

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

APPROVAL PAGE

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

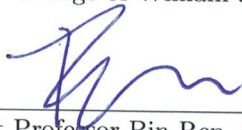
Assistant Professor Xu Liu, Computer Science
College of William & Mary



Professor Weizhen Mao, Computer Science
College of William & Mary



Professor Andreas Stathopoulos, 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

	Handling parallel programs	12
2.5	Algorithmic Refinement	13
2.5.1	Reducing Time Overhead	13
	Approximate hashing	13
	Selective instruction instrumentation	14
	Bursty sampling	14
2.5.2	Reducing Space Overhead	14
2.5.3	Providing Insights for Optimization	15
	Metric to quantify redundancy	16
2.6	Experiments	16
2.6.1	410.bwaves	17
2.6.2	456.hmmmer	19
2.6.3	434.zeusmp and 173.applu	20
2.6.4	Sweep3D	21
3	RedSpy: Exploring Value Locality in Software	24
3.1	Introduction	24
3.2	Related Work	26
	3.2.1 Traditional Value Profiling	26
	3.2.2 Other Redundancy Optimization Techniques	28
3.3	Methodology	28
	Exploiting Value Locality.	30
	Limitations:	30
3.4	Detection of Value Redundancies	30
	3.4.1 Temporal Redundancy	30
	Memory Temporal Redundancy.	31
	Register Temporal Redundancy.	31
	Value Approximation in Temporal Redundancy.	32

	Metric of Temporal Redundancy.	32
3.4.2	Spatial Redundancy	33
	Spatial Redundancy in Memory.	34
	Spatial Redundancy in Registers.	34
	Approximating Spatial Redundancy.	34
3.5	Recording and Reporting Redundancy	34
	Attributing Memory-temporal Redundancies.	35
	Attributing Register-temporal Redundancies.	35
	Attributing Memory-spatial Redundancies.	36
	Attributing Register-spatial Redundancies.	36
	Sampling for Low Overhead.	36
	Handling Parallel Programs.	36
	Presentation.	36
3.6	Experiments	37
	Volume of Redundancy.	37
	Sampling Accuracy.	39
	REDSPY Overhead.	39
3.7	Case Studies	40
	3.7.1 SPEC CPU2006 h264ref	41
	3.7.2 NWChem	42
	3.7.3 Rodinia LavaMD	42
	3.7.4 Rodinia Hotspot	43
	3.7.5 Rodinia Backprop	44
	3.7.6 Rodinia Particlefilter	44
	3.7.7 Comparison with Other State-of-the-art Tools	45
4	Watching for Software Inefficiencies with WITCH	46
	4.1 Introduction	46

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

4.8.1	NWChem-6.3	65
4.8.2	Caffe-1.0	66
4.8.3	GNU Binutils-2.27	67
4.8.4	SPEC OMP2012 367.imagick	67
4.8.5	Discussion on Other Optimizations	68
5	Conclusion	69
	Thesis Confirmation	70
5.1	Innovation Highlights	70
	Instrument with Sliding Window	70
	Temporal and Spatial Value Locality	70
	Approximation Checking for Floating Points	70
	Monitoring Consecutive Accesses with Sampling	70
	New Linux Kernel Patch to Better Support Debug Registers	71
5.2	Research Highlights	71
	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 <i>RVN</i> 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	REDSKY's space and time overheads in the unit of times (\times) on SPEC CPU2006 benchmarks.	40
3.5	Overview of performance improvement guided by REDSKY on different platforms.	40
3.6	REDSKY vs. other tools: whether value redundancies identified by REDSKY can be identified by other tools.	45
4.1	Runtime slowdown (\times) and memory bloat (\times) 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 $\langle operator, VN(\hat{S}[0]), VN(\hat{S}[1]), \dots \rangle$ 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	<p>Detecting dead writes using WITCH. The client, DeadCraft, subscribes to the precise PMU store event with a desired sample period. ① PMU counter overflows triggering an interrupt. ② WITCH handles the signal, extracts the calling context (C_{watch}) of the interrupt and the address accessed (M), and offers the triplet $\langle C_{watch}, M, AccessType \rangle$ to DeadCraft. ③ DeadCraft asks WITCH to monitor subsequent load or store to M. ④ WITCH sets a watchpoint to monitor M, and the execution continues ⑤ Program accesses M, which causes a CPU trap. ⑥ WITCH handles the trap signal, extracts the calling context (C_{trap}), and offers the triplet $\langle C_{trap}, M, AccessType \rangle$ to DeadCraft. ⑦ If the <i>AccessType</i> is a store, DeadCraft infers a dead write and attributes it to $\langle C_{watch}, C_{trap} \rangle$.</p>	51
4.2	<p>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.</p>	56
4.3	<p>(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.</p>	59
4.4	<p>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.</p>	63
4.5	<p>Comparison of dead writes with different number of debug registers. Error bars are for different (100K - 100M) sampling intervals.</p>	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 to 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

1. $x = a^1 * b^2$	$\langle *, 1, 2 \rangle = 4$; $VN(x) = 4$
2. $x = x + c^3$	$\langle +, 3, 4 \rangle = 5$; $VN(x) = 5$
3. $d = a$	$VN(d) = VN(a) = 1$
4. $e = b$	$VN(e) = VN(b) = 2$
5. $y = d * e$	$\langle *, 1, 2 \rangle$ redundant! $VN(y) = 4$
6. $z = y + c$	$\langle +, 3, 4 \rangle$ redundant! $VN(z) = 5$

Figure 2.1: An example of value numbering.

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 $(*c + *d)$ 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 `CalleeModule` in Listing 2.3 called from the function `CallerModule` makes the mod computation on Line 11 redundant. However, since `CalleeModule` and `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 `icc`, nor `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.

```

1 /* a and c alias each other */
2 /* b and d alias each other */
3 int AliasRedundancy(int * a, int *b,
4 int *c, int *d){
5     int v1 = *a + *b;
6     int v2 = *c + *d;
7     return v1 + v2;
8 }

```

Listing 2.1: Redundancy due to aliasing.

```

1 void PathSensitiveRedundancy(){
2 BB1:    v1 = a + b;
3     if(cont){
4 BB2:    a = c;
5         b = d;
6     }
7 BB3:    v2 = c + d;
8 }

```

Listing 2.2: Path-sensitive redundancy.

```

1 -----Compilation Unit 1-----
2 int CalleeModule(int a, int b){
3     if (a%b == 0)
4         ...
5     else
6         ...
7 }
8 -----Compilation Unit 2-----
9 void CallerModule(int a, int b){
10    CalleeModule(a, b);
11    if (a%b == 0)
12        ...
13    else
14        ...
15 }

```

Listing 2.3: Redundancy across compilation units.

```

1 /* callee function */
2 int Callee(int *a, int *b, int scaleFactor){
3     for(int i = 0 ; i , N; i++){
4         a[i] = a[i] * N;
5         b[i] = b[i] * N;
6     }
7 }
8 /* caller function */
9 void Caller(){
10    int * a = new int[N];
11    int * b = new int[N];
12    // Init a;
13    memcpy(b, a, N * sizeof(int));
14    Callee(a, b, alpha /*scaleFactor*/);
15 }

```

Listing 2.4: Input-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 neighbor elements in each direction. We partially eliminate the redundancies by reusing the addition result computed in the `j` direction¹. 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

¹Redundancies can be further reduced with transformation in `i` direction.

```

1 for ( i = T; i < N - T; i++) {
2   for ( j = T; j < M - T; j++){
3     temp = 0;
4     for ( k = 1; k < T; k++)
5       temp += matrix[i-k][j] + matrix[i][j-k] + matrix[i+k][j] + matrix[i][j+k];
6     matrix[i][j] += temp;
7   }
8 }

```

Listing 2.5: Redundancies in a stencil code.

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

Program	Orig.	Optimize.	%Reduction
#Instructions(billion)	72.8	39.64	45.6
#cycles(billion)	32.74	20.97	35.8

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

```
1: /* data structures */
2: uint64_t gValue = 0;
3: unordered_map<Key_t, uint64_t> VNMap;
4: I := instruction about to execute;
5: P := I's operator;
6: T := I's target operand;
7: S := {I's source operands in order};
8: for each source operand S[i] in S do
9:   if S[i] is not already assigned a VN then
10:    VN(S[i]) := ++gValue;
11:   end if
12: end for
13: if —S— == 1 and P copies source operand S[0] to T then
14:   VN(T) := VN(S[0]);
15: else
16:   if P is commutative then
17:     $\hat{S}$  := sort( VN(S[0]), VN(S[1]), ...);
18:   else
19:     $\hat{S}$  := {VN(S[0]), VN(S[1]), ...};
20:   end if
21:   hashKey := Hash(P,  $\hat{S}$ [0],  $\hat{S}$ [1], ...);
22:   if VNMap contains hashKey then
23:     record redundancy in the RedundancyTable ;
24:   else
25:     VNMap[hashKey] := ++gValue;
26:   end if
27:   if T  $\neq$  NULL then
28:     VN(T) := VNMap[hashKey];
29:   end if
30: 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.

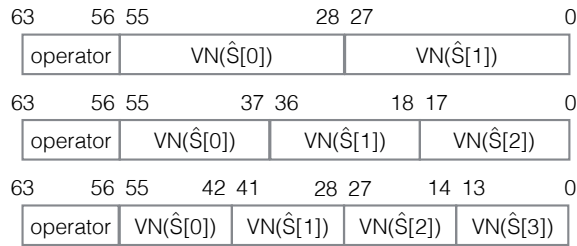


Figure 2.2: Hashing $\langle operator, VN(\hat{S}[0]), VN(\hat{S}[1]), \dots \rangle$ to a 64-bit integer.

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 invalidated 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 $44\times$ runtime overhead and $8\times$ 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

Program	% Redundant Fraction (\hat{R})			
	Min	Max	Average	GeoMean
bzip2	9.56	20.72	15.34	14.61
gcc	18.40	21.28	20.28	20.23
mcf	4.16	10.12	6.27	5.76
hmmer	18.42	20.27	19.53	19.51
libquantum	0.64	6.69	2.92	1.84
h264ref	12.91	13.16	13.07	13.07
omnetpp	21.52	22.81	22.29	22.28
astar	26.82	28.13	27.68	27.68
bwaves	3.12	6.73	4.67	4.43
zeusmp	4.29	4.75	4.51	4.51
gamess	15.50	18.26	16.77	16.73
milc	6.23	6.58	6.45	6.45
gromacs	8.89	11.32	9.75	9.69
leslie3d	3.60	4.21	3.86	3.85
namd	3.13	3.18	3.16	3.16
soplex	9.67	20.07	13.58	3.85
povray	18.28	20.20	19.56	19.54
calculix	12.84	24.75	18.80	17.83
gemsRDTD	4.04	11.95	6.79	5.95
tonto	13.97	18.40	15.47	15.34
lbm	8.28	9.00	8.57	8.56
wrf	5.26	7.72	6.62	6.54
sphinx3	3.46	12.09	6.94	6.04
applu	4.67	16.11	9.11	7.90
GeoMean	7.82	12.59	10.16	9.68

Table 2.3: Overhead of *RVN* with sampling

Program	Time	Space
bwaves	41.8×	5.8×
zeusmp	27.6×	7.4×
hmmer	46.5×	15.3×
applu	45.4×	7.3×
sweep3d	44.7×	4.1×

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

Program	Procedures:loops	gcc (-O2)			gcc (PGO)			icc (-O2)		
		%Cycle [†]	%Ins. [†]	WS [‡]	%Cycle	%Ins.	WS	%Cycle	%Ins.	WS
410.bwaves	block_solver.f:loop(167)	-97.1	-97.3		-96.9	-97.1		-55.1	-52.6	
	jacobian_lam.f:loop(30)	-16.0	-16.8	1.07×	-16.0	-16.8	1.12×	-6.3	-2.9	1.04×
	shell_lam.f	-13.6	-15.2		-10.9	-13.5		-9.1	-3.9	
434.zeusmp	lorentz.f:loop(675 or 552)	-22.0	-6.3		-6.7	-<0.1		-4.2	+8	
	forces.f:loop(466)	-15.8	+6.9	1×	-10.8	-6.9	1×	-13.45	+4.6	1×
	pdv.f:loop(317)	-6.5	-10.5		-5.6	-12.8		-18.1	+29.8	
456.hmmmer	hmmcalibrate.c:loop(499)	-13.9	-7.2		-6.5	-15.8		-5.9	-8.8	
	fast_algorithms.c:loop(119)	-14.1	-16.9	1.06×	-7.3	-15.7	1.06×	-6.1	-8.9	1.09×
173.applu	applu.f:loop(2660)	-30.1	-3.4	1×	-9.1	+2.8	1×	-0.58	+0.44	1×
sweep3d	sweep.f:loop(397)	-26.3	+4.3	1.08×	-21.0	-9.2	1.05×	-39.6	-6.2	1.22×

[†]The percentages (%) of cycle and instruction reduction (-) and increment (+) due to redundancy elimination.

[‡]WS means whole-program speedup due to redundancy elimination.

