ZeroSpy: Exploring Software Inefficiency with Redundant Zeros

Xin You*, Hailong Yang*†, Zhongzhi Luan*, Depei Qian* and Xu Liu‡
Beihang University, China*, North Carolina State University, USA‡
State Key Laboratory of Mathematical Engineering and Advanced Computing, Wuxi, China†
{youxin2015,hailong.yang,07680,depeiq}@buaa.edu.cn*, xliu88@ncsu.edu‡

***Abstract*—Redundant zeros cause inefficiencies in which the zero values are loaded and computed repeatedly, resulting in unnecessary memory traffic and identity computation that waste memory bandwidth and CPU resources. Optimizing compilers is difficult in eliminating these zero-related inefficiencies due to limitations in static analysis. Hardware approaches, in contrast, optimize inefficiencies without code modification, but are not widely adopted in commodity processors. In this paper, we propose ZeroSpy - a fine-grained profiler to identify redundant zeros caused by both inappropriate use of data structures and useless computation. ZeroSpy also provides intuitive optimization guidance by revealing the locations where the redundant zeros happen in source lines and calling contexts. The experimental results demonstrate ZeroSpy is capable of identifying redundant zeros in programs that have been highly optimized for years. Based on the optimization guidance revealed by ZeroSpy, we can achieve significant speedups after eliminating redundant zeros.**

***Index Terms*—Redundant Zero, Software Inefficiency, Performance Profiling and Optimization**

I. Introduction

Nowadays, production software in both industry and academia is becoming increasingly complex as they consist of a large number of library dependencies as well as sophisticated control and data flows. Such complexities can easily lead to unexpected inefficiency, which prevents the software from achieving optimal performance. Often, software inefficiency involves redundant operations, such as repeatedly loading the same values from memory [1], writing values never used [2], overwriting values in the same location with intermediate values never referenced [3], and repeatedly computing the same value [4], [5]. In addition, a large number of applications [6], [7] take sparse data as the inputs, which can result in a significant waste of resources if dense data structures are used. One of the fundamental reasons for the above inefficiencies is that there are instructions and data structures frequently dealing with zeros.

Listing 1 shows a simple example. The code first initializes two integer arrays of A and B (line 1-3) and then computes array C by subtracting array A and B (line 5-6). The redundant zeros appear in the computation where memory operations frequently read zeros from the array A. Such memory loads can be avoided by compressing A with a sparse data structure. Furthermore, values of array B range from 0 to 1000, so a few significant bytes of array B's values are always 0, which are redundant zeros as well, leading to a fraction of memory loads wasted. Instead of using a 32-bit integer, a 16-bit integer is enough to fulfill the computation, which yields better cache usage and vectorization potentials [1].

```
1 for(int i=0;i<1000;++i) {
2     A[i] = 0; B[i] = i;
3 }
4 ...
5 for(int i=0;i<1000;++i)
6 ►   C[i] = A[i]-B[i];
```

Listing 1. An example code working on redundant zeros, where all variables are 32-bit integers.

Actually, a larger number of real-world applications (e.g., deep neural network and high-efficiency video coding) have already been reported to contain a significant amount of redundant zeros and achieved significant speedup from the corresponding optimization. For instance, in the area of deep neural network, researchers have proposed both hardware [11] and software [7] approaches to automatically detect the sparsity of the network in order to apply specific optimizations for achieving better performance. For high-efficiency video coding applications, better detection of all-zero blocks can improve the performance by skipping computations on these blocks [6].

Modern optimizing compilers usually employ a set of redundancy elimination techniques, such as value numbering [12], common subexpression elimination [13], and constant propagation [14]. However, they suffer from the limitation of myopic optimization scopes and imprecise static analysis on pointers and aliases. Link-time optimization [15], [16] can enlarge the visibility of compiler optimization, but the performance improvement after optimization remains constrained. Combining these static optimizing techniques, profile-guided optimization (PGO) [17] was proposed to optimize the generated code with profiled performance data. However, these techniques do not identify redundant zeros involved in the memory and computation operations for further code optimization.

Performance analysis tools such as Perf [18], HPCToolkit [19], VTune [20], gprof [21], CrayPat [22] and OProfile [23] can monitor program execution to report performance metrics including CPU cycles, cache misses, and arithmetic intensity to guide optimization. Other fine-grained profiling tools such as RedSpy [4] and LoadSpy [1] can identify redundant memory stores and loads. Although these tools can report program hotspots and resource under-utilization, they do not identify inefficiencies involved with redundant zeros and provide corresponding optimization guidance.

[1]Unlike existing work [8]–[10] that reduces floating-point precisions, we do not suffer from any accuracy loss by only removing unnecessary zeros.

SC20, November 9-19, 2020, Is Everywhere We Are
978-1-7281-9998-6/20/$31.00 ©2020 IEEE

Hardware approaches have also been proposed to detect and optimize redundant zeros. Dusser et al. [24], [25] reported frequent zero values existing in cache and memory and proposed Zero-Content Augmented cache (ZCA cache) as well as Decoupled Zero-Compressed memory (DZC memory) for redundancy elimination. Redundant zeros are exploited in eDRAM [26] to save energy by avoiding refreshing zero chunks. A zero-aware cache algorithm (Zero-Chunk) [27] was proposed to deal with redundant zeros. However, the major drawback of these hardware approaches is that they require special hardware extensions and are not directly applicable to commodity servers running real-world applications.

Given the existing software and hardware approaches, it is still difficult to identify the inefficiencies caused by redundant zeros that are buried deep due to layers of software abstractions. Thus, we propose ZeroSpy, a practical tool to profile redundant zeros stored in memory and reused for computation. ZeroSpy works with fully optimized binary executables and provides useful metrics (e.g., *redmap*, *redundancy fraction*) to guide performance optimization. ZeroSpy reports redundant zeros with their calling contexts and associated source lines as well as the data access patterns. For optimization, ZeroSpy presents the top candidate instructions and data structures with redundant zeros, and derives a metric *redmap* to represent the data access pattern. With the detailed profiles provided by ZeroSpy, developers can obtain useful insights to guide the optimization of the code regions and data structures involving frequent redundant zeros for better performance.

In summary, this paper makes the following contributions:

- We observe the fact that operating redundant zeros pervasively exists in modern software packages, which usually indicates performance inefficiencies. Furthermore, we categorize the provenance of redundant zeros to motivate different code optimization guidances produced by ZeroSpy.
- We develop ZeroSpy, a practical tool with reasonable profiling overhead to identify arithmetic and memory operations dealing with redundant zeros. In addition, we provide useful metrics to quantify the effect of redundant zeros and pinpoint code regions and data structures with high redundant zeros, which provide intuitive guidance for code optimization.
- We evaluate ZeroSpy with well-known benchmarks and real applications. We demonstrate the effectiveness of ZeroSpy by identifying redundant zeros in these programs and perform optimization guided by ZeroSpy for significant speedups.

This paper is organized as follows. Section II reviews related work and distinguishes our approach. Section III defines redundant zeros and categorizes them according to their root causes. Section IV presents our implementation of ZeroSpy to detect redundant zeros in real-world programs. Section V shows the experimental results to evaluate ZeroSpy. Section VI shows some case studies to prove the usage of ZeroSpy. Section VII presents some discussions of ZeroSpy. Section VIII presents some conclusions of this paper.

II. Related Work

A. Hardware Solutions to Remove Redundancies

There are many research approaches to profile and optimize data sparsity from the hardware aspect. Dusser et al. [24], [25] reported the data sparsity in cache and memory that there are zero values loaded from and written to memory frequently and these zero values wasted memory bandwidth. Correspondingly, they proposed Zero-Content Augmented cache (ZCA cache) and Decoupled Zero-Compressed memory (DZC memory). In their approach, each level of cache consists of a conventional cache augmented with zero-content cache, which utilizes an address tag and a validate tag for null data. They also attached a zero detection unit both in the path of load and write to detect zero values as well as gave control signals to maintain the zero-content cache. A similar idea is applied to compress physical memory in their proposed DZC memory.

Manohar and Kapoor [26] developed a new refreshing strategy of eDRAM by detecting the zero chunks and skipping the corresponding refreshment, which results in notable energy saving. Gao et al. [27] proposed an efficient cache algorithm named Zero-Chunk, which detects fingerprints with all-zero remainders and aggregates them into corresponding containers to improve the cache hit ratio as well as save the memory space. Miguel et al. [28], [29] proposed a new cache design to store multiple similar blocks with a single data array entry in order to reduce the data redundancy. Rollback-Free Value Prediction (RFVP) [30] exploited approximate computation by predicting the requested values for certain safe-to-approximate load operations missed in the cache.

