimize via static analysis, especially in the presence of pointers and aliases.

Moreover, computation redundancies can highly depend on the input. In the extreme case, when the matrix in Listing 2.5 is sparse, most of the additions are redundant, because they always perform addition with zero. This kind of redundancies cannot be identified by the static analysis, whereas a profiler looking for redundant computations during the program execution can detect such redundancies.

It is worth noting that existing dynamic profilers based on the widely used cycle per instruction (CPI) metric, do not accurately quantify the computation inefficiency. Low CPI does not necessarily mean the computation is efficient. In fact, in this case, compared

### Table 2.1

<table>
<thead>
<tr>
<th>Instruction Reduction</th>
<th>CPU Cycles Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>80%</td>
<td>70%</td>
</tr>
</tbody>
</table>

1Redundancies can be further reduced with transformation in i direction.
Listing 2.5: Redundancies in a stencil code.

Table 2.1: Performance improvement after eliminating redundancies in the stencil code.

<table>
<thead>
<tr>
<th>Program</th>
<th>Orig.</th>
<th>Optimize</th>
<th>%Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Instructions(billion)</td>
<td>72.8</td>
<td>39.64</td>
<td>45.6</td>
</tr>
<tr>
<td>#cycles(billion)</td>
<td>32.74</td>
<td>20.97</td>
<td>35.8</td>
</tr>
</tbody>
</table>

to the optimized code whose CPI is 0.53, the original code, as shown in Listing 2.5, has a lower CPI—0.45. Thus, a dynamic profiler that pinpoints redundant computations is essential to understand execution inefficiencies.

2.3 Related Work

There are a variety of techniques applied in compilers to eliminate redundant computations. Elaborating all of these techniques, however, is outside the scope of this work. In this section, we review the most related work that adopt either static or dynamic analysis.

Static analysis of computation redundancies Cooper et al. [28] extended traditional value numbering algorithm to identify inter-iteration redundant computations in loops. They assigned value numbers to array references in a loop to identify redundant computations across loop iterations. They integrated this new value numbering technique to perform enhanced scalar replacement at compile time and achieved good performance improvement.

Deitz et al. [29] extended the common sub-expression elimination to identify redundant computations across loop iterations. Their approach analyzes instructions only with sum-of-products operators and memory reference operands. Moreover, their approach requires stringent index expressions for array references, which limits its applicability.

Luo et al. [76] developed equivalent computation elimination (ECE) to remove redundant computations in multi-dimensional stencil code. Their approach can identify and eliminate redundancies that reside in deep loop nests.

Hundt et al. [45] developed MAO, a compiler-independent tool to identify and eliminate useless and redundant computations. MAO statically analyzes a program’s assembly code with a sliding window, looking for a predefined set of patterns that define inefficient computations. The small size of sliding windows limits MAO’s optimization scope, preventing it from identifying inter-procedural and inter-module inefficiencies.

All these static approaches suffer from limitations related to aliasing, optimization scope, and insensitivity to input and execution contexts, as described in Section 2.2. In
contrast, RVN is a dynamic approach, which is compiler-independent. RVN can identify more computation redundancies than static approaches.

**Dynamic analysis of redundancies** Chabbi et al. [20] developed DeadSpy tool to identify dead stores. They track every memory store to pinpoint the ones that are never loaded before subsequent stores. They associate pairs of instructions involved in a dead write with their calling contexts and source code locations to guide manual program optimizations. Unlike DeadSpy, RVN detects redundancies in arithmetic-logic, as well as load and store instructions.

Butts et al. [13] developed a hardware-based method to track CPU-bound operations and identify useless computations in a program. They do not provide detailed feedback for optimization. In contrast, RVN also monitors memory operations and provides rich optimization guidance.

Moreover, both of these two approaches pinpoint useless computations, but not redundant computations. Redundant computations are the same computations that are performed more than once, which are not necessarily useless computations. To the best of our knowledge, RVN is the first dynamic tool that identifies all kinds of computation redundancies.

### 2.4 Basic Runtime Value Numbering Methodology

A straightforward implementation of RVN is to adapt the existing static value numbering algorithm, as shown in Algorithm 2.1. RVN maintains a counter $gValue$ to assign unique numbers to different values computed in an execution, and a map ($VNMap$) to record the value number associated with each unique computation (Line 2 and 3). At runtime, RVN monitors every instruction under execution. It decodes operators as well as both source and target operands. Typically, an instruction has multiple source operands and one target operand. If the source operands do not have a value number already assigned, RVN assigns a unique number obtained by incrementing $gValue$ to each of them (Line 8-12).

If an operator copies a source operand to a target operand, RVN assigns the value number of the source operand to the target operand, which means that the value is passed from one operand to another without change. RVN constructs a key ($hashKey$) formed by the operator and the value numbers of the source operands (Line 21). If the operator is commutative, such as addition and multiplication, RVN sorts all the value numbers of the source operands (Line 17) to expose potential redundancies via computation re-association [11]. If the operator is not commutative, such as subtraction and division, RVN retains the order of the operands. Then RVN uses this key to index the computation of this instruction into the global $VNMap$. If the $hashKey$ is already in the map, RVN records a redundancy because the computation is performed earlier. Otherwise, RVN inserts the $hashKey$ in the map and associates it with a new value number. This value number is also assigned to the target operand. The choice of the $Hash$ function greatly influences the performance of RVN, which we discuss later. In rare cases, an x86 instruction can have multiple target operands; RVN gives each target operand a new value (not shown in Algorithm 1).
Algorithm 1 Algorithm for RVN, invoked before each instrumented instruction

```
/* data structures */
uint64 gValue = 0;
unordered_map<Key, uint64> VNMap;
I := instruction about to execute;
P := I’s operator;
T := I’s target operand;
S := {I’s source operands in order};
for each source operand S[i] in S do
  if S[i] is not already assigned a VN then
    VN(S[i]) := ++gValue;
  end if
end for
if --S-- == 1 and P copies source operand S[0] to T then
  VN(T) := VN(S[0]);
else
  if P is commutative then
    ˆS := sort( VN(S[0]), VN(S[1]), ...);
  else
    ˆS := {VN(S[0]), VN(S[1]), ...};
  end if
  hashKey := Hash(P, ˆS[0], ˆS[1], ...);
  if VNMap contains hashKey then
    record redundancy in the RedundancyTable ;
  else
    VNMap[hashKey] := ++gValue;
  end if
  if T ≠ NULL then
    VN(T) := VNMap[hashKey];
  end if
end if
```

On each instruction, RVN records the instruction pointer where the value number was computed. On encountering a redundant computation, RVN, records a pair of instruction pointers where the previous value number was computed, and the current instruction where the redundant computation occurred, into a table—RedundancyTable. RedundancyTable is keyed by the pair of instructions, and the value stored in each entry of the RedundancyTable is the number of times the redundancy happened at the same pair of instructions. Pairs with higher redundancy values are reported first for the developer inspection.

2.4.1 Implementation Details

We have implemented RVN using Intel’s dynamic instrumentation framework—Pin [2] . In the following paragraphs, we elaborate some of the implementation details of RVN in the context of Pin.

**Efficiently maintaining value numbers**  RVN leverages Pin to decode each instruction to obtain its operator and operands. An x86 instruction can have one operator and one or more operands on both left- and right-hand sides. Operands can fall into three categories: registers, immediate numbers and memory references. Pin uses a unique integer to encode each register. We use these encoding bits to index into a table to fetch the value number associated with a register. It incurs only $O(1)$ time overhead to fetch the value number of
a specific register. Since immediate numbers are encoded in the instruction stream, we use their values as keys to map to the assigned value numbers. The number of registers in the system is constant, while there is only a small amount of immediate numbers in a binary. Consequently, maps for both registers and immediate numbers have negligible sizes.

As for operands with memory references, we assign a value number to each effective address used in the instruction, regardless of the operand’s addressing mode. Associating value numbers with effective addresses disambiguates memory aliasing due to indirection employed by programs. The number of memory addresses used in a program is large. To efficiently access the value number of a memory address, we use the page-table-based shadow memory technique [20]. Shadow memory creates shadow bytes that are associated with every memory byte used in the program. These shadow bytes are invisible to the original program and are used to record the value numbers for operand with memory references. The shadow memory allows RVN to obtain the value number for each memory address in O(1) time.

**Handling operand aliases** Aliases can exist in operands that reference registers or memory. In x86 architectures, different segments of a register can be accessed via different register names. For example, the register AL is the lower 8 bits of AX, AX is the lower 16 bits of EAX, and EAX is the lower 32 bits of RAX. We handle the value number assignment to these registers as follows. If the value number of a smaller-scoped register, e.g., AH is updated, then the value number of all the large-scoped registers, such as AX, EAX, and RAX that enclose the bits of AH are also updated. If a larger-scope register, e.g., AX updates its value number, RVN updates the value numbers of all its constituent smaller-scoped registers AL and AH (in this particular case, updating the value number of AX will also update the value numbers of EAX, and RAX).

A similar issue occurs for operands that reference memory. Typically, write to a location and the corresponding read from the location are of the same size, and hence RVN obtains the value number associated with the starting address of an operand, irrespective of the size of the memory operand. One may improve the precision of RVN by maintaining the value numbers for each byte of a location accessed by an instruction at the cost of additional overhead. Our experience using RVN demonstrated no need for byte-level value numbers for larger memory accesses.

**Handling parallel programs** RVN runs out-of-the-box for programs parallelized by MPI [80]. Each process performs the RVN individually and records the analysis results into separate files without any interference. Results are merged in a postmortem fashion. Adapting RVN for multithreaded applications is complicated, since updates to shared locations by one thread need to be informed to other threads. In our prototype, we ignore multiple threads in the same address space, and instead, each thread performs RVN independently by maintaining thread-private RVN data structures (gValue, VNMap, RedundancyTable, and the shadow memory). Our approach is justified, since computational redundancies that might arise between two independent threads are both rare and hard to eliminate without incurring synchronization overheads.
2.5 Algorithmic Refinement

The basic RVN algorithm has high space and time overheads. The hashing technique and heavyweight instrumentation highly impact the runtime overhead. Moreover, the number of entries in VNMap can quickly grow large causing unaffordable space overhead. Finally, the assembly-level data gathered from the basic RVN algorithm needs to be associated with full calling contexts and source code for consumption by application developers. We address these three issues in the following subsections.

2.5.1 Reducing Time Overhead

**Approximate hashing** The hash function used to compute the hash key from operands and operators of an instruction is one of the governing factors in the overhead of RVN. A binary blob formed by all bits of the operator and all operands is collision free, but very slow, whereas a fixed-width hash that does not pay attention to the contents of the operator or operands can be fast but too inaccurate due to hash collisions. We trade-off speed for accuracy, but do so prudently to keep the collision to a minimal level. Our hash is a 64-bit fixed-width key formed from the operator and set of operands, but the hash function pays weightage to the operator and operands that play a significant role in forming a unique key.

Figure 2.2 shows, how we compute a 64-bit hash of a computation by incorporating operator and operands of the x86 ISA. The highest 8 bits represent the instruction operator. From our experiments, such 8 bits can cover all the frequently-used operators without conflict; other lengthy operators are not commonly used. The following 56 bits encode the value number of each operand. For a typical x86 instruction, there can be two or more operands. We evenly divide the 56 bits to include all the truncated operand values. For example, if an instruction has two operands, the least 28 bits of each operand value will be used by the hash function. Two non-redundant instructions that have the same operator and their operands share the same least several bits but not the significant ones, may hash to the same 64-bit value. To evaluate the effectiveness of our hashing strategy, we measured the hash collision in 15 SPEC CPU2006 benchmarks. The geometric mean of collisions was only 0.095%, which corroborates the accuracy of our hashing.
Selective instruction instrumentation  In an application, not all kinds of instructions triggering redundancies can be removed legally. For example, a comparison instruction consistently executed in a spin loop is not a candidate for optimization. The analysis of such instructions not only produces false positives, but also incurs high runtime overhead. Therefore, RVN does not monitor comparison and control flow instructions. RVN also ignores stack maintenance operations such as push and pop.

Bursty sampling  Despite the aforementioned filtering, keeping the RVN instrumentation enabled for the entire application can add high overhead. The source of the overhead is the size of the hash table used for storing all value numbers generated during the execution. A large number of keys in VNMap slow down the lookup performed during each instruction.

We develop a sampling method to reduce instruction instrumentation. RVN periodically enables and disables instruction instrumentation to reduce runtime overhead at the cost of sacrificing some measurement accuracy. To balance the overhead and accuracy, we define two thresholds: enable interval ($E$)—the interval during which the measurement is enabled, and disable interval ($D$)—the interval during which the measurement is disabled. This type of bursty sampling is effective for loop-based programs or programs with repeated execution patterns because it does not miss significant redundancies that occur with high frequencies [119]. To avoid blind spots during sampling, RVN randomizes the last a few bits of sampling thresholds. Our experiments show that enable interval of 100 million instructions and disable interval of 1 billion instructions gives a good balance between overhead and accuracy.

In the sampling approach, all the bookkeeping data structures needs to be invalided from one enabled interval to the next enabled interval. For example, the shadow memory needs to be cleared before starting each sample. Intuitively, this is time consuming because traversing all the shadow memory bytes is non-trivial. We leverage the feature of value numbering, to overcome this issue without incurring a significant overhead. Whenever the monitoring is reenabled, the global counter gValue for value number assignment is not reset, which means that the value number increases monotonically. Therefore, we record the starting value in the global counter at the beginning of each monitoring period. If we find the value number in the shadow memory is smaller than this starting value, this value number is from one of the previous sampling period and is invalidated with the new value number.

2.5.2 Reducing Space Overhead

The basic RVN inserts a 64-bit hash value for each dynamic instruction instance into VNMap. Because a typical processor executes billions of instructions per second, the hash map can quickly use up the whole memory space. To reduce the space overhead, we bound RVN’s memory consumption, making it not depend on the number of dynamic instructions. We refine the RVN implementation as follows. We only maintain the most recent $N$ hash values for each static instruction, where $N$ is a configurable parameter by the user. Therefore, the size of VNMap is proportional to the number of static instructions.
To limit the number of entries per instruction in VNMap, we employ the shadow memory technique, again. In this case, the address of the instruction acts as the key for the shadow memory. The contents of the shadow memory contain the last $N$ value numbers produced by that instruction. Once we examine an instruction $I$, we update the oldest value number produced by $I$'s computation with the current value number of the computation produced by $I$, if this value is different from any of the last $N$ value numbers produced by $I$. This incurs $O(N)$ runtime overhead ($O(1)$, when $N$ is 1). With this optimization, with $N = 1$, the RVN consumes $8 \times$ memory for our tested programs, on average.

It is worth noting that maintaining only a subset of value numbers per instruction may miss some redundancies. For example, if a hash value $V$ of an instruction is replaced by the new one, the RVN cannot identify the subsequent computation with the hash value $V$ as a redundancy. In practice, such redundancy omission occurs when there is a large time interval between executions of the two instructions in the redundancy pair. Usually, such redundancies are difficult to optimize due to the long distance between the two instructions. When a previous instance of an instruction is redundant with its future instance, we might miss such redundancies if they are separated by more than $N$ execution instances of the same instruction. For stencil-based codes, a rule of thumb is to choose $N$ to equal the number of points in a dimension that the stencil is computed on.

### 2.5.3 Providing Insights for Optimization

To make RVN useful for application developers, we need to associate the redundancy information collected by RVN with the source code and execution contexts. Doing so involves mapping a redundancy pair of instructions to their source locations, their enclosing functions, along with the call paths from `main` leading to the current pair of functions. The calling context provides insights into redundancies across procedures and pinpoints “hot” call paths where redundancies are pervasive.

Collecting the calling context for each instruction and efficiently storing it is a complicated problem. We leverage the CCTLib library [18]—a call path collection library for Pin tools—to efficiently collect and store the calling context of each instruction throughout the execution. CCTLib is tailored for fine-grained instrumentation tools that require frequent call path collection. RVN queries the CCTLib on each instruction to obtain a 32-bit handle that uniquely represents the full calling context of the current instruction. Internally, CCTLib maintains a calling context tree (CCT) [6] and the handle points to one of its tree nodes. A path implied by a node in the CCT to the root of the tree provides a unique call path. CCTLib also provides APIs for associating each instruction along the call path to the source locations, which helps RVN provide the source-level mapping along with call-site-level attribution for each redundant computation.

On encountering a redundant computation (Line 23 in Algorithm 1), RVN creates a 64-bit key (rKey) formed by a pair of two 32-bit calling context handles, one for the current instruction and one for the previous instruction where the computation was already performed. RVN inserts rKey into RedundancyTable. If the same redundancy was already reported for the given pair of contexts previously, RVN increments the value associated with the key in RedundancyTable.
Metric to quantify redundancy  To quantify the extent of redundant computations in an execution, we define the Redundancy Fraction ($R$) as the total number of dynamic redundant instructions out of the total number of dynamic instructions executed. The Redundancy Fraction is the fraction of total dynamic instructions that are redundant.

$$R = \frac{\text{Total Redundant Instructions}}{\text{Total Instructions}}$$

With bursty sampling, RVN detects the redundant operations only during the sampling period. We approximate the redundant fraction $\hat{R}$ as:

$$\hat{R} = \frac{\sum_{i=0}^{N} \text{Redundant Instructions in Sample } i}{\sum_{i=0}^{N} \text{Total Instructions in Sample } i}.$$ 

$\hat{R}$ may be slightly different from the real redundancy fraction $R$ because we can miss redundancies when monitoring is disabled due to sampling technique.