```

1 ros=q(1, ip1, jp1, kp1)
2 us=q(2, ip1, jp1, kp1)/ros
3 vs=q(3, ip1, jp1, kp1)/ros
4 ws=q(4, ip1, jp1, kp1)/ros
5 a1=(1.0d0/ros-1.0d0/ro)/step
6 mu=(mu +((gm-1.0d0)*(q(5, ip1, jp1, kp1)/ros-0.5d0*(us*us+vs*vs+ws*ws)))*0.75d0)/2.0d0

```

Listing 2.6: Redundant power 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 significantly reduce 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 `gcc -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 0x7f1116acc810:__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 0x7f1116acc810:__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`.

```

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

```

Listing 2.7: Redundant `mod` in `bwaves`.

2.6.2 456.hmmmer

RVN pinpoints significant redundancies between Lines 2 and 4 in the loop shown in Listing 2.9². 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-

²DeadSpy [20] also found these redundancies as dead writes.

```

1 !perform k = 1 case
2 do k=2, nz-1
3 km1=k-1; kp1=k+1; j = 1; jm1 = ny; jp1 = 2
4 !perform i loop
5 do j = 2, ny - 1
6   jm1 = j - 1; jp1 = j + 1
7   !perform i loop
8   enddo
9   j = ny; jm1 = ny - 1; jp1 = 1
10  !perform i loop
11  enddo
12 !perform k = nz case

```

Listing 2.8: Optimized code in bwaves.

```

1 for (k = 1; k <= M; k++) {
2   mc[k] = mpp[k-1] + tpmm[k-1];
3   if ((sc = ip[k-1] + tpim[k-1]) > mc[k]) mc[k] = sc;
4   if ((sc = dpp[k-1] + tpdm[k-1]) > mc[k]) mc[k] = sc;
5   if ((sc = xmb + bp[k]) > mc[k]) mc[k] = sc;
6   mc[k] += ms[k];
7   if (mc[k] < -INFTY) mc[k] = -INFTY;
8   .....
9}

```

Listing 2.9: Memory write redundancies in hmmer.

pilation. 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 PG0`, 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

```

1  mov    (%r10,%rax,4),%ecx
2  add    0x0(%r13,%rax,4),%ecx    #mpp[k-1]+tpmm[k-1]
3  mov    %ecx, 0x4(%rdx)         #assign mc[k]
4  mov    0x18(%rsp),%rbx
5  mov    (%r9,%rax,4),%r15d
6  add    (%rbx,%rax,4),%r15d    #dpp[k-1]+tpdm[k-1]
7  mov    0x20(%rsp),%rbx
8  cmp    %ecx,%r15d             #%ecx is mc[k]
9  cmovge %r15d, %ecx
10 mov    %ecx, 0x4(%rdx)        #assign mc[k]

```

Listing 2.10: The assembly binary of the code in listing 2.9.

```

1 for (k = 1; k <= M; k++) {
2   int tmpmc = mpp[k-1] + tpmm[k-1];
3   if ((sc = ip[k-1] + tpim[k-1]) > tmpmc) tmpmc = sc;
4   if ((sc = dpp[k-1] + tpdm[k-1]) > tmpmc) tmpmc = sc;
5   if ((sc = xmb + bp[k]) > tmpmc) tmpmc = sc;
6   tmpmc += ms[k];
7   if (tmpmc < -INFTY) mc[k] = -INFTY;
8   else mc[k] = tmpmc;
9   .....
10}

```

Listing 2.11: Remove Memory write redundancies in hmmer.

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 ASCII 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 `sig` 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`, `phijb`, 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

```

1 do 11 i=ibeg-1,iend
2   d1b2oo(i) = ( ( g2b (i) * b2(i, j, k) )**2 - ( g2b (i-1) * b2(i-1, j, k) )**2 )
3             * g2ai (i) * g2ai (i)
4   d1b2po(i) = ( ( g2b (i) * b2(i, j+1, k) )**2 - ( g2b (i-1) * b2(i-1, j+1, k) )
5             **2 ) * g2ai (i) * g2ai (i)
6   d1b3oo(i) = ( ( g31b(i) * b3(i, j, k) )**2 - ( g31b(i-1) * b3(i-1, j, k) )**2 )
7             * g31ai(i) * g31ai(i)
8   d1b3op(i) = ( ( g31b(i) * b3(i, j, k+1) )**2 - ( g31b(i-1) * b3(i-1, j, k+1) )
9             **2 ) * g31ai(i) * g31ai(i)
10  d2b3oo(i) = ( ( g32b(j) * b3(i, j, k) )**2 - ( g32b(j-1) * b3(i, j-1, k) )**2 )
11             * g32ai(j) * g32ai(j)
12  d2b3op(i) = ( ( g32b(j) * b3(i, j, k+1) )**2 - ( g32b(j-1) * b3(i, j-1, k+1) )
13             **2 ) * g32ai(j) * g32ai(j)
14      .....
15 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.

```

1      tmp1 = g32ai(j) * g32ai(j)
2      tmp4 = (g2b(ibeg-2) * b2(ibeg-2, j, k))**2
3      tmp6 = (g2b(ibeg-2) * b2(ibeg-2, j+1, k))**2
4      tmp8 = (g31b(ibeg-2) * b3(ibeg-2, j, k))**2
5      tmp10 = (g31b(ibeg-2) * b3(ibeg-2, j, k+1))**2
6      do 11 i=ibeg-1,iend
7          tmp2 = g2ai(i) * g2ai(i)
8          tmp3 = g31ai(i) * g31ai(i)
9          tmp5 = (g2b(i) * b2(i,j,k))**2
10         tmp7 = (g2b(i) * b2(i, j+1, k))**2
11         tmp9 = (g31b(i) * b3(i,j,k))**2
12         tmp11 = (g31b(i) * b3(i, j, k+1))**2
13
14         d1b2oo(i) = ( tmp5 - tmp4 ) * tmp2
15         d1b2po(i) = ( tmp7 - tmp6 ) * tmp2
16         d1b3oo(i) = ( tmp9 - tmp8 ) * tmp3
17         d1b3op(i) = ( tmp11 - tmp10 ) * tmp3
18
19         tmp4 = tmp5
20         tmp6 = tmp7
21         tmp8 = tmp9
22         tmp10 = tmp11
23         .....
24 11  continue

```

Listing 2.13: Remove inter-iteration redundancies in zeusmp.

```

1  do i = i0, i1, i2
2      ci = mu(m)*hi(i)
3      dl = ( sigt(i,j,k) + ci + cj + ck )
4      dl = 1.0 / dl
5      ql = ( phi(i) + ci*phiir + cj*phijb(i,lk,mi) + ck*phikb(i,j,mi) )
6      phi(i) = ql * dl
7      phiir = 2.0d+0*phi(i) - phiir
8      phii(i) = phiir
9      phijb(i,lk,mi) = 2.0d+0*phi(i) - phijb(i,lk,mi)
10     phikb(i,j,mi) = 2.0d+0*phi(i) - phikb(i,j,mi)
11 enddo

```

Listing 2.14: Redundnat 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

```

1 for (int i = 0 ; i < N; i++) {
2  /* Func() is side-effect free */
3  A[i] = 2 * Func(i);
4  /* use of A[i]. Line 3 is not a dead store */
5  ... = A[i];
6  /* A[i] gets the same value as after line 3 */
7  A[i] = Func(i)+Func(i);
8  /* use of A[i]. Line 7 is not a dead store */
9  ... = A[i];
10 }

```

Listing 3.1: Redundancy not detected by classic value profiling [14, 15, 35, 113] , DeadSpy [20], and RVN.

demonstrated in our case studies in §3.7, various forms of program inefficiencies—e.g., sub-optimal implementation choice and poor algorithmic choice—often manifest as redundant operations. Hence, runtime tracking different forms of redundancies offers visibility into program inefficiencies and hence offers new avenues to tune codes.

The classical value profiling [14, 15, 35, 113] has focused on identifying frequently occurring values at different program granularities such as functions, basic blocks, and instructions. While this classical approach helps identify a unit of code with potential for data or branch speculation, the approach does not render itself useful in identifying redundancies in execution—e.g., the value keeps changing unpredictably, but it is same as the one generated elsewhere in the program. For example, in Listing 3.1, the value of `A[i]` keeps changing and has no predictability. However, there is a redundancy between line 3 and line 7, which write the same value to `A[i]`. Existing fine-grained profilers [20] do not detect this redundancy. DeadSpy [20], which tracks accesses to every memory location, cannot recognize the redundancy between line 3 and line 7 because of an intervening “load” of the location `A[i]`. RVN, which assigns unique values to computations analogous to value numbering [10], assigns different values to the computations on line 3 and line 7 and hence fails to recognize the redundancy in this case.

Our work distinguishes from prior value profiling efforts by focusing on how a location often gets overwritten with the same (or approximately the same) value, regardless of the instructions involved in the computation. We classify value locality into two kinds: temporal and spatial. *Temporal value locality* indicates that the same value overwrites the same storage location; *spatial value locality* indicates that the nearby storage locations share a common value.

One can exploit value locality to eliminate redundant computations and tune performance. Redundancy arising from temporal value locality can be eliminated by removing redundant computations and redundant data movements. Redundancy arising from spatial value locality can be eliminated by memorization [83]—remember the value computed with a storage location and reuse it if the same computation is performed on an adjacent location. Not all redundancies seen in an execution need be eliminated. In our experience, a high fraction of redundancy demands an investigation and often yields a path to code tuning. Value locality is often a symptom of some kinds of redundancy; we use the terms value locality and redundancy interchangeably in this work.