In some specific domains such as machine learning, Peng et al. [7] proposed to exploit tensor sparsity in the deep neural network at runtime. They profiled several mini-batches during training to identify the sparsity of certain neurons in the network and divided them into sparse neurons to perform sparse matrix multiplication and dense neurons for normal GEMM. Another approach proposed by Lascorz et al. [11] combined the zero-aware neural network accelerator (Bit-Tactical) and software scheduling method to address the data sparsity. Their hardware accelerator was designed to skip the zero activation by bit scale, whereas the proposed scheduler was designed to skip processing zero weights as much as possible. Research works in this category are specific to their application areas and cannot provide useful insights for other applications in general.

Unlike these approaches, ZeroSpy is a pure software profiler without any special hardware extension, which is ready to be deployed in commodity systems. Several approaches investigate values used for computation relying on hardware features available in commodity CPU processors. Burrows et al. [31] sampled values in Digital Continuous Profiling Infrastructure [32] with hardware performance counters. Wen et al. [33] identified redundant memory operations by combining the performance monitoring units and debug registers available on x86 processors. However, these approaches work on a small set of processors and do not pinpoint redundant zeros as ZeroSpy.

B. *Software Solutions to Remove Redundancies*

Value profilers are developed to pinpoint redundant computations, which are highly related to ZeroSpy. The first value profiler perhaps was proposed by Calder et al. [34]–[36] on DEC Alpha Accessors, which can instrument the program code and pinpoint invariant or semi-invariant variables stored in registers or memory locations. The following research [37] proposed a variant of this value profiler. Wen et al. [4] developed a fine-grained profiler (RedSpy) to identify silent writes and guide the optimization. RedSpy detects redundancies in computation results as well as data movements. It can also pinpoint the redundant code regions by reporting the redundancy with calling contexts as well as their locations in source code. Furthermore, Su et al. [1] developed another fine-grained profiler (LoadSpy) to identify redundant loads in software. They claimed that in software, there is inefficiency of loading the same values across instructions. LoadSpy tracked every load instruction to pinpoint these redundant loads and reported pairs of instructions involving redundant loads. In addition, Tan et al. [38] developed CIDetector and proved that dead and redundant operations still exist even using modern optimizing compilers. Unlike ZeroSpy, RedSpy, LoadSpy and CIDetector do not pinpoint zero-related redundant arithmetic and memory operations. ZeroSpy complements existing value profilers to increase the coverage of value-related inefficiencies.

In addition, there are approaches focusing on exploiting redundant zeros by reducing the length of data types. Stephenson et al. [39] proposed a bitwidth analysis methodology to statically identify the maximum required bitwidth of variables to reduce the length of data type as much as possible. They claimed that their approach could result in a positive impact in area, speed, and power consumption when evaluated on a targeted FPGA platform. On the other hand, Precimonious [8] was proposed to trade off the floating-point precision between accuracy and performance. Unlike these previous works, which can only deal with redundancies existing in data types, ZeroSpy can also detect redundancies caused by inappropriate use of data structures as well as zero-agonistic instructions. Moreover, our approach does not need any source code for zero redundancy detection.

Another research area targets optimizing redundant zeros in high-efficiency video coding (HEVC) programs, where the coefficient used in the techniques often consists of a large chunk of zeros. Many research works [6], [40], [41] exploited all-zero block detection to skip the redundant computation with zeros for better performance. Meanwhile, Yang et al. [42] observed that costly zero initialization was applied to maintain memory safety by modern programming languages such as Java, and they proposed several optimization strategies to reduce such inefficiency of redundant zeros in Java. Instead, ZeroSpy is a general tool that works on any binary executables, not limited to specific domains.

III. Deep Analysis of Redundant Zeros

We use the following three terms to define and quantify redundant zeros.

Definition 1 (Redmap)**.** Let $B_1B_2...B_n$ be the byte representation of a value V, where B_n is the most significant byte of V. The *redmap* of V is defined as a ***bit vector*** $b_1b_2...b_n$, where b_i $(i = 1, 2, ..., n)$ is bit 0 if $B_i = 0$, otherwise b_i is bit 1.

Definition 2 (Redundant Zero)**.** A value V contains redundant zero when a sub-sequence $b_k...b_n$ exists in the *redmap* $b_1b_2...b_n$ of V, where $b_i = 0$ $(1 \leq k \leq i \leq n)$. The number of redundant zeros in V is defined as the length of the longest $b_k...b_n$. If the value $V = 0$, we call it *fully redundant zeros*.

Definition 3 (Redundancy Fraction)**.** The redundancy fraction R is defined as the ratio of the number of redundant zeros to the total bytes loaded from memory during the entire program execution.

Since the B_n/b_n in Definition 1 and 2 represents the most significant byte, Definition 1 and 2 identify a value as redundant zero if and only if the most significant $(n-k)$ bytes of the value are zeros, other than any zero bytes in the value. Moreover, the redundant zeros identified by Definition 2 are zeros that can be eliminated without affecting the correctness of the program (e.g., Fig. 1). Similarly, a load has redundant zeros if its loaded value has redundant zeros.

A. *Pervasive Existence of Redundant Zeros*

Redundant Zeros frequently occur in real-world software packages. We measure redundant zeros with the tool to be described in Section IV in popular benchmark suites such as SPEC CPU2017 [43], NPB [44], Rodinia [45], and CORAL-2 [46]. We compile all these benchmarks with gcc 9.2.0 -O3. Benchmarks in SPEC CPU2017 have 37.1% redundant zeros for integer operations and 0.83% for floating-point operations on average. The maximum redundancy fractions of SPEC CPU2017 benchmarks are 63.0% and 10.3% for integer and floating-point operations, respectively. We observe similar results in NPB, Rodinia, and CORAL-2, whose redundancy fractions averaged across all benchmarks are respectively 6.6%, 20.2%, and 24.2% for integer operations, and 0.37%, 0.63% and 3.2% for floating-point operations.

Such large amounts of redundant zeros open an opportunity for code optimization to avoid most redundant computations based on zeros. In the next section, we analyze the root causes of redundant zeros due to various impropriate programming practices and provide corresponding solutions to improve the performance.

B. *Root Causes of Redundant Zeros*

Many redundant zeros are caused by poor programming practices. We categorize the existence of redundant zeros in software due to their root causes.

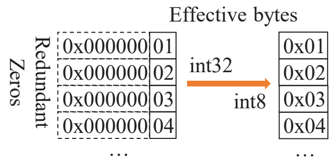
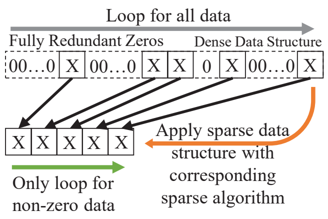


Fig. 1. An example of redundant zeros from data with more than enough storage. The values on left are stored in 32-bit integer with the most significant three bytes always being zero. It can be optimized by downgrading the 32-bit integer to 8-bit integer (one byte).

a) Data with More than Enough Storage: Fig. 1 shows how data with more than enough storage contributes to the redundant zeros during program execution. The values (on the left) stored in 32-bit integers always contain zeros on their most significant three bytes. When instructions load these values from memory, they have to load three useless bytes of redundant zeros in order to use only one effective byte for each value. In addition, from the aspect of data locality, the redundant zeros waste the limited resources along the cache hierarchy and thus may lead to a higher cache miss rate or even frequent page fault. All these potential inefficiencies can result in the performance penalties of the program. In practice, we observe this kind of redundant zeros in *648.exchange2* benchmark of SPEC CPU2017, which is described in detail in Section VI.

b) Inappropriate Use of Data Structures: Another cause of software inefficiency with redundant zeros is the inappropriate use of data structure. Some software chooses the dense data structure for its computation even when the data is sparse with few effective values, ignoring the property of the data it is handling. In such a case, a large fraction of fully redundant zeros with the data structures can be detected during the execution. As shown in the upper part of Fig. 2, if sparse data is stored in a dense data structure, the loop for loading and computing each data results in a large number of fully redundant zeros, which turns to useless loads and computation instructions. However, if we can store the data in a sparse data structure and apply the corresponding sparse algorithm, then the computation only needs to iterate through non-zero data and thus eliminates the inefficiencies with the dense data structure. We observe this kind of redundant zeros in *649.fotnonik* benchmark of SPEC CPU2017, which is described in detail in Section VI.

c) Zero-agonistic Computation: Redundant zero in this category is the inefficiency at the instruction level, which means instructions load redundant zeros from memory and compute with these redundant zeros. Such computations are useless and thus waste the computation resources. Listing 2 is a code region with high fraction of redundant zeros in *603.bwaves* benchmark of SPEC CPU2017. The most frequent redundant zeros are detected in line 51 of Listing 2, which involves expensive computation instructions such as *power*. After a comprehensive analysis, the redundant zeros come from three variables: u, v and w. These three values are often loaded as zeros, and the results of the corresponding computations can be certain as zeros. The useless computation with



Loop for all data
Fully Redundant Zeros
Dense Data Structure
00…0
X
00…0
X
X
0
X
00…0
X
X X X X X
Apply sparse data structure with corresponding sparse algorithm
Only loop for non-zero data

Fig. 2. An example of redundant zeros with inappropriate use of data structure. The original dense data structure contains many zeros and thus leads to high redundancy fraction of fully redundant zeros. Instead, using sparse data structure and the corresponding sparse algorithm can significantly reduce the amount of fully redundant zeros as well as mitigate such kind of software inefficiency.

```
34  do k=1,nzj
37    do j=1,ny
39      do i=1,nx
44        u=q(2,i,j,kq)/ro
45        v=q(3,i,j,kq)/ro
46        w=q(4,i,j,kq)/ro
...       ...
51        mu=((gm-1.)*(q(5,i,j,kq)/ro-0.5*(u*u+v*v+w*w)))**0.75
...       ...
134     enddo
135   enddo
136 enddo
```

Listing 2. Code region with high fraction of redundant zeros in *603.bwaves* benchmark of SPEC CPU2017.

redundant zeros worths performance optimization, especially when the code region involves heavy computation. We observe this kind of redundant zeros in *644.nab* benchmark of SPEC CPU2017, heartwall benchmark of Rodinia, and ShengBTE, which are described in detail in Section VI.