2.6 Experiments

We evaluate RVN on an 8-core Intel Nehalem processor clocked at 2.93GHz attached with 48GB DDR3 memory. We apply RVN to benchmarks from SPEC CPU2000 [107] and CPU2006 [104] benchmark suites with reference inputs, as well as one parallel HPC benchmark: Sweep3D [46]. We compile these programs using gcc 4.7 with -O2 option and profile-guided optimization (PGO) as well as icc 11.1 with default -O2 option. Table 2.2 shows that these codes have non-trivial redundant computations.

We optimize the redundant computations pinpointed by RVN for four benchmarks from SPEC CPU2000 and CPU2006, as well as Sweep3D. As shown in Table 2.3, RVN incurs about 44× runtime overhead and 8× space overhead, on average to profile these benchmarks. Table 2.4 shows the performance improvement for individual loops and the whole program of these benchmarks with both test and reference inputs. We leverage HPCToolkit [5] to glean hardware events, such as CPU cycles and graduated instructions to further understand the speedups due to redundancy elimination. From the table, we can see the significant reduction of CPU cycles consumed in the optimized loops in each benchmark. However, the number of instructions associated with these loops are not always reduced as we expect. The principle reason is that our optimizations break the SIMD instruction generation by compilers, leading the code to executing more instructions. We elaborate this issues in Section 2.6.3. Moreover, the overall speedups for 434.zeusmp and 173.applu are very small because the redundant computations in both benchmarks are not in hot loops.

In the remaining section, we describe our findings via the RVN profiler in each benchmark and discuss the code optimizations for eliminating redundancies.
Table 2.2: Redundant Fraction for SPEC Benchmarks

<table>
<thead>
<tr>
<th>Program</th>
<th>Min</th>
<th>Max</th>
<th>Average</th>
<th>GeoMean</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2</td>
<td>9.56</td>
<td>20.72</td>
<td>15.34</td>
<td>14.61</td>
</tr>
<tr>
<td>gcc</td>
<td>18.40</td>
<td>21.28</td>
<td>20.28</td>
<td>20.23</td>
</tr>
<tr>
<td>mcf</td>
<td>4.16</td>
<td>10.12</td>
<td>6.27</td>
<td>5.76</td>
</tr>
<tr>
<td>hmmer</td>
<td>18.42</td>
<td>20.27</td>
<td>19.53</td>
<td>19.51</td>
</tr>
<tr>
<td>libquantum</td>
<td>0.64</td>
<td>6.69</td>
<td>2.92</td>
<td>1.84</td>
</tr>
<tr>
<td>h264ref</td>
<td>12.91</td>
<td>13.16</td>
<td>13.07</td>
<td>23.13</td>
</tr>
<tr>
<td>omnetpp</td>
<td>21.52</td>
<td>22.81</td>
<td>22.29</td>
<td>22.28</td>
</tr>
<tr>
<td>astar</td>
<td>26.82</td>
<td>28.13</td>
<td>27.68</td>
<td>27.68</td>
</tr>
<tr>
<td>bwaves</td>
<td>3.12</td>
<td>6.73</td>
<td>4.67</td>
<td>4.43</td>
</tr>
<tr>
<td>zeusmp</td>
<td>4.29</td>
<td>4.75</td>
<td>4.51</td>
<td>4.51</td>
</tr>
<tr>
<td>gamedd</td>
<td>15.50</td>
<td>18.26</td>
<td>16.77</td>
<td>16.73</td>
</tr>
<tr>
<td>milc</td>
<td>6.23</td>
<td>6.58</td>
<td>6.45</td>
<td>6.45</td>
</tr>
<tr>
<td>gromacs</td>
<td>8.89</td>
<td>11.32</td>
<td>9.75</td>
<td>9.69</td>
</tr>
<tr>
<td>leslie3d</td>
<td>3.60</td>
<td>4.21</td>
<td>3.86</td>
<td>3.85</td>
</tr>
<tr>
<td>namd</td>
<td>3.13</td>
<td>3.18</td>
<td>3.16</td>
<td>3.16</td>
</tr>
<tr>
<td>soplex</td>
<td>9.67</td>
<td>20.07</td>
<td>13.58</td>
<td>3.85</td>
</tr>
<tr>
<td>povray</td>
<td>18.28</td>
<td>20.20</td>
<td>19.56</td>
<td>19.54</td>
</tr>
<tr>
<td>calculix</td>
<td>12.84</td>
<td>24.75</td>
<td>18.80</td>
<td>17.83</td>
</tr>
<tr>
<td>gamsRDTD</td>
<td>4.04</td>
<td>11.90</td>
<td>6.79</td>
<td>5.95</td>
</tr>
<tr>
<td>tonto</td>
<td>13.97</td>
<td>18.40</td>
<td>15.47</td>
<td>15.34</td>
</tr>
<tr>
<td>lbm</td>
<td>8.28</td>
<td>9.00</td>
<td>8.57</td>
<td>8.56</td>
</tr>
<tr>
<td>wrf</td>
<td>5.26</td>
<td>7.72</td>
<td>6.62</td>
<td>6.54</td>
</tr>
<tr>
<td>sphinx3</td>
<td>3.46</td>
<td>12.09</td>
<td>6.94</td>
<td>6.04</td>
</tr>
<tr>
<td>applu</td>
<td>4.67</td>
<td>16.11</td>
<td>9.11</td>
<td>7.90</td>
</tr>
<tr>
<td>GeoMean</td>
<td>7.82</td>
<td>12.59</td>
<td>10.16</td>
<td>9.68</td>
</tr>
</tbody>
</table>

Table 2.3: Overhead of RVN with sampling

<table>
<thead>
<tr>
<th>Program</th>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>bwaves</td>
<td>41.8×</td>
<td>5.8×</td>
</tr>
<tr>
<td>zeusmp</td>
<td>27.3×</td>
<td>7.4×</td>
</tr>
<tr>
<td>hmmer</td>
<td>46.5×</td>
<td>15.3×</td>
</tr>
<tr>
<td>applu</td>
<td>45.4×</td>
<td>7.3×</td>
</tr>
<tr>
<td>sweep3d</td>
<td>44.7×</td>
<td>4.1×</td>
</tr>
</tbody>
</table>

2.6.1 410.bwaves

Figure 2.3 shows a redundancy pair with their full calling contexts identified by our RVN profiler. This redundancy occurs many times during the execution. The `movsdq` instruction continuously loads the same data from memory to register %xmm0. This redundancy, however, occurs inside the math library used by the application code. The instruction causing the redundancy by itself provides no useful information for tuning. Beyond this, we neither have source code access, nor can one optimize it for just one workload. The calling contexts collected by RVN, however, provide rich insights: the redundancy occurs in a `pow` function, called at line 47 in file `jacobian_lam.f`. Listing 2.6 shows the source code at the call site of `pow`. Our further study shows that this piece of code is in a loop nest (denoted as `jacobian_lam.f:loop(30)` in Table 2.4). The base value used for the power of 0.75d0 computation remains the same across loop iterations, which is the cause of redundant `pow` computations. To remove this redundancy, we modify the code to reuse the value from the previous call to the `pow` function if the base value remains unchanged from the previous call to the current. The optimization can achieve significant performance
Table 2.4: Performance Improvement

<table>
<thead>
<tr>
<th>Program</th>
<th>Procedures:loops</th>
<th>ggc (-O2)</th>
<th>ggc (PGO)</th>
<th>icc (-O2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>%Cycle†</td>
<td>%Ins.†</td>
<td>WS‡</td>
<td>%Cycle</td>
</tr>
<tr>
<td>410.bwaves</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>block_solver.f</td>
<td>-97.1</td>
<td>-97.3</td>
<td>1.07×</td>
<td>-96.9</td>
</tr>
<tr>
<td>jacobion     lam.f</td>
<td>-16.0</td>
<td>-16.8</td>
<td></td>
<td>-16.0</td>
</tr>
<tr>
<td>shell_lam.f</td>
<td>-13.6</td>
<td>-15.2</td>
<td></td>
<td>-10.9</td>
</tr>
<tr>
<td>434.zeusmp</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lorentz.f</td>
<td>-22.0</td>
<td>-6.3</td>
<td></td>
<td>-6.7</td>
</tr>
<tr>
<td>forces.f</td>
<td>-15.8</td>
<td>+6.9</td>
<td>1×</td>
<td>-10.8</td>
</tr>
<tr>
<td>pdv.f</td>
<td>-6.5</td>
<td>-10.5</td>
<td></td>
<td>-5.6</td>
</tr>
<tr>
<td>456.hmmer</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hmmcalibrate.c</td>
<td>-13.9</td>
<td>-7.2</td>
<td></td>
<td>-6.5</td>
</tr>
<tr>
<td>fast_algorithms.c</td>
<td>-14.1</td>
<td>-16.9</td>
<td>1.06×</td>
<td>-7.3</td>
</tr>
<tr>
<td>173.applu</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>applu.f</td>
<td>-30.1</td>
<td>-3.4</td>
<td>1×</td>
<td>-9.1</td>
</tr>
<tr>
<td>sweep3d</td>
<td>-26.3</td>
<td>+4.3</td>
<td>1.08×</td>
<td>-21.0</td>
</tr>
</tbody>
</table>

†The percentages (%) of cycle and instruction reduction (-) and increment (+) due to redundancy elimination.
‡WS means whole-program speedup due to redundancy elimination.

Listing 2.6: Redundant pow function calls in bwaves

improvement by reducing the costly function call pow in the loop as shown in Table 2.4.

Listing 2.7 shows another significant redundancy in block_solver:loop(167) identified by RVN. The tool finds that the mod operations in Lines 5 and 6 are redundant with themselves. Further studies in the source code show that the computation of jm1 and jp1 are loop invariants for the outer-most k loop. The computation involves heavyweight mod operations that can significantly degrade the program’s performance. Completely eliminating this redundancy is difficult because recomputing jm1 and jp1 are necessary in the inner loop. The values of jm1 and jp1, however, can be determined without the mod operation. For iterations, 2 through ny-1, jm1 is j-1 and jp1 is j+1. The values are special for the first and last iteration of the j loop, hence we peel those iterations. Listing 2.8 shows the j loop after the optimization. The similar optimizations also apply to the inner-most i loop and outer-most k loop. As shown in Table 2.4, our optimization can significant reduces the loop’s execution time and instruction number.

Besides these two major redundancies, we also optimize some other minor redundancies identified by RVN. For the whole program compiled with ggc -O2, we are able to reduce division and multiplication instructions by 74.8% and 9.3%, respectively; we also reduce memory loads and stores by 13.4% and 14.5%, respectively. As a result, we get 1.07× speedup for the whole bwaves benchmark. With PGO and icc, our optimization can also achieve 1.12× and 1.04× speedups.
movsdq 0x8(%rdi,%r10,8), %xmm0:__mul::0
callq 0x7f1116aca6b0:__dvd::0
callq 0x7f1116aca840:__mpexp::0
callq 0x7f1116acaf40:__mplog::0
callq 0x7f1116acb330:__slowpow::0
callq 0x7f1116ac810:__ieee754_pow_sse2::0
callq 0x7f1116a8d610:pow::0
callq 0x400ab0:jacobian_:jacobian_lam.f:47
callq 0x404380:shell_:shell_lam.f:193
callq 0x405470:MAIN__:flow_lam.f:63
callq 0x402660:main:flow_lam.f:67

********************REDUNDANT WITH ***********************

movsdq 0x8(%rdi,%r10,8), %xmm0:__mul::0
callq 0x7f1116aca6b0:__dvd::0
callq 0x7f1116aca840:__mpexp::0
callq 0x7f1116acaf40:__mplog::0
callq 0x7f1116acb330:__slowpow::0
callq 0x7f1116ac810:__ieee754_pow_sse2::0
callq 0x7f1116a8d610:pow::0
callq 0x400ab0:jacobian_:jacobian_lam.f:47
callq 0x404380:shell_:shell_lam.f:193
callq 0x405470:MAIN__:flow_lam.f:63
callq 0x402660:main:flow_lam.f:67

Figure 2.3: A redundancy pair reported in bwaves.

Listing 2.7: Redundant mod in bwaves.

```
1    do k=1,nz
2       km1=mod(k+nz-2,nz)+1; kp1=mod(k,nz)+1
3    do j=1,ny
4       jm1=mod(j+ny-2,ny)+1; jp1=mod(j,ny)+1
5    do i=1,nx
6       im1=mod(i+nx-2,nx)+1; ip1=mod(i,nx)+1
7       ...
8  enddo
9  enddo
```

2.6.2 456.hmmer

RVN pinpoints significant redundancies between Lines 2 and 4 in the loop shown in Listing 2.9\(^2\). To better understand the causes of these redundancies, we investigate the loop’s assembly code in Listing 2.10. RVN points that a pair of redundant assignments in red in line 3 and 10. The two assignments write the same values in %ecx to the same memory location if the value in %ecx is not changed by the conditional move instruction in blue. However, during the program execution, the condition for this instruction in blue is seldom true, which prevents %ecx from receiving a new value and consistently causes the instruction in line 10 to be redundant. Similar redundant memory stores occur multiple times in this benchmark.

To remove such redundant assignments, we refactor the source code as shown in Listing 2.11. We introduce a temporary scalar variable to maintain the intermediate values of sc instead of frequently overwriting the same memory location with the same sc value. The optimization reduces 69% memory stores in the loop in Listing 2.9 with the gcc -O2 com-

\(^2\)DeadSpy [20] also found these redundancies as dead writes.
Listing 2.8: Optimized code in bwaves.

```c
for (k = 1; k <= M; k++) {
    mc[k] = mpp[k-1] + tpmm[k-1];
    if ((sc = ip[k-1] + tpim[k-1]) > mc[k]) mc[k] = sc;
    if ((sc = dpp[k-1] + tpdm[k-1]) > mc[k]) mc[k] = sc;
    if ((sc = xmb + bp[k]) > mc[k]) mc[k] = sc;
    mc[k] += ms[k];
    if (mc[k] < -INFTY) mc[k] = -INFTY;
    .......
}
```

Listing 2.9: Memory write redundancies in hmmer.

Compilation. After optimizing all the redundancies pointed by RVN, this benchmark achieves significant speedups: $1.06 \times$, $1.06 \times$, and $1.09 \times$ to the code generated by `gcc -O2`, `gcc PGO`, and `icc -O2`, respectively.

It is worth noting that compilers do not optimize the redundant memory writes because they cannot know whether the array `mc` aliases with others, such as `ip`, `tpim`, `tpdm`, and `bp`. Consequently, compilers end up retaining dead memory writes. In reality, these arrays do not alias with one another and hence our optimization is safe.

### 2.6.3 434.zeusmp and 173.applu

Listing 2.12 shows the code suffering from cross-iteration computation redundancies identified by RVN. For example, the computation in line 2 is redundant because the computation is already done at line 3 in the previous iteration. The similar redundancies appear multiple times in this loop and some other loops. We remove these redundancies, via the scalar replacement [16] technique by introducing temporary variables that hold the results computed in the current iteration and reuse them in the next iteration, as shown in Listing 2.13. From Table 2.4, we can see the redundancy elimination can significant speedup these loops by up to 30%.

The 173.applu benchmark from SPEC CPU2000 also suffers from a very similar redundancy. The `rhs` procedure has similar inter-iteration redundancies and requires the same scalar replacement optimization as for the code of zeusmp shown in Listing 2.12 and 2.13.

It is worth noting that our optimization does not reduce the instruction numbers in both benchmarks. The reason is that the original code (e.g., Listing 2.12) has regular access patterns and unit strides. Compilers can easily generate SIMD instructions in these loops. However, the optimized code (e.g., Listing 2.13) introduces many scalars, which
prevents compilers from generating SIMD instructions, leading to more instructions in these optimized loops. This observation is even obvious for code generated by ICC, which has better SIMD generation support compared to GCC. Since the performance gains by eliminating redundant computations and memory accesses in 434.zeusmp are larger than the losses due to hurting the SIMD generation, we still see significant speedups for all the optimized loops in 434.zeusmp.

2.6.4 Sweep3D

Sweep3D, an ASCI benchmark, solves a 1-group time-independent discrete ordinates 3D cartesian geometry neutron transport problem. It is written in Fortran and parallelized with MPI. RVN monitors both sequential and MPI versions with the default input data and reports significant computation redundancies in the loop highlighted in Listing 2.14. RVN identifies redundant computations, such as additions, multiplications, divisions, subtractions, and memory movements existing in all the code ranging from Line 2 to 10.

Further studies on the code show that arrays mu, hi, and sigt have identical elements. Consequently, the computations from Line 2 to 4 are loop invariant. We hoist these three lines of code outside of the loop. Further, we discover that arrays phi, phiib, and phikb can be divided into a few continuous segments with the same beginning and ending indices; and each of these arrays has elements in each segment with the same value. Therefore, we always perform a conditional check for the segment boundaries and reuse the value computed at the beginning of each segment to avoid redundant computations in Line 5 to 10.