The definition of both kinds of value locality can be relaxed from *the same value* to *approximately the same value* if approximate computation results can be tolerated. Samadi

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

```

1 int Temp(int a, int b){
2   int m = a * a;
3   int n = b * b;
4   int v1 = m - n;
5   c = a - b;
6   d = a + b;
7   v1 = c * d;
8   return v1;
9 }

```

Listing 3.2: Code example of temporal value locality.

```

1 void Spat(){
2   int * a = new int[N];
3   int * b = new int[N];
4
5   for(i=0; i<N; ++i){
6     a[i] = i/2 + 1;
7     b[i] = Foo(a[i]);
8   }
9 }

```

Listing 3.3: Code example of spatial value locality.

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 $a[i + 1]$ ($i = 0, 2, 4, \dots$) 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* $v=v'$ ($v \approx v'$ 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 $\sim 4\text{KB}$, 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 `AX`, `EAX`, or `RAX`, it affects the subsequent values read at `AL`, `AH`, `AX`, `EAX`, and `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 `shadow_A` of 64 bits for the entire alias group `AL`, `AH`, `AX`, `EAX`, and `RAX`. The writes to `AL`, `AX`, `EAX`, and `RAX` simply result in different sized writes into `shadow_A`. Writes to `AH` is a special case that writes to bits 8-15 of `shadow_A`.

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. REDSPY uses XED [51] 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 `IARG_REG_VALUE` mechanism, hence REDSPY uses `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 [50], which cannot even be read via `IARG_REG_CONST_REFERENCE` in Pin. In such situations, REDSPY resorts to using Pin’s heavyweight API `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, REDSPY fetches the generated 128-bit (XMM), 256-bit (YMM), or 512-bit (ZMM) values via Pin’s `IARG_REG_CONST_REFERENCE` argument passing. REDSPY, then compares the previously stored value with the current value. For efficiency, REDSPY uses SIMD for approximate equality comparison, which involves a SIMD subtraction followed by a SIMD division. Subsequently, 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.

Metric of Temporal Redundancy. 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:

$$\begin{aligned}
 R_{load}^{precise} &= \frac{\sum \text{non-FP bytes redundantly loaded from memory}}{\sum \text{non-FP bytes loaded from memory}} \\
 R_{load}^{approx} &= \frac{\sum \text{FP bytes redundantly loaded from memory}}{\sum \text{FP bytes loaded from memory}} \\
 R_{store}^{precise} &= \frac{\sum \text{non-FP bytes redundantly written to memory}}{\sum \text{non-FP bytes written to memory}} \\
 R_{store}^{approx} &= \frac{\sum \text{FP bytes redundantly written to memory}}{\sum \text{FP bytes written to memory}} \\
 R_{reg}^{precise} &= \frac{\sum \text{non-FP bytes redundantly generated in registers}}{\sum \text{non-FP bytes generated in registers}} \\
 R_{reg}^{approx} &= \frac{\sum \text{FP bytes redundantly computed in registers}}{\sum \text{FP bytes computed in registers}}
 \end{aligned}$$

Overall redundancy is:

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

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

$$\begin{aligned}
 R_{ins}^{precise} &= \frac{\sum \text{Non-FP dynamic instructions generating redundant value}}{\sum \text{Dynamic instructions executed}} \\
 R_{ins}^{approx} &= \frac{\sum \text{FP dynamic instructions generating redundant value}}{\sum \text{Dynamic instructions executed}}
 \end{aligned}$$

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 `AL`, 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,

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

$\hat{R}_{store}^{precise}$, $\hat{R}_{reg}^{precise}$, \hat{R}_{load}^{appx} , \hat{R}_{store}^{appx} , \hat{R}_{reg}^{appx} , $\hat{R}_{ins}^{precise}$, and \hat{R}_{ins}^{appx} 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.

Machines	Intel-SandyBridge	AMD	Intel-Xeon-Phi	IBM-POWER7
Processor	Xeon E5-4650@2.7GHz	Opteron 6168@1.6GHz	Xeon Phi 3100@1.1GHz	POWER7@3.5GHz
SMT x Cores	2 x 8	1 x 12	4 x 57	4 x 8
L1/L2/L3 Cache	32KB/256KB/20MB	64K/512K/10MB	32KB/512KB/NA	32K/256K/4MB
Memory	256GB DDR3	128GB DDR3	6GB GDDR5	256GB DDR3
Compiler	gcc 4.8.5 -O3 PGO	gcc 4.8.3 -O3 PGO	icc 15.0.0 -O3 PGO	xlc 13.1.0 -O2 PDF

Table 3.1: Machine configurations.

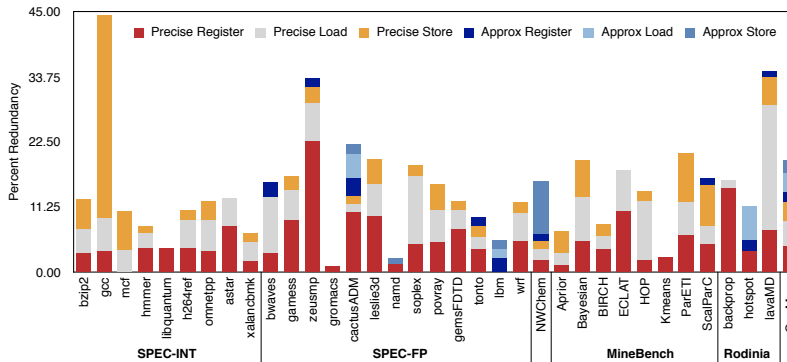


Figure 3.1: Breakdown of redundant bytes written in different benchmark suites.

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.

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%

Column 1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Program	Precise							Approximate						
	Fraction			%redundancy				Fraction			%redundancy			
	%reg	%load	%store	$R_{reg}^{precise}$	$R_{load}^{precise}$	$R_{store}^{precise}$	$R_{ins}^{precise}$	%reg	%load	%store	R_{reg}^{appx}	R_{load}^{appx}	R_{store}^{appx}	R_{ins}^{appx}
bzip2	44	35	19	7.6	11	28	8.7	<0.1	<0.1	<0.1	<0.1	<0.1	4.5	<0.1
gcc	32	26	42	11	23	83	21	<0.1	<0.1	<0.1	0.9	<0.1	80	<0.1
mcf	29	53	18	2.5	7	38	6.8	<0.1	<0.1	<0.1	<0.1	<0.1	<0.1	<0.1
hmmer	54	33	12	7.7	7.4	10	5.6	0.2	0.1	<0.1	6.5	<0.1	12	<0.1
libquantum	61	26	12	6.7	2.2	7.2	4.8	0.4	<0.1	<0.1	7.2	<0.1	20	<0.1
h264ref	48	39	11	8.7	13	14	6.5	0.1	<0.1	0.1	6.1	<0.1	40	<0.1
omnetpp	39	35	20	9.2	16	16	7.2	1.1	4.2	1	21	9.5	43	2.2
astar	63	32	4.9	13	15	9.9	8.3	0.1	<0.1	<0.1	1.3	<0.1	33	<0.1
xalancbmk	41	50	9.1	4.4	6.9	15	3.9	<0.1	0.15	<0.1	19	0.1	27	<0.1
SPEC-INT MAX	--	--	--	13	23	83	21	--	--	--	19	9.5	80	2.2
bwaves	39	26	2.2	8.6	37	29	8.1	24	3.7	4.5	11	0.4	20	3.6
garnet	48	27	7.1	19	19	33	11	11	5.2	2.2	3.8	1.8	30	1.3
mlc	53	9.8	4.2	0.9	2.8	8.5	1.7	21	8.5	3.4	2.5	1.7	26	2.3
zeusmp	55	14	5	41	49	54	31	20	4.4	1.4	7.4	2.2	23	2.4
gromacs	58	5.7	2.3	1.8	7.3	5.8	2.1	28	4.2	1.8	0.7	0.6	7.1	0.9
cactusADM	32	19	5.8	32	8.3	20	13	17	20	6.3	19	20	28	8.6
leslie3d	39	34	11	25	16	38	17	13	2.9	1	5.2	0.9	39	1.3
namd	55	8.2	2	2.6	0.5	3.6	1.7	25	7.5	2.3	1.3	3.5	47	2.2
soplex	43	31	6.8	11	38	30	9.1	9	8.8	2	8.2	7.9	11	2.2
povray	38	21	11	14	26	44	12	18	10	2	2.6	5.3	23	1.8
calculix	64	8.5	0.9	0.6	1.3	15	1.1	23	3.9	0.4	3.6	0.4	12	1.4
gemsFTD	44	30	8	17	11	21	11	15	1.6	0.9	3.6	0.4	30	1.2
tonto	45	19	7.5	8.6	12	24	6.7	18	6.7	3	7.5	4.8	17	2.9
lbn	66	3.9	2.2	0.3	4	6.8	1.1	19	7.3	1.8	13	21	93	8.3
wrf	40	22	5.5	14	21	39	9.7	28	4.2	1.3	3.4	1.8	35	2.8
sphinx3	63	11	1.5	0.4	5.9	47	2.1	22	2.4	<0.1	2.9	0.3	21	<0.1
SPEC-FP MAX	--	--	--	41	49	54	31	--	--	--	19	21	93	8.6
NWChem	28	39	5.7	7.5	4.7	24	6.7	12	5.2	10	10	1.4	89	7.5
Aprior	60	25	15	2	7.6	26	4.3	<0.1	<0.1	<0.1	10	<0.1	<0.1	<0.1
Bayesian	37	38	20	14	20	33	12	2.2	1.7	1.1	<0.1	<0.1	83	0.5
BIRCH	42	21	7.2	9.5	11	28	6.1	18	11	1.2	2.9	5.2	20	1.2
ECLAT	44	51	5.8	24	14	1.7	8.4	<0.1	<0.1	<0.1	1.6	<0.1	82	<0.1
HOP	42	22	4.4	5	45	42	10	25	5.7	1	0.6	<0.1	54	2.2
Kmeans	67	17	1.2	4	3.6	44	4	8.8	<0.1	0.6	0.2	<0.1	36	1.1
ParETI	55	30	15	12	19	57	14	<0.1	<0.1	<0.1	<0.1	<0.1	<0.1	<0.1
SVM-RFE	55	41	0.1	0.1	1.3	13	1.3	0.4	<0.1	<0.1	1.3	<0.1	46	0.1
ScalParC	53	20	16	9	16	44	11	9.4	2	<0.1	12	4	<0.1	1.4
MineBench MAX	--	--	--	24	45	57	14	--	--	--	12	5.2	83	2.2
backprop	44	41	1.2	33	3.6	80	9.2	12	1.4	<0.1	6.3	<0.1	23	1.4
hotspot	26	12	2.9	13	7.8	0.1	4.7	42	17	<0.1	4.7	35	<0.1	5.3
lavaMD	26	26	5.7	28	84	82	21	31	7.8	3.7	3.5	4.9	20	3
particlefilter	43	3.9	0.6	1.7	24	9.1	1.3	8.2	44	0.1	7.9	1.2	11	0.4
Rodinia MAX	--	--	--	33	84	82	21	--	--	--	7.9	35	23	5.3
All-MAX	--	--	--	41	84	83	31	--	--	--	21	35	93	8.6

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 $R_{reg}^{precise}$, $R_{load}^{precise}$, $R_{store}^{precise}$, and $R_{ins}^{precise}$ components. Column 12, 13, 14, and 15, respectively, decompose the observed approximate redundancies into R_{reg}^{appx} , R_{load}^{appx} , R_{store}^{appx} , and R_{ins}^{appx} 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% $R_{store}^{precise}$ in `gcc` is because of a repeated zero initialization of a large array, which has been well studied elsewhere [20]. We observe that R_{store} is often

Program	Overhead		Redundancy	
	w/o	w/	(w/o sampling)	(w/ sampling)
	sampling	sampling	R_{total}	\hat{R}_{total}
bzip2	39×	15×	8.2%	8.4%
bwaves	82×	20×	17%	17%
zeusmp	65×	8.4×	31%	31%
gromace	71×	6.6×	1.6%	1.6%
cactusADM	105×	8.6×	13%	13%
backprop	55×	25×	7.4%	8.4%
lavaMD	279×	74×	6.6%	6.6%
particlefilter	167×	68×	0.8%	0.6%

Table 3.3: Comparing overhead and redundancy with sampling enabled and disabled. The sampling covers 1% instructions.