IV. Detection of Redundant Zeros

Manually identifying redundant zeros used in programs is tedious, which requires strong domain knowledges. Instead, we develop a tool — ZeroSpy to automate the analysis for fully optimized binary code. ZeroSpy provides *code-centric* and *data-centric* analyses to pinpoint both instructions and data objects that are involved in redundant zero computation. The above two analysis provide different angles to understand the redundant zeros and cannot be substituted by each other. Note that we do not handle registers in Zerospy, because the memory behaviors caused by software inefficiency impact the program performance more significantly. Thus, Zerospy mainly focuses on the detection and optimization of redundant zeros in memory. We first describe the implementation of the two analyses and then discuss how ZeroSpy handles parallelism and provides optimization guidance.

A. Code-centric Analysis

Fig. 3 shows the detection of instructions dealing with redundant zeros, as known as the **code-centric** mode of ZeroSpy. We use *INS_InsertPredicatedCall* with *IPOINT_BEFORE* from PIN tool [47] to instrument the analysis routine before each memory load instruction to extract its values. We handle integer and floating-point values differently.

a) Integer values: If the memory load instruction loads an integer value, we insert an *INT* routine to analyze the loaded value, where we directly convert it to redmap and count byte by byte to obtain the number of redundant zeros defined in Definition 1 and Definition 2. To accelerate the analysis, we

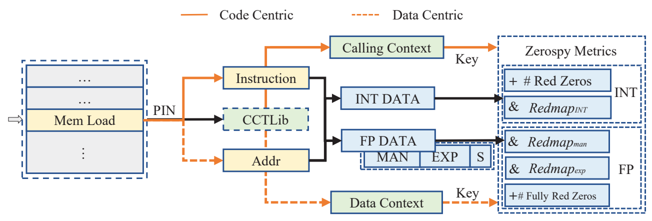


Fig. 3. The detection workflow for instructions (code-centric) and data (data-centric) dealing with redundant zeros.

apply reduction on the bits of each loaded bytes with 64-bit bitwise operations if available. In addition, we enumerate all possible *redmaps* (2^n, where $n = 1, 2, 4, 8$) for representing data size less than 8 bytes, and calculate the number of redundant zeros of each *redmap*. The above information is stored in separate cache tables to avoid counting the redundant zeros when the *redmap* is referenced during profiling. We can obtain the redmap R_{INT} of an instruction ins in the same calling context C (obtained using CCTLib [48]) by accumulating each *redmap* as shown in Equation 1.

$$R_{INT}(ins, C) = R_{INT}(ins, C) \& (b_1 b_2 .. b_n) \tag{1}$$

The accumulated redmap of an instruction indicates the memory access pattern of the instruction in its calling context. Besides, the number of redundant zeros of the instruction is aggregated to the total number of redundant zeros in each statements, loops, and functions, according to their calling contexts. Note that Zerospy will not identify redundant zeros if the sign bit of an integer is set. This is because, from the aspect of binary instrumentation, Zerspy cannot determine if the most significant bit is a valid data bit or a sign bit, thus it is reasonable to make conservative action to skip redundant zero detection if the sign bit is set.

b) Floating-point values: For floating-point instructions, the redmap of a floating-point value is meaningless if directly converted regardless of the IEEE floating-point format [49]. As shown in Fig. 3, a floating-point value is composed of three parts: mantissa, exponent, and sign. Only mantissa and exponent are the potential targets for software inefficiency when they contain redundant zeros. Therefore, we only convert mantissa and exponent to redmap. The redmap R_{FP} of an instruction ins in the same calling context C is obtained by accumulating each part of redmap R_{man} and R_{exp} with bit representations of the floating-point value ($k = 23, n = 31$ for *single* and $k = 52, n = 63$ for *double*) as shown in Equation 2. For fully redundant zeros of a floating point value, we treat the value V as 0 if and only if all bits are zero regardless of the sign bit.

$$R_{FP}(ins, C) = \begin{cases} R_{man}(ins, C) = R_{man}(ins, C) \& (b_1 b_2 ... b_k) \\ R_{exp}(ins, C) = R_{exp}(ins, C) \& (b_{k+1} ... b_n) \end{cases} \tag{2}$$

Note that, the number of redundant zeros of a single floating-point value has little interest for performance optimization because we cannot downgrade its representation (using fewer bits) without loss of accuracy. Instead, the number of fully redundant zeros are more interesting with floating-point values, because it can indicate the sparsity of a data structure or data loaded by the instruction. Therefore, we only accumulate the total number of fully redundant zeros of a floating-point instruction in the same calling context. Note that for double values, in IEEE 754 definition of double and single floating point values, the exponent of the value is actually 2^{e-1023} for double and 2^{e-127} for single, where e is the value of exponent bits for double/single value. Therefore, if the most significant bits in exponent for a double are all zeros, it indicates higher precision instead, and we cannot safely convert double into single without precision loss. Thus, redundant zeros cannot be used to determine whether a double can be reduced into a single.

B. Data-centric Analysis

ZeroSpy identifies redundant zeros residing in a block of memory, such as a data object, a memory page, or a cache line. If the redundancy faction is high, such memory blocks have high sparsity.

1) Data Object Sparsity: The detection for redundant zeros in Fig. 3 identifies the data structure with spatial sparsity, which is the **data-centric** mode of ZeroSpy. The difference of data-centric detection from code-centric detection is that the metrics (redmap and the number of redundant zeros) are accumulated if and only if they are from the same data object. The data object is obtained by tracking the data of *DYNAMIC_OBJECT* and *STATIC_OBJECT* with CCTLib [48], which represents variables both allocated dynamically (e.g. from the *malloc() family*) and declared statically.

For each memory load instruction, a redmap R_{INT} (R_{FP} for floating point) of the loaded integer value (and $b_1 b_2 ... b_n$ for the bit representation of redmap) is generated, and then accumulated by and operation when it is from the same data context D as well as the same memory address M as shown in Equation 3 and 4.

$$R_{INT}(M, D) = R_{INT}(M, D) \& (b_1 b_2 .. b_n) \tag{3}$$

$$R_{FP}(M, D) = R_{FP}(M, D) \& (V \equiv 0) \tag{4}$$

2) Page Level Sparsity: From the aspect of operating system and architecture, the redundant zeros residing in pages also worth investigation for mitigating the inefficiency in operating systems and architecture. For page-level sparsity analysis, we instrument every load instruction, and M1 and M2 are addresses referenced by two load instructions. These two load instructions are considered to read from the same page if M1 and M2 are pointing to the same page. In our implementation, we consider two memory addresses M_1 and M_2 in the same page when $M_1 \& (\sim 0xFFF) = M_2 \& (\sim 0xFFF)$. Here, we consider the page size to be 4KB. Therefore, the number of redundant zeros of a memory load instruction is accumulated when it is considered to be on the same page.

3) Cache line Level Sparsity: The aforementioned method can also be applied to redundant zero detection at the cache level, where we choose $0x3F$ as the mask on a loaded memory address. As the common size of the cacheline in modern CPU architecture is 64 bytes, our approach can detect the fraction of redundant zeros at the cache level. A high fraction of redundant zeros at both page and cache level is a symptom of inefficiency in the layer of operating system and architecture, which indicates the redundant zeros waste the limited resources and thus hurt overall performance.

For better portability, the settings of page/cache line sizes in Zerospy are defined as macros and can be provided as compilation parameters when adapting to different platforms. The corresponding masks are calculated based on the customized page/cache line sizes. The capability of ZeroSpy pinpointing page and cache line sparsity in programs complements the existing software and hardware approaches [24]–[27] that compress redundant zeros for better performance. We will integrate ZeroSpy with these approaches in the future to automate the optimization. In this study, page- and cache-level profiling also provide optimization guidance in case studies such as *649.fotonik3d* and *NWChem*.