Our optimizations speed up the loop by up to 39.6% (ICC). Because this loop is the hottest in the program, the entire application is improved by up to 22% (ICC) when the
do $i=ibeg-1,iend$
  
  $d1b2oo(i) = ( ( g2b (i) * b2(i, j, k) )**2 - ( g2b (i-1) * b2(i-1, j, k) )**2 ) * g2ai (i) * g2ai (i)$
  
  $d1b2po(i) = ( ( g2b (i) * b2(i, j+1, k) )**2 - ( g2b (i-1) * b2(i-1, j+1, k) )**2 ) * g2ai (i) * g2ai (i)$
  
  $d1b3oo(i) = ( ( g31b(i) * b3(i, j, k) )**2 - ( g31b(i-1) * b3(i-1, j, k) )**2 ) * g31ai(i) * g31ai(i)$
  
  $d1b3op(i) = ( ( g31b(i) * b3(i, j+1, k) )**2 - ( g31b(i-1) * b3(i-1, j+1, k) )**2 ) * g31ai(i) * g31ai(i)$
  
  $d2b3oo(i) = ( ( g32b(j) * b3(i, j, k) )**2 - ( g32b(j-1) * b3(i, j-1, k) )**2 ) * g32ai(j) * g32ai(j)$
  
  $d2b3op(i) = ( ( g32b(j) * b3(i, j, k+1) )**2 - ( g32b(j-1) * b3(i, j-1, k+1) )**2 ) * g32ai(j) * g32ai(j)$
  
  continue

Listing 2.12: Inter-iteration redundancies in zeusmp.

code is running sequentially. For the parallel execution with 48 processes on an AMD Magny-Cour machine, our optimization is still effective: 1.16× and 1.10× speedups (not shown in Table 2.4) for the whole program compiled by gcc -O2 and icc -O2, respectively.
Listing 2.13: Remove inter-iteration redundancies in zeusmp.

Listing 2.14: Redundant computations in Sweep3D.
Chapter 3

RedSpy: Exploring Value Locality in Software

3.1 Introduction

Sophisticated flow of control and a hierarchy of component libraries have increased the complexity of modern software productions which often introduces inefficiencies which prevent applications from achieving optimal performance.

In this work, we focus on wasteful data movement, which we refer to as “redundancy”. The term “redundancy” should not be misinterpreted as “resiliency” for fault-tolerance.

Compilers often fail to eliminate many kinds of redundancies since the myopic view of the program limits their analysis to a small scope—individual functions or files. Link-time optimization [36, 56] can offer better visibility; however, the analysis is still conservative. Layers of abstractions, dynamically loaded libraries, multi-lingual components, aggregate types, aliasing, sophisticated flows of control, and combinatorial explosion of execution paths make it practically impossible for compilers to obtain a holistic view of an application to apply its optimizations.

Orthogonal to static analysis is the coarse-grained runtime profiling that identifies program hot spots. Performance analysis tools such as HPCToolkit [5], VTune [49], gprof [39], OProfile [95], and CrayPAT [30] monitor code execution to identify hot code regions, idle CPU cycles, arithmetic intensity, and different level of cache misses, to name a few. These tools can recognize the utilization (saturation or underutilization) of hardware resources, but they cannot inform whether a resource is being used in a fruitful manner or not.

The solution to the limitations of static compiler analysis and coarse-grained profiling is a less commonly employed paradigm of fine-grained program monitoring. Unlike coarse-grained profilers, fine-grained analysis involves microscopic monitoring of each dynamic instruction, its operands, memory accesses, and runtime values. A key advantage of microscopic program-wide monitoring is that it can identify redundancies notwithstanding user-level program abstractions. Furthermore, as identified in prior work, RVN, and
for (int i = 0 ; i < N; i++) {
    /* Func() is side-effect free */
    A[i] = 2 * Func(i);
    /* use of A[i]. Line 3 is not a dead store */
    ... = A[i];
    /* A[i] gets the same value as after line 3 */
    A[i] = Func(i)+Func(i);
    /* use of A[i]. Line 7 is not a dead store */
    ... = A[i];
}
et al. [99] pointed out that approximate results by reusing similar values can significantly save computation operations, yielding more than a $2.5 \times$ speedup with tolerable accuracy loss. Thus, exploiting value locality shows promising performance gains by eliminating redundancies or approximating computations. However, this technique is overlooked by optimizing compilers; similarly, none of the existing coarse-grained or fine-grained profilers recognize the potential for approximate computations.

Value locality pervasively exists in several code bases, which opens a wide avenue for performance tuning. Table 3.2 summarizes the maximum redundancies we observed in each of SPEC CPU2006 integer and floating-point reference benchmarks, Rodinia suite, MineBench, and NWChem. Redundancies in loads, stores, and computations can be as high as 39%, 79%, and 82%, respectively.

In this work, we propose REDSPY, a fine-grained profiler to pinpoint and quantify value locality (exact and approximate) in executions. REDSPY works on fully optimized binary executables and instruments instructions with Intel Pin [75]. REDSPY attributes each redundancy instance to its provenance—a pair of instructions (one generating the old value and one re-generating the same value), their source lines along with their calling contexts. REDSPY presents the context pairs in the order of frequency of redundancies to easy investigating top inefficiencies. Guided by REDSPY, we are able to eliminate redundant operations in critical code bases and achieve speedups as high as $2.2 \times$. We also show our optimizations are architecture independent and demonstrate their benefits on multiple processor architectures and compilers.

We make the following contributions in this work:

- We develop a tool (REDSPY) to pinpoint redundancies arising from the temporal and spatial locality of values including the potential for approximate computing.

- We develop techniques that provide rich performance insights, which include metrics and provenance of redundancies that serve to focus on tuning code regions involved in high redundancies.

- We demonstrate significant speedups in important code bases by exploiting value locality identified by REDSPY.

- We tackle important implementation challenges related to aliasing, SIMD, and floating-point registers. We build a practical tool that ensures moderate profiling overhead. REDSPY is open sourced [1].

### 3.2 Related Work

We review the related work from two aspects. §3.2.1 reviews existing value profilers. §3.2.2 shows other approaches on eliminating redundant operations.

#### 3.2.1 Traditional Value Profiling

Lipasti et al. [69, 68] proposed value locality and exploited it in a hardware extension—the value prediction unit. Their work was concerned with same instruction frequently
loading the same value from memory and producing same value into a register. Lepak and Lipasti [62] introduced the concept of “silent stores”—stores that overwrite the value already existing in memory. Silent stores do not change the system state. They developed a hardware mechanism to “squash” silent stores by converting every store instruction into a three-operation sequence—a load, a comparison, and a conditional store (if the store is not silent). Their scheme resulted in 33% reduction in cache line write-back, and 6.3% speedup on average. In a follow-up work [63], the same authors repurposed the data-cache Error Checking and Correcting (ECC) code’s hardware logic, which allowed them to avoid a potentially expensive load operation introduced in the earlier scheme. Furthermore, they proposed exploiting idle cache read ports for store verification. The two new techniques in conjunction could detect more than 90% of silent stores.

There are more hardware approaches. Miguel et al. [82, 81] proposed hardware extensions to identify approximate load values; Yazdanbakhsh et al. proposed RFVP [118], a hardware approach to exploit approximate computation.

Our work differs from all these hardware-based approaches in the following ways: first, RedSpy is a pure software tool and does not need any hardware changes; second, RedSpy detects redundancies not only in loads and stores (cache or memory) but also in computations performed in processor registers and any combination of these and allows approximation in each case; third, while the hardware approaches attempt to silently hide inefficiencies, RedSpy aims to highlight code regions causing inefficiencies to help developers tune their code, which can lead to higher speedups.

Bell et al. [9] explored silent stores with source code analysis and compiler optimization levels. Like us, they inferred that the root cause of silent stores is often algorithmic in nature.

Calder et al. [14, 15, 35] proposed 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 locations. A variant of this value profiler is proposed in a later research [113]. Unlike RedSpy, their approach (1) does not identify spatial and approximate redundancies, (2) does not recognize redundancies when the value changes in the same storage location, and (3) does not provide calling context of instructions that have redundant values.

Muth et al. [84] proposed value profiling for code specialization. Their approach, however, identifies the redundant values in registers only. Oh et al. [92] automatically specialized loops in script programs based on patterns. They collected value profiles to identify static instructions that always produce the same value. Their approach cannot identify optimization opportunities for partially redundant values.

Chung et al. [25] developed a procedure-level value profiler, which identifies redundant values passed to the same function as parameters multiple times. Kamio and Masuhara [58] proposed a similar method-level value profiling in JAVA programs. These two approaches omit the redundancies that happen elsewhere, e.g., individual instructions or loops.

Burrows et al. [12] used hardware performance counters to sample values in Digital Continuous Profiling Infrastructure (DCPI) [7]. Their approach incurs low runtime overhead, but its sampling technique captures only the currently occurring value. It cannot identify redundancies since it does not maintain a history of values in a storage location.

Henry et al. proposed MAQAO VPROF [42], which profiles values in high-performance
computing code bases. VPROF monitors hot loops or functions and captures the frequencies of each value computed. However, VPROF requires extensive manual effort. VPROF does not capture calling contexts of redundant function calls. VPROF works only at function-level granularity, which is coarse-grained.

Unlike existing value profilers, RedSpy has four distinct features. First, RedSpy is the only value profiler that tracks the history of values occurring in a storage location, which allows it to recognize value redundancy. Second, it identifies and exploits both temporal and spatial value locality. Third, it provides rich information including calling contexts and redundancy metrics associated with program source code. Fourth, RedSpy not only identifies redundant computations but also explores opportunities for approximate computing.

3.2.2 Other Redundancy Optimization Techniques

Compilers employ a variety of techniques, e.g., value numbering, constant propagation, and partial redundancy elimination, to eliminate redundant operations. Elaborating these compiler techniques is outside the scope of this work. Beyond these classical compiler techniques, there exist many static analysis techniques [28, 29, 76, 45] to identify redundant computation. However, these static approaches suffer from limitations related to aliasing, optimization scope, and insensitivity to execution contexts. In this section, we only review profiling techniques beyond value profiling.

Chabbi and Mellor-Crummey [20] developed DeadSpy to identify execution-wide dead stores. DeadSpy tracks every memory operation to pinpoint a store operation that is not loaded before a subsequent store to the same location. They associate pairs of instructions involved in a “dead write” with their calling contexts and source code locations to guide manual program optimizations. DeadSpy is value agnostic. Unlike DeadSpy, RedSpy detects redundancies arising in computations (registers) and data movement (memory) operations.

Butts et al. [13] developed a hardware-based method to track CPU-bound operations and identify useless computations in a program. They do not provide detailed feedback for optimization. In contrast, RedSpy uses a software method to monitor memory operations and provides rich optimization guidance.

Our previous work, RVN, assigns symbolic values to dynamic instructions and identifies redundancies on the fly. RVN effectively performs symbolic equivalence at runtime but does not inspect actual runtime-generated values. Hence, RVN misses out on certain opportunities that RedSpy can detect by explicitly inspecting values generated at runtime. Furthermore, RVN essentially performs a tracing of instructions and incurs heavy space and time overheads, whereas RedSpy performs profiling and hence incurs much less space and time overheads.

3.3 Methodology

At a high level, RedSpy tracks values present in every storage location (registers and memory) and checks if a newly generated value is same as the one that already existed at
Listing 3.2: Code example of temporal value locality.

```c
int Temp(int a, int b){
    int m = a * a;
    int n = b * b;
    int v1 = m - n;
    int c = a - b;
    int d = a + b;
    v1 = c * d;
    return v1;
}
```

Listing 3.3: Code example of spatial value locality.

```c
void Spat(){
    int *a = new int[N];
    int *b = new int[N];
    for (i=0; i<N; ++i){
        a[i] = i/2 + 1;
        b[i] = Foo(a[i]);
    }
}
```

the same storage location. We relax the “same location” to “nearby 'locations’ and ‘same value” to “approximately same” values. REDSpy uses Intel’s Pin dynamic-instrumentation framework [75] to instrument binaries for runtime value tracking.

Listing 3.2 shows an example with temporal value locality. The values assigned to v1 at line 4 and 7 are the same because of the identity $a \times a - b \times b == (a - b) \times (a + b)$. Thus, the value of v1 shows temporal locality, resulting in redundant computations at line 5-7. To identify temporal value locality, REDSpy inspects the value(s) generated by each instruction instance, whether computations or data movement and compares the previous value at the target location(s), whether memory or register, with the newly generated value. If the two values are the same, then REDSpy flags such operation pairs as redundant. On each instance of redundancy, REDSpy records the pair `<previous calling context, current calling context>` into a table of redundancies. Associated with each context pair is a frequency metric that records how often the same pair produces redundant values. We describe the details of our metric later in this section. The context pairs with high frequencies are the targets of optimization.

Relaxing the logic to detect approximation is conceptually straightforward: instead of bitwise equivalence, REDSpy checks if the old and new values are within a threshold percentage difference. REDSpy applies the approximation only for floating-point computations since integer values may have other semantic meanings in a program, e.g., branch decisions, switch tables, among others.

Challenges in identifying temporal value locality reside in handling complex instructions such as SIMD in modern architectures, handling registers with aliases (e.g., EAX vs. AX in x86), segregating floating-point computations from the rest, and maintaining a moderate runtime overhead of the analysis. §3.4.1 details the techniques to address these challenges.

Listing 3.3 shows spatial value locality, where values computed for $a[i]$ and $a[i + 1]$ ($i = 0, 2, 4, ...$) are always the same. Thus, the computation on $a[i + 1]$ at line 7 is always redundant. To identify spatial value locality, REDSpy investigates the values stored in a segment of memory, e.g., an array. After initialization or a series of intensive writes, REDSpy compares the values of adjacent memory elements. If most of these values are identical, then they exhibit the spatial locality. REDSpy also periodically checks register contents for the uniqueness of its values. Relaxing to value approximation is similar to that of temporal value locality, which checks whether the adjacent values are within a threshold percentage difference. §3.4.2 offers more details of spatial value locality.
Exploiting Value Locality. RedSpy is a profiler, which only pinpoints locality. Exploiting value locality requires code transformation. If a redundancy is due to temporal value locality, one can remove the computation that repeatedly produces the same or similar values to the same storage location. If a redundancy is due to spatial value locality, one can reuse the computation result from one array element to other elements, as SPMD computation on different array elements is often the same. Redundancies captured at runtime may be input specific or input agnostic. The application developer needs to make the design choice on how to optimize the code. We show several examples of how we exploit value locality in our case studies in §3.7.

Limitations: First, RedSpy does not distinguish optimizable vs. non-optimizable value redundancies. RedSpy may have false positives where the values are accidentally identical. However, we easily filter accidental value collisions by attributing redundancies to the calling context pairs responsible for generating the last and current values. Thus, only those contexts that frequently lead to redundancies are optimization candidates. Only a handful of top contexts account for a vast majority of redundancy found in executions; it is not worth exploring contexts that contribute to a small fraction of the overall redundancy. Second, RedSpy detects only intra-thread redundancies. We can extend our analysis to detect inter-thread redundancies by introducing extra synchronization in RedSpy analysis routines for memory operations—required to ensure atomicity of analysis routines and application code.

3.4 Detection of Value Redundancies

3.4.1 Temporal Redundancy

Temporal redundancy may occur both in registers and memory. Our implementation differs based on whether the target location of instruction is a register or the memory. For example, the target location of a load instruction is a register; the target location of a store instruction is the memory; the target location of a register-to-register computation is a register. Furthermore, x86 poses other complex scenarios since the target location of some computations can also be memory. We use the term “write” to mean generation of a new value either into a register or memory. For example, loading a value from memory into a register is a “register write”.

To flag an instance of a write as redundant, we need to instrument every instruction and analyze the newly written value immediately after the instruction execution. Intel’s Pin provides facilities to identify the register or effective memory address along with the size of the operation to its analysis routines and allows tools to instrument either before or after any instruction. An inspection of the target location performed immediately after an instruction would tell us the newly generated value at the target location. The challenge, however, is in knowing the previous value at the same location. There are two possibilities on how one can capture the previous value:

1. Option 1: Insert instrumentation before an instruction to capture the value just before the instruction execution and store it in a temporary buffer, or
2. Option 2: Record the last written value of every location into a “shadow” memory location.

Option 1 has relatively higher time overhead since it would instrument both before (IPOINT-before in Pin terminology) and after (IPOINT-after in Pin terminology) an instruction but it has an O(1) space overhead. Option 2 has relatively lower time overhead since it would instrument only after an instruction, but it has O(N) space overhead where N is the number of unique storage bytes accessed in the program.

We use the best of both strategies. Since memory writes are relatively less frequent compared to register writes but the amount of addressable memory is very large, we use Option 1 when the target of a write is a memory location. Since register writes are very frequent, and the number of registers is much smaller, we use Option 2 when the target of a write is a register.

**Memory Temporal Redundancy.** As stated before, we insert instrumentation before and after a memory write operation to identify redundancies. A complication with this strategy is that for instructions that have the auto-increment/decrement [50] semantics, the effective address computed immediately after an instruction is not same as the one used by the instruction. To be precise, Pin’s IARG_MEMORYWRITE_EA argument to an analysis function when used with IPOINT_AFTER location computes the effective address after the instruction, not the effective address used by the instruction itself. Thus, our analysis routine executed after an instruction would get an incorrect effective address.

To make Option 1 work, we need to capture the effective address e and the value v’ at e before an instruction’s execution into a buffer, say b. With this information, the analysis performed immediately after the instruction can compare the new value v at e with the previous value v’ captured in b for the number of bytes that the instruction writes and flag redundancy iff v=v’ (v≈v’ for approximate computations).

A few rare instructions may update more than one memory location. To accommodate multi-memory location updates, we dedicate multiple buffers to remember the effective addresses and the old values. We dedicate eight such buffers. In our experience, we have never encountered any instruction updating more than four disjoint locations in the x86_64 architecture. The largest value written is 512 bytes in the fxsave instruction. Thus, the total buffer size for all fields is ~4KB, which is much less compared to shadowing each byte.

**Register Temporal Redundancy.** As stated before, we insert instrumentation only after a register write operation to identify register-level redundancies. On most occasions, REDSPY used the lightweight IARG_REG_VALUE argument-passing technique in Pin, which presents the register value at runtime to an analysis function.  