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× and 9×, 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

Program	bzip2	gcc	mcf	hmmr	libquantum	h264ref	omnetpp	astar	xalancbmk	bwaves	garness	imic	zeusmp	gromace	cactusADM	leslie3D	namd	soplex	povray	calculix	gemsFDTD	torito	lbm	wrf	sphinx3	GeoMean
Time Overhead	19	20	7	14	12	34	11	11	31	14	23	6	8	8	6	7	12	12	24	11	8	16	6	15	12	12
Space Overhead	5.6	27	4	19	8	11	10	4	148	4	16	6	8	16	2	4	8	3	63	12	5	61	8	6	5	9

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

Redundancy types	Programs	Problematic procedures:loops	Intel SandyBridge		AMD WS	Xeon Phi WS	POWER 7 WS
			WS [‡]	PS [‡]			
Memory Temporal	464.h264ref	mv-search.c:loop(394)	1.34 \times	23%	1.36 \times	1.26 \times	1.27 \times
	backprop	bpnn.adjust_weights	1.01 \times	13%	1.14 \times	1.00 \times	1.08 \times
	NWChem*	tce_mo2e_trans.F:240	1.19 \times	9%	1.53 \times	–	–
Memory Spatial	particlefilter	particle_filter.c:loop(487)	1.10 \times	8%	1.04 \times	1.01 \times	1.05 \times
Register Temporal	lavaMD	kernel_cpu.c:loop(117)	1.50 \times	37%	1.64 \times	1.34 \times	2.72 \times
Spatial Approximation	hotspot	hotspot_openmp.cpp:loop(44)	2.21 \times	69%	2.19 \times	1.16 \times	1.63 \times

[‡]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.

instruction mix has more SIMD or x87 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 4.3 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 4.3, it is evident that REDSPY can guide exploiting value locality in various programs yielding significant performance gains. In addition to time savings, Table 4.3 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

```

-----
movq 0x18(%rsp), %rdi: SetupFastFullPelSearch:mv-search.c:419
FastFullPelBlockMotionSearch:mv-search.c:963
BlockMotionSearch:mv-search.c:2615
PartitionMotionSearch:mv-search.c:3272
encode_one_macroblock:rdopt.c:3096
encode_one_slice:slice.c:253
code_a_picture:image.c:236
frame_picture:image.c:798
encode_one_frame:image.c:409
main:lencod.c:413
*****REDUNDANT WITH *****
movq 0x18(%rsp), %rdi: SetupFastFullPelSearch:mv-search.c:419
FastFullPelBlockMotionSearch:mv-search.c:963
BlockMotionSearch:mv-search.c:2615
PartitionMotionSearch:mv-search.c:3272
encode_one_macroblock:rdopt.c:3096
encode_one_slice:slice.c:253
code_a_picture:image.c:236
frame_picture:image.c:798
encode_one_frame:image.c:409
main:lencod.c:413
-----

```

Figure 3.2: A redundancy pair reported in h264ref.

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 `abs_x`, `img_height`, and `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 is shown in Table 4.3.

```

412 for (pos = 0; pos < max_pos; pos++) {
413     ...
414     if(...) PelYline_11 = FastLine16Y_11;
415     else PelYline_11 = UMVLine16Y_11;
416
417     for (blk_y = 0; blk_y < 4; blk_y++) {
418         for (y = 0; y < 4; y++) {
419             refptr = PelYline_11(ref_pic, abs_y++, abs_x, img_height, img_width);
420             ... } ... } ...}

```

Listing 3.4: Temporal redundancy in SetupFastFullPelSearch function in h264ref.

```

-----
movapsx  %xmm1, -0x20(%rax):dfill_:src/util/dfill.f:12
tce_mo2e_trans_:src/tce/tce_mo2e_trans.F:240
text:src/tce/tce_energy.F:1326
tce_energy_fragment_:src/tce/tce_energy_fragment.F:101
.
.
.
main:nwchem-6.3/src/nwchem.F:347
*****REDUNDANT WITH *****
movapsx  %xmm1, -0x20(%rax):dfill_:src/util/dfill.f:12
tce_mo2e_trans_:src/tce/tce_mo2e_trans.F:240
text:src/tce/tce_energy.F:1326
tce_energy_fragment_:src/tce/tce_energy_fragment.F:101
.
.
.
main:nwchem-6.3/src/nwchem.F:347
-----

```

Figure 3.3: A redundancy pair reported in NWChem.

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 $1.53\times$ 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

```
238 c getting piece of atomic 2-e integrals
239 c zeroing ---
240 call dfill(work1, 0.0d0, dbl_mb(k_work1), 1)
241 call dfill(work2, 0.0d0, dbl_mb(k_work2), 1)
```

Listing 3.5: Temporal redundant memory writes in NWChem.

```
169 for (...)
170   for (...)
171     for (i=0; i<NUMBER_PAR_PER_BOX; i=i+1){
172       for (j=0; j<NUMBER_PAR_PER_BOX; j=j+1){
173         r2 = rA[i].v + rB[j].v - DOT(rA[i],rB[j]);
174         u2 = a2*r2;
175         vij= exp(-u2);
176         fs = 2.*vij;
177         .....}}}
```

Listing 3.6: Temporal redundant function call in lavaMD.

that the redundancy occurs at line 175, where frequently on consecutive invocations `exp()` return the same value, which is written to the register holding `vij`. With further investigation, we inferred that `r2` is often assigned the same value at line 173. Since `a2` is a loop invariant, `u2`, which is derived from `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 `exp()`, which is assigned to `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 `r2`'s value has not changed from the previous iteration, we reuse the value of `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 $1.50\times$ 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. REDSPY identifies high approximate spatial value locality in a two-dimensional array `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 `temp` array outside the nested loop. The spatial redundancy introduced $\sim 7\times$ runtime overhead. REDSPY pinpoints that all the values in `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 `Hotspot`. Instead of performing computation on all the cells of `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 $2.21\times$ speedup and 69% energy saving for the entire program.

```

44 for (r = 0; r < row; r++) {
45   for (c = 0; c < col; c++) {
46     ...
47   {
48     delta = (step / Cap) * (power[r*col+c] +
49       (temp[(r+1)*col+c]+temp[(r-1)*col+c]-2.0*temp[r*col+c]) / Ry +
50       (temp[r*col+c+1]+ temp[r*col+c-1] - 2.0*temp[r*col+c]) / Rx +
51       (amb_temp - temp[r*col+c]) / Rz);
52   }
53   result[r*col+c] =temp[r*col+c]+ delta;
54 }
55 }

```

Listing 3.7: Approximate spatial value locality in Hotspot.

```

321 for (j = 1; j <= ndelta; j++) {
322   for (k = 0; k <= nly; k++) {
323     new_dw = ((ETA * delta[j] * ly[k])+(MOMENTUM * oldw[k][j]));
324     w[k][j] += new_dw;
325     oldw[k][j] = new_dw;
326   }
327 }

```

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

`Particle_filter` 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`

```

486 #pragma omp parallel for shared(CDF, Nparticles, xj, yj, u, arrayX, arrayY) private(
      i, j)
487 for(j = 0; j < Nparticles; j++){
488     i = findIndex(CDF, Nparticles, u[j]);
489     if(i == -1)
490         i = Nparticles-1;
491     xj[j] = arrayX[i];
492     yj[j] = arrayY[i];
493 }

```

Listing 3.9: Spatial value locality in Particle_filter.

Program	REDSKY vs. other tools			
	DeadSpy [20]	RVN	Paraprox [99]	ThinLTO [56]
464.h264ref	No	No	No	Partial
NWChem	Yes	Yes	No	-
backprop	No	No	No	No
hotspot	No	No	Yes	No
lavaMD	No	Yes	No	No
particlefilter	No	No	No	No

Table 3.6: REDSKY vs. other tools: whether value redundancies identified by REDSKY 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× 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 REDSKY 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 REDSKY. 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. REDSKY 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].

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

```

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

```

Listing 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 \times) and consume enormous (6-100 \times) 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 $40\text{-}280\times$ 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 workaroud 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.

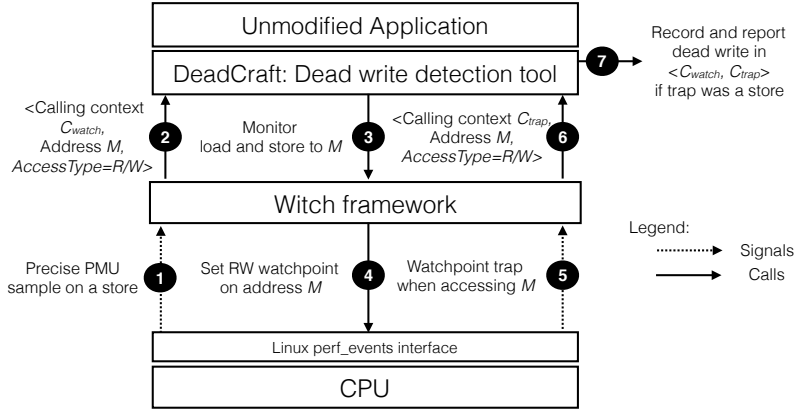


Figure 4.1: Detecting dead writes using WITCH. The client, **DeadCraft**, subscribes to the precise PMU store event with a desired sample period. ① PMU counter overflows triggering an interrupt. ② WITCH handles the signal, extracts the calling context (C_{watch}) of the interrupt and the address accessed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

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

4.4 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.¹ When the program accesses M next time, a CPU trap happens. WITCH handles the watchpoint exception, captures information associated with the trap, associates any information given by the client at the watchpoint subscribe time, and gives control to the client tool for appropriate actions.

We use our dead store detection tool—`DeadCraft`, shown in Figure 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_{watch} . In the PMU overflow handler, WITCH offers the triplet $\langle C_{watch}, M, AccessType \rangle$ to `DeadCraft`. `DeadCraft` memorizes the tuple and in-turn asks WITCH to set a `RW_TRAP` watchpoint W at M . The normal execution continues. W traps when the program accesses M next time; we defer discussing another sample happening before the trap to Section 4.4.1. Let the address accessed in the trap be M and let its calling context be C_{trap} . WITCH handles the trap and offers the triplet $\langle C_{trap}, M, AccessType \rangle$ to `DeadCraft`. If a *load* causes a trap, `DeadCraft` treats it as a useful operation and disables the watchpoint. If a *store* causes a trap, however, `DeadCraft` infers the store seen in the context C_{watch} as a dead store. It attributes a “unit” of dead store to the calling context pair $\langle C_{watch}, C_{trap} \rangle$.