C. Handling Parallel Program

ZeroSpy can handle the parallel programs with multiple threads and processes. For multi-threaded programs, each thread handles its local profiled data, including calling context trees and profile of redundant zeros. Therefore, it does not require extra synchronization across parallel threads in the code-centric mode of ZeroSpy. However, in the data-centric mode, ZeroSpy builds a map to maintain all the allocated data objects. Inserting or deleting this map due to the object allocation and deallocation in different threads requires synchronization. For multi-processed programs, each process has its own memory space, and the profiling results are separated by their PIDs. Executing multiple processes on a single node or multiple nodes does not affect the results of identifying redundant zeros. For a parallel HPC execution, we can identify a program with inefficiency if a process of the program loads a large number of redundant zeros. Thus, there is no need to merge the reports from all processes. Alternatively, the users can write a customized post-mortem tool to automate the analysis of the generated reports in parallel HPC execution.

D. Reporting Redundant Zeros

ZeroSpy reports redundant zeros with code- and data-centric attribution and ranks them using redundancy fraction and redmap. When the program terminates, ZeroSpy sorts the profiled data according to the number of redundant zeros and present the top few candidates to the user. The user typically investigates the calling contexts of the top few redundant zeros, which are often responsible for the software inefficiency.

As shown in Fig. 4, the format of the reported redundant zeros by ZeroSpy can be classified into three types: redundant zeros in *1) static object*, *2) dynamic object* and *3) integer/floating-point instructions*. In data-centric mode, the redundant zeros are reported in the format shown in Fig. 4 (a) and (b). For static objects, the name of the object is shown at the beginning of the record, which could be different from the name defined in source code due to the compiler naming strategy. For dynamic objects, ZeroSpy reports the location of the memory allocation through its calling context instead of the object name. The format for both static and dynamic object reports the data size, data not accessed (question marks in Fig. 4 (a) and (b)) and redundant zeros at a per-byte basis. The redmap proposed in Section III is reported from **the least** to **the most** significant bytes (from left to right in the format) with maximum number of 128 bytes. In addition, the top five data objects with maximum redundant zeros are recorded in the profile generated by ZeroSpy with their redmaps printed for offline analysis.

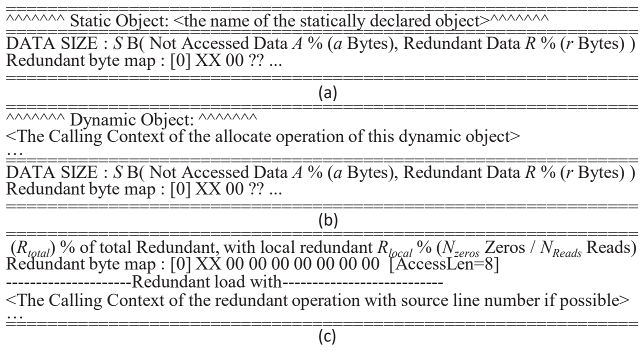

```
================================================================
^^^^^^^ Static Object: <the name of the statically declared object>^^^^^^^
================================================================
DATA SIZE : S B( Not Accessed Data A % (a Bytes), Redundant Data R % (r Bytes) )
Redundant byte map : [0] XX 00 ?? ...
================================================================
                              (a)
================================================================
^^^^^^^ Dynamic Object: ^^^^^^^
<The Calling Context of the allocate operation of this dynamic object>
...
================================================================
DATA SIZE : S B( Not Accessed Data A % (a Bytes), Redundant Data R % (r Bytes) )
Redundant byte map : [0] XX 00 ?? ...
================================================================
                              (b)
================================================================
 (R_total) % of total Redundant, with local redundant R_local % (N_zeros Zeros / N_Reads Reads)
Redundant byte map : [0] XX 00 00 00 00 00 00 00  [AccessLen=8]
---------------------Redundant load with---------------------------
<The Calling Context of the redundant operation with source line number if possible>
...
================================================================
                              (c)
```

Fig. 4. The format of the redundant zero report generated by ZeroSpy (a) for static object, (b) for dynamic object, and (c) for integer/floating point instructions.

Whereas in code-centric mode, the format of the record generated by ZeroSpy is shown in Fig. 4 (c). In the first line of the report, R_{total} is the fraction of redundant zeros of the reported instruction among all detected redundant zeros, and R_{local} is the fraction of redundant zeros among the execution of this particular instruction. The redmap of the instruction is shown from the least to the most significant bytes, similar to data-centric mode. The calling context with source line number is listed alongside the instruction dealing with redundant zeros, which guides the programmer for code optimization.

V. Evaluation

A. Experiment Setup

We evaluate ZeroSpy on an Intel machine with the detailed hardware and software configurations shown in Table I. To evaluate ZeroSpy and report redundant zeros, we choose SPEC CPU2017 [43], CORAL-2 [46], NAS Parallel Benchmarks (NPB) [50], Rodinia [45], CortexSuite [51] and a production software package ShengBTE [52]. For SPEC CPU2017 benchmarks, we all use *ref* inputs. For CORAL-2, NPB, CortexSuite, and Rodinia, we use the default input dataset given in the benchmark suites. For ShengBTE, we use official QE input released in the software package, which spends most of the time in computation. For all benchmarks, we use GCC compiler with *-O3* optimization for compilation. We run each

TABLE I
THE HARDWARE AND SOFTWARE CONFIGURATIONS.

Machine	Intel-Broadwell
Processor	Xeon E5-2680v4@2.40GHz
Cores	14
Memory	256 GB DDR4
L1/L2/L3 Cache	448 KB/3.5MB/35MB
Compiler	gcc 9.2.0 -O3
System	Linux 3.10.0-514.el7.x86_64

parallel program with 14 threads (1 CPU) and collect the proposed metrics (on average) representing redundant zeros.

B. Identifying Redundant Zeros

Fig. 5 shows the fraction of redundant zeros in the evaluated programs detected by ZeroSpy. The blue and green bars indicate the redundant zeros of integer values and floating-point values, respectively. The redundancy of each program reported in Fig. 5 is averaged if there are multiple datasets evaluated. The ***AVG*** bar is the average redundancy across all evaluated programs, which shows that there is 23.43% integer redundant zeros and 1.43% floating-point redundant zeros in our evaluation.

The detailed redundancy of each program is listed in Table II. The redundancy is further divided into *INT* for integer redundant zeros and *FP* for floating-point redundant zeros. We observe that the fraction of integer redundant zeros is always larger than floating-point for the integer benchmarks in SPEC CPU2017. Whereas, for some floating-point benchmarks, the fraction of floating-point redundant zeros is larger than integer, while for others, it is the opposite. It is easy to understand that the integer/floating-point redundant zeros dominate in integer/floating-point benchmarks. However, for those floating-point benchmarks whose integer redundant zeros dominate, the reason could be that accessing a floating-point value is almost always associated with an integer value (e.g., accessing a floating-point value within an array using an integer index), where the floating-point value itself contains few redundant zeros but the associated integer value. Furthermore, we can see from Table II that some of the programs contain large fraction of redundant zeros, such as *gcc* (over 50% integer redundancy) and *fotonik3d* (over 10% floating-point redundancy). The above redundant zeros detected by ZeroSpy can guide the program optimization, which is further discussed in Section VI.

The heatmaps shown in Fig. 6 represent the redundant zeros detected by ZeroSpy from the aspect of data, which is a more intuitive visualization of sparsity with different scales of redundant zeros. The heatmap contains two types of values: the black cells indicate the non-zero values, and the white cells indicate the zero values. For better illustration, we draw the heatmaps in rectangle, of which width and height are the same as the square root of the evaluated array size. Fig. 6 (a) is the heatmap of the top redundant array in *603.bwaves*, which contains 9.99% fully redundant zeros. Fig. 6 (b) is the heatmap of 649.fotonik3d with 99.04% fully redundant zeros. The higher redundancy usually indicates higher data sparsity, which reveals the opportunity to be optimized with sparse data structures. In addition, the heatmap also shows the statistic distribution of data patterns and presents insights for optimization strategies specified to the data sparsity.

Moreover, we evaluate ZeroSpy at the cache and page level. Table III shows the fraction of redundant zeros at cache and page level by thresholding the local redundancy from fully to larger than 50%. We only demonstrate redundancies of SPEC CPU2017 benchmarks due to the space constraint. As shown in Table III, the fraction of fully redundant zeros at cache level is 48.23% at maximum and 7.82% on average. At the page level, the fraction of fully redundant zeros is 52% at maximum and 7.1% on average. The above results indicate the caching strategy, such as fast shared "zero cache" or "zero page" could be quite effective in eliminating these fully redundant zeros, and alleviate about 45% and 40% load at cache and page respectively, which could result in notable performance improvement. Actually, similar ideas have already been proposed in [24], [25] and our evaluation results coincide with existing research. When we relax the threshold of the local redundancy to 50%, more than 10% of data buffered in cache and page can be regarded as redundant zeros. The results reveal a common pattern of over-provisioned data types that occupy more memory bandwidth and cache capacity than needed. The hardware or OS designer can propose appropriate mechanisms (e.g., compressed cache [53]) by reducing or eliminating such inefficiencies at cache and page level identified by Zerospy for better performance. Redundant zeros in cache/page are interesting phenomenons to study the caching designs to improve the efficiency at the cache and page level. We hope these results present useful insights for system and architecture research.