X86 architectures have aliased registers where different segments of a register can be accessed via different register names. For example, the register AL is the lower 8 bits of AX, AH is the higher 8 bits of AX, AX is the lower 16 bits of EAX, and EAX is the lower 32 bits of RAX. If an instruction updates AL, it affects the subsequent value read at AX, EAX, and RAX, but it does not affect the value read at AH. If an instruction updates AH, it affects the subsequent value read at AX, EAX, and RAX, but it does not affect the value read at AL. If
an instruction updates \texttt{AX}, \texttt{EAX}, or \texttt{RAX}, it affects the subsequent values read at \texttt{AL}, \texttt{AH}, \texttt{AX}, \texttt{EAX}, and \texttt{RAX}.

To hold the previous values of registers, we dedicate shadow value registers equal in number to the physical registers on the target processor. We handle register aliasing by creating aliases in the shadow registers so as to mirror the exact aliasing present in the physical registers. For example, we maintain only one real shadow register \texttt{shadow.A} of 64 bits for the entire alias group \texttt{AL}, \texttt{AH}, \texttt{AX}, \texttt{EAX}, and \texttt{RAX}. The writes to \texttt{AL}, \texttt{AX}, \texttt{EAX}, and \texttt{RAX} simply result in different sized writes into \texttt{shadow.A}. Writes to \texttt{AH} is a special case that writes to bits 8-15 of \texttt{shadow.A}.

\textbf{Value Approximation in Temporal Redundancy.} To accommodate approximate redundancy, we relax the strict “equal to” operation to “approximately equal to” for floating-point operations. The approximation can be either ignoring a few lower-order bits or allowing a threshold percentage accuracy. We implement the threshold-based approximation and set the accuracy to be 99\% in our evaluation. The threshold is a user tunable parameter.

On modern x86 processors, the floating-point operations can be performed either on the x87 coprocessor or via the SIMD engine. The x87 coprocessor uses an 80-bit extended precision representation whereas SIMD engines can work on either 32-bit single precision or 64-bit double precision quantities. \textsc{RedSpy} uses XED \cite{XED} to decode an instruction and classify it into an x87, single-precision, double-precision, or non-floating-point category.

If an instruction falls into the x87 category, we inspect the 80-bit registers that are the target of an x87 instruction. We check the higher 16 bits (sign bit and exponent) for exact equality and check the lower 64 bits (significand) to be within the threshold of accuracy. Unfortunately, Pin does not allow reading non-general-purpose registers with its lightweight \texttt{IARG\_REG\_VALUE} mechanism, hence \textsc{RedSpy} uses \texttt{IARG\_REG\_CONST\_REFERENCE} to read the top of the x87 stack. A few x87 instructions operate on more than one register of the x87 coprocessor stack \cite{XED}, which cannot even be read via \texttt{IARG\_REG\_CONST\_REFERENCE} in Pin. In such situations, \textsc{RedSpy} resorts to using Pin’s heavyweight API \texttt{PIN\_GetContextRegval}. Fortunately, the use of x87 instructions is rare on code-generated for modern x86_64 processors.

If an instruction is a SIMD single-precision or double-precision category, \textsc{RedSpy} fetches the generated 128-bit (XMM), 256-bit (YMM), or 512-bit (ZMM) values via Pin’s \texttt{IARG\_REG\_CONST\_REFERENCE} argument passing. \textsc{RedSpy}, then compares the previously stored value with the current value. For efficiency, \textsc{RedSpy} uses SIMD for approximate equality comparison, which involves a SIMD subtraction followed by a SIMD division. Subsequently, \textsc{RedSpy} reports redundancies found in the constituent SIMD components separately.

All through, we optimize the instrumentation for the common case and accomplish all specialization with C++ template meta-programming and template specialization to produce minimal instrumentation code tailored for each kind of instruction. This scheme lowers runtime overhead.

\textbf{Metric of Temporal Redundancy.} \textsc{RedSpy} measures the volume of temporal redundancy in an execution as the fraction of bytes that are redundantly produced to the total
bytes produced by the program. More specifically, if an instruction produces a value $V$ of length $N$ bytes at its target location $L$ (whether memory or register) and if and only if the previous value ($V'$) at $L$ was already $V$, i.e., all $N$ bytes match, then RedSpy treats the currently produced value as a temporal redundancy of $N$ bytes. If fewer than $N$ bytes match, then it is not considered as redundant. Intuitively, sub-write-size redundancy is not actionable by the programmer. Note, however, that the previous value $V'$ of $N$ bytes might have been generated by multiple shorter writes, a single write longer than $N$ bytes, or more commonly a single write of $N$ bytes.

A redundant computation is usually cheaper than a redundant data movement. Furthermore, the volume of data generated within registers is far more than the volume of data moved between CPUs and memory. Hence, we classify the redundancy into load redundancy, store redundancy, and register redundancy. RedSpy provisions for approximate computation by allowing new values generated in floating-point (FP) operations to approximately match the previously present values. RedSpy decomposes the redundancy into “precise” vs. “approximate”. The definitions below show how redundancy is decomposed into various categories:

\[
R_{\text{load}}^{\text{precise}} = \frac{\sum \text{non-FP bytes redundantly loaded from memory}}{\sum \text{non-FP bytes loaded from memory}}
\]

\[
R_{\text{load}}^{\text{appx}} = \frac{\sum \text{FP bytes redundantly loaded from memory}}{\sum \text{FP bytes loaded from memory}}
\]

\[
R_{\text{store}}^{\text{precise}} = \frac{\sum \text{non-FP bytes redundantly written to memory}}{\sum \text{non-FP bytes written to memory}}
\]

\[
R_{\text{store}}^{\text{appx}} = \frac{\sum \text{FP bytes redundantly written to memory}}{\sum \text{FP bytes written to memory}}
\]

\[
R_{\text{reg}}^{\text{precise}} = \frac{\sum \text{non-FP bytes redundantly generated in registers}}{\sum \text{non-FP bytes generated in registers}}
\]

\[
R_{\text{reg}}^{\text{appx}} = \frac{\sum \text{FP bytes redundantly computed in registers}}{\sum \text{FP bytes computed in registers}}
\]

Overall redundancy is:

\[
R_{\text{total}} = \frac{\sum \text{bytes of value redundantly generated}}{\sum \text{bytes of value generated}}
\]

In addition to measuring the volume of redundant data, RedSpy also computes the fraction of instructions involved in redundant computations as below:

\[
R_{\text{ins}}^{\text{precise}} = \frac{\sum \text{Non-FP dynamic instructions generating redundant value}}{\sum \text{Dynamic instructions executed}}
\]

\[
R_{\text{ins}}^{\text{appx}} = \frac{\sum \text{FP dynamic instructions generating redundant value}}{\sum \text{Dynamic instructions executed}}
\]

### 3.4.2 Spatial Redundancy

Spatial value redundancy ensues when same values appear in the neighborhood of storage locations. Spatial redundancy can also occur in memory or registers. Inspecting the neighborhood on each write, however, is extremely expensive and creates noisy results.
Instead of automatically inspecting neighborhood locations on each write, we let the application programmer insert “instrumentation hooks” that tell RedSpy when to inspect a neighborhood of locations.

**Spatial Redundancy in Memory.** For spatial memory redundancy, RedSpy focuses on array type elements. RedSpy knows the data object that any memory access belongs to and the size of the entire object—this is captured by performing a binary analysis on the static data present in the binary and intercepting memory allocation routines such as `malloc`, `calloc`, `realloc`, `posix_memalign`, and `free` for dynamically allocated data. RedSpy, however, does not know the size of each array element, e.g., the size of a structure and its fields in an array of structures. Without knowing the size of an element, it is not possible to inspect the neighboring elements. RedSpy relies on explicit user instrumentation to inform the starting address, stride, and size of an element whenever it wants to check for spatial redundancy; the size of the entire array is not necessary. We recommend inserting the hook at the end of a computation phase once all array elements are updated, e.g., after initialization or after a time step.

RedSpy’s spatial analysis scans the entire array of elements and looks for the uniqueness of the values in the array. RedSpy computes the redundancy in an array as the fraction of the number of non-unique values to the total number of elements:

\[
S = \frac{\text{Total elements} - \text{Unique elements}}{\text{Total elements}}
\]

After analysis, if \( S \) is above a set threshold (20% in our implementation), RedSpy records and reports such redundancies. Similar to temporal redundancy metric, we decompose \( S \) into precise and approximate redundancies.

**Spatial Redundancy in Registers.** For spatial register redundancy, RedSpy groups architectural registers into general-purpose, floating-point, and SIMD. At user-chosen hook points, RedSpy inspects each register group and computes redundancy in each group based on the uniqueness of the values.

**Approximating Spatial Redundancy.** We provision for approximation in spatial redundancies by relaxing our comparison to be within a threshold of accuracy from the base value (99% in our experiments). The user hook is responsible for informing whether to perform an approximate comparison for memory locations and SIMD registers; floating-point registers are always considered for approximate comparison.

### 3.5 Recording and Reporting Redundancy

In addition to identifying value redundancy, as a profiler, RedSpy needs to record the provenance of redundancies. Showing the source lines and full calling contexts of the previous write and the new write that overwrites it with (approximately) the same value offers detailed insights into diagnosing and understanding the causes of redundancies such
as algorithmic and data structure choice. We have found calling context to be very useful, especially when redundancy manifests deep inside a common library (e.g., `memset`) called from many call sites in a large code base. With this objective, RedSpy needs to capture the full calling context on each write operation so that it can be used when a redundancy may be detected subsequently. RedSpy uses CCTLib [18] for efficiently collecting calling contexts on every instruction and associates them with source code using the DWARF [4] information. RedSpy can query for the calling context for every monitored instruction instance and in return, it gets a four-byte `ContextHandle` from CCTLib.

Once a temporal redundancy happens, RedSpy records the pair of calling contexts involved in the redundancy. The pair has two 32-bit components—the calling context of the last write operation and the calling context of the current write operation. We maintain a hash table where the key is a 64-bit context pair and the value is the redundancy metric—bytes redundant in that context pair.

**Attributing Memory-temporal Redundancies.** RedSpy dedicates a shadow memory of four bytes for each memory byte the program touches and stores the current `ContextHandle` obtained from CCTLib in the shadow memory. RedSpy uses a previously developed efficient two-level page-table mapping strategy [20] to store and retrieve the `ContextHandle` in a constant time. If a memory write instruction is found to be redundant, RedSpy reports the context pair involved in the redundancy. With this logic, whenever a redundancy is found while writing n bytes to a memory address, say M, RedSpy can immediately fetch the n previous contexts from `shadow[M:M+n-1]` that caused the formation of the same value at the same location. Often, these n contexts are the same, which allows us to specialize our code.

**Attributing Register-temporal Redundancies.** RedSpy dedicates a set of shadow registers (shadow context registers) each of which maintains the calling context of the last write to each register. Handling aliases in shadow context registers is more involved. We treat AX, EAX, and RAX as single `super register`. We dedicate a single shadow context register $C_s$ for a super group. We dedicate one shadow context register $C_{al}$ for AL and another shadow context register $C_{ah}$ for AH. If an instruction writes to a super group register, then RedSpy records the calling context in its corresponding shadow context register $C_s$ and in addition it records the context in $C_{al}$ and $C_{ah}$ since updating a super register implicitly updates the other two registers.

If an instruction writes to AX, then RedSpy records the calling context in its corresponding shadow context register $C_{al}$ and in addition it records the context in $C_s$ since updating AL implicitly updates the super registers (but not AH). If an instruction writes to AH, then RedSpy records the calling context in its corresponding shadow context register $C_{ah}$ and in addition it records the context in $C_s$ since updating AH implicitly updates the super registers. Other aliased registers such as RBX, RCX, and RDX are handled in the same way. With this logic, whenever a redundancy is found while writing to a register, say $R$, the corresponding context stored in the shadow context register $C_R$ immediately fetches the previous calling context that had created the same value at the same location. We ignore special cases for multiple different writes combining to form a larger value that becomes redundant since it incurs more bookkeeping and runtime overhead.
**Attributing Memory-spatial Redundancies.** If the spatial redundancy in a user-chosen hook location is above a threshold, RedSpy records the location of redundancy (the calling context of the hook location) and the data object that exhibits the redundancy along with the number of bytes redundant. The data object will be the name for static objects or the calling context of the allocation site for dynamically allocated objects.

**Attributing Register-spatial Redundancies.** If the register-spatial redundancy percentages in a register group is above a user-specified threshold (e.g., 20%) at a user specified hook point, RedSpy records the redundancy percentages in each such register group and associates them with the calling context of the hook point.

**Sampling for Low Overhead.** As a fine-grained analyzer, RedSpy has a relatively high runtime overhead (80× on average). RedSpy adopts a bursty sampling mechanism to further reduce its overhead [119]. 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. From our experiment, 1% sampling with WINDOW_ENABLE=1 million and WINDOW_DISABLE=100 million yields a good balance between overhead and analysis accuracy. With bursty sampling, RedSpy aggregates the redundancy found only when the sampling is enabled. For example,

\[
\hat{R}_{\text{load}}^{\text{precise}} = \sum_{i=0}^{N} \frac{\sum \text{non-FP bytes redundantly loaded in sample } i}{\sum \text{non-FP bytes loaded in sample } i}
\]

\[
\hat{R}_{\text{store}}, \hat{R}_{\text{reg}}, \hat{R}_{\text{load}}^{\text{appx}}, \hat{R}_{\text{store}}^{\text{appx}}, \hat{R}_{\text{reg}}^{\text{appx}}, \hat{R}_{\text{ins}}^{\text{precise}}, \text{ and } \hat{R}_{\text{ins}}^{\text{appx}} \text{ are analogously defined.}
\]

We evaluate the sampling accuracy in the next section. Values are not carried over from one interval to the next—shadow registers are cleared at the start of a new monitoring interval.

Start and end of sampling also serve as the points where RedSpy inspects the register spatial redundancies without requiring a user hook.

**Handling Parallel Programs.** RedSpy works for both multi-threaded and multi-processed executions. RedSpy monitors each thread and process individually without introducing any synchronization and hence its analysis scales perfectly. A post-mortem profile merging phase aggregates metrics in different calling contexts from different threads and processes albeit retaining individual thread’s contribution to identify imbalance, if any.

**Presentation.** RedSpy apportions redundancy to its contributing context pairs. On program termination, RedSpy sorts the redundancies accumulated in different context pairs and presents them in the order of their contribution. Users, typically, need to inspect only a top few (3-5) redundancy pairs to identify significant causes of inefficiencies, if any.
3.6 Experiments

We evaluate RedSpy on four platforms: Intel SandyBridge, AMD Opteron, Intel Xeon Phi, and IBM POWER7. Table 3.1 shows the machine configurations and the compilers used. We evaluate RedSpy on the following programs and benchmarks: SPEC CPU2006 [104] integer and floating-point benchmarks, Rodinia [91] and MineBench [3] parallel benchmark codes, and NWChem-6.3 [111] MPI computational chemistry code. For SPEC CPU2006 we use the reference inputs; for Rodinia and MineBench, we use the default datasets released with the suites. For NWChem, we use the QM-CC aug-cc-pvdz input, which spends most cycles in computation. We use profile-guided optimization (PGO) as our baseline so as to detect redundancies remaining only after applying any automatic optimization. An exception is NWChem, which has a complicated build process; hence we use only -O3 for NWChem. We use the same input(s) both for PGO training and testing.

We run every parallel program with four threads (or processes for NWChem) pinned to cores on the same socket, and average the numbers across all threads (or processes). We do not use the simultaneous multi-threading (SMT) feature. We collect the previously mentioned metrics and profiling overhead in these benchmarks. We also explore the sampling accuracy on a subset of benchmarks.

Table 3.1: Machine configurations.

<table>
<thead>
<tr>
<th>Machines</th>
<th>Intel-SandyBridge</th>
<th>AMD</th>
<th>Intel-Xeon-Phi</th>
<th>IBM-POWER7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Xeon <a href="mailto:E5-4650@2.7GHz">E5-4650@2.7GHz</a></td>
<td>Opteron <a href="mailto:6108@1.6GHz">6108@1.6GHz</a></td>
<td>Xeon Phi <a href="mailto:31000@1.1GHz">31000@1.1GHz</a></td>
<td><a href="mailto:POWER7@3.5GHz">POWER7@3.5GHz</a></td>
</tr>
<tr>
<td>SMT x Cores</td>
<td>2 x 8</td>
<td>1 x 12</td>
<td>4 x 57</td>
<td>4 x 8</td>
</tr>
<tr>
<td>L1/L2/L3 Cache</td>
<td>32KB/256KB/20MB</td>
<td>64KB/512KB/10MB</td>
<td>32KB/612KB/NA</td>
<td>32KB/256KB/4MB</td>
</tr>
<tr>
<td>Memory</td>
<td>256GB DDR3</td>
<td>128GB DDR3</td>
<td>6GB GDDR5</td>
<td>256GB DDR3</td>
</tr>
<tr>
<td>Compiler</td>
<td>gcc 4.8.5 -O3 PG0</td>
<td>gcc 4.8.3 -O3 PG0</td>
<td>icc 15.0.0 -O3 PG0</td>
<td>xlc 13.1.0 -O2 PDF</td>
</tr>
</tbody>
</table>

Figure 3.1: Breakdown of redundant bytes written in different benchmark suites.