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

4.4.1 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.

```

1 for(int i = 1; i <= 100K; i++){
2   array[i] = 0;
3 }
4 for(int j = 1; j <= 100K; j++){
5   array[j] = j;
6 }

```

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 $\{S_1, S_2, \dots, S_{k-1}\}$ with $1/(k-1)$ probability each. At the end of k^{th} sample, the probability of monitoring any sampled address S_i , $1 \leq i \leq (k-1)$ of the prior $(k-1)$ samples is:

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

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 α finds a place in a watchpoint, it can affect the subsequent samples. If no watchpoint has triggered for H samples when α is sampled, the expected number of samples before α will be replaced is $1.7H$, which follows from the sum of harmonic series. The number of debug registers does not influence α .

The number of consecutive PMU samples that are not monitored form a “blindspot” window; the longer the window is, the larger the probability of missing bugs. In our experience, many software in practice often have very short windows. For example, in the SPEC CPU2006 [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 $\langle 3, 11 \rangle$, $\langle 11, 3 \rangle$, $\langle 7, 8 \rangle$, and $\langle 8, 7 \rangle$. That is, 25% each. This expectation in quantification is not preserved with our sampling scheme because of a mixture of sparse monitoring (lines 3 and 11) and dense monitoring (lines 7 and 8). As soon as a watchpoint traps on Line 7, a debug register frees up; every subsequent PMU sample in the `k` loop will find a free debug register. Hence, there will be a disproportionately large number of dead writes recorded for the line pairs $\langle 7, 8 \rangle$ and $\langle 8, 7 \rangle$ compared to rest.

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

```

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

```

Listing 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 $\langle 3, 11 \rangle$ as ten.

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

$$\hat{D} = \frac{\sum_i \sum_j \text{waste in } \langle C_i, C_j \rangle}{\sum_i \sum_j \text{waste in } \langle C_i, C_j \rangle + \sum_i \sum_j \text{use in } \langle C_i, C_j \rangle} \quad (4.1)$$

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

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-

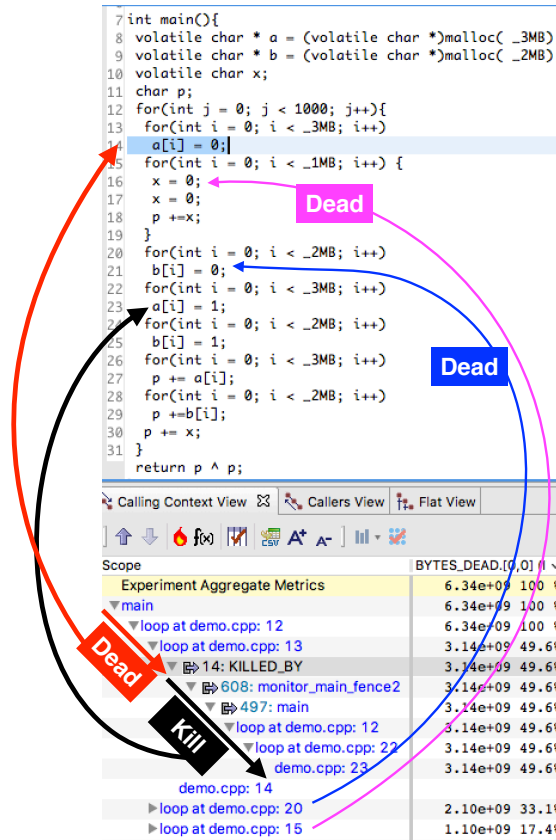


Figure 4.2: `a[]` and `b[]` and `x` are involved in dead writes in 3:2:1 ratio (50%:33%:17%), respectively. The sampling interval is 50K stores. Our proportional, context-sensitive scheme apportions dead writes in near perfect ratio.

portional 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 $\langle 16, 17 \rangle$.

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 ($\sim 3\%$) 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 $\sim 5\%$ 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 ($\sim 5\%$ 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,

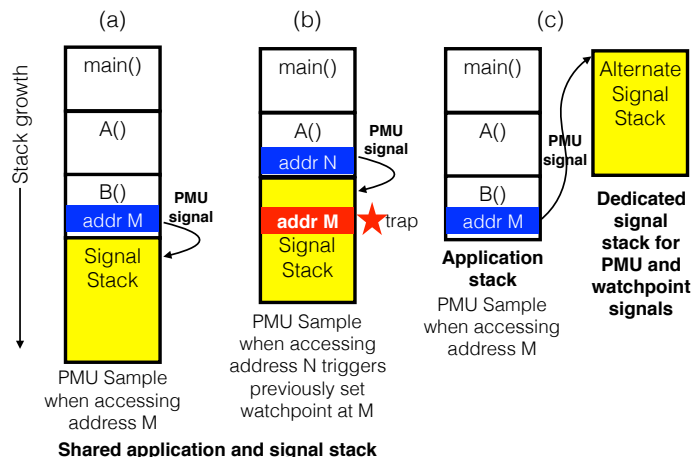


Figure 4.3: (a) A PMU sample happens in a deeper call stack when `B()` is accessing address `M`; signal handler sets a watchpoint to monitor the address `M`. (b) A shallower application call stack, function `A()`, triggers another PMU sample, the signal handler is established in a location that overwrites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

one watchpoint exception handler stack frame may trap another watchpoint.

We avoid this problem by establishing a separate signal-handler stack frame for both PMU signal handler and watchpoint exception handler using the Linux `sigaltstack` facility [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 $\langle C_{watch}, C_{trap} \rangle$ with proportional units of silent stores. Proportionality computation follows the previously discussed proportional attribution heuristic. To identify opportunities for approximate computation, for the floating-point operations, **SilentCraft** performs approximate equality check within a user-specified precision level. **SilentCraft** infers that a datum is a floating-point value by disassembling the instruction accessing the address.

SilentCraft quantifies the store redundancy \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]. HPCViewer, the graphical interface, enables navigating the CCT and the corresponding source code ordered by the monitored metrics. A top-down view shows a call path C starting from `main` to a leaf function with the breakdown of metrics at each level. WITCH tools discussed here need to attribute metrics to calling context *pairs* $\langle C_{watch}, C_{trap} \rangle$. Merely attributing a metric to two independent contexts loses the association between two related contexts during postmortem inspection. To maintain a correlation between a *source* context (e.g., dead) and target (killing) context, WITCH appends a copy of the *target* calling context to a source calling context. For example, if a store in context `main->A->B` is overwritten by another store in context `main->C->D`, DeadCraft constructs a synthetic calling context: `main->A->B->KILLED_BY->main->C->D`. The dead write metrics will be attributed to the leaf of this call chain. These synthetic call chains make it easy to visually navigate the CCT and focus on top redundancy pairs. Figure 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 `MEM_UOPS_RETIRED:ALL_STORES` whereas LoadCraft uses `MEM_UOPS_RETIRED:ALL_LOADS`. In our experiments, we use the nearest prime number for the shown sampling intervals, which is the recommended method in PMU sampling. The raw data from our experiments are available online [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-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 `hmmcr` 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.

`1bm` has $\sim 100\%$ silent stores and silent loads, but it has negligible dead stores. `1bm` 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

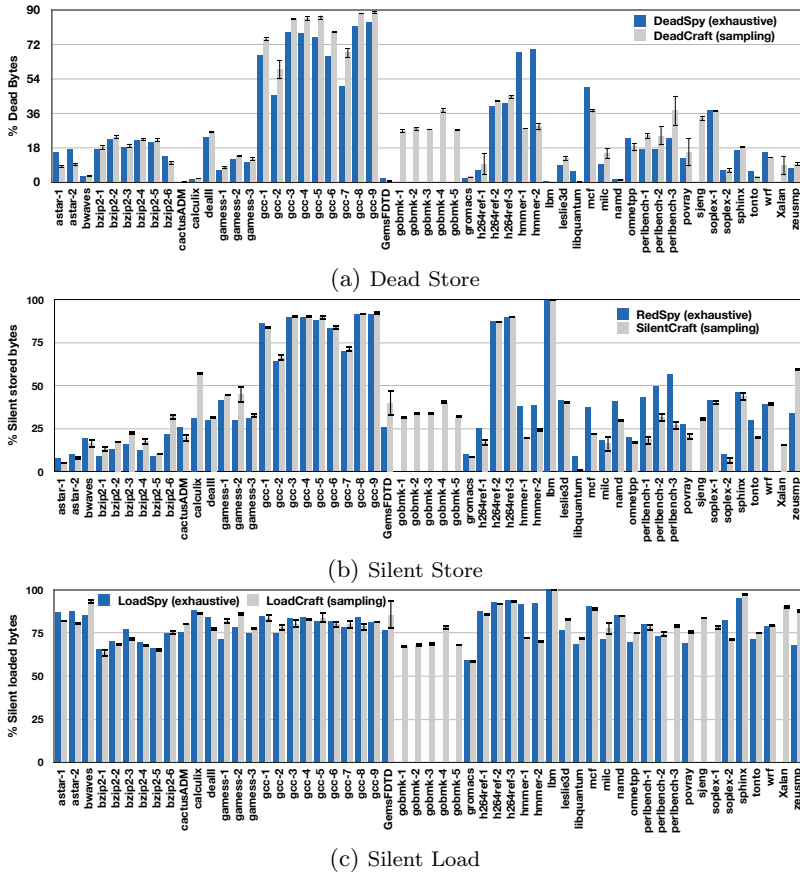


Figure 4.4: WITCH tools vs. instrumentation tools on SPEC CPU2006. Error bars capture different sampling rates. Ground truth instrumentation data is unavailable for `gobmk`, `sjeng`, and `Xalan` since they ran out of memory. The benchmarks with multiple inputs (e.g., `bzip2`) appear multiple times with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

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 `xalanbmk`, `sjeng`, and `gobmk` incur higher space and time overheads; and their instrumentation counterparts do not run to completion. Recursive codes with inefficiencies (e.g., `SilentCraft` on `gobmk` and `LoadCraft` on `xalanbmk`) exacerbate memory bloat due

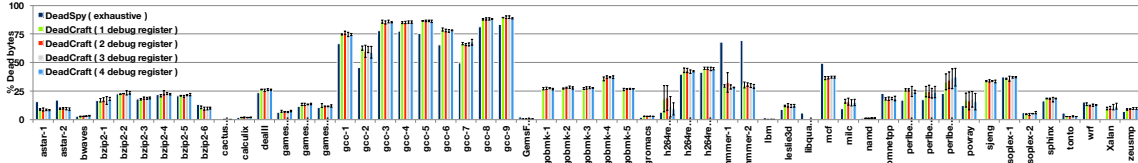


Figure 4.5: Comparison of dead writes with different number of debug registers. Error bars are for different (100K - 100M) sampling intervals.

