REDSPY: Exploring Value Locality in Software

Shasha Wen
College of William & Mary
swen@cs.wm.edu

Milind Chabbi
Hewlett Packard Labs
milind.chabbi@hpe.com

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

Abstract

Complex code bases with several layers of abstractions have abundant inefficiencies that affect the execution time. *Value redundancy* is a kind of inefficiency where the same values are repeatedly computed, stored, or retrieved over the course of execution. Not all redundancies can be easily detected or eliminated with compiler optimization passes due to the inherent limitations of the static analysis.

Microscopic observation of whole executions at instruction- and operand-level granularity breaks down abstractions and helps recognize redundancies that masquerade in complex programs. We have developed REDSPY—a fine-grained profiler to pinpoint and quantify redundant operations 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. REDSPY provides intuitive optimization guidance by apportioning redundancies to their provenance—source lines and execution calling contexts. REDSPY pinpointed dramatically high volume of redundancies in programs that were optimization targets for decades, such as SPEC CPU2006 suite, Rodinia benchmark, and NWChem—a production computational chemistry code. Guided by REDSPY, we were able to eliminate redundancies that resulted in significant speedups.

Categories and Subject Descriptors C.4 [*Performance of systems*]: Measurement techniques, Performance attributes

Keywords Value profiling, redundancy detection and elimination, approximate computing, performance tools, Pin, CCTLib.

ASPLOS '17, April 08 - 12, 2017, Xi'an, China

ISBN 978-1-4503-4465-4/17/04…$15.00

DOI: http://dx.doi.org/10.1145/3037697.3037729

1. Introduction

Production software systems, including HPC applications, have become increasingly complex since they employ sophisticated flow of control and a hierarchy of component libraries. This complexity often introduces inefficiencies in the software stacks, which prevent applications from achieving the "bare-metal" performance. The inefficiencies can manifest as expensive re-computation of already computed values, unnecessary computation, unnecessary data movement, and excessive synchronization, among others. Inefficiencies can arise from developers' inattention to performance, suboptimal implementation choice, suboptimal code generation, inappropriate choice of algorithms, among others. In this paper, we focus on wasteful computations and wasteful data movement, which we refer to as "redundancy". The term "redundancy" should not be misinterpreted as "resiliency" for fault-tolerance.

Optimizing compilers are adept at eliminating redundant operations by techniques such as common subexpression elimination [18], value numbering [41], constant propagation [47], among others. However, they have a myopic view of the program, which limits their analysis to a small scope—individual functions or files. Link-time optimization [21, 27] 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. Thus, compilers often fail to eliminate many kinds of inefficiencies.

Orthogonal to static analysis is the coarse-grained runtime profiling that identifies program hot spots. Performance analysis tools such as HPCToolkit [6], VTune [5], gprof [22], OProfile [31], and CrayPAT [19] 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

```
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 1: Redundancy not detected by classic value profiling [12, 13, 20, 45] , DeadSpy [14], and RVN [48].

of the floating-point unit. They may, in fact, mislead us by acclaiming such loop with a high IPC (instructions executed per cycle) metric.

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 [14, 48] and demonstrated in our case studies in §7, various forms of program inefficiencies—e.g., suboptimal 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 [12, 13, 20, 45] 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 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 [14, 48] do not detect this redundancy. DeadSpy [14], 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 [48], which assigns unique values to computations analogous to value numbering [9], 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 [38]—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 paper.

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. [42] pointed out that approximate results by reusing similar values can significantly save computation operations, yielding more than a 2.5× 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 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 paper, 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 [34]. 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×. 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 paper:

- 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].

We organize the rest of the paper as follows: §2 reviews existing work and distinguishes REDSPY's approaches. §3 overviews the methodology adopted by REDSPY. §4 elaborates on the detection mechanisms of temporal and spatial value locality. §5 details REDSPY's recording and reporting strategies for intuitive analysis. §6 empirically evaluates REDSPY for detected redundancies and its overhead. §7 explores several important code bases where we apply optimizations guided by the insights from REDSPY. Finally, §8 presents some conclusions and previews some future work.

2. Related Work

We review the related work from two aspects. §2.1 reviews existing value profilers. §2.2 shows other approaches on eliminating redundant operations.

2.1 Traditional Value Profiling

Lipasti et al. [32, 33] 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 [29] 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 [30], 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. [36, 37] proposed hardware extensions to identify approximate load values; Yazdanbakhsh et al. proposed RFVP [49], 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. [8] 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. [12, 13, 20] 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 [45]. 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. [39] proposed value profiling for code specialization. Their approach, however, identifies the redundant values in registers only. Oh et al. [40] 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. [16] developed a procedure-level value profiler, which identifies redundant values passed to the same function as parameters multiple times. Kamio and Masuhara [28] 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. [10] 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 [23], 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

```
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 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: Code example of spatial value locality.