Volume of Redundancy. Figure 3.1 shows the temporal redundancy observed in the aforementioned benchmark suites on Intel SandyBridge machine compiled with gcc -O3 PGO. Each bar with different colors in the figure quantifies the value redundancy and its decomposition in individual programs. The bar of GeoMean over all the benchmarks reveals that among the total bytes written, 4.5% are redundant integer register writes, 4.5%
We observe that redundancies can be decomposed into floating-point stores, resulting in the total value redundancy as high as 17%. Among these, 3.4% are redundant floating-point loads, and 2.3% are redundant floating-point register writes. Table 3.2 shows the breakdown of temporal redundant bytes and redundant instructions in different benchmark suites.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Fraction Redundant</th>
<th>Redundant Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>gromacs</td>
<td>42.7</td>
<td>34.3</td>
</tr>
<tr>
<td>milc</td>
<td>93.9</td>
<td>89.5</td>
</tr>
<tr>
<td>xalancbmk</td>
<td>41.4</td>
<td>34.3</td>
</tr>
<tr>
<td>SPEC-INT</td>
<td>4.7</td>
<td>4.7</td>
</tr>
</tbody>
</table>

Table 3.2: Breakdown of temporal redundant bytes and redundant instructions in different benchmark suites. Are redundant integer loads, 3% are redundant integer stores, 1.7% are redundant floating-point register writes, 3.4% are redundant floating-point loads, and 2.3% are redundant floating-point stores, resulting in the total value redundancy as high as 17%.

Table 3.2 further breaks down the temporal redundancy in categories. The volume of temporal redundancy is classified into accurate (col 2-8) and approximate (col 9-15). The Fraction column (col 2-4) under precise category shows the percentage breakdowns of data generated within registers (col 2), loaded from memory (col 3), and stored to memory (col 4) via non-floating-point operations. The Fraction column (col 9-11) under approximate category analogously shows the breakdown for floating-point operations. Column 5, 6, 7, and 8 respectively decompose the observed precise redundancies into $R_{\text{precise}}$, $R_{\text{load}}$, $R_{\text{store}}$, and $R_{\text{ins}}$ components. Column 12, 13, 14, and 15, respectively, decompose the observed approximate redundancies into $R_{\text{approx}}$, $R_{\text{load}}$, $R_{\text{store}}$, and $R_{\text{ins}}$ components. Bold texts summarize the maximum redundancy observed in each benchmark suite.

There are several benchmarks with a high volume of redundancies (e.g., gcc and h264ref). For example, 83% $R_{\text{precise}}$ in gcc is because of a repeated zero initialization of a large array, which has been well studied elsewhere [20]. We observe that $R_{\text{store}}$ is often
higher than $R_{reg}$ and $R_{load}$. $R_{ins}$ is always lower than the rest since this metric is computed for all instructions without decomposing the contributions from different categories. $R_{precise}$ is always higher than $R_{approx}$ in SPEC CPU2006 integer benchmarks compared to the floating-point benchmarks and vice-versa.

h264ref shows 13% load redundancy and 14% store redundancy; NWChem shows 89% store redundancy. We investigate them in the next section. A few benchmarks show high redundant loads; e.g., lavaMD has 26% data generation due to loads of which 84% are redundant loads.

We do not report spatial redundancy because it depends on programmers choosing their hook placement. In §3.7, we, however, show two case studies, Hotspot and Particle_filter, which have high spatial redundancy.

Sampling Accuracy. To assess the accuracy of bursty sampling technique adopted by RedSpy, we selected eight representative benchmarks (bzip2, bwaves, zeusmp, gromacs, cactusADM, backdrop, lavaMD, particlefilter). The benchmarks cover integer, floating-point, iterative, non-iterative, HPC, and non-HPC benchmarks with high and low redundancies. We compare RedSpy’s redundancy volume with and without bursty sampling in Table 3.3. At the previously mentioned 1% sampling rate, all these benchmarks show negligible variation from full monitoring. We have further analyzed pairs of redundancies that RedSpy reports in both settings; the rank ordering of the top ten redundancy locations is almost always the same. An exception is gromacs, which shows 50% variation in its top contributors since (1) the total redundancy is very small, and (2) the top contributors account for less than 1% redundancy. With this data, we infer that RedSpy’s bursty sampling strategy does not lose accuracy in detecting and reporting value redundancies where they matter.

RedSpy Overhead. Table 3.4 shows the space and time overhead of RedSpy on SPEC CPU2006 benchmarks on the Intel SandyBridge platform. The average time and space overheads are $12 \times$ and $9 \times$, respectively. Some benchmarks, e.g., gcc, suffer from high memory overhead, which is due to the high space overhead of CCTLib [18] for applications that have a deep and large calling context tree. For most benchmarks with moderate call chains, RedSpy incurs $\sim 5 \times$ memory overhead, on average. Time overhead is usually high when (1) the redundancy is high since it results in more hash-table updates, or (2) the
3.2 Reduction Techniques

In this section, we evaluate a few cases with high value redundancy seen in the previous section, investigate the causes of redundancies, and optimize them. Table 3.3 overviews the performance improvements after optimizing redundancies seen in several programs on various platforms. For parallel programs (LavaMD, Backprop, Hotspot, and NWChem), we show the improvements when the application is run with all cores on each machine. The machine configurations are the same as shown in Table 3.1. As before, we use PGO for the baseline code and also for the code after our manual transformations. The training set used for PGO is the same input used for testing in all case studies. We do not apply PGO to NWChem due to its complicated build process.

From Table 3.3, it is evident that RedSpy can guide exploiting value locality in various programs yielding significant performance gains. In addition to time savings, Table 3.4 shows energy reduction (measured by RAPL [114] on Intel SandyBridge).

In the following subsections, we elaborate on how we employed RedSpy to identify redundancies in these codes and also discuss our optimization techniques. At the end of this section, we compare the ability of RedSpy with existing software-based redundancy elimination techniques described in §3.2, including (1) DeadSpy [20], which identifies dead stores, (2) RVN, which pinpoints redundant computation via symbolic execution, (3) ParaProx [99], a compiler technique to identify approximate computing opportunities in OpenCL codes, and (4) LLVM-ThinLTO [56], a link-time optimization technique across

---

### Table 3.4: RedSpy’s space and time overheads in the unit of times (⇥) on SPEC CPU2006 benchmarks.

<table>
<thead>
<tr>
<th>Redundancy types</th>
<th>Programs</th>
<th>Problematic procedures:loops</th>
<th>Intel SandyBridge</th>
<th>AMD WS</th>
<th>Xeon Phi WS</th>
<th>POWER 7 WS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Temporal</td>
<td>464, h264ref</td>
<td>mv-search.c:loop(394)</td>
<td>1.34⇥</td>
<td>23%</td>
<td>1.36⇥</td>
<td>1.26⇥</td>
</tr>
<tr>
<td></td>
<td>backprop</td>
<td>bprop_adjust_weights</td>
<td>1.01⇥</td>
<td>13%</td>
<td>1.14⇥</td>
<td>1.00⇥</td>
</tr>
<tr>
<td></td>
<td>NWChem†</td>
<td>tcc_mathc:trans.F:240</td>
<td>1.19⇥</td>
<td>9%</td>
<td>1.53⇥</td>
<td>—</td>
</tr>
<tr>
<td>Memory Spatial</td>
<td>particlefilter</td>
<td>particle_filter.c:loop(237)</td>
<td>1.10⇥</td>
<td>8%</td>
<td>1.04⇥</td>
<td>1.01⇥</td>
</tr>
<tr>
<td>Register Temporal</td>
<td>lavaMD</td>
<td>kernel_cpu.c:loop(117)</td>
<td>1.50⇥</td>
<td>37%</td>
<td>1.64⇥</td>
<td>1.34⇥</td>
</tr>
<tr>
<td>Spatial Approximation</td>
<td>hotspot</td>
<td>hotspot_openmp.cpp:loop(44)</td>
<td>2.21⇥</td>
<td>69%</td>
<td>2.19⇥</td>
<td>1.16⇥</td>
</tr>
</tbody>
</table>

†WS means whole-program speedup due to redundancy elimination while PS means whole-program power saving.

NWChem was run only on SandyBridge and AMD without PGO due to its highly complex and laborious installation procedure.

Table 3.5: Overview of performance improvement guided by RedSpy on different platforms.

<table>
<thead>
<tr>
<th>Program</th>
<th>Time Overhead</th>
<th>Space Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip2</td>
<td>19.207</td>
<td>5.627</td>
</tr>
<tr>
<td>gcc</td>
<td>14.123</td>
<td>4.198</td>
</tr>
<tr>
<td>md</td>
<td>11.514</td>
<td>10.144</td>
</tr>
<tr>
<td>namd</td>
<td>8.768</td>
<td>5.148</td>
</tr>
<tr>
<td>astar</td>
<td>8.816</td>
<td>8.166</td>
</tr>
<tr>
<td>bzip2</td>
<td>7.816</td>
<td>7.166</td>
</tr>
<tr>
<td>astar</td>
<td>4.198</td>
<td>4.198</td>
</tr>
<tr>
<td>bzip2</td>
<td>2.198</td>
<td>2.198</td>
</tr>
<tr>
<td>astar</td>
<td>1.198</td>
<td>1.198</td>
</tr>
<tr>
<td>bzip2</td>
<td>0.198</td>
<td>0.198</td>
</tr>
<tr>
<td>astar</td>
<td>0.198</td>
<td>0.198</td>
</tr>
</tbody>
</table>

Instruction mix has more SIMD or x86 instructions, which require heavyweight Pin APIs to pass runtime values to RedSpy’s analysis routines.

3.7 Case Studies

In this section, we evaluate a few cases with high value redundancy seen in the previous section, investigate the causes of redundancies, and optimize them. Table 3.3 overviews the performance improvements after optimizing redundancies seen in several programs on various platforms. For parallel programs (LavaMD, Backprop, Hotspot, and NWChem), we show the improvements when the application is run with all cores on each machine. The machine configurations are the same as shown in Table 3.1. As before, we use PGO for the baseline code and also for the code after our manual transformations. The training set used for PGO is the same input used for testing in all case studies. We do not apply PGO to NWChem due to its complicated build process.

From Table 3.3, it is evident that RedSpy can guide exploiting value locality in various programs yielding significant performance gains. In addition to time savings, Table 3.4 shows energy reduction (measured by RAPL [114] on Intel SandyBridge).

In the following subsections, we elaborate on how we employed RedSpy to identify redundancies in these codes and also discuss our optimization techniques. At the end of this section, we compare the ability of RedSpy with existing software-based redundancy elimination techniques described in §3.2, including (1) DeadSpy [20], which identifies dead stores, (2) RVN, which pinpoints redundant computation via symbolic execution, (3) ParaProx [99], a compiler technique to identify approximate computing opportunities in OpenCL codes, and (4) LLVM-ThinLTO [56], a link-time optimization technique across
different compilation units.

3.7.1 SPEC CPU2006 h264ref

h264ref is a reference implementation of the H.264 advanced video coding standard, a sequential C code. RedSpy reports \( \sim 39\% \) bytes are loaded from memory to registers, of which 13\% are redundant. Figure 3.2 shows the top pairs with calling contexts involved in the load redundancy. Both contexts happen to be the same location. Listing 3.4 shows function SetupFastFullPelSearch in file mv-search.c. This surrounding loop nest accounts for 55\% of the total running time.

The function pointer PelYline\(_{11}\) is assigned to either Fastline16Y\(_{11}\) or UMVLine16Y\(_{11}\). Both of these functions accept \texttt{abs.x}, \texttt{img.height}, and \texttt{img.width} as their arguments, whose values are loop invariants in the two-level loop nest from Line 417 to 420. Thus, at the call site on line 419, the same values are loaded for usage in the callee resulting in a large number of redundant loads. In addition, there is significant store redundancy because of the same values being written to the stack (not shown).

The compiler fails to eliminate this redundancy since the callee is invoked via a function pointer and the callee routines are not present in the same file.

Despite the cache locality, the redundancy is expensive since most of the time is spent in this loop nest. We eliminate the redundancy by inlining the function calls: we create two loop nests each one with a direct function call instead of using function pointers and move the target functions to the same compilation unit as their call site. This optimization saves 45\% cycles and reduces 44\% instructions for this loop yielding a 1.34\times speedup for the whole program on Intel SandyBridge. The optimization saves 23\% power. The improvements observed on other machines in shown in Table 4.3.
for (pos = 0; pos < max_pos; pos++) {
    ... if(...) PelYline_11 = FastLine16Y_11;
else PelYline_11 = UMVLine16Y_11;
for (blky = 0; blky < 4; blky++) {
    for (y = 0; y < 4; y++) {
        refptr = PelYline_11(ref_pic, abs_y++, abs_x, img_height, img_width);
        ... } ... } ...}

Listing 3.4: Temporal redundancy in SetupFastFullPelSearch function in h264ref.

3.7.2 NWChem

NWChem is a production computational chemistry package from Pacific Northwest National Laboratory, which implements several quantum mechanics and molecular mechanics methods. It is a six-million-line application written primarily in Fortran and C and parallelized with MPI. REDSPY reports that over 50% of memory writes are redundant.

Listing 3.5 shows the high portion redundancy in routine tce_mo2e_trans with culprit calling context pair in Figure 3.3. The top redundant memory writes occur in the function call to dfill, which zeroes two arrays work1 and work2. REDSPY reports that most of the redundancy was in initializing the work2 array.

Calls to redundant dfill repeat more than 200K times, resulting in redundantly writing 500GB data. By consulting NWChem developers, we identified that the buffer size was larger than necessary, and the zero initialization was unnecessary, leading to the redundant writes in the same location. Subsequently, NWChem developers eliminated the unnecessary initialization from the code base. Table 4.3 shows the speedup of the execution is 1.53× after optimization with the gcc compiler on our AMD platform.

3.7.3 Rodinia LavaMD

LavaMD is an OpenMP benchmark, which calculates particle potential and relocation between particles in a three-dimensional space. REDSPY identifies a loop nest, shown in Listing 3.6, which accounts for more than 60% value redundancy in registers. Moreover, this loop nest accounts for more than 90% of the total execution time. REDSPY pinpoints
that the redundancy occurs at line 175, where frequently on consecutive invocations \texttt{exp()} return the same value, which is written to the register holding \texttt{vij}. With further investigation, we inferred that \texttt{r2} is often assigned the same value at line 173. Since \texttt{a2} is a loop invariant, \texttt{u2}, which is derived from \texttt{r2} often has the same value. Consequently, the code keeps recomputing the expensive exponentiation of a value that infrequently changes in the loop. The output of \texttt{exp()}, which is assigned to \texttt{vij}, shows a high fraction of redundancy.

To exploit this temporal value locality in registers, we transform the code by adding a conditional check before line 173. If we find \texttt{r2}'s value has not changed from the previous iteration, we reuse the value of \texttt{vij} computed from the previous iteration. This optimization reduces the CPU cycles and instructions consumed in this loop nest by 33\% and 35\% respectively resulting in a 1.50× speedup and 37\% energy saving for the entire program.

### 3.7.4 Rodinia Hotspot

Hotspot estimates processor temperatures based on architectural floorplan and simulated power measurements. \textsc{RedSpy} identifies high approximate spatial value locality in a two-dimensional array \texttt{temp}, which stores the temperature values of processor cell partitions. This array updates the new cell temperatures based on the stencil computation with their neighbor cells, as shown in Listing 3.7. We placed the instrumentation hook to inspect the \texttt{temp} array outside the nested loop. The spatial redundancy introduced ∼7× runtime overhead. \textsc{RedSpy} pinpoints that all the values in \texttt{temp} are approximately identical, with less than 1\% difference between values in adjacent cells.

We employed an approximate computing technique to exploit the spatial value redundancy observed in \textsc{Hotspot}. Instead of performing computation on all the cells of \texttt{temp}, we perform computation on only the first and the middle element. Other cells simply reuse the value computed for one representative element in their halves. The approximation error, quantified by the mean relative error across all the cell values is less than 0.6\%. This approximation based on the spatial value locality yields a 2.21× speedup and 69\% energy saving for the entire program.
for (r = 0; r < row; r++) {
    for (c = 0; c < col; c++) {
        delta = (step / Cap) * (power[r*col+c] +
            temp[(r+1)*col+c]+temp[(r-1)*col+c]-2.0*temp[r*col+c]) / Ry +
            (temp[r*col+c+1]+ temp[r*col+c-1] - 2.0*temp[r*col+c]) / Rx +
            (amb_temp - temp[r*col+c]) / Rz);
        result[r*col+c] =temp[r*col+c] + delta;
    }
}

Listing 3.7: Approximate spatial value locality in Hotspot.

for (j = 1; j <= ndelta; j++) {
    for (k = 0; k <= nly; k++) {
        new_dw = ((ETA * delta[j] * ly[k]) +(MOMENTUM * oldw[k][j]));
        w[k][j] += new_dw;
        oldw[k][j] = new_dw;
    }
}

Listing 3.8: Temporal redundant array updating in backprop.

3.7.5 Rodinia Backprop

Backprop implements backward propagation machine learning algorithm to trains the weights of connecting nodes in a neural network. Backprop is an OpenMP program written in C. The code in Listing 3.8 shows a top temporal store redundancy identified by RedSpy. The redundancy happens when updating the value new_dw at line 323. This nested loop is accessed twice during the whole execution. During the first visit, the loop iterates 17 times and populates new_dw with non-zero values. However, during the second visit, the total number of loop iterations is over one billion and this time new_dw is always zero. Since adding zero does not change value, the entire array w is always written with the same value (redundancy at line 324).

To avoid redundancy, we execute the update of w[k][j] line 324 conditionally if and only if new_dw is non-zero. This optimization avoids a redundant load, a redundant addition, and a redundant store. Our optimization for this loop yields a small speedup (1.01⇥) but nontrivial energy saving (13%) for the entire program on Intel SandyBridge. On the AMD machine, this optimization yields a 1.14⇥ speedup for the whole program.

3.7.6 Rodinia Particlefilter

Particlefilter estimates the location of a target object using a Bayesian method. It is an OpenMP program written in C. RedSpy reports high spatial value locality of the array xj and yj in Listing 3.9. We position the instrumentation hook after this loop, which introduces <12⇥ runtime overhead. RedSpy detects that the index i computed at line 488 seldom changes in consecutive iterations; hence, many elements in xj or yj are assigned to the same value.