C. Overhead

The overhead of ZeroSpy is listed in Table II. The average time overhead of all evaluated benchmarks is 72.31× in code-centric mode and 118.51× in data-centric mode. Some benchmarks such as *631.deepsjeng* in SPEC CPU2017, result in high time overhead of 335×. Such high overhead is due to the deep and large calling context tree when deriving the calling context using CCTLib. In general, the profiling overhead appears to be high when there is a large fraction of redundant zeros existing in the program. In addition, the SIMD and x87 instructions also lead to heavy PIN API calls and thus increase the time overhead. Since Zerospy is a dynamic profiler based on instrumentation, and the runtime overhead is similar to the well-accepted profilers [3], [4]. To further reduce the runtime overhead of Zerospy, we can apply a bursty sampling mechanism [54] to tradeoff profiling accuracy and overhead in the future. Moreover, the average memory overhead is 108.94× in code-centric mode and 14.26× in data-centric mode. The *nn* program in Rodinia results in an extremely high memory overhead of 2656.65×. This is because the *nn* program itself only consumes within 1 KB memory, whereas the data structures maintained by ZeroSpy and Pin take up several hundreds of KB. Although the memory

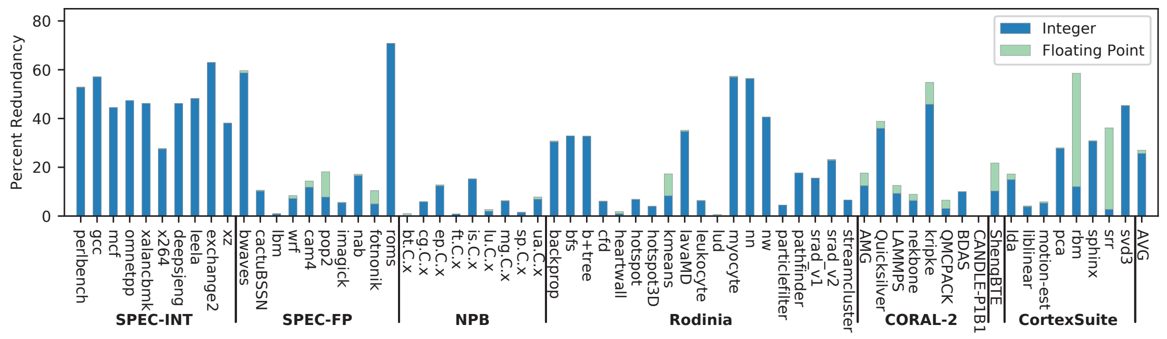


Fig. 5. The redundant zeros reported by ZeroSpy in the evaluated programs.

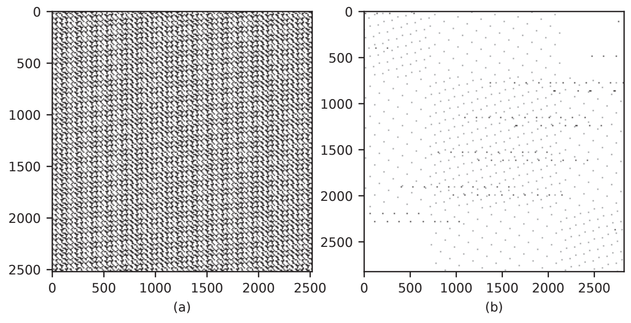
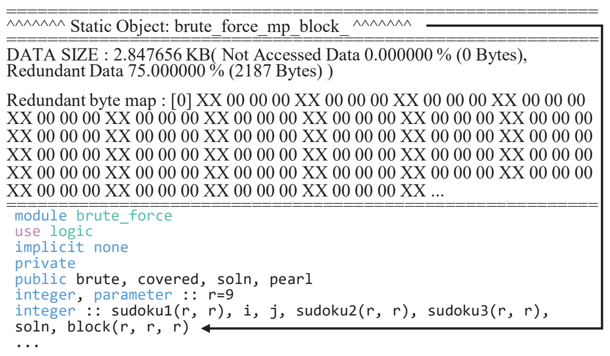


Fig. 6. The heatmap generated based on the redmap of the top candidate arrays containing redundant zeros reported by ZeroSpy ((a) 9.99% redundant zeros in *603.bwaves* and (b) 99.04% redundant zeros in *649.fotonik3d*). The white cells indicate zeros and black cells indicate non-zeros.



```
==========================================================
^^^^^^^ Static Object: brute_force_mp_block_ ^^^^^^^
==========================================================
DATA SIZE : 2.847656 KB( Not Accessed Data 0.000000 % (0 Bytes),
Redundant Data 75.000000 % (2187 Bytes) )

Redundant byte map : [0] XX 00 00 00 XX 00 00 00 XX 00 00 00 XX 00 00 00
XX 00 00 00 XX 00 00 00 XX 00 00 00 XX 00 00 00 XX 00 00 00 XX 00 00 00
XX 00 00 00 XX 00 00 00 XX 00 00 00 XX 00 00 00 XX 00 00 00 XX 00 00 00
XX 00 00 00 XX 00 00 00 XX 00 00 00 XX 00 00 00 XX 00 00 00 XX 00 00 00
XX 00 00 00 XX 00 00 00 XX 00 00 00 XX 00 00 00 XX 00 00 00 XX 00 00 00
XX 00 00 00 XX 00 00 00 XX 00 00 00 XX 00 00 00 XX ...
==========================================================
 module brute_force
 use logic
 implicit none
 private
 public brute, covered, soln, pearl
 integer, parameter :: r=9
 integer :: sudoku1(r, r), i, j, sudoku2(r, r), sudoku3(r, r),
 soln, block(r, r, r)
 ...
```

Fig. 7. The top candidate of redundant zeros in exchange2 reported by ZeroSpy, where **brute_force_mp_block** is the statically declared three-dimensional variable *block* in **brute_force** module.

overhead of *nn* seems quite high, the absolute memory usage when running with ZeroSpy is still within hundreds of KB, which is small enough to fit in the memory.

VI. Case Studies

In this section, we investigate the programs with high redundancy and propose corresponding optimizations guided by ZeroSpy. Table IV overviews the programs we investigated. We compile and run these programs with the same configuration described in Section V-A. We use RAPL [55] to measure energy consumption. We safely revise the code to avoid redundant computations without changing the semantics. Especially, we rely on minimum domain knowledge from program analysis to guarantee the correctness. Moreover, we specialize in the code to maintain both optimized and original paths. We have the code execute the optimized path only when we can guarantee the correctness, otherwise the code goes to the original path. All evaluated programs are whether highly optimized at industrial strength (e.g., 648.exchange2 in SPEC CPU2017) or real-world applications (e.g., NWChem with over a million LOC) in production usage. Zerospy can still identify the software inefficiency caused by redundant zeros and improve the performance by 3.03%-52.8% as shown in Table IV.

A. SPEC CPU2017 exchange2

Exchange2 is a sudoku puzzle generator included in SPEC CPU2017 benchmark suite. It is written in Fortran without parallelization. Due to its goal of generating sudoku games, the computation of exchange2 deals with integers instead of floating points. ZeroSpy reports over 50% integer redundant zeros, but little in floating point. Fig. 7 shows the top candidate of redundant zeros in exchange2 reported by ZeroSpy. The result indicates that the ***block*** variable in **brute_force** module involves large fraction of redundant zeros (75%).

From the pattern in the redmap, we can infer that the redundant zeros come from the data type that is more than enough to store the value, for the most significant three bytes of the integer data type are always zero. Since this variable is to store the number of sudoku games (ranging from 0 to 9), it is feasible to reduce the data type to one-byte integer in order to eliminate the redundant zeros. Similar redundancies are also reported with variables ***sudoku1***, ***sudoku2*** and ***sudoku3*** in **brute_force** module and thus we apply the same optimization to these variables. The speedup achieved by our optimization is 9.8%, as shown in Table IV.

B. SPEC CPU2017 fotonik3d

Fotonik3d is a floating-point benchmark that uses the finite-difference time-domain method for the Maxwell equations in order to compute the transmission coefficient of a photonic

TABLE II
THE FRACTION OF REDUNDANT ZEROS AND PROFILING OVERHEAD REPORTED BY ZEROSPY FOR BOTH CODE-CENTRIC AND DATA-CENTRIC ANALYSES.