Benchmark	ant1	ant2	bwaves	bjp2	calculusDM	calculus	death	games	gcc	GenSDTD	gobmk	gromacs	h24ref	hanner	libn	lelak3d	libquantum	mcf	milc	nand	omnetpp	perlbenc	povray	sjeng	soplex	sphinx3	tonto	vrf	yabachmk	zeusmp	GeoMean	Median	
Original Time (second)	139	303	64	371	635	246	50	24	297	71	317	138	160	342	215	173	221	458	318	182	65	101	367	86	423	408	312	158	360				
Original Memory Usage (MB)	232	875	562	664	118	795	22	459	22	831	30	16	38	16	411	125	95	1677	681	48	171	400	7	176	279	44	36	695	421	512			
Dead store	Slowdown (times)	DeadSpy	22.65	31.70	32.32	29.93	33.40	54.70	40.00	43.57	26.36	-	26.95	59.67	52.01	28.86	31.70	27.87	21.69	14.61	22.18	37.80	71.91	71.13	-	26.45	26.16	34.85	24.60	-	19.95	32.48	30.82
	Memory bloat (times)	DeadSpy	6.47	6.40	6.27	5.84	8.53	14.70	8.20	32.26	6.12	-	9.80	11.93	20.65	6.06	6.25	9.49	6.00	6.26	6.55	6.84	45.75	38.66	-	17.54	7.48	16.76	6.70	-	6.03	9.87	7.16
Silent store	Slowdown (times)	RedSpy	16.33	17.62	23.75	45.64	26.17	23.60	33.00	200.07	41.24	-	25.60	101.66	26.66	14.53	39.43	23.67	10.76	10.94	16.94	27.97	59.71	67.50	-	23.60	16.46	29.32	33.00	-	27.66	29.10	26.42
	Memory bloat (times)	SilentCraft	1.01	1.01	1.00	1.05	1.01	1.00	1.05	1.03	1.02	1.00	1.02	1.02	1.03	1.00	1.01	1.03	1.00	1.00	1.02	1.01	1.04	1.03	1.00	1.02	1.00	1.01	1.04	1.13	1.00	1.02	1.01
Silent load	Slowdown (times)	LoadSpy	30.00	87.70	53.00	123.00	75.30	81.30	100.00	51.80	69.60	-	39.80	185.00	95.30	15.10	98.60	36.80	26.90	26.90	46.10	36.00	82.00	156.00	-	31.20	60.10	54.10	81.90	-	51.00	58.66	57.10
	Memory bloat (times)	LoadCraft	1.04	1.00	2.16	1.69	1.09	1.00	1.12	1.04	1.08	1.00	1.06	1.04	1.04	1.00	1.09	1.01	1.00	1.02	1.58	1.00	1.51	1.05	1.00	1.00	1.01	1.05	1.10	1.86	1.08	1.13	1.04
Silent load	Slowdown (times)	LoadSpy	6.80	6.50	4.00	5.90	8.49	14.40	12.00	50.10	6.30	-	12.60	18.40	23.90	6.20	4.90	50.20	6.09	6.40	6.80	8.20	184.00	1051.00	-	13.20	9.30	36.70	5.20	-	6.30	13.52	8.35
	Memory bloat (times)	LoadCraft	1.03	1.01	1.03	1.00	1.08	1.01	1.36	1.03	1.01	2.01	1.43	1.36	1.64	1.01	1.07	1.11	1.00	1.01	1.13	1.04	1.02	3.55	1.14	1.05	1.19	1.87	1.02	24.93	1.02	1.33	1.05

Table 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 watchpoint traps and inefficiency reporting cost, 3) most PMU samples find a free debug register and incur the cost of arming it, and finally 4) LoadCraft sets the RW_TRAP watchpoint (x86 does not support break on load watchpoint), which triggers a spurious exception on a store. Table 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 \checkmark prefix) and confirms previously found performance issues [20]. In this section, we describe four case studies covering the analyses by the three WITCH tools.

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

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

Benchmark Information		WITCH	WS*
program	problem code	Inefficiencies (Client)	
gcc [104]	cselib.c:cselib_init	Poor data structure (DS)	1.33×
bzip2 [104]	blocksort.c:mainGtU_init	Poor code generation# (DS)	1.07×
hmmer [104]	fast_algorithms.c:loop(119)	No-vectorization (DS/SS)	1.28×
h264ref [104]	mv-search.c:loop(394)	Missed inlining (SL)	1.27×
✓ povray [104]	csg.cpp:loop(248)	Missed inlining (DS)	1.08×
✓ Chombo [27]	PolytropicPhysicsF.ChF:(434)	Inattention to perf. (DS)	1.07×
✓ botsspar [105]	sparselu.c:fwd	Redundant computation (SL)	1.15×
✓ imagick [105]	magick_effect.c:loop(1482)	Redundant computation (SL)	1.6×
✓ SMB [86]	msgrate.c:cache_invalidate	Redundant computation (SL)	1.47×
backprop [91]	bpnn_adjust_weights	Redundant computation (SS)	1.20×
lavaMD [91]	kernel_cpu.c:loop(117)	Redundant computation (SL)	1.66×
✓ vacation [85]	client.c:loop(198)	Redundant computation (SL)	1.31×
NWChem-6.3 [111]	tce_mo2e_trans.F(240)	Useless initialization (DS/SS)	1.43×
✓ Caffe-1.0 [52]	pooling_layer.cpp(289)	Redundant computation (SS)	1.06×
✓ Binutils-2.27 [38]	dwarf2.c(1561)	Linear search algorithm (SL)	10×
✓ Kallisto-0.43 [79]	KmerHashTable.h(131)	Poor hashing (SL)	4.1×

*WS means whole-program speedup after problem elimination.

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

✓ newly found issues via WITCH

used gcc-4.1.2

Table 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.

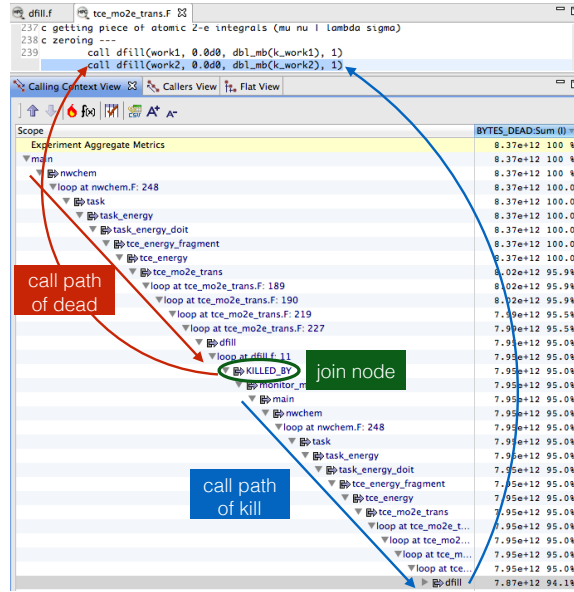


Figure 4.6: The pair of dead and kill stores with full contexts reported by WITCH’s dead store client.

```

1 for (int n = 0; n < top[0]->num(); ++n) {
2   for (int c = 0; c < channels_; ++c) {
3     for (int ph = 0; ph < pooled_height_; ++ph) {
4       for (int pw = 0; pw < pooled_width_; ++pw) {
5         ...
6         for (int h = hstart; h < hend; ++h) {
7           for (int w = wstart; w < wend; ++w) {
8 ►      bottom_diff[h * width_ + w] +=
9          top_diff[ph * pooled_width_ + pw] / pool_size;
10      }}}}
11      ...}}

```

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

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-

```

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

```

Listing 4.5: Redundant loads in binutils-2.27 dwarf2.c file. Linear searches load same the values from same locations.

sults. This optimization, with less than 2% accuracy loss, yields $1.16\times$ and $2.23\times$ speedups for the pooling and normalization layers, respectively. The entire program obtains a $1.06\times$ 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 with 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 $10\times$. 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,

```

1 for (y=0; y < (ssize_t) image->rows; y++) {
2   for (x=0; x < (ssize_t) image->columns; x++) {
3     for (v=0; v < (ssize_t) width; v++) {
4       for (u=0; u < (ssize_t) width; u++) {
5▶         pixel.red+>(*k)*kernel_pixels[u].red;
6▶         pixel.green+>(*k)*kernel_pixels[u].green;
7▶         pixel.blue+>(*k)*kernel_pixels[u].blue;
8           k++;
9 }}}}

```

Listing 4.6: Redundant loads to different fields of structure `pixel` and array `kernel_pixels` in `367.imagick`

which saves a memory load from address `k`, a multiplication, and a memory load to the field of `pixel`. This optimization yields a $1.6\times$ 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 $4.1\times$ 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 $1.3\times$ speedup. The results from our tools showed us that SPEC CPU2006 `1bm` is an excellent candidate for approximate computing; we applied loop perforation [101] to `1bm` and obtained $1.25\times$ 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. $\sim 5\%$) 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

- [1] CCTLib. <https://github.com/CCTLib/>.
- [2] Intel Pin. <https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool>.
- [3] NU-MineBench suite. <http://cucis.ece.northwestern.edu/projects/DMS/MineBench.html>.
- [4] The DWARF Debugging Standard. <http://www.dwarfstd.org>.
- [5] L. ADHIANTO, S. BANERJEE, M. FAGAN, M. KRENTEL, G. MARIN, J. MELLOR-CRUMMEY, AND N. R. TALLENT. HPCToolkit: Tools for Performance Analysis of Optimized Parallel Programs. *Concurrency Computation : Practice Experience*, 22(6):685–701, April 2010.
- [6] GLENN AMMONS, THOMAS BALL, AND JAMES R. LARUS. Exploiting hardware performance counters with flow and context sensitive profiling. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–96, NY, NY, USA, 1997. ACM.
- [7] JENNIFER M. ANDERSON, LANCE M. BERG, JEFFREY DEAN, SANJAY GHEMAWAT, MONIKA R. HENZINGER, SHUN-TAK A. LEUNG, RICHARD L. SITES, MARK T. VANDEVOORDE, CARL A. WALDSPURGER, AND WILLIAM E. WEIHL. Continuous Profiling: Where Have All the Cycles Gone? *ACM Trans. Comput. Syst.*, 15(4):357–390, November 1997.
- [8] R. BARIK AND V. SARKAR. Interprocedural Load Elimination for Dynamic Optimization of Parallel Programs. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 41–52, Sept 2009.
- [9] G. B. BELL, K. M. LEPAK, AND M. H. LIPASTI. Characterization of Silent Stores. In *Proceedings 2000 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.PR00622)*, pages 133–144, 2000.
- [10] P. BRIGGS, K. D. COOPER, AND L. T. SIMPSON. Value numbering. *Software-Practice and Experience*, 27(6):701–724, June 1997.