To exploit this spatial value locality, we reuse the value of xj[j-1] for xj[j] if i...
#pragma omp parallel for shared(CDF, Nparticles, xj, yj, u, arrayX, arrayY) private(i, j)
for(j = 0; j < Nparticles; j++){
  i = findIndex(CDF, Nparticles, u[j]);
  if(i == -1) i = Nparticles - 1;
  xj[j] = arrayX[i];
  yj[j] = arrayY[i];
}

Listing 3.9: Spatial value locality in Particle_filter.

<table>
<thead>
<tr>
<th>Program</th>
<th>DeadSpy</th>
<th>RVN</th>
<th>Paraprox</th>
<th>ThinLTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>464.h264ref</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Partial</td>
</tr>
<tr>
<td>NWChem</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>–</td>
</tr>
<tr>
<td>backprop</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>hotspot</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>lavaMD</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>particlefilter</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 3.6: RedSpy vs. other tools: whether value redundancies identified by RedSpy can be identified by other tools.

remains the same in these two adjacent iterations. This optimization saves the loads from arrayX, which is randomly accessed over more than 9,000 elements. Hence, saving the loads from this array reduces cache misses. This optimization yields a $1.10 \times$ speedup for the entire program.

### 3.7.7 Comparison with Other State-of-the-art Tools

Table 3.6 shows whether the redundancies detected by RedSpy could have been detected by other state-of-the-art tools. DeadSpy and RVN can find redundancy if there are dead writes or symbolic computational equivalence, respectively. DeadSpy can identify the NWChem redundancy since it is a dead write in addition to a redundant write. RVN can identify the redundancy in NWChem and lavaMD since they have a symbolic equivalence. ParaProx can identify and exploit the approximate computing opportunity available in the OpenCL version of hotspot, but not the OpenMP version.

Finally, we used LLVM ThinLTO [56] to assess whether link-time optimization (LTO) could have eliminated any of the redundancies found by RedSpy. ThinLTO inlined the indirect function call in 464.h264ref via PGO but introduced a condition check in the loop body resulting in only 8% speedup. RedSpy continued to find redundancies at the same place. Our hand-optimization cloned the loops and hoisted the condition check out of the loops resulting in 45% speedup. All other redundancies are algorithmic in nature and hence not exploitable by compilers even with a global view of the program. In fact, we noticed that sometimes the redundancy fraction increased with ThinLTO; this is because LTO can reduce the number of operations by eliminating some “language and abstraction overheads” but cannot reduce redundant operations arising from algorithmic inefficiencies. Hence, the number of redundant operations relative to the total number of operations increases [9].

45
Chapter 4

Watching for Software
Inefficiencies with WITCH

4.1 Introduction

Large, layered, production software is complex due to a hierarchy of component libraries and sophisticated control flow. Even the high-performance computing (HPC) software achieves only 5-15% of peak performance on modern supercomputers [93, 31, 33]. Inefficiencies inherent in complex software [20, 54, 117, 61, 102, 57, 108, 116] significantly contribute to this abysmal performance. Software inefficiencies may arise during design (e.g., inappropriate choice of algorithms and data-structures), implementation (e.g., developers’ inattention to performance and use of heavyweight APIs), or translation (e.g., detrimental compiler optimizations and lack of tuning for an architecture).

Inefficiencies, whatever their origin, often manifest as computations whose results may not be used [13, 100], re-computation of already computed values, unnecessary data movement [20, 62, 74, 77], and excessive synchronization [17, 110]. Inefficiencies involving the memory subsystem are particularly egregious because of limited bandwidth shared by multiple cores and high access latencies. Repeated initialization, register spill and restore on hot paths, lack of inlining hot functions, missed optimization opportunities due to aliasing, computing and storing already computed or sparingly changing values, and contention and false sharing [40, 70, 73, 72] (in multi-threaded codes), are some of the common prodigal uses of the memory subsystem. Although compiler literature is rich with optimization to eliminate inefficiencies, in practice, layers of abstractions, dynamic libraries, multi-lingual components, aggregate types, aliasing, and combinatorial explosion of execution paths handicap optimizing compilers in delivering top application performance. Additionally, algorithmic and data structural deficiencies also appear as useless memory operations [20, 54, 117, 61, 102, 116].

Coarse-grained profilers such as VTune [49], HPCToolkit [5], gprof [39], Oracle Solaris Studio [96], Oprofile [95], Perf [66], and CrayPAT [30] identify execution “hotspots”. They
void loop_regs_scan(struct loop *loop, ...){
    ....
    last_set=(rtx *) xcalloc(regs->num, sizeof (rtx));
    /* Scan the loop, recording register usage */
    for (each instruction in loop){
        ....
        if (GET_CODE (PATTERN (insn)) == SET || ...)
            count_one_set (...,last_set ,...);
        ....
        if (end of basic block)
            memset(last_set,0,regs->num*sizeof(rtx));
    } ...
}

Listing 4.1: Dead stores in SPEC CPU2006 gcc due to an inappropriate data structure. The function iterates over the basic blocks in a loop scanning for the registers used. Line 3 allocates and zero initializes a 16K-element 132KB array representing the virtual registers. The loop body accesses only a few (< 2) array elements since basic blocks are typically small. At the end of each basic block (Line 11) the code zero initializes the same array for the use in the next basic block. Line 11 is repeatedly involved in dead stores.

attribute measurements such as CPU cycles, stalls, arithmetic intensity, and cache misses, obtained from hardware performance monitoring units (PMUs) to the source code. On the positive side, they introduce little runtime overhead and do not materially perturb execution. On the negative side, hotspots fail to distinguish efficient vs. inefficient resource usage. The SPEC CPU2006 [104] gcc code, shown in Listing 4.1, repeatedly zero initializes a 132KB array, most of which is already zero. None of these profilers detects this as wasted work. Ironically, a hotspot may have no further optimization scope (e.g., a highly optimized linear algebra library); and conversely, a code region acclaimed by a profiler with high arithmetic intensity (a goodness metric) may perform useless computations.

Fine-grained profilers such as DeadSpy [20], Toddler [88], Cachetor [87], and our previous works RVN, and RedSpy analyze dynamic instructions with specific objectives—detect useless computation or data movement. They can identify inefficiencies not detected by coarse-grained profilers. In Listing 4.1, they can pinpoint the source code location that re-initializes an already initialized array and quantify the wasted work. On the positive side, they offer visibility into wasted work. On the negative side, they significantly slow execution down (10-80⇥) and consume enormous (6-100⇥) extra memory.

Despite their effectiveness, the high overhead of fine-grained inefficiency detection tools has blocked them from better scalability. It is necessary to make such tools common enough to run on large scale real applications.

We developed Witch—a lightweight inefficiency-detection framework—to address this issue. Witch combines the best of both worlds—low overhead of coarse-grained profilers and inefficiency detection of fine-grained profilers. Our key observation is that an important class of inefficiency detection schemes, explored previously via fine-grained profilers [20], requires monitoring consecutive accesses to the same memory location. For example, detecting repeated initialization—a dead write [20]—requires monitoring store after store without an intervening load to the same location.

Witch samples addresses accessed by a program using hardware PMUs. Witch intercepts the subsequent access(es) to the sampled memory locations using hardware debug registers. The result is (1) the ability to observe consecutive accesses to the same memory location to detect myriad inefficiencies, and (2) no code or binary instrumentation and
hence low overhead. We show the benefit of this concept by building various inefficiency-detection tools (witchcraft) atop Witch. There are various challenges in making it practical, which we detail and address in Section 4.4 and 4.5.

The idea generalizes to detect other kinds of inefficiencies—updating a location with a value already present at the location (aka silent store [63, 62]) and loading an unchanged value from memory that was previously loaded [8, 88, 103] (poor register usage). Sharing addresses sampled by one thread with another thread enables building Witch-based tools for multi-threaded programs. In this work, we restrict ourselves to describing the Witch framework and three tools that detect inefficiencies in a thread of execution. We make the following contributions:

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

4.2 Related Work and Motivation

There is a vast literature in detecting and eliminating software inefficiencies. We classify these techniques into hardware and software approaches. The hardware approaches [69, 68, 62, 63, 82, 81, 118] introduce new hardware components to detect and eliminate computations whose results are never used or elide memory operations that do not change the contents of their target memory cells. Our focus is on software approaches, which do not need any hardware modification.

Classic compiler optimizations such as value numbering [98], constant propagation [115], and common subexpression elimination [29] eliminate several inefficiencies. Recently, static analysis has been used in detecting performance bugs [94, 89]. Static analysis, typically, suffers from limitations related to aliasing, optimization scope, and input and context insensitivities. A thorough literature review of static analysis is not pertinent.

The dynamic analysis addresses the limitation of static analysis. Chabbi and Mellor-Crummey [20] show that dead writes are a common symptom of myriad inefficiencies. Their tool, DeadSpy, tracks every memory operation to identify store operations that are never loaded (dead) before a subsequent store (kill) to the same location. DeadSpy associates pairs of instructions involved in a dead store (dead-kill pair) with their calling contexts and source code locations to guide manual optimizations. Using DeadSpy, the authors identify
inefficiencies arising from inappropriate data structure choice, optimization inhibiting code shape, inattention to performance, and poor compiler code generation. They improve the performance of several systems by eliminating dead writes. DeadSpy’s exhaustive monitoring typically introduces more than $28 \times$ slowdown and consumes more than $9 \times$ extra memory on average.

Our previous works RVN, REDSPY detect CPU- and memory-bound inefficiencies arising from redundant computation, missed inlining opportunities, layers of abstractions, and redundant stores. With exhaustively monitoring, our tools incur $40-280 \times$ runtime overhead. By periodically enabling and disabling monitoring (bursty sampling [43]), we can bring it down to a manageable $12 \times$ slowdown and $9 \times$ memory bloat.

Toddler [88] focuses on identifying repetitive memory load sequences across loop iterations at the cost of $10 \times$ slowdown. LDoctor [103] reduces Toddler’s overhead using a combination of ad-hoc sampling and static analysis techniques. However, it only analyzes a small number of suspicious loops identified by profiling, and hence does not work for systematically detecting inefficiencies in the whole program.

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

**Tools Based on Hardware Debug Registers:** Erickson et al. [34] use hardware debug registers [55, 78] to detect data races in the Windows kernel. Jiang et al. [53] extend it to the Linux. They sample memory access instructions and set watchpoints to detect conflicting accesses. They use code breakpoints to intercept random instructions and use them to monitor memory accesses for a time window. Liu et al. [71] developed DoubleTake, which uses debug registers to identify buffer overflow, use after free, and memory leaks. Pesterev et al. developed DProf [97], which combines PMU and hardware debug registers to capture the data flow across runtime objects. DProf suffers from limited debug registers; it runs a program multiple times to achieve higher coverage. These approaches focus on detecting the presence or absence of a bug; they are not concerned with quantifying the frequency of a bug or prioritizing the importance of a bug, which become necessary in performance analysis tools. WITCH addresses these quantification and attribution problems necessary for performance tools.

Kasikci et al. [60] describe a spatially unbiased sampling scheme to trace cold code for code coverage. In contrast, WITCH develops a temporally unbiased sampling scheme to monitor memory locations. Kasikci et al. dynamically rewrite the first instruction of every basic block with the int 3 breakpoint instruction, which causes a trap; there is no hardware limit on how many blocks they can monitor. Breakpoints set in hot code regions drive their sampling, and they throttle too frequently trapping breakpoints. In contrast, WITCH does not modify the binary (not even at runtime), it uses the PMU as its sampling engine, but it has to workaround the limited number of debug registers.
4.3 Background and Terminology

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

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

Intel SandyBridge and its successors support Precise Event-Based Sampling (PEBS) [47]. A PMU captures a snapshot of the user-visible register state including the program counter (PC) and the effective address (EA) accessed by the instruction on an event overflow. AMD Instruction-Based Sampling (IBS) [32] and PowerPC Marked Event Sampling (MRK) [106] offer commensurate capabilities.

**Hardware Debug Registers:** Hardware debug registers [55, 78] enable trapping the CPU execution for debugging when the PC reaches an address (breakpoint) or an instruction accesses a designated address (watchpoint). One can program debug registers with different addresses, widths, and conditions that will cause the CPU to trap on reaching the programmed conditions. Today’s x86 processors have four debug registers. If used for the break-on-data-access (store, or load-or-store), on x86 processors, the trap occurs after the instruction execution. Hence, if a store instruction results in a trap, the contents of the target memory will contain the results of the store operation.

**Linux Perf events:** Linux offers a standard interface to program and sample PMUs using the `perf_event_open` system call [65] and the associated `ioctl` calls. The Linux kernel can deliver a signal to the thread whose PMU event overflows. The user code can `mmap` a circular buffer into which the kernel keeps appending the PMU data on each sample. Linux 2.6.33 and its successors incorporate the debug registers in the `perf_event` interface, however, the support has several limitations, which we discuss and fix in our work. We implement Witch on Intel processors with the PEBS facility. It is straightforward to extend Witch to work on AMD with IBS and PowerPC with MRK.

**Call Path Profiling:** Call path profiling [41] is a profiling technique where runtime events (e.g., cache misses) are attributed to the full call path seen at the time of the event. Call path profiling offers insightful details in complex applications with deep call chains. The calling context of an event is a set of active procedure frames when the event happens. A calling context begins at a process or thread entry function such as `main` and ends at the instruction pointer (IP) of the instruction that triggers the event. The alternative, flat profiling, merely attributes events to the leaf function involved in the event, which introduces ambiguities when the same leaf function (e.g., `memset`) can be invoked from multiple contexts.
Figure 4.1: Detecting dead writes using Witch. The client, DeadCraft, subscribes to the precise PMU store event with a desired sample period. ① PMU counter overflows triggering an interrupt. ② Witch handles the signal, extracts the calling context (C\text{watch}) of the interrupt and the address accessed (M), and offers the triplet (C\text{watch}, M, AccessType) to DeadCraft. ③ DeadCraft asks Witch to monitor subsequent load or store to M. ④ Witch sets a watchpoint to monitor M, and the execution continues ⑤ Program accesses M, which causes a CPU trap. ⑥ Witch handles the trap signal, extracts the calling context (C\text{trap}), and offers the triplet (C\text{trap}, M, AccessType) to DeadCraft. ⑦ If the AccessType is a store, DeadCraft infers a dead write and attributes it to (C\text{watch}, C\text{trap}).

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

### 4.4 Methodology and Design

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

**Summary:** PMU samples that include the effective address accessed in a sample provide the knowledge of the addresses accessed in an execution. Given this effective address, a hardware debug register allows us to keep an eye on (watch) a location and recognize what the program subsequently does to such location. Since the hardware can monitor a small number of locations at a time, reservoir sampling [112] allows monitoring a subset of previously seen addresses without any temporal bias. Finally, we scale the measurements taken for a few monitored samples in a calling context to other unmonitored samples in the same calling context; the scaling is based on the observation that the code behavior in a calling context typically remains the same.
Details: Precise PMU samples drive Witch. Client tools subscribe to PMU events of their choice. On each PMU sample, the client obtains the memory address M accessed in the sample. Clients subscribe to a watchpoint at the sampled address in the signal handler and continue their execution. When the program accesses M next time, a CPU trap happens. Witch handles the watchpoint exception, captures information associated with the trap, associates any information given by the client at the watchpoint subscribe time, and gives control to the client tool for appropriate actions.

We use our dead store detection tool—DeadCraft, shown in Figure 4.1—as a running example to illustrate our methodology. The ideas generalize to any tool built atop Witch. A store followed by another store to the same address is an instance of a dead store. A store followed by a load to the same address is not a dead store. A software instrumentation tool such as DeadSpy [20] maintains a large shadow memory where it stores the last operation performed on each byte of the original program. A write→write transition in a shadow byte indicates an instance of a dead write.

DeadCraft mimics the behavior of DeadSpy but on a subset of addresses seen in PMU samples. DeadCraft samples the PMU store events at a chosen frequency. Let the address accessed in a PMU sample be M and let the calling context where the sample happens be \( C_{\text{watch}} \). In the PMU overflow handler, Witch offers the triplet \( \langle C_{\text{watch}}, M, \text{AccessType} \rangle \) to DeadCraft. DeadCraft memorizes the tuple and in-turn asks Witch to set a RW_TRAP watchpoint \( W \) at \( M \). The normal execution continues. \( W \) traps when the program accesses \( M \) next time; we defer discussing another sample happening before the trap to Section 4.4.1. Let the address accessed in the trap be \( M \) and let its calling context be \( C_{\text{trap}} \). Witch handles the trap and offers the triplet \( \langle C_{\text{trap}}, M, \text{AccessType} \rangle \) to DeadCraft. If a load causes a trap, DeadCraft treats it as a useful operation and disables the watchpoint. If a store causes a trap, however, DeadCraft infers the store seen in the context \( C_{\text{watch}} \) as a dead store. It attributes a “unit” of dead store to the calling context pair \( \langle C_{\text{watch}}, C_{\text{trap}} \rangle \).

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

4.4.1 Challenge with Samples Intervening Accesses

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

Consider the dead store example in Listing 4.2. Assume the loop index variables i and j are in registers, the sampling period is 10K stores, and the number of debug register is one. The first sample happens in the i loop when accessing \&array[10K]. DeadCraft sets a watchpoint to monitor \&array[10K] since a debug register is available. The second sample happens when accessing \&array[20K]. Since the watchpoint armed for address \&array[10K] is still active, there is no room to monitor \&array[20K].