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*.

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 paper. Beyond these classical compiler techniques, there exist many static analysis techniques [17, 18, 24, 35] 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 [14] 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. [11] 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.

Wen et al. [48] developed a runtime value numbering tool (RVN), which 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. 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 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 [34] to instrument binaries for runtime value tracking.

Listing 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. §4.1 details the techniques to address these challenges.

Listing 3 shows spatial value locality, where values computed for $a[i]$ and $a[i+1]$ $(i = 0, 2, 4, ...)$ are always the same. Thus, the computation on $a[i+1]$ at line 7 is always redundant. To identify *spatial value locality*, REDSPY investigates the values stored in a segment of memory, e.g., an array. After initialization or a series of intensive writes, REDSPY compares the values of adjacent memory elements. If most of these values are identical, then they exhibit the spatial locality. REDSPY also periodically checks register contents for the uniqueness of its values. Relaxing to value approximation is similar to that of temporal value locality, which checks whether the adjacent values are within a

threshold percentage difference. §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 §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.

4. Detection of Value Redundancies

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 [25] semantics, the effective address computed immediately after an instruction is not same as the one used by the instruction. To be precise, Pin's `IARG_MEMORYWRITE_EA` argument to an analysis function when used with `IPOINT_AFTER` location computes the effective address after the instruction, *not* the effective address used by the instruction itself. Thus, our analysis routine executed *after* an instruction would get an incorrect effective address.

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

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

fields is ∼4KB, which is much less compared to shadowing each byte.

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

`X86` architectures have aliased registers where different segments of a register can be accessed via different register names. For example, the register `AL` is the lower 8 bits of `AX`, `AH` is the higher 8 bits of `AX`, `AX` is the lower 16 bits of `EAX`, and `EAX` is the lower 32 bits of `RAX`. If an instruction updates `AL`, it affects the subsequent value read at `AX, EAX,` and `RAX`, but it does not affect the value read at `AH`. If an instruction updates `AH`, it affects the subsequent value read at `AX, EAX`, and `RAX`, but it does not affect the value read at `AL`. If an instruction updates `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 [26] 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 [25], 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:

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

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

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

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

$$R_{reg}^{precise} = \frac{\sum \text{non-FP bytes redundantly generated in registers}}{\sum \text{non-FP bytes generated in registers}}$$
$$R_{reg}^{appx} = \frac{\sum \text{FP bytes redundantly computed in registers}}{\sum \text{FP bytes computed in registers}}$$

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:

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

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.

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 [15] for efficiently collecting calling contexts on every instruction and associates them with source code using the DWARF [2] 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 [14] 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 [50]. 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.

6. Experiments

We evaluate REDSPY on four platforms: Intel SandyBridge, AMD Opteron, Intel Xeon Phi, and IBM POWER7. Table 1 shows the machine configurations and the compilers used. We evaluate REDSPY on the following programs and benchmarks: SPEC CPU2006 [43] integer and floating-point benchmarks, Rodinia [4] and MineBench [3] parallel benchmark codes, and NWChem-6.3 [44] 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,

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, Memory	32KB/256KB/20MB, 256GB DDR3	64K/512K/10MB, 128GB DDR3	32KB/512KB/NA, 6GB GDDR5	32K/256K/4MB, 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 1: Machine configurations.

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	$\hat{R}^{precise}_{reg}$	$\hat{R}^{precise}_{load}$	$\hat{R}^{precise}_{store}$	$\hat{R}^{precise}_{ins}$	%reg	%load	%store	$\hat{R}^{appx}_{reg}$	$\hat{R}^{appx}_{load}$	$\hat{R}^{appx}_{store}$	$\hat{R}^{appx}_{ins}$
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**	--	--	--	**21**	**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
gamess	48	27	7.1	19	19	33	11	11	5.2	2.2	3.8	1.8	30	1.3
milc	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
gemsFDTD	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
lbm	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 2: Breakdown of temporal redundant bytes and redundant instructions in different benchmark suites.

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 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% 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%.