Program	% Redundancy		Time Overhead		Memory Overhead	
	INT	FP	CC	DC	CC	DC
600.perlbench	53.91	0.02	72.70	148.02	4.78	3.59
	51.10	0.03	87.80	227.26	4.53	3.00
	53.45	0.00	75.19	224.89	9.29	6.80
602.gcc	52.88	0.04	66.44	180.22	20.40	12.07
	59.51	0.18	86.66	156.20	53.63	33.66
	58.73	0.15	85.63	157.99	54.89	30.79
605.mcf	44.51	0.00	28.31	55.54	1.05	1.71
620.omnetpp	47.36	0.00	29.96	120.39	2.83	3.22
623.xalancbmk	46.19	0.03	38.19	174.72	1.72	3.43
625.x264	27.67	0.06	57.71	146.08	2.97	1.76
	27.08	0.34	60.09	149.78	9.22	4.28
	27.66	0.25	62.34	162.06	3.95	2.44
631.deepsjeng	46.19	0.00	335.05	232.79	1.03	1.26
641.leela	48.21	0.05	45.91	38.92	225.25	151.70
648.exchange2	63.00	0.00	80.52	81.31	169.23	42.25
657.xz	34.97	0.00	32.19	57.35	1.14	3.17
	41.26	0.00	26.81	46.88	1.26	1.88
SPEC INT MAX	63.00	0.34	335.05	232.79	225.25	151.70
603.bwaves	58.35	0.92	100.98	93.11	1.21	1.70
	59.17	0.90	88.16	94.91	1.21	1.71
607.cactuBSSN	10.24	0.24	60.17	94.80	1.42	1.10
619.lbm	0.93	0.03	12.73	18.54	1.70	4.95
621.wrf	7.23	1.14	15.14	29.48	22.34	11.06
627.cam4	11.84	2.52	27.02	68.24	5.82	4.58
628.pop2	7.83	10.33	44.79	85.14	2.15	2.50
638.imagick	5.45	0.17	86.06	216.73	1.83	8.49
644.nab	16.67	0.46	75.91	171.00	5.18	7.39
649.fotnonik	5.01	5.39	23.98	59.93	1.24	2.92
654.roms	70.82	0.31	25.07	67.65	1.16	2.72
SPEC FP MAX	70.82	10.33	100.98	216.73	22.34	11.06
NPB MEDIAN	5.88	0.18	59.17	78.72	3.97	15.02
NPB MAX	15.25	0.89	196.09	229.35	148.30	29.77
Rodinia MEDIAN	15.57	0.03	66.84	84.79	62.86	15.73
Rodinia MAX	57.00	8.95	450.39	524.62	2656.65	48.70
AMG	11.39	5.05	25.97	66.29	5.02	4.40
	13.52	5.22	34.92	71.58	27.13	3.55
Quicksilver	36.01	2.84	64.55	159.55	1.62	2.99
LAMMPS	9.29	3.27	39.43	76.43	2.50	19.47
nekbone	6.45	2.47	79.55	295.02	2.53	45.08
kripke	45.85	8.96	102.09	207.32	1.80	7.00
QMCPACK	3.08	3.45	26.94	-	1.11	-
BDAS	10.01	0.00	100.01	40.35	50.61	6.56
CANDLE - P1B1	0.01	0.00	42.86	-	4.93	-
CORAL-2 MAX	45.85	8.96	102.09	295.02	50.61	45.08
ShengBTE	10.27	11.47	63.47	173.49	5.91	40.76
CortexSuite MAX	49.46	46.48	138.56	241.32	276.85	179.65
AVG	25.66	1.33	72.31	118.51	108.94	14.26
MEDIAN	22.88	0.18	64.17	99.52	4.47	6.56

*** Floating Point (FP), Integer (INT), Code-Centric Mode (CC), Data-Centric Mode (DC), BDAS uses princomp-cxx version, '-': Failed to run due to out of memory (e.g., deep call chain or frequent data allocation). Due to space constraint, only the maximum values for NPB, Rodinia and CortexSuite benchmarks are reported.**

waveguide. After profiling with ZeroSpy, 5.39% of floating-point redundant zeros are reported. Zeropsy reports that the redundant zeros come from dynamically allocated arrays such as ***E_x***, ***E_y***, ***E_z***, ***H_x***, *H_y* and *H_z*. These arrays contain large fraction of fully redundant zeros, which indicates these arrays are **sparse**. The cause of software inefficiency of fotonik3d is that it handles sparse data with dense data structure as well as dense algorithm. The inappropriate use of data structure leads to redundant computations.

Our optimization is to change the dense data structure to a more efficient sparse data structure. The original dense data structure of ***E_z*** is a three-dimension array in the order of *(x,y,z)*. We observe that the x-z plane contains fully redundant zeros. To optimize, we first transpose the original data structure to *(x,z,y)* so that the data is continuous in the x-z plane. Then we only store the non-zero values along the x-z plane and index along the y-axis. The corresponding algorithm is also proposed to access the data in a sparse format, which eliminates redundant computations. The proposed optimization results in 52.8% speedup with *ref* dataset.

TABLE III
BREAKDOWN OF LOCAL REDUNDANY REPORTED BY ZEROSPY AT CACHE AND PAGE LEVEL.

	Redundant Cache (LR) %				Redundant Page (LR) %			
No.	100	<90	<70	<50	100	<90	<70	<50
600	0.13	0.29	0.51	1.05	0.01	0.11	0.21	0.60
	0.04	0.06	0.28	11.11	0.01	0.02	0.05	6.70
	0.06	0.44	0.64	1.21	0.00	0.05	0.14	0.46
602	0.24	0.30	0.66	5.21	0.02	0.04	0.13	4.80
	2.66	3.09	5.64	11.61	1.05	1.89	2.59	6.32
	2.17	2.61	4.78	10.61	1.09	1.28	1.79	5.18
603	27.27	27.27	66.91	80.24	21.37	21.37	66.56	83.12
	27.43	27.43	66.98	80.28	21.54	21.54	66.63	83.16
605	0.00	0.00	0.00	0.01	0.00	0.00	0.00	0.00
607	15.31	15.31	15.75	15.89	14.89	14.93	15.45	15.65
619	0.09	0.09	0.09	0.09	1.45	1.45	1.45	1.45
620	3.64	4.73	5.64	13.07	0.00	3.77	6.23	7.04
621	17.63	19.56	21.98	24.37	10.20	15.53	20.12	24.60
623	0.56	0.68	0.78	1.09	0.00	0.00	0.02	0.04
625	0.19	0.39	1.70	3.42	0.02	0.18	1.29	3.46
	0.92	1.37	6.30	8.50	0.01	0.44	6.02	7.27
	0.26	0.37	0.96	3.07	0.01	0.16	0.68	2.93
627	9.41	10.54	12.47	15.47	3.55	7.52	11.73	15.61
628	35.37	36.03	39.25	44.51	10.98	18.94	31.66	46.22
631	5.83	40.16	71.10	90.35	0.00	0.00	45.77	99.55
638	26.66	26.77	27.33	30.48	52.00	52.15	52.77	55.53
641	3.39	3.57	3.92	4.65	3.17	3.21	3.47	4.05
644	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05
648	2.43	3.72	4.69	7.07	0.55	4.40	7.14	10.99
649	48.23	49.91	54.04	58.62	46.27	48.61	54.27	60.19
654	7.88	7.88	7.89	7.90	7.86	7.88	7.89	7.92
657	0.64	0.70	1.34	3.53	0.00	0.00	0.03	0.08
	0.52	0.52	0.57	0.67	1.49	1.51	2.07	3.15

*** Local Redundancy (LR), SPEC No.: 600.perlbench, 602.gcc, 603.bwaves, 605.mcf, 607.cactuBSSN, 619.lbm, 621.wrf, 623.xalancbmk, 625.x264, 627.cam4, 628.pop2, 631.deepsjeng, 638.imagick, 641.leela, 644.nab, 648.exchange2, 649.fotnonik, 654.roms, 657.xz.**

```
local redundant 50.000000 % (46388660 Zeros/92777320 Reads)
==== Redundant byte map : [mantiss | exponent | sign] ====
XX XX XX XX XX XX XX | XX XX | X
===== [AccessLen=8, typesize=8] =======
-------------------Redundant load with---------------------
300734:0x404d2c:vmulsdq  0x68(%rsp), %xmm0, %xmm0:egb._omp_fn.1:
    cpu2017/benchspec/CPU/644.nab_s/src/eff.c:2314
298714:0x4097d4:callq  0x404250:mme34:cpu2017/benchspec/CPU/644.
    nab_s/src/eff.c:1890
...
```

Listing 3. The top candidate of redundant zeros in nab reported by ZeroSpy.

C. SPEC CPU2017 nab

Nucleic Acid Builder (nab) is a molecular modeling program included in SPEC CPU2017 benchmark suite. The computation-intensive nature of nab is typical in life science simulation. ZeroSpy reports that nab contains 16.67% integer redundant zeros and 0.46% floating-point redundant zeros. Listing 3 shows the top candidate of redundant zeros in nab reported by ZeroSpy.