1A client may set a watchpoint at an address derived from the sampled address or any other address instead of the sampled address itself.
for (int i = 1; i <= 100K; i++) {
    array[i] = 0;
}

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

Listing 4.2: Long distance inefficiencies: All (say 4) watchpoints will be armed when sampling at 10K store in the first four samples taken in the i loop. A naive replacement will not trigger a single watchpoint due to many samples taken in the i loop before reaching the j loop. WITCH ensures each sample equal probability to survive.

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

Monitoring a new sample may help detect a new, previously unseen problem whereas continuing to monitor an old, already-armed address may help detect a problem separated by many intervening operations. We should detect both. But, we do not know when in the future a watchpoint may trap, if at all. Our solution strikes a balance between new vs. old by being unbiased in choosing among the previously accessed addresses (reservoir sampling [112]), and we rely on multiple such unbiased samples taken over a repetitive execution to capture both scenarios. We first show our approach for a single debug register and then generalize it for an arbitrary but finite number of debug registers.

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

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

$$= \frac{1}{k-1} \times \frac{k-1}{k} = \frac{1}{k} = Pr[monitoring \ S_k]$$

Any time a watchpoint traps and the client chooses to disarm the watchpoint, and the probability is reset to 1.0, which ensures that the immediately next sample is monitored. Naturally, if every watchpoint triggers before the next sample, we will monitor every address seen in every sample.
In a system with \( N \) debug registers, on a new sample, we populate any unused debug register as long as we find one. If no debug register is freed up in a window of \( N \) consecutive samples, there will be no room for the \((N+1)\)th sample. We install the sample \( S_{N+1} \) with \( N/N+1 \) probability. If the choice is to install \( S_{N+1} \), we randomly choose one of the \( N \) debug registers and replace it with \( S_{N+1} \). It follows that at the end of \( S_{N+1} \), the probability of monitoring any sample \( S_i \), \( 1 \leq i \leq N + 1 \), is \( N/N+1 \).

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

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

The number of consecutive PMU samples that are not monitored form a “blindspot” window; the longer the window is, the larger the probability of missing bugs. In our experience, many software in practice often have very short windows. For example, in the SPEC CPU2006 [104] reference benchmarks, on an Intel Haswell machine, we found the largest blind-spot to be, typically, extremely small (< 0.02% of the total samples in a program), and the worst case was 0.5% of the total samples in the mcf benchmark.

### 4.4.2 Challenges with Proportional Attribution

Consider the code in Listing 4.3. For brevity, line numbers represent contexts. 25% PMU samples will be attributed to each of Line 3, 7, 8, and 11. If the outer loop executes 1K times and if the sampling period is 10K store operations, each of these lines will get approximately 10K PMU samples. The number of sampled dead writes should be 10K for each line pair \([3,11]\), \([11,3]\), \([7,8]\), and \([8,7]\). That is, 25% each. This expectation in quantification is not preserved with our sampling scheme because of a mixture of sparse monitoring (lines 3 and 11) and dense monitoring (lines 7 and 8). As soon as a watchpoint traps on Line 7, a debug register frees up; every subsequent PMU sample in the \( k \) loop will find a free debug register. Hence, there will be a disproportionately large number of dead writes recorded for the line pairs \([7,8]\) and \([8,7]\) compared to rest.

We solve this problem with a context-sensitive approximation. The code behavior is typically same in a calling context; hence, an observation made by monitoring an address accessed in a calling context can approximately represent other unmonitored samples occurring in the same calling context. If in a sequence of \( N \) samples occurring in a calling context \( C \), only one sample is monitored through a debug register, we scale the observation made for the monitored sample by \( N \) to approximate the behavior of the remaining \( N - 1 \) unmonitored samples taken at \( C \). In this scheme, in a sequence of ten PMU samples taken at line 3, only one is monitored through a debug register, and that address leads to a dead
for(... many iterations ...){
  for(int i = 1; i <= 100K; i++){
    array[i] = 0;
  }
  // p and q alias to the same location
  for(int k = 1; k <= 100K; k++){
    *p = 0; // dead write
    *q = 0;
  }
  for(int j = 1; j <= 100K; j++) {
    array[j] = 0;
  }
}

Listing 4.3: 100K stores in the i loop are dead by the overwriting j loop, but only a few watchpoints survive between these two loops. 100K writes to *p are also dead but trigger many more watchpoints at *q. "Witch" applies a proportional attribution heuristic by accounting the samples taken in a context.

write with line 11, we scale and record number of dead writes between lines 〈3,11〉 as 10.

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

\[
\tilde{D} = \frac{\sum_i \sum_j \text{waste in } \langle C_i, C_j \rangle}{\sum_i \sum_j \text{waste in } \langle C_i, C_j \rangle + \sum_i \sum_j \text{use in } \langle C_i, C_j \rangle}
\] (4.1)

The metric is similar to the “deadness” \(D\) metric described in [18]; instead of deriving the metric by measuring every load and store, we are approximating. Equation 4.1 is an optional feature available for the clients of "Witch"; not all clients need this kind of proportional attribution.

In Listing 4.3, when a watchpoint traps for the first time on Line 11 (= \(C_{\text{trap}}\)), and if there were 10 PMU samples accumulated at the source Line 3 (= \(C_{\text{watch}}\)), we attribute \(10 \times 10K \times 4\) bytes = 400K bytes of dead writes to the line pair \(\langle 3,11\rangle\). This scheme allows the dead writes metric to catchup with the PMU samples, resulting in proportional attribution. Thus, even though we have very few watchpoints, we use PMU samples in a context to approximate the dead writes in that context. If multiple watchpoints were simultaneously set from the same calling context at different addresses, we proportionally distribute the samples among them.

Figure 4.2 shows "Witch"'s attribution of dead writes in a more complex scenario, which perfectly matches our expectation of 50%:33%:17% dead writes to \(a:b:x\). Without pro-
Figure 4.2: \( a[] \) and \( b[] \) and \( x \) are involved in dead writes in 3:2:1 ratio (50%:33%:17%), respectively. The sampling interval is 50K stores. Our proportional, context-sensitive scheme apportions dead writes in near perfect ratio.

Through proportional attribution, we noticed a biased attribution of 5%:2%:93%. With random sampling, rather than our equal probability sampling, 100% samples get attributed to the line pair \( (16, 17) \).

4.4.3 Limitations

Witch employs Monte-Carlo experiments to approximately model real-world observations and suffers from the limitations of any sampling system. Insufficient samples can result in overestimation or underestimation. Witch cannot monitor register-to-register operations. Witch cannot hide the deficiencies of the underlying PMU used to drive its sampling: on some Intel architectures, sporadically, the shadow sampling effect [64, 90, 23] may hide a short latency store behind a long latency store. This behavior can bias the samples to favor long latency stores.

Witch can simultaneously monitor only as many memory locations as the number of debug registers. This physical constraint often is not a problem in practice as we show in our evaluation. However, an adversary may be able to construct a program where the effects of limited registers can be more pronounced.
Witch’s context-sensitive attribution is an optional feature available for its tools. It approximates the behavior of one monitored sample in a context to many samples taken in the same context. If very few monitored samples in a context are used to approximate the behavior of a large number of samples with different traits in that context, it can result in noticeable overestimation or underestimation.

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

4.5 Design and Implementation

We implement Witch in the open-source HPCToolkit [5] performance analysis tools suite. HPCToolkit works on multi-lingual, multi-threaded, and multi-process, fully optimized applications on multiple programming models such as MPI and OpenMP. On a PMU sample, HPCToolkit’s profiler, hpcrun, walks the sampled thread’s call stack using an on-the-fly binary analysis technique and attributes the measurements to the sampled call path. hpcrun introduces negligible runtime overhead (~3%) and consumes only a few megabytes of memory space for its metrics data when sampling at ~200 samples/second/thread [109].

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

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

Precise PC of a Watchpoint: Some clients need the precise instruction pointer of the instruction triggering the watchpoint, for example, to distinguish a load from a store.
when a RW_TRAP watchpoint triggers. The BREAKPOINT event in Linux perf_event is not a PMU event and hence the Intel PEBS support, which otherwise provides the precise register state, is unavailable for a watchpoint. Although the watchpoint causes a trap immediately after the instruction execution, the PC seen in the signal handler context (contextPC) is one ahead of the actual PC (precisePC) that causes the trap. In the x86 variable instruction set ISA, it is non-trivial to derive the precisePC, even though it is just one instruction before the contextPC. A software solution is to find the function enclosing contextPC and disassemble every instruction till we reach the contextPC. This solution may fail with linear disassembly due to 1) data embedded in instruction and 2) missing function bounds [109]. Furthermore, it can be time-consuming if the function body is large.

Our solution depends on the Last Branch Record (LBR) facility [47] provided by Intel Nehalem and its successors, which is exposed through the Linux perf_event interface. LBR tracks taken branches throughout CPU execution and continuously records the <from:to> pairs of instruction pointers in a fixed-size in-CPU circular buffer. WITCH exploits the LBR facility by modifying the perf_event implementation inside the Linux kernel. Linux perf_event already has the facility to construct the precise PC by disassembling the instructions starting from the “to” field of the last entry in the LBR until the disassembly reaches the contextPC. Disassembling a basic block is “feather light” compared to full function disassembly. We reuse this component with PERF_TYPE_SOFTWARE to construct the precisePC when a watchpoint trap event happens. The kernel makes the precisePC available to WITCH’s watchpoint exception handler in the ring buffer associated with the event on each watchpoint trap. This reduces ~5% runtime overhead.

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

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

Setting a watchpoint on the application stack address has a corner case. On a PMU sample, the profiler’s overflow signal handler, by default, shares the same stack as the application thread. In Figure 4.3(a), assume M is the sampled stack address. Assume we set a watchpoint at M. If the next PMU sample is taken with a shallower stack (Figure 4.3(b)), and the signal stack frame overwrites M; it spuriously triggers the watchpoint. Similarly,
Figure 4.3: (a) A PMU sample happens in a deeper call stack when \( B() \) is accessing address \( M \); signal handler sets a watchpoint to monitor the address \( M \). (b) A shallower application call stack, function \( A() \), triggers another PMU sample, the signal handler is established in a location that overwrites \( M \), triggering a spurious watchpoint. (c) An alternate signal stack for PMU signal handler and watchpoint signal handler solves the problem.

We avoid this problem by establishing a separate signal-handler stack frame for both PMU signal handler and watchpoint exception handler using the Linux `sigaltstack` facility [67]. The `sigaltstack` facility allows each thread in a process to define an alternate signal stack in a user-designated memory region. We use alternate stack to handle PMU and watchpoint signals as shown in Figure 4.3(c). All other signals continue to use the default stack unless specified otherwise by the application.

### 4.6 Witchcraft: Client Tools of Witch

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

#### 4.6.1 SilentCraft: Silent Store Detection

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

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

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

**SilentCraft** quantifies the store redundancy $\hat{R}$ in an execution analogous to **DeadCraft** (Equation 4.1); two consecutive stores with unchanged values (approximately the same for floating point values) contribute to the “waste” and contribute to the “use” otherwise.

### 4.6.2 LoadCraft: Load-after-load Detection

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

**LoadCraft** samples PMU load events. The rest of the functionality is similar to that of **SilentCraft**, except that it requests a watchpoint for load access on the monitored location. **Witch** uses RW_TRAP because x86 machines do not offer a trap-on-load watchpoint. If a watchpoint triggers on a store operation, **Witch** merely drops it. **LoadCraft** quantifies the load redundancy $\hat{L}$ in an execution analogous to **DeadCraft** (Equation 4.1), where two consecutive loads with (approximately) unchanged values contribute to the “waste” and different values contribute to the “use”.

### 4.6.3 Witchcrafts on Multi-threading

Debug registers and PMUs are per CPU core and virtualized for each software thread. All the previously discussed **Witch** tools work on multi-threaded codes; they, however, track intra-thread inefficiencies only. If a thread $T_1$ configures a watchpoint at address $M$, a trap occurs only in $T_1$; other threads remain unaffected whether they access $M$ or not. Sharing addresses accessed by one thread with another thread allows building several tools for multi-threaded applications. Atop **Witch**, we have developed Feather [21]—a tool to detect false sharing in parallel programs.
4.6.4 Discussion

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

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

4.6.5 Presentation

HPCToolkit maintains all sampled call paths in a compact calling context tree (CCT) format [6]. HPCToolkit, the graphical interface, enables navigating the CCT and the corresponding source code ordered by the monitored metrics. A top-down view shows a call path \( C \) starting from main to a leaf function with the breakdown of metrics at each level. Witch tools discussed here need to attribute metrics to calling context pairs \(<C_{\text{watch}}, C_{\text{trap}}\>\). Merely attributing a metric to two independent contexts loses the association between two related contexts during postmortem inspection. To maintain a correlation between a source context (e.g., dead) and target (killing) context, Witch appending a copy of the target calling context to a source calling context. For example, if a store in context main\( \rightarrow \)A\( \rightarrow \)B is overwritten by another store in context main\( \rightarrow \)C\( \rightarrow \)D, DeadCraft constructs a synthetic calling context: main\( \rightarrow \)A\( \rightarrow \)B\( \rightarrow \)KILLED\( \rightarrow \)main\( \rightarrow \)C\( \rightarrow \)D. The dead write metrics will be attributed to the leaf of this call chain. These synthetic call chains make it easy to visually navigate the CCT and focus on top redundancy pairs. Figure 4.2 in Section 4.4 depicts this scheme.

4.7 Evaluation

We evaluate Witch on a 2-socket, 18-core Intel Xeon E5-2699 v3 (Haswell) CPU clocked at 2.30GHz running Linux 4.8.0. The machine has 128GB DDR3 RAM. Simultaneous multi-threading (SMT) facility is not used in our experiments. All experiments use GCC v5.4.1 tools with -O3 and profile-guided optimization (PGO) to ensure the highest level of optimization. DeadCraft and SilentCraft use the PMU event \texttt{MEM\_UOPS\_RETIRED:ALL\_STORES} whereas LoadCraft uses \texttt{MEM\_UOPS\_RETIRED:ALL\_LOADS}. In our experiments, we use the nearest prime number for the shown sampling intervals, which is the recommended method in PMU sampling. The raw data from our experiments are available online [19].

Two aspects are critically important in evaluating Witch: accuracy and overhead compared to the exhaustive instrumentation techniques. We use SPEC CPU2006 reference benchmarks for this aspect of evaluation.
Accuracy: For accuracy, we need to answer three questions: (1) how accurate are the results compared to exhaustive monitoring, (2) how does the accuracy vary with sampling rates, and (3) how stable are the sampled results from one run to another.

The quantitative metric of dead writes is the percent of dead stores $\hat{D}$ (bytes overwritten without reading) as described in Equation 4.1, which we compare against the ground-truth dead stores $D$ from DeadSpy [20, 22]. We compare the percent of silent stores $\hat{R}$ from SilentCraft against the ground-truth exhaustive monitoring metric $R$ from RedSpy. No prior tool exists to compare against LoadCraft; hence we implemented an exhaustive load-value redundancy detection tool called LoadSpy. We compare the percent of silent loads $\hat{L}$ from LoadCraft against the ground-truth exhaustive monitoring metric $L$ from LoadSpy. RedSpy also performs redundancy detection in registers, which we disabled for our evaluation. To assess the accuracy of our sampling clients against the ground-truth, we disable the bursty sampling used by RedSpy. SilentCraft, LoadCraft, RedSpy, and LoadSpy use 1% precision when comparing floating point values.

Figure 4.4 compares the total redundancies found by different sampling vs. exhaustive monitoring tools. The error bars represent the metric values at different sampling rates for Witch tools, i.e., 100K (high), 500K, 1M, 5M, 10M, and 100M (low) events per PMU interrupt. Clearly, the sampling rate, when chosen with some care, does not significantly affect the results. The sampling tools are highly accurate in almost all cases. There are, however, some exceptions. DeadCraft and SilentCraft on hmmer and calculix suffer from shadow sampling effects [64, 90, 23], where high latency stores hide low latency stores. GemsFDTD, perlbench, and zeusmp have many small inefficiencies scattered all over the code, leading to inaccuracies in SilentCraft. We ran each benchmark 10 times at 5M sampling rate (not shown) and the maximum standard deviations were 2.27%, 1.89%, and 0.77% for DeadCraft, SilentCraft, and LoadCraft respectively, which proves the run-to-run sampling stability.

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

To assess the effectiveness of reservoir sampling, we vary the number of debug registers from one to four and compare the redundancy metrics against the ground truth. Figure 4.5 shows that the number of debug registers has little practical influence in DeadCraft on the quality of results except h264ref, which shows better results with four debug registers. The online compendium [19] corroborates this observation on SilentCraft and LoadCraft.

To assess the effectiveness of our proportional attribution based on samples taken in a context, we compared the accuracy with and without this feature at different sampling rates and also with different number of debug registers with all three tools (not shown); we also compared it against the ground truth. In general, the feature did not make significantly positive or negative impact. GemsFDTD and perl were exceptions, where having the feature improved the accuracy.

To further understand the accuracy, we compared the rank ordering and percentage contribution of the top N redundancy pairs between DeadSpy and DeadCraft; we chose
N to add up to 90% of redundancy observed in execution. No single metric suffices to compare this type of complex data. We used edit distance and set difference of the top N contexts and also compared weights at each position. Our measurements [19] show that only a handful of context pairs account for the majority of redundancies and their rank ordering and individual weights match the exhaustive monitoring.