Program	Overhead		Redundancy	
	w/o sampling	w/ sampling	(w/o sampling) R_{total}	(w/ sampling) $\hat{R}_{total}$
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: Comparing overhead and redundancy with sampling enabled and disabled. The sampling covers 1% instructions.

Table 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 $R^{precise}_{reg}$, $R^{precise}_{load}$, $R^{precise}_{store}$, and $R^{precise}_{ins}$ components. Column 12, 13, 14, and 15, respectively, decompose the observed approximate redundancies into R^{appx}_{reg}, R^{appx}_{load}, R^{appx}_{store}, and R^{appx}_{ins} components. Bold texts summarize the maximum redundancy observed in each benchmark suite.

There are several benchmarks with a high volume of redundancies (e.g., `gcc` and `h264ref`). For example, 83% $R^{precise}_{store}$ in `gcc` is because of a repeated zero initialization of a large array, which has been well studied elsewhere [14]. We observe that R_{store} is often higher than R_{reg} and R_{load}. R_{ins} is always lower than the rest since this metric is computed for all instructions without decomposing the contributions from different categories. $R^{precise}$ is always higher than R^{appx} 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 §7, we, however, show two case studies, `Hotspot` and `Particle_filter`, which have high spatial redundancy.

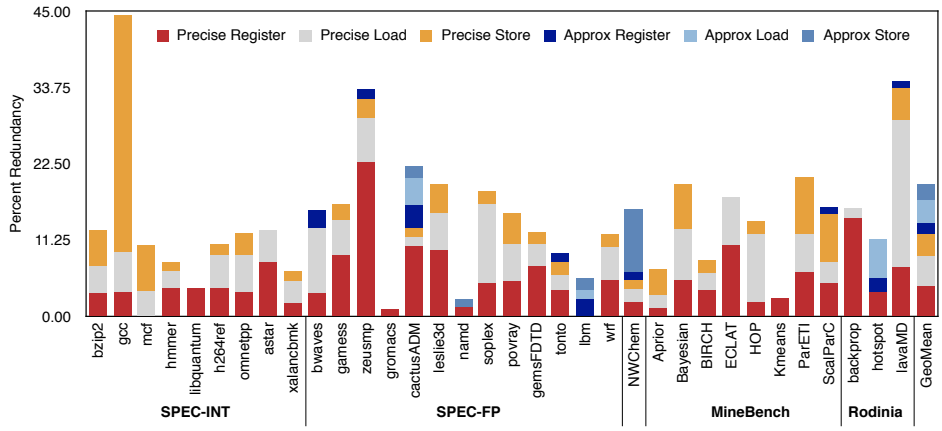

Figure 1: Breakdown of redundant bytes written in different benchmark suites.

Program	bzip2	gcc	mcf	hmmer	libquantum	h264ref	omnetpp	astar	xalancbmk	bwaves	gamess	milc	zeusmp	gromace	cactusADM	leslie3D	namd	soplex	povray	calculix	gemsFDTD	tonto	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 4: REDSPY's space and time overheads in the unit of times (×) on SPEC CPU2006 benchmarks.

Sampling Accuracy. To assess the accuracy of bursty sampling technique adopted by REDSPY, we selected eight representative benchmarks (`bzip2`, `bwaves`, `zeusmp`, `gromaces`, `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. 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 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 [15] for applications that have a deep and large calling context tree. For most benchmarks with moderate call chains, REDSPY incurs ∼ 5× 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 instruction mix has more SIMD or x87 instructions, which require heavyweight Pin APIs to pass runtime values to REDSPY's analysis routines.

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 5 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 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 5, it is evident that REDSPY can guide exploiting value locality in various programs yielding significant performance gains. In addition to time savings, Table 5 shows energy reduction (measured by RAPL [46] 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 §2, including (1) DeadSpy [14], which identifies dead stores, (2) RVN [48], which pinpoints redundant computation via symbolic execution, (3) ParaProx [42], a compiler technique to identify approximate computing opportunities in OpenCL codes, and (4) LLVM-ThinLTO [27], a link-time optimization technique across different compilation units.

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

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

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

Table 5: Overview of performance improvement guided by REDSPY on different platforms.

7.1 SPEC CPU2006 h264ref

`h264ref` is a reference implementation of the H.264 advanced video coding standard, a sequential C code. REDSPY reports ∼39% bytes are loaded from memory to registers, of which 13% are redundant. Figure 2 shows the top pairs with calling contexts involved in the load redundancy. Both contexts happen to be the same location. Listing 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× speedup for the whole program on Intel SandyBridge. The optimization saves 23% power. The improvements observed on other machines in shown in Table 5.