The identified code region leads to redundant zeros is shown in Listing 4, where the value of ****kappa*** is usually observed to be zero (50% reported by ZeroSpy in Listing 3). As the fraction of fully redundant zeros is high, we can use *if/else* statement to skip the useless computations dealing

TABLE IV
SUMMARY OF PERFORMANCE IMPROVEMENT GUIDED BY ZEROSPY.

Redundancy types	Programs	Directed by	Problematic procedures/variables	speedup	power saving
Data with more than enough storage	641.leela [43]	CC	FastState.cpp:176	8.7%	7.57%
	648.exchange2 [43]	CC	brute_force_mp_block	9.8%	5.90%
	JM [56]	CC/DC	bi_context_type	13.0%	14.1%
Inappropriate use of data structure	649.fotonik3d [43]	DC/CP	*E_z*:yeemain.fppized.f90:112	52.8%	34.85%
	gcc (SPEC CPU2006) [43]	DC	*last_set*:loop_regs_scan()	11.3%	9.8%
	srr [51]	CC	SRREngine.c:25	58.8%	38.7%
	BT [50]	DC	*work_lhs*:z_solver.f	4.4%	4.5%
Useless computation	644.nab [43]	CC	eff.c:189	7.1%	5.88%
	heartwall [45]	CC	kernel.c:57	9.6%	6.79%
	ShengBTE [52]	CC	ShengBTE.f90:476	9.46%	10.04%
	638.imagick [43]	CC	morphology.c:2982	25%	28.8%
	NWChem [57]	DC/CP	tce mo2e trans.F:240	20%	10%
	backprop [45]	CC	backprop.c:323	40.8%	29.4%
	Stack RNN [58]	CC	StackRNN.h:loop(352,365)	26.80%	26.73%
	xgboost [59]	CC	updater_colmaker.cc:if(422,434)	3.03%	2.55%
	QuEST [60]	CC	QuEST_cpu.c:2120	8.77%	8.93%

*** Code-Centric Mode (CC), Data-Centric Mode (DC), Cacheline level and Page level detection (CP)**

```
for (j = 0; j < prm->Natom; j++) {
  for (k = 0; k < npairs; k++) {
    ...
    fgbk = -(*kappa) * KSCALE / fgbi;
    expmkf = exp(fgbk) / (*diel_ext);
    temp6 = qiqj * temp4 * (dielfac + fgbk * expmkf);
    ...
  }
}
```

Listing 4. The redundant zeros in the *egb* function of nab.

```
if(*kappa==0) {
  for (j = 0; j < prm->Natom; j++) {
    for (k = 0; k < npairs; k++) {
      ...
      expmkf = 1 / (*diel_ext);
      temp6 = qiqj * temp4 * dielfac;
      ...
    }
  }
} else {
  ... // same as before
}
```

Listing 5. The optimized code region in the *egb* function of nab.

with redundant zeros. To reduce the branching overhead, the value of ***kappa** is evaluated before entering the two nested loops. The optimized code is shown in Listing 5, and similar optimizations are also applied to other code regions with similar redundancies. After our optimization, nab achieves 7.1% performance speedup.

D. QuEST

QuEST [60] is the cutting-edge high performance simulator of universal quantum circuits, which is capable of simulating quantum circuits with multi-qubit controlled gates. Over 80% redundant zeros in floating point are reported by Zerospy and the large fraction of redundant zeros can be attributed to the code region listed in Listing 6. The values of *alphaReal*, *alphaImag*, *betaReal*, and *betaImag* are frequently observed to be zero. Moreover, the above values can be classified into four cases: all zeros, only *alphaReal* and *alphaImag* are zeros, only *betaReal* and *betaImag* are zeros, and the rest. Since these values are loop invariant, we can use *if/else* statements outside the loop to skip useless computations with redundant zeros. With our optimization, QuEST achieves a 8.77% speedup.

E. ShengBTE

ShengBTE [52] is a phonon computation software to compute the lattice contribution to the thermal conductivity of bulk crystalline solids by solving the Boltzmann transport equation.

```
for (thisTask=0; thisTask<numTasks; thisTask++) {
  ...
  stateVecReal[indexUp] = alphaReal*stateRealUp - alphaImag*
      stateImagUp
    - betaReal*stateRealLo - betaImag*stateImagLo;
  stateVecImag[indexUp] = alphaReal*stateImagUp + alphaImag*
      stateRealUp
    - betaReal*stateImagLo + betaImag*stateRealLo;
  ...
  stateVecReal[indexLo] = betaReal*stateRealUp - betaImag*
      stateImagUp
    + alphaReal*stateRealLo + alphaImag*stateImagLo;
  stateVecImag[indexLo] = betaReal*stateImagUp + betaImag*
      stateRealUp
    + alphaReal*stateImagLo - alphaImag*stateRealLo;
}
```

Listing 6. The code region in the *statevec_controlledCompactUnitaryLocal* function of QuEST.

It is written in Fortran and parallelized with MPI. Over 10% redundant zeros in integer are reported by ZeroSpy. Zerospy reports that the redundant zeros come from the most significant 7 bytes of the loaded value by instructions in *Vp_minus*, as shown in Listing 7, which is invoked for over millions of times by outer loops. ZeroSpy reports that most redundant zeros come from the indexing of the floating-point array by three loop variables: tt, ss and rr (line 126-128 of Listing 7).

The redundant code region is the inner three loops of a four-level nested loop in *Vp_minus*, and the number of iterations of the inner three loops is always three. Therefore, the redundant zeros from the loop variables can be eliminated by unrolling the inner three loops. However, the GCC compiler fails to do so, which seems intuitive. After investigating the assembly codes, we observe the compiler first optimizes the innermost loop with vectorization, which results in a long segment of vectorized code. The long vectorized codes prevent further loop unrolling of the nested loops. Based on the above observation, we manually unroll the inner three loops, and the redundant zeros are eliminated as the loop variables (tt, ss and rr) are inlined as immediate values to the instructions. In addition, the compiler is still able to generate vectorized codes for outer loop. The similar optimization is applied to the *Vp_plus* function, in which ZeroSpy identifies frequent redundant zeros. Our optimization yields a 9.46% speedup finally.

F. Stack RNN

Stack RNN [58] is a C++ based project originating from Facebook AI research, which applies memory stack to op-

```
Vp0=0.
do rr=1,3
  do ss=1,3
    do tt=1,3
      Vp0=Vp0+Phi(tt,ss,rr,ll)*&
        eigenvect(q,i,tt+3*(Index_i(ll)-1))*&
        conjg(eigenvect(qprime,j,ss+3*(Index_j(ll)-1)))*&
        conjg(eigenvect(qdprime,k,rr+3*(Index_k(ll)-1)))
    end do
  end do
end do
```

Listing 7. The redundant zeros in the *Vp_minus* function of ShengBTE.

```
350 for(my_int i=_TOP_OF_STACK;i<_TOP_OF_STACK+_STACK_SIZE-1;i++)
351 {
352 ▶  _pred_err_stack[s][i+1] += _err_stack[s][i] * _act[s][itm
      ][pop];
353 }
355 for(my_int i=_TOP_OF_STACK;i<_TOP_OF_STACK+_STACK_SIZE-1;i++)
356 {
357 ▶  _err_act[s][pop] += _err_stack[s][i] * _stack[s][old_it][
      i+1];
358 }
359 _err_act[s][pop] += _err_stack[s][_TOP_OF_STACK + _STACK_SIZE
      - 1] * EMPTY_STACK_VALUE;
363 for(my_int i=_TOP_OF_STACK+1;i<_TOP_OF_STACK+_STACK_SIZE;i++)
364 {
365 ▶  _pred_err_stack[s][i-1] += _err_stack[s][i] * _act[s][itm
      ][push];
366 }
367 for(my_int i=_TOP_OF_STACK+1;i<_TOP_OF_STACK+_STACK_SIZE;i++)
368 {
369 ▶  _err_act[s][push] += _err_stack[s][i] * _stack[s][old_it
      ][i-1];
370 }
```

Listing 8. Redundant zeros detected in Stack RNN. Most elements of array $\_err\_stack$ equal zero, resulting in spatial load redundancy.

timize and extend a recurrent neural network. We evaluate Stack RNN by profiling its built-in application train_add with ZeroSpy. ZeroSpy quantifies a floating-point redundancy fraction of 78.48%, and pinpoints that the top redundant zeros are detected in four loops as shown in Listing 8.

The cause of the frequent redundant zeros is that each of the four loops accesses array $\_err\_stack$, and most elements of the array $\_err\_stack$ are zeros, resulting in identity computation at lines 352, 357, 365 and 369 shown in Listing 8. We employ a conditional check to avoid the computation of identities. Moreover, we also eliminate the redundant loads of zeros by loop fusion, which fuses the four loops into one so that array $\_err\_stack$ is only loaded once. These two optimizations together eliminate 10% of the memory loads, and 15% of the redundant memory loads, yielding a 27% speedup for the whole program.