- [11] PRESTON BRIGGS AND KEITH D. COOPER. Effective Partial Redundancy Elimination. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 159–170, 1994.
- [12] M. BURROWS, Ú ERLINGSSON, S-T. A. LEUNG, M. T. VANDEVOORDE, C. A. WALDSPURGER, K. WALKER, AND W. E. WEIHL. Efficient and Flexible Value Sampling. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, pages 160–167, New York, NY, USA, 2000. ACM.
- [13] J. ADAM BUTTS AND GURI SOHI. Dynamic Dead-instruction Detection and Elimination. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 199–210, 2002.
- [14] BRAD CALDER, PETER FELLER, AND ALAN EUSTACE. Value profiling. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 30, pages 259–269, Washington, DC, USA, 1997. IEEE Computer Society.
- [15] BRAD CALDER, PETER FELLER, AND ALAN EUSTACE. Value Profiling and Optimization. *Journal of Instruction Level Parallelism*, 1, 1999.
- [16] STEVE CARR AND KEN KENNEDY. Scalar Replacement in the Presence of Conditional Control Flow. *Software Practice and Experience*, 24:51–77, 1992.
- [17] MILIND CHABBI, WIM LAVRIJSEN, WIBE DE JONG, KOUSHIK SEN, JOHN MELLOR-CRUMMEY, AND COSTIN IANCU. Barrier Elision for Production Parallel Programs. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP 2015, pages 109–119, New York, NY, USA, 2015. ACM.
- [18] MILIND CHABBI, XU LIU, AND JOHN MELLOR-CRUMMEY. Call Paths for Pin Tools. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 76:76–76:86, 2014.
- [19] MILIND CHABBI, XU LIU, AND SHASHA WEN. WitchTools. <https://github.com/WitchTools/Witch.git>, 2017.
- [20] MILIND CHABBI AND JOHN MELLOR-CRUMMEY. DeadSpy: a tool to pinpoint program inefficiencies. In *Proceedings of the 10th International Symposium on Code Generation and Optimization*, pages 124–134, 2012.
- [21] MILIND CHABBI, SHASHA WEN, AND XU LIU. Featherlight On-the-fly False Sharing Detection. In *Proceedings of the 23th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP 2018, New York, NY, USA, 2018. ACM.
- [22] MILIND CHABBI, SHASHA WEN, XU LIU, ET AL. CCTLib. <https://github.com/CCTLib/CCTLib>, 2014.
- [23] DEHAO CHEN, NEIL VACHHARAJANI, ROBERT HUNDT, SHIH-WEI LIAO, VINODHA RAMASAMY, PAUL YUAN, WENGUANG CHEN, AND WEIMIN ZHENG. Taming

- Hardware Event Samples for FDO Compilation. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '10*, pages 42–52, New York, NY, USA, 2010. ACM.
- [24] FRED CHOW, SUN CHAN, ROBERT KENNEDY, SHIN-MING LIU, RAYMOND LO, AND PENG TU. A New Algorithm for Partial Redundancy Elimination Based on SSA Form. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation, PLDI '97*, pages 273–286, 1997.
- [25] EUI-YOUNG CHUNG, LUCA BENINI, AND GIOVANNI DE MICHELI. Energy Efficient Source Code Transformation based on Value Profiling. In *PROC. INTERNATIONAL WORKSHOP ON COMPILERS AND OPERATING SYSTEMS FOR LOW POWER*, 2000.
- [26] CLIFF CLICK. Global code motion/global value numbering. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 246–257, 1995.
- [27] P COLELLA, DT GRAVES, ND KEEN, TJ LIGOCKI, DF MARTIN, PW MC-CORQUODALE, D MODIANO, PO SCHWARTZ, TD STERNBERG, AND B VAN STRAALLEN. Chombo Software Package for AMR Applications - Design Document. *Lawrence Berkeley National Laboratory Technical Report LBNL-6616E*, 2013.
- [28] KEITH COOPER, JASON ECKHARDT, AND KEN KENNEDY. Redundancy elimination revisited. In *Proceedings of the 17th International Conference on Parallel architectures and compilation techniques*, pages 12–21, 2008.
- [29] STEVEN J DEITZ, BRADFORD L CHAMBERLAIN, AND LAWRENCE SNYDER. Eliminating redundancies in sum-of-product array computations. In *Proceedings of the 15th International Conference on Supercomputing*, pages 65–77, 2001.
- [30] LUIZ DEROSE, BILL HOMER, DEAN JOHNSON, STEVE KAUFMANN, AND HEIDI POXON. Cray performance analysis tools. In *Tools for High Performance Computing*, pages 191–199. Springer Berlin Heidelberg, 2008.
- [31] JACK DONGARRA AND MICHAEL A HEROUX. Toward a new metric for ranking high performance computing systems. *Sandia Report, SAND2013-4744*, 312:150, 2013.
- [32] PAUL J. DRONGOWSKI. Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors. <https://pdfs.semanticscholar.org/5219/4b43b8385ce39b2b08ecd409c753e0efafe5.pdf>, November 2007.
- [33] RYUSUKE EGAWA, KAZUHIKO KOMATSU, SHINTARO MOMOSE, YOKO ISOBE, AKIHIRO MUSA, HIROYUKI TAKIZAWA, AND HIROAKI KOBAYASHI. Potential of a modern vector supercomputer for practical applications: performance evaluation of sx-ace. *The Journal of Supercomputing*, Mar 2017.
- [34] JOHN ERICKSON, MADANLAL MUSUVATHI, SEBASTIAN BURCKHARDT, AND KIRK OLYNYK. Effective Data-race Detection for the Kernel. In *Proceedings of the 9th*

- USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 151–162, Berkeley, CA, USA, 2010. USENIX Association.
- [35] PETER T. FELLER. *Value Profiling for Instructions and Memory Locations*. Master dissertation, 1998.
 - [36] MARY F. FERNÁNDEZ. Simple and Effective Link-time Optimization of Modula-3 Programs. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, pages 103–115, New York, NY, USA, 1995. ACM.
 - [37] FREE SOFTWARE FOUNDATION. objdump. <https://sourceware.org/binutils/docs/binutils/objdump.html>, 2017.
 - [38] GNU. GNU Binutils. <https://www.gnu.org/software/binutils/>, 2014. September 2014.
 - [39] SUSAN L. GRAHAM, PETER B. KESSLER, AND MARSHALL K. MCKUSICK. Gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, pages 120–126, New York, NY, USA, 1982. ACM Press.
 - [40] STEPHAN M. GÜNTHER AND JOSEF WEIDENDORFER. Assessing cache false sharing effects by dynamic binary instrumentation. In *WBIA '09: Proceedings of the Workshop on Binary Instrumentation and Applications*, pages 26–33, New York, NY, USA, 2009. ACM.
 - [41] ROBERT J. HALL. Call Path Profiling. In *Proceedings of the 14th International Conference on Software Engineering*, ICSE '92, pages 296–306, New York, NY, USA, 1992. ACM.
 - [42] SYLVAIN HENRY, HUGO BOLLORÉ, AND EMMANUEL OSERET. Towards the Generalization of Value Profiling for High-Performance Application Optimization. http://sylvain-henry.info/home/files/papers/shenry_2015_vprof.pdf.
 - [43] M. HIRZEL AND T. CHILIMBI. Bursty tracing: A framework for low overhead temporal profiling. In *ACM Workshop on Feedback-Directed and Dynamic Optimization*, pages 117–226. ACM, 2001.
 - [44] JUSTIN HOLEWINSKI, LOUIS-NOËL POUCHET, AND P. SADAYAPPAN. High-performance Code Generation for Stencil Computations on GPU Architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing*, pages 311–320, 2012.
 - [45] ROBERT HUNDT, EASWARAN RAMAN, MARTIN THURESSON, AND NEIL VACHHARAJANI. MAO—An extensible micro-architectural optimizer. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 1–10, 2011.
 - [46] DOE ACCELERATED STRATEGIC COMPUTING INITIATIVE. Sweep3D benchmark code. http://www.llnl.gov/asci_benchmarks/asci/limited/sweep3d/asci_sweep3d.html.

- [47] INTEL. Intel Microarchitecture Codename Nehalem Performance Monitoring Unit Programming Guide. <https://software.intel.com/sites/default/files/m/5/2/c/f/1/30320-Nehalem-PMU-Programming-Guide-Core.pdf>, 2010.
- [48] INTEL. Intel Math Kernel Library (MKL). <https://software.intel.com/en-us/intel-mkl>, 2015.
- [49] INTEL. Intel VTune. <https://software.intel.com/en-us/intel-vtune-amplifier-xe>, 2017.
- [50] INTEL CORP. Intel 64 and IA-32 Architectures Software Developers Manual. <http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf>.
- [51] INTEL CORP. Intel X86 Encoder Decoder Software Library. <https://software.intel.com/en-us/articles/xed-x86-encoder-decoder-software-library>.
- [52] YANGQING JIA, EVAN SHELHAMER, JEFF DONAHUE, SERGEY KARAYEV, JONATHAN LONG, ROSS GIRSHICK, SERGIO GUADARRAMA, AND TREVOR DARRELL. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.
- [53] YUNYUN JIANG, YI YANG, TIAN XIAO, TIANWEI SHENG, AND WENGUANG CHEN. DRDDR: A Lightweight Method to Detect Data Races in Linux Kernel. *The Journal of Supercomputing*, 72(4):1645–1659, April 2016.
- [54] GUOLIANG JIN, LINHAI SONG, XIAOMING SHI, JOEL SCHERPELZ, AND SHAN LU. Understanding and Detecting Real-world Performance Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 77–88, New York, NY, USA, 2012. ACM.
- [55] MARK SCOTT JOHNSON. Some Requirements for Architectural Support of Software Debugging. In *Proceedings of the First International Symposium on Architectural Support for Programming Languages and Operating Systems*, ASPLOS I, pages 140–148, New York, NY, USA, 1982. ACM.
- [56] TERESA. JOHNSON, MEHDI. AMINI, AND XINLIANG DAVID LI. ThinLTO: Scalable and Incremental LTO. In *Proceedings of International Symposium on Code Generation and Optimization*, Austin, Texas, USA, 2017.
- [57] MILAN JOVIC, ANDREA ADAMOLI, AND MATTHIAS HAUSWIRTH. Catch me if you can: Performance bug detection in the wild. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 155–170, New York, NY, USA, 2011. ACM.
- [58] TAKAHIRO KAMIO AND HIDEHIKO MASAHURA. A Value Profiler for Assisting Object-Oriented Program Specialization. In *Proceedings of Workshop on New Approaches to Software Construction*, 2004.

- [59] IAN KARLIN, ABHINAV BHATELE, JEFF KEASLER, BRADFORD L. CHAMBERLAIN, JONATHAN COHEN, ZACHARY DEVITO, RIYAZ HAQUE, DAN LANEY, EDWARD LUKE, FELIX WANG, DAVID RICHARDS, MARTIN SCHULZ, AND CHARLES STILL. Exploring traditional and emerging parallel programming models using a proxy application. In *27th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2013)*, Boston, USA, May 2013.
- [60] BARIS KASIKCI, THOMAS BALL, GEORGE CANDEA, JOHN ERICKSON, AND MADANLAL MUSUVATHI. Efficient tracing of cold code via bias-free sampling. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference, USENIX ATC'14*, pages 243–254, Berkeley, CA, USA, 2014. USENIX Association.
- [61] CHARLES KILLIAN, KARTHIK NAGARAJ, SALMAN PERVEZ, RYAN BRAUD, JAMES W. ANDERSON, AND RANJIT JHALA. Finding Latent Performance Bugs in Systems Implementations. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE '10*, pages 17–26, New York, NY, USA, 2010. ACM.
- [62] K. M. LEPAK AND M. H. LIPASTI. On the Value Locality of Store Instructions. In *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)*, pages 182–191, Jun 2000.
- [63] KEVIN M. LEPAK AND MIKKO H. LIPASTI. Silent Stores for Free. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture, MICRO 33*, pages 22–31, New York, NY, USA, 2000. ACM.
- [64] LEVINTHAL, DAVID. Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors, Version 1.0. https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf, 2009.
- [65] LINUX. perf_event_open - Linux man page. https://linux.die.net/man/2/perf_event_open, 2012.
- [66] LINUX. Linux perf tool. https://perf.wiki.kernel.org/index.php/Main_Page, 2015.
- [67] LINUX. SIGALTSTACK. <http://man7.org/linux/man-pages/man2/sigaltstack.2.html>, 2017.
- [68] MIKKO H. LIPASTI AND JOHN PAUL SHEN. Exceeding the Dataflow Limit via Value Prediction. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture, MICRO 29*, pages 226–237, Washington, DC, USA, 1996. IEEE Computer Society.
- [69] MIKKO H. LIPASTI, CHRISTOPHER B. WILKERSON, AND JOHN PAUL SHEN. Value Locality and Load Value Prediction. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS VII*, pages 138–147, New York, NY, USA, 1996. ACM.