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 5 shows the high portion redundancy in routine `tce_mo2e_trans` with culprit calling context pair in Figure 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 5 shows the speedup of the execution is 1.53× after optimization with the `gcc` compiler on our AMD platform.

```
------------------------------------------------------------
 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 2: A redundancy pair reported in h264ref.

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

Listing 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: A redundancy pair reported in NWChem.

```
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 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 6: Temporal redundant function call in lavaMD.

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 6, which accounts for more than 60% value redundancy in registers. Moreover, this loop nest accounts for more than 90% of the total execution time. REDSPY pinpoints that the redundancy occurs at line 175, where frequently on consecutive invocations `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× speedup and 37% 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 7: Approximate spatial value locality in Hotspot.

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 7. We placed the instrumentation hook to inspect the `temp` array outside the nested loop. The spatial redundancy introduced ∼7× 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× speedup and 69% energy saving for the entire program.

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 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,

```
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 8: Temporal redundant array updating in backprop.

```
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 9: Spatial value locality in Particle_filter.

Program	RedSpy vs. other tools			
	DeadSpy [14]	RVN [48]	Paraprox [42]	ThinLTO [27]
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 6: RedSpy vs. other tools: whether value redundancies identified by RedSpy can be identified by other tools.

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.

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 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` 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.

7.7 Comparison with Other State-of-the-art Tools

Table 6 shows whether the redundancies detected by RedSpy could have been detected by other state-of-the-art tools. DeadSpy and RVN can find redundancy if there are dead writes or symbolic computational equivalence, respectively. DeadSpy can identify the NWChem redundancy since it is a dead write in addition to a redundant write. RVN can identify the redundancy in NWChem and lavaMD since they have a symbolic equivalence. ParaProx can identify and exploit the approximate computing opportunity available in the OpenCL version of `hotspot`, but not the OpenMP version used in this paper.

Finally, we used LLVM ThinLTO [27] to assess whether link-time optimization (LTO) could have eliminated any of the redundancies found by RedSpy. ThinLTO inlined the indirect function call in `464.h264ref` via PGO but introduced a condition check in the loop body resulting in only 8% speedup. RedSpy continued to find redundancies at the same place. Our hand-optimization cloned the loops and hoisted the condition check out of the loops resulting in 45% speedup. All other redundancies are algorithmic in nature and hence not exploitable by compilers even with a global view of the program. In fact, we noticed that sometimes the redundancy fraction increased with ThinLTO; this is because LTO can reduce the number of operations by eliminating some "language and abstraction overheads" but cannot reduce redundant operations arising from algorithmic inefficiencies. Hence, the number of redundant operations relative to the total number of operations increases [8].

8. Conclusions and Future Work

This paper presents RedSpy, a fine-grained profiler that explores value locality during program execution. RedSpy identifies value locality occurring over time in same storage locations (temporal) and in the neighborhood of a storage location (spatial). RedSpy tracks redundancies occurring in both memory and registers. RedSpy pinpoints the causes of redundancies by annotating the calling context pairs where redundancies frequently occur. Furthermore, RedSpy incorporates techniques to recognize when floating-point values are approximately the same, thus offering new venues to tune code for approximate computations.

RedSpy incurs a modest runtime and memory overhead making it practical to track execution-wide redundancies in complex and long-running parallel programs. Guided by RedSpy, with little effort, we were able to obtain significant speedup in many important, complex codes bases unfamiliar to us. The optimizations we developed to exploit value locality are architecture independent and demonstrate their benefits on several micro architectures.

Our future work involves (1) backward slicing from the provenance of redundancy to explore the entire sequence of wasteful operations ending in a redundancy, (2) exploring inter-thread value redundancy, and (3) designing a GUI for RedSpy to ease data interpretation.

Acknowledgements

The authors thank Haris Volos for his suggestions on the paper and the anonymous reviewers for their useful feedback.

References

[1] CCTLib. https://github.com/CCTLib/.

[2] The DWARF Debugging Standard. http://www.dwarfstd.org.

[3] NU-MineBench Suite. http://cucis.ece.northwestern.edu/projects/DMS/MineBench.html.

[4] Rodinia Benchmark Suite. http://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php/Rodinia:Accelerating_Compute-Intensive_Applications_with_Accelerators.