**Overhead:** Table 4.1 shows the runtime slowdown and memory bloat of sampling vs. exhaustive monitoring. Slowdown (memory bloat) is the ratio of the runtime (peak memory usage) under monitoring to the runtime (peak memory usage) of the corresponding native execution. We show the average values for the same benchmark with multiple inputs. We used the sampling period of one in 5M stores and one in 10M loads (since loads are more common), which we found to be highly effective. Two critical things to observe about the sampling tools are 1) their overheads are at least an order of magnitude less than the exhaustive instrumentation tools, and 2) they introduce negligible overhead. Deep recursive codes such as xalan, sjeng, and gombk incur higher space and time overheads; and their instrumentation counterparts do not run to completion. Recursive codes with inefficiencies (e.g., SilentCraft on gobmk and LoadCraft on xalan) exacerbate memory bloat due
this section, we describe four case studies covering the analyses by the three tools (denoted by STAMP [85] and full applications—NWChem [111], Ca
e [52], GNU Binutils [38], and

![Figure 4.5: Comparison of dead writes with different number of debug registers. Error bars are for different (100K - 100M) sampling intervals.](image)

Table 4.1: Runtime slowdown ($\times$) and memory bloat ($\times$) over native execution: Witch (DeadCraft, SilentCraft, LoadCraft) vs. exhaustive monitoring tools (DeadSpy, RedSpy, LoadSpy).

to large calling context trees. Codes with a very small memory footprint (e.g., povray) show higher memory bloat because of some basic pre-allocated data structures used in our tools.

LoadCraft has higher overhead compared to the other two tools since 1) loads are more common than stores, 2) a high fraction of loading the same value leads to more wakeup traps and inefficiency reporting cost, 3) most PMU samples find a free debug register and incur the cost of arming it, and finally 4) LoadCraft sets the RW_TRAP wakeup point (x86 does not support break on load watchdog), which triggers a spurious exception on a store. Table 4.2 shows the geometric mean and median of the slowdown and memory bloat at different sampling periods in SPEC CPU2006.

4.8 Case Studies

The lightweight nature of Witch tools allowed us to apply it on an array of benchmark suites—SPEC CPU2006 [104], SPEC OMP2012 [105], NERSC Trinity [86], Rodinia [91], and STAMP [85] and full applications—NWChem [111], Caffe [52], GNU Binutils [38], and Kallisto RNA sequencing [79]. Table 4.3 summarizes the new performance bugs found by our tools (denoted by prefix) and confirms previously found performance issues [20]. In this section, we describe four case studies covering the analyses by the three Witch tools.

![Table 4.1: Runtime slowdown ($\times$) and memory bloat ($\times$) over native execution: Witch (DeadCraft, SilentCraft, LoadCraft) vs. exhaustive monitoring tools (DeadSpy, RedSpy, LoadSpy).](image)

![Table 4.2: Geometric mean and median of the slowdown and memory bloat at different sampling periods in SPEC CPU2006.](image)
the same problem.

overhead whereas the fine-grained profiler, DeadSpy, incurs

bug, which was hiding in the large code base, is now fixed.

same location. We eliminate this unnecessary initialization, yielding a 1.43

and the zero initialization was unnecessary, leading to the dead and killing writes in the

calls to

store pair in the call of function

lines of code written primarily in Fortran and C and parallelized with MPI [80]. We use

quantum mechanics and molecular mechanics methods. NWChem consists of six million

4.8.1 NWChem-6.3

NWChem [111] is a production computational chemistry package, which implements several

quantum mechanics and molecular mechanics methods. NWChem consists of six million

lines of code written primarily in Fortran and C and parallelized with MPI [80]. We use

the QM-CC aug-cc-pvdz input and eight MPI processes in our studies.

DeadCraft reports that more than 60% of memory stores are dead. Figure 4.6 shows

the full calling contexts of the top (94% contribution to total dead writes) dead and killing

store pair in the call of function dfill, which zeroes the array work2. With the given input,
calls to dfill repeat more than 200K times, resulting in writing 500GB data that are never

used. With further analysis, we identified that the size of work2 was larger than necessary,
and the zero initialization was unnecessary, leading to the dead and killing writes in the

same location. We eliminate this unnecessary initialization, yielding a 1.43× speedup. This
bug, which was hiding in the large code base, is now fixed. Witch incurs only 6% runtime
overhead whereas the fine-grained profiler, DeadSpy, incurs > 10× slowdown identifying

<table>
<thead>
<tr>
<th>GeoMean</th>
<th>DeadCraft</th>
<th>SilentCraft</th>
<th>LoadCraft</th>
</tr>
</thead>
<tbody>
<tr>
<td>Median</td>
<td>Slowdown</td>
<td>Memory bloat</td>
<td>Slowdown</td>
</tr>
<tr>
<td>100M</td>
<td>1.00/1.00</td>
<td>1.12/1.03</td>
<td>1.01/1.00</td>
</tr>
<tr>
<td>10M</td>
<td>1.01/1.01</td>
<td>1.19/1.05</td>
<td>1.01/1.00</td>
</tr>
<tr>
<td>5M</td>
<td>1.02/1.01</td>
<td>1.23/1.05</td>
<td>1.02/1.01</td>
</tr>
<tr>
<td>1M</td>
<td>1.05/1.03</td>
<td>1.40/1.05</td>
<td>1.06/1.03</td>
</tr>
<tr>
<td>500K</td>
<td>1.09/1.03</td>
<td>1.48/1.06</td>
<td>1.09/1.04</td>
</tr>
</tbody>
</table>

Table 4.2: Geomean and median of slowdown and memory bloat of Witch tools at different sampling rates on SPEC CPU2006.

<table>
<thead>
<tr>
<th>Benchmark Information</th>
<th>Witch</th>
<th>WS*</th>
</tr>
</thead>
<tbody>
<tr>
<td>program</td>
<td>problem code</td>
<td>Inefficiencies (Client)</td>
</tr>
<tr>
<td>gcc [104]</td>
<td>cseblb.c.csebb_init</td>
<td>Poor data structure (DS)</td>
</tr>
<tr>
<td>hiper2 [104]</td>
<td>blocksort.c.mainGtU_init</td>
<td>Poor code generation# (DS)</td>
</tr>
<tr>
<td>hmer [104]</td>
<td>fast_algorithms.c:loop(119)</td>
<td>No-vectorization (DS/SS)</td>
</tr>
<tr>
<td>h264ref [104]</td>
<td>mv-search.c:loop(394)</td>
<td>Missed inlining (SL)</td>
</tr>
<tr>
<td>povray [104]</td>
<td>csg.cpp:loop(248)</td>
<td>Missed inlining (DS)</td>
</tr>
<tr>
<td>Chombo [27]</td>
<td>PolytropicPhysics.F.ChF:(434)</td>
<td>Inattention to perf. (DS)</td>
</tr>
<tr>
<td>botsspar [105]</td>
<td>sparselu.c.fwd</td>
<td>Redundant computation (SL)</td>
</tr>
<tr>
<td>imagick [105]</td>
<td>magick_effect.c:loop(1482)</td>
<td>Redundant computation (SL)</td>
</tr>
<tr>
<td>SMB [86]</td>
<td>msgrate:ccache_invalidate</td>
<td>Redundant computation (SL)</td>
</tr>
<tr>
<td>backprop [91]</td>
<td>bpn_adjust_weights</td>
<td>Redundant computation (SS)</td>
</tr>
<tr>
<td>lavaMD [91]</td>
<td>kernel_cpu.c:loop(117)</td>
<td>Redundant computation (SL)</td>
</tr>
<tr>
<td>vacation [85]</td>
<td>client.c:loop(199)</td>
<td>Redundant computation (SL)</td>
</tr>
<tr>
<td>NWChem-6.3 [111]</td>
<td>tcp_nw2e_trans.F(240)</td>
<td>Useless initialization (DS/SS)</td>
</tr>
<tr>
<td>Cafeto-1.0 [52]</td>
<td>pooling_layer.cpp(289)</td>
<td>Redundant computation (SS)</td>
</tr>
<tr>
<td>Binutils-2.27 [38]</td>
<td>dwarf2.c(1561)</td>
<td>Linear search algorithm (SL)</td>
</tr>
<tr>
<td>Kalisto-0.43 [79]</td>
<td>KmerHashTable.h(131)</td>
<td>Poor hashing (SL)</td>
</tr>
</tbody>
</table>

WS* means whole-program speedup after problem elimination.
# DS means dead store, SS means silent store, SL means silent load.

newly found issues via Witch # used gcc-4.1.2

Table 4.3: Performance improvement guided by Witch.

4.8.1 NWChem-6.3

NWChem [111] is a production computational chemistry package, which implements several quantum mechanics and molecular mechanics methods. NWChem consists of six million lines of code written primarily in Fortran and C and parallelized with MPI [80]. We use the QM-CC aug-cc-pvdz input and eight MPI processes in our studies.

DeadCraft reports that more than 60% of memory stores are dead. Figure 4.6 shows the full calling contexts of the top (94% contribution to total dead writes) dead and killing store pair in the call of function dfill, which zeroes the array work2. With the given input, calls to dfill repeat more than 200K times, resulting in writing 500GB data that are never used. With further analysis, we identified that the size of work2 was larger than necessary, and the zero initialization was unnecessary, leading to the dead and killing writes in the same location. We eliminate this unnecessary initialization, yielding a 1.43× speedup. This bug, which was hiding in the large code base, is now fixed. Witch incurs only 6% runtime overhead whereas the fine-grained profiler, DeadSpy, incurs > 10× slowdown identifying the same problem.
Figure 4.6: The pair of dead and kill stores with full contexts reported by Witch’s dead store client.

Listing 4.4: Silent stores to array `bottom_diff` in Caffe.

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

4.8.2 Caffe-1.0

We apply SilentCraft on the deep learning framework Caffe [52]. We study the OpenMP C++ CPU version, which uses Intel MKL [48] to parallelize its computation kernels. We use the CIFAR-10 dataset to train the CIFAR network with 0.9 momentum, 4e-3 weight decay, 1e-3 learning rate, 128 batch size. We run Caffe with eight threads.

SilentCraft attributes 25% of total memory stores as redundant in a loop nest belonging to a major computation kernel in pooling and normalization layers (Listing 4.4). The memory stores to the array `bottom_diff` (Line 8) account for 17% of total silent stores. A large portion of elements in `top_diff` are zeroes; hence the same values overwrite the existing values in the same memory location of `bottom_diff`. The iteration over all the elements of `bottom_diff` in the four-level nested loop amplifies the fraction of silent stores. We optimize this code by introducing a check for the value in `top_diff`. If it is a zero, we bypass a division, an addition, and a memory store. This optimization speeds up the pooling layer by 1.16× the normalization layers by 1.34×. We observe 1.03× speedup for the entire program. We further relax the check for the absolute value in `top_diff` with a small delta 1e-7 rather than 0. If it is smaller, we bypass the computation for approximate re-
Listing 4.5: Redundant loads in `binutils-2.27` `dwarf2.c` file. Linear searches load the same values from same locations.

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

### 4.8.3 GNU Binutils-2.27

GNU Binutils [38] is a collection of binary tools used by many binary analysis tools such as Pin [75] and command-line tools such as `objdump` [37]. Disassembling an object file containing many functions using `objdump` with `-d -S -l` flags (map assembly to symbol and source lines) is unusually slow. We profile `objdump` in `binutils-2.27` using `LoadCraft` by disassembling the LULESH-2.0 [59] binary, which contains many functions. `LoadCraft` identifies 96% of the loads in the program as loading the same value from the same location. The top contributor is the Line 5 (Listing 4.5) in the function `lookup_address_in_function_table` with 70% redundant loads attributed to it. The function performs a linear scan over the addresses covered by each line of each function, maintained as a linked list, looking for the best match for a given address range.

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

### 4.8.4 SPEC OMP2012 367.imagick

SPEC OMP2012 367.imagick [105] is an OpenMP software to manipulate bitmap images. With the ref input and eight threads, `LoadCraft` reports that more than 99% of total memory loads are redundant and 85% of the redundant loads are associated with the loop nests shown in Listing 4.6.

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

4.8.5 Discussion on Other Optimizations

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

We presented a subset of programs where we found inefficiencies using witchcraft. Kallisto-0.43 [79] is an important RNA-sequencing software where LoadCraft found more than 98% redundant loads. The problem was a large, linear-probing hash-table with excessive hash collisions. We fixed Kallisto by reducing the load factor on the hash table and gained 4.1× speedup. Vacation is a STAMP [85] transactional memory benchmark, where we found unnecessary calls to a hash-table lookup of an item that was already found in the previous line of the code. Memoizing the result of the previous lookup resulted in 1.3× speedup. The results from our tools showed us that SPEC CPU2006 lbm is an excellent candidate for approximate computing; we applied loop perforation [101] to lbm and obtained 1.25x speedup with insignificant (7.7e-5%) accuracy loss.
Chapter 5

Conclusion

The hardware nowadays is developing quite fast. Softwares need to be efficient at all scales to take full use of the hardware resources. However, inefficiencies arise due to various causes. Unnecessary operation is typical representative among various inefficiencies occurring in the code level. Compilers which we rely on traditionally help to eliminate some redundancies with static analysis which is not enough since compilers do optimizations conservatively with limited view of scope. Classical runtime profiling tools focus on revealing how resources are used by using which lots of manual efforts are in need to root cause inefficiencies. Due to the severity of the problem and limitations of existing methodologies, this thesis states that new profiling techniques are in need to pinpoint where the unnecessary operations happen.

This thesis includes three profilers, RVN, RedSpy and Witch exposing different kinds of unnecessary operations from different points of view. RVN implements value numbering technique at runtime and helps use to identify some significant computation redundancies in several benchmarks. RedSpy is another fine-grained profiler exploring value locality during program execution. Value locality occurs over time in same storage locations (temporal) and in the neighborhood of a storage location (spatial). RedSpy can monitor data manipulation in both memory and registers. RedSpy incorporates techniques to recognize when floating-point values are approximately the same, thus offering new venues to tune code for approximate computations. Witch is the first lightweight resource wastage investigation framework in the market. Witch employs PMU sampling to get the first touch to memory locations and then set hardware debug registers to monitor the sampled locations with which, Witch is able to monitor consecutive memory accesses and explore useless data manipulation. Atop Witch, inefficiency-detection tools are build which are at least an order of magnitude faster than the state-of-the-art exhaustive-instrumentation tools with the same capabilities. Guided by RVN, RedSpy and Witch, with little effort, we are able to achieve great speedup in many important, complex codes bases unfamiliar to us. The optimizations we applied are platform compatible and are demonstrated on several architectures. We further demonstrate the effectiveness of our tools in several parallel software projects that were subject of optimization for decades.
Thesis Confirmation  This thesis proposes new profiling methodologies exposing different kinds of unnecessary operations to improve the code quality of software. All the new profilers work on executables fully optimized by compilers. Thus, problems reported by these profilers are failed by compile time analysis. These new profilers are also able to detect problematic code sections missed by state-of-the-art performance profilers like bwaves, gcc and h264ref from SPEC 2006. More experiments were conducted and have demonstrated that our new profilers are necessary and widely useful.

5.1 Innovation Highlights

All the three profilers included in this thesis are open-sourced. RVN and REDSPY are available from https://github.com/CCTLib while WITCH is available from https://github.com/WitchTools/. Besides the three profilers, more key contributions are summarized as follows.

Instrument with Sliding Window  Monitoring every single instruction would blow up the runtime and memory overhead significantly. RVN provides a sliding window implementation allowing one to tune instrumentation proportion. The higher the proportion, the more comprehensive result, and the higher overhead. This instrumentation flexibility allows users to customize the analysis.

Temporal and Spatial Value Locality  REDSPY demonstrates the existence and significance of value locality. Value locality happens when the same data has already present at the same storage location (temporal locality) or adjacent locations (spatial locality). Different from traditional instruction-based analysis, REDSPY looks into the inefficiencies through a completely new channel, the data’s point of view. With this new analysis mode, REDSPY would be able to expose more inefficiencies in the application where previous works failed.

Approximation Checking for Floating Points  When exploring value locality, instead of checking whether “same” data are manipulated, REDSPY provides approximation comparison for floating points data objects. In most approximation programming, result with little variation (e.g. ~5%) is completely acceptable. By examining whether “similar” data are frequently written to (or read from) the same (or adjacent) locations, one will export more opportunities in trading off the accuracy for better performance. The similarity can be adjusted based on how developers can tolerate the variation.

Monitoring Consecutive Accesses with Sampling  In previous works, when one needs to monitor the consecutive memory accesses, we either use instrumentation or tracing both of which have non-ignorable overhead. WITCH proposes a new approach that allows us to monitor the consecutive memory accesses with negligible extra CPU and memory usage. WITCH combines the PMU and debug registers. PMU makes it possible to sample memory accesses (get first access), and then debug register can be set to watch on the following-on accesses to that sample (get second access). On top of this framework, different crafts
can be build to detect various unnecessary memory operations including dead write, silent write, and silent load.

**New Linux Kernel Patch to Better Support Debug Registers** System calls are provided to open, set, close the debug registers. In Witch, we need to frequently reset the debug registers to monitor new coming samples. With existing support, we have to close the debug register first and then reopen it to watch on new memory locations. Massive closing/reopening would introduce significant overhead which is non transparent in sampling based analysis. Witch modifies the Linux kernel and adds support for resetting the debug registers online without closing and reopening it.

### 5.2 Research Highlights

Our paper “RVN: Pinpointing Redundant Computations” got accepted by PACT’15. Paper “RedSpy: Exploring Value Locality in Software” was published in ASPLOS’17 and nominated as the best paper candidate. Paper “Watching for Software Inefficiencies with Witch” got accepted by ASPLOS’18 and is nominated to ACM SIGs for CACM Research Highlights.
Bibliography


[23] **DEHAO CHEN, NEIL VACHHARAJANI, ROBERT HUNDT, SHIH-WEI LIAO, VINODHA RAMASAMY, PAUL YUAN, WENGUANG CHEN, AND WEIMIN ZHENG.** Taming


[34] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective Data-race Detection for the Kernel. In Proceedings of the 9th


81