- [70] CHIEN-LUNG LIU. False Sharing Analysis for Multithreaded Programs. Master’s thesis, National Chung Cheng University, July 2009.
- [71] TONGPING LIU, CHARLIE CURTSINGER, AND EMERY D. BERGER. DoubleTake: Fast and Precise Error Detection via Evidence-based Dynamic Analysis. In *Proceedings of the 38th International Conference on Software Engineering, ICSE ’16*, pages 911–922, New York, NY, USA, 2016. ACM.
- [72] TONGPING LIU AND XU LIU. Cheetah: Detecting False Sharing Efficiently and Effectively. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization, CGO ’16*, pages 1–11, New York, NY, USA, 2016. ACM.
- [73] TONGPING LIU, CHEN TIAN, HU ZIANG, AND EMERY D. BERGER. Predator: Predictive False Sharing Detection. In *Proceedings of 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP’14*, New York, NY, USA, 2014. ACM.
- [74] X. LIU AND J. MELLOR-CRUMMEY. Pinpointing data locality bottlenecks with low overhead. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 183–193, April 2013.
- [75] CHI-KEUNG LUK, ROBERT COHN, ROBERT MUTH, HARISH PATIL, ARTUR KLAUSER, GEOFF LONEY, STEVEN WALLACE, VIJAY JANAPA REDDI, AND KIM HAZELWOOD. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’05*, pages 190–200, New York, NY, USA, 2005. ACM.
- [76] YU-LONG LUO AND GUANGMING TAN. Optimizing stencil code via locality of computation. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, pages 477–478, 2014.
- [77] GABRIEL MARIN AND JOHN MELLOR-CRUMMEY. Pinpointing and Exploiting Opportunities for Enhancing Data Reuse. In *IEEE Intl. Symposium on Performance Analysis of Systems and Software, ISPASS ’08*, pages 115–126, Washington, DC, USA, 2008. IEEE Computer Society.
- [78] R. E. MCLEAR, D. M. SCHEIBELHUT, AND E. TAMMARU. Guidelines for Creating a Debuggable Processor. In *Proceedings of the First International Symposium on Architectural Support for Programming Languages and Operating Systems, ASPLOS I*, pages 100–106, New York, NY, USA, 1982. ACM.
- [79] PALL MELSTED, HAROLD PIMENTEL, AND LIOR PACTHER. Near-optimal RNA-Seq quantification. <https://github.com/makaho/kallisto>, 2014.
- [80] Message Passing Interface Forum. *MPI-2: Extensions to the Message Passing Interface Standard*, 1997.

- [81] JOSHUA SAN MIGUEL, JORGE ALBERICIO, ANDREAS MOSHOVOS, AND NATALIE ENRIGHT JERGER. Doppelganger: A Cache for Approximate Computing. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 50–61, New York, NY, USA, 2015. ACM.
- [82] JOSHUA SAN MIGUEL, MARIO BADR, AND NATALIE ENRIGHT JERGER. Load Value Approximation. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 127–139, Washington, DC, USA, 2014. IEEE Computer Society.
- [83] JACK MOSTOW AND DONALD COHEN. Automating Program Speedup by Deciding What to Cache. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence - Volume 1, IJCAI'85*, pages 165–172, San Francisco, CA, USA, 1985. Morgan Kaufmann Publishers Inc.
- [84] ROBERT MUTH, SCOTT A. WATTERSON, AND SAUMYA K. DEBRAY. Code Specialization Based on Value Profiles. In *Proceedings of the 7th International Symposium on Static Analysis, SAS '00*, pages 340–359, London, UK, UK, 2000. Springer-Verlag.
- [85] TAKUYA NAKAIKE, REI ODAIRA, MATTHEW GAUDET, MAGED M. MICHAEL, AND HISANOBU TOMARI. Quantitative Comparison of Hardware Transactional Memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture, ISCA '15*, pages 144–157, New York, NY, USA, 2015. ACM.
- [86] NERSC. NERSC-8 / Trinity Benchmarks. <http://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks/>, 2016.
- [87] KHANH NGUYEN AND GUOQING XU. Cachetor: Detecting Cacheable Data to Remove Bloat. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013*, pages 268–278, New York, NY, USA, 2013. ACM.
- [88] A. NISTOR, L. SONG, D. MARINOV, AND S. LU. Toddler: Detecting performance problems via similar memory-access patterns. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 562–571, May 2013.
- [89] ADRIAN NISTOR, PO-CHUN CHANG, COSMIN RADOI, AND SHAN LU. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1, ICSE '15*, pages 902–912, Piscataway, NJ, USA, 2015. IEEE Press.
- [90] ANDRZEJ NOWAK, AHMAD YASIN, AVI MENDELSON, AND WILLY ZWAENEPOEL. Establishing a Base of Trust with Performance Counters for Enterprise Workloads. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference, USENIX ATC '15*, pages 541–548, Berkeley, CA, USA, 2015. USENIX Association.
- [91] UNIVERSITY OF VIRGINIA. Rodinia benchmark suite. http://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php/Rodinia:Accelerating_Compute-Intensive_Applications_with_Accelerators, 2015.

- [92] TAEWOOK OH, HANJUN KIM, NICK P. JOHNSON, JAE W. LEE, AND DAVID I. AUGUST. Practical Automatic Loop Specialization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 419–430, New York, NY, USA, 2013. ACM.
- [93] LEONID OLIKER, ANDREW CANNING, JONATHAN CARTER, JOHN SHALF, AND STPHANE ETHIER. Scientific Application Performance on Leading Scalar and Vector Supercomputing Platforms. *International Journal of High Performance Computing Applications*, 2006.
- [94] OSWALDO OLIVO, ISIL DILLIG, AND CALVIN LIN. Static detection of asymptotic performance bugs in collection traversals. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 369–378, New York, NY, USA, 2015. ACM.
- [95] OPROFILE DEVELOPMENT TEAM. OProfile. <http://oprofile.sourceforge.net>, 2008.
- [96] ORACLE. Oracle Solaris Studio. <http://www.oracle.com/technetwork/server-storage/solarisstudio/overview/index.html>, 2017.
- [97] ALEKSEY PESTEREV, NICKOLAI ZELDOVICH, AND ROBERT T. MORRIS. Locating Cache Performance Bottlenecks Using Data Profiling. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 335–348, New York, NY, USA, 2010. ACM.
- [98] B. K. ROSEN, M. N. WEGMAN, AND F. K. ZADECK. Global Value Numbers and Redundant Computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 12–27, 1988.
- [99] MEHRZAD SAMADI, DAVOUD ANOUSHE JAMSHIDI, JANGHAENG LEE, AND SCOTT MAHLKE. Paraprox: Pattern-based Approximation for Data Parallel Applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 35–50, New York, NY, USA, 2014. ACM.
- [100] JOHN S. SENG AND DEAN M. TULLSEN. Architecture-level power optimization—what are the limits? *J. Instruction-Level Parallelism*, 7, 2005.
- [101] STELIOS SIDIROGLOU-DOUSKOS, SASA MISAILOVIC, HENRY HOFFMANN, AND MARTIN RINARD. Managing Performance vs. Accuracy Trade-offs with Loop Perforation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 124–134, New York, NY, USA, 2011. ACM.
- [102] CONNIE U. SMITH AND LLOYD G. WILLIAMS. Software Performance Antipatterns. In *Proceedings of the 2Nd International Workshop on Software and Performance*, WOSP '00, pages 127–136, New York, NY, USA, 2000. ACM.

- [103] L. SONG AND S. LU. Performance Diagnosis for Inefficient Loops. In *2017 39th International Conference on Software Engineering (ICSE)*, May 2017.
- [104] SPEC CORPORATION. SPEC CPU2006 benchmark suite. <http://www.spec.org/cpu2006>, 2007. 3 November 2007.
- [105] SPEC CORPORATION. SPEC OMP2012 benchmark suite. <https://www.spec.org/omp2012/>, 2015. May 2015.
- [106] M. SRINIVAS, B. SINHARROY, R. J. EICKEMEYER, R. RAGHAVAN, S. KUNKEL, T. CHEN, W. MARON, D. FLEMMING, A. BLANCHARD, P. SESHADRI, J. W. KELLINGTON, A. MERICAS, A. E. PETRUSKI, V. R. INDIKURU, AND S. REYES. IBM POWER7 performance modeling, verification, and evaluation. *IBM JRD*, 55(3):4:1–4:19, May-June 2011.
- [107] STANDARD PERFORMANCE EVALUATION CORPORATION. SPEC CPU2000 benchmark suite. <http://www.spec.org/cpu2000/>. 29 April 2005.
- [108] C. STEWART, K. SHEN, A. IYENGAR, AND J. YIN. EntomoModel: Understanding and Avoiding Performance Anomaly Manifestations. In *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 3–13, Aug 2010.
- [109] NATHAN R. TALLENT, JOHN MELLOR-CRUMMEY, AND MICHAEL W. FAGAN. Binary analysis for measurement and attribution of program performance. In *Proceedings of the 2009 ACM PLDI*, pages 441–452, NY, NY, USA, 2009. ACM.
- [110] NATHAN R. TALLENT, JOHN M. MELLOR-CRUMMEY, AND ALLAN PORTERFIELD. Analyzing Lock Contention in Multithreaded Applications. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '10*, pages 269–280, New York, NY, USA, 2010. ACM.
- [111] M. VALIEV, E.J. BYLASKA, N. GOVIND, K. KOWALSKI, T.P. STRAATSMA, H.J.J. VAN DAM, D. WANG, J. NIEPLOCHA, E. APRA, T.L. WINDUS, AND W.A. DE JONG. NWChem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications*, 181(9):1477 – 1489, 2010.
- [112] JEFFREY S. VITTER. Random Sampling with a Reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, March 1985.
- [113] SCOTT A. WATTERSON AND SAUMYA K. DEBRAY. Goal-Directed Value Profiling. In *Proceedings of the 10th International Conference on Compiler Construction, CC '01*, pages 319–333, London, UK, UK, 2001. Springer-Verlag.
- [114] VINCE WEAVER. Reading RAPL energy measurements from linux. <http://web.eece.maine.edu/~vweaver/projects/rapl/>.
- [115] MARK N. WEGMAN AND F. KENNETH ZADECK. Constant Propagation with Conditional Branches. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(2):181–210, April 1991.

- [116] GUOQING XU, NICK MITCHELL, MATTHEW ARNOLD, ATANAS ROUNTEV, EDITH SCHONBERG, AND GARY SEVITSKY. Finding Low-utility Data Structures. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 174–186, New York, NY, USA, 2010. ACM.
- [117] CONG YAN, ALVIN CHEUNG, JUNWEN YANG, AND SHAN LU. Understanding Database Performance Inefficiencies in Real-world Web Applications. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, CIKM '17, pages 1299–1308, New York, NY, USA, 2017. ACM.
- [118] AMIR YAZDANBAKHSI, GENNADY PEKHIMENKO, BRADLEY THWAITES, HADI ESMAELZADEH, ONUR MUTLU, AND TODD C MOWRY. RFVP: Rollback-free Value Prediction with Safe-to-approximate Loads. *ACM Transactions on Architecture and Code Optimization (TACO)*, 12(4):62, 2016.
- [119] YUTAO ZHONG AND WENYAO CHANG. Sampling-based program locality approximation. In *Proceedings of the 7th International Symposium on Memory Management*, pages 91–100, 2008.