[5] Intel VTune. https://software.intel.com/en-us/intel-vtune-amplifier-xe.

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

[7] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous Profiling: Where Have All the Cycles Gone? *ACM Trans. Comput. Syst.*, 15(4):357–390, Nov 1997.

[8] 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.

[9] P. Briggs, K. D. Cooper, and L. T. Simpson. Value Numbering. *Software–Practice and Experience*, 27(6):701–724, Jun 1997.

[10] M. Burrows, U. 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.

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

[12] B. Calder, P. Feller, and A. 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.

[13] B. Calder, P. Feller, and A. Eustace. Value Profiling and Optimization. *Journal of Instruction Level Parallelism*, 1, 1999.

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

[15] M. Chabbi, X. Liu, and J. Mellor-Crummey. Call Paths for Pin Tools. In *Proc. of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 76:76–76:86, 2014.

[16] E.-Y. Chung, L. Benini, and G. D. Micheli. Energy Efficient Source Code Transformation based on Value Profiling. In *Proceedings of International Workshop on Compilers and Operating Systems for Low Power*, 2000.

[17] K. Cooper, J. Eckhardt, and K. Kennedy. Redundancy Elimination Revisited. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 12–21, New York, NY, USA, 2008. ACM.

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

[19] L. DeRose, B. Homer, D. Johnson, S. Kaufmann, and H. Poxon. Cray Performance Analysis Tools. In *Tools for High Performance Computing*, pages 191–199. Springer Berlin Heidelberg, 2008.

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

[21] M. 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.

[22] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A Call Graph Execution Profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, pages 120–126, New York, NY, USA, 1982. ACM.

[23] S. Henry, H. Bolloré, and E. Oseret. Towards the Generalization of Value Profiling for High-Performance Application Optimization. http://sylvain-henry.info/home/files/papers/shenry_2015_vprof.pdf.

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

[25] Intel Corp. Intel 64 and IA-32 Architectures Software Developer's Manual. https://software.intel.com/en-us/articles/intel-sdm, .

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

[27] T. Johnson, M. Amini, and X. D. Li. ThinLTO: Scalable and Incremental LTO. In *Proceedings of International Symposium on Code Generation and Optimization*, Austin, Texas, USA, 2017.

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

[29] 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.

[30] K. M. Lepak and M. 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.

[31] J. Levon *et al.* OProfile. `http://oprofile.sourceforge.net`.

[32] M. H. Lipasti and J. P. 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.

[33] M. H. Lipasti, C. B. Wilkerson, and J. P. 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.

[34] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. 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.

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

[36] J. S. Miguel, M. Badr, and N. E. 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.

[37] J. S. Miguel, J. Albericio, A. Moshovos, and N. E. Jerger. DoppelgÄNger: 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.

[38] J. Mostow and D. 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.

[39] R. Muth, S. A. Watterson, and S. 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, 2000. Springer-Verlag.

[40] T. Oh, H. Kim, N. P. Johnson, J. W. Lee, and D. 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.

[41] 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.

[42] M. Samadi, D. A. Jamshidi, J. Lee, and S. 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.

[43] SPEC Corporation. SPEC CPU2006 Benchmark Suite. `http://www.spec.org/cpu2006`. 3 November 2007.

[44] M. Valiev, E. Bylaska, N. Govind, K. Kowalski, T. Straatsma, H. V. Dam, D. Wang, J. Nieplocha, E. Apra, T. Windus, and W. de Jong. NWChem: A Comprehensive and Scalable Open-source Solution for Large Scale Molecular Simulations. *Computer Physics Communications*, 181(9):1477 – 1489, 2010.

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

[46] V. Weaver. Reading RAPL Energy Measurements from Linux. `http://web.eece.maine.edu/~vweaver/projects/rapl/`.

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

[48] S. Wen, X. Liu, and M. Chabbi. Runtime Value Numbering: A Profiling Technique to Pinpoint Redundant Computations. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 254–265, Oct 2015.

[49] A. Yazdanbakhsh, G. Pekhimenko, B. Thwaites, H. Esmaeilzadeh, O. Mutlu, and T. C. Mowry. RFVP: Rollback-free Value Prediction with Safe-to-approximate Loads. *ACM Transactions on Architecture and Code Optimization (TACO)*, 12(4):62, 2016.

[50] Y. Zhong and W. Chang. Sampling-based Program Locality Approximation. In *Proceedings of the 7th International Symposium on Memory Management*, pages 91–100, 2008.