G. JM

JM [56] is a reference implementation of HEVC applications [6] for H.264/AVC encoding and decoding. We evaluate the video encoding program in JM with Zerospy and observe 28.3% integer redundant zeros from the report. Zerospy identifies that the data copies of data structure *bi_context_type* (Listing 9) lead to a large fraction of redundant zeros (over 20%). The data structure consists of one 64-bit integer, and two 8-bit integers with padding of 6 bytes by default. The zero-padding of the data structure wastes 37.5% of memory bandwidth as well as cache capacity. To optimize, we annotate the data structure with *__attribute__((packed))*, which yields 13.0% performance speedup.

VII. Discussions

ZeroSpy instruments every memory access instruction, so it does not generate any false positive or false negative. However, as a tool, ZeroSpy reports the inefficiencies and provides rich information to understand the inefficiencies, but

```
struct bi_context_type
{
  unsigned long  count;
  byte state; // index into state-table CP
  unsigned char  MPS; // Least Probable Symbol 0/1 CP
};
```

Listing 9. Data structure with a large fraction of redundant zeros in JM.

it is the programmers' responsibility to apply the optimization. Currently, Zerospy is restricted to Intel platforms due to the PIN-based implementation. In the future, we consider rebasing our implementation on DynamoRio [61], which can support broader platforms (e.g., Intel, AMD, and ARM). We further discuss other aspects in the usage of ZeroSpy.

a) Optimization efforts: To admit that Zerospy is a performance tool guiding for optimization, using it does require basic knowledge of optimization (e.g., data locality). Guided by ZeroSpy, a Ph.D. student typically requires a few hours to a few days to investigate the profile and optimize a program. For example, the student used one hour to optimize *648.exchange2*, while *649.fotonik3d* took up a week for optimization. It is worth noting that the student has no domain knowledge of any optimized applications.

b) Actionable optimization guidance: It is worth noting that we do not optimize all programs with a large volume of redundant zeros. First, some of the code requires significant domain knowledge to optimize the algorithms or data structures. We will send our profiles to the application developers to explore better code optimization. Second, some redundant zeros are introduced intentionally due to security issues. For example, some pages are always initialized to zeros before any usage. ZeroSpy reports all redundant zeros, without distinguishing actionable ones.

c) Input dependency: As a dynamic profiler, ZeroSpy measures program execution with specific inputs. Such input dependency is common for dynamic analysis approaches [1], [4], [18]–[23]. Thus, ZeroSpy requires users to run the programs with typical inputs that can represent the programs' real behaviors.

d) Understanding Zerospy report: Users can analyze the report, identify the type of root cause, and then reference the optimizations presented in specific case studies to guide the optimization of their own applications. Zerospy also provides a simple root cause diagnosis and recommended optimization receipts in its report.

VIII. Conclusion

In this paper, we present ZeroSpy, a fine-grained profiler that explores the software inefficiency caused by redundant zeros. ZeroSpy identifies instructions and data objects dealing with redundant zeros, which reveals significant potentials for performance optimization. ZeroSpy provides rich information such as redundancy metrics, calling contexts, and source code attribution for intuitive optimization guidance. ZeroSpy works on fully optimized binary executables with reasonable overhead, so it is applicable to real-world production codebases. Based on the guidance provided by ZeroSpy, we are able to optimize a number of well-known benchmarks and real-world applications, yielding significant speedups. ZeroSpy is publicly available at [62].

Acknowledgements

This work is supported by National Key Research and Development Program of China (Grant No. 2016YFB1000304), National Natural Science Foundation of China (Grant No. 61502019 and 61732002) and the Open Project Program of the State Key Laboratory of Mathematical Engineering and Advanced Computing (Grant No. 2019A12). Hailong Yang is the corresponding author.

References

[1] P. Su, S. Wen, H. Yang, M. Chabbi, and X. Liu, "Redundant loads: a software inefficiency indicator," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, J. M. Atlee, T. Bultan, and J. Whittle, Eds. IEEE / ACM, 2019, pp. 982–993. [Online]. Available: https://dl.acm.org/citation.cfm?id=3339628

[2] K. M. Lepak and M. H. Lipasti, "On the value locality of store instructions," in *27th International Symposium on Computer Architecture (ISCA 2000), June 10-14, 2000, Vancouver, BC, Canada*, A. D. Berenbaum and J. S. Emer, Eds. IEEE Computer Society, 2000, pp. 182–191. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/ISCA.2000.854389

[3] M. Chabbi and J. M. Mellor-Crummey, "Deadspy: a tool to pinpoint program inefficiencies," in *10th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2012, San Jose, CA, USA, March 31 - April 04, 2012*, C. Eidt, A. M. Holler, U. Srinivasan, and S. P. Amarasinghe, Eds. ACM, 2012, pp. 124–134. [Online]. Available: https://doi.org/10.1145/2259016.2259033

[4] S. Wen, M. Chabbi, and X. Liu, "REDSPY: exploring value locality in software," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, Y. Chen, O. Temam, and J. Carter, Eds. ACM, 2017, pp. 47–61. [Online]. Available: https://doi.org/10.1145/3037697.3037729

[5] S. Wen, X. Liu, and M. Chabbi, "Runtime value numbering: A profiling technique to pinpoint redundant computations," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, Oct 2015, pp. 254–265.

[6] B. Lee, J. Jung, and M. Kim, "An all-zero block detection scheme for low-complexity hevc encoders," *IEEE Transactions on Multimedia*, vol. 18, no. 7, pp. 1257–1268, 2016.

[7] K. Peng, S. Fu, Y. Liu, and W. Hsu, "Adaptive runtime exploiting sparsity in tensor of deep learning neural network on heterogeneous systems," in *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, SAMOS 2017, Pythagorion, Greece, July 17-20, 2017*, Y. N. Patt and S. K. Nandy, Eds. IEEE, 2017, pp. 105–112. [Online]. Available: https://doi.org/10.1109/SAMOS.2017.8344617

[8] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, "Precimonious: Tuning assistant for floating-point precision," in *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2013, pp. 1–12.

[9] H. Menon, M. O. Lam, D. Osei-Kuffuor, M. Schordan, S. Lloyd, K. Mohror, and J. Hittinger, "Adapt: algorithmic differentiation applied to floating-point precision tuning," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE Press, 2018, p. 48.

[10] G. Marin, C. McCurdy, and J. S. Vetter, "Diagnosis and optimization of application prefetching performance," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*. ACM, 2013, pp. 303–312.

[11] A. Delmas Lascorz, P. Judd, D. M. Stuart, Z. Poulos, M. Mahmoud, S. Sharify, M. Nikolic, K. Siu, and A. Moshovos, "Bit-tactical: A software/hardware approach to exploiting value and bit sparsity in neural networks," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: ACM, 2019, pp. 749–763. [Online]. Available: http://doi.acm.org/10.1145/3297858.3304041

[12] P. Briggs, K. D. Cooper, and L. T. Simpson, "Value numbering," *Software: Practice and Experience*, vol. 27, no. 6, pp. 701–724, 1997.

[13] 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 2001, Sorrento, Napoli, Italy, June 16-21, 2001*, M. M. Furnari and E. Gallopoulos, Eds. ACM, 2001, pp. 65–77. [Online]. Available: https://doi.org/10.1145/377792.377807

[14] M. N. Wegman and F. K. Zadeck, "Constant propagation with conditional branches," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 2, pp. 181–210, 1991. [Online]. Available: https://doi.org/10.1145/103135.103136

[15] M. F. Fernandez, "Simple and effective link-time optimization of modula-3 programs," in *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI), La Jolla, California, USA, June 18-21, 1995*, D. W. Wall, Ed. ACM, 1995, pp. 103–115. [Online]. Available: https://doi.org/10.1145/207110.207121

[16] T. Johnson, M. Amini, and D. X. Li, "Thinlto: scalable and incremental LTO," in *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*, V. J. Reddi, A. Smith, and L. Tang, Eds. ACM, 2017, pp. 111–121. [Online]. Available: http://dl.acm.org/citation.cfm?id=3049845

[17] R. Gupta, E. Mehofer, and Y. Zhang, "Profile-guided compiler optimizations," in *The Compiler Design Handbook: Optimizations and Machine Code Generation*, Y. N. Srikant and P. Shankar, Eds. CRC Press, 2002, pp. 143–174. [Online]. Available: https://doi.org/10.1201/9781420040579.ch4

[18] A. C. De Melo, "The new linuxperftools," in *Slides from Linux Kongress*, vol. 18, 2010.

[19] L. Adhianto, S. Banerjee, M. W. Fagan, M. Krentel, G. Marin, J. M. Mellor-Crummey, and N. R. Tallent, "HPCTOOLKIT: tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010. [Online]. Available: https://doi.org/10.1002/cpe.1553

[20] J. Reinders, "Vtune performance analyzer essentials," *Intel Press*, 2005.

[21] S. L. Graham, P. B. Kessler, and M. K. McKusick, "gprof: a call graph execution profiler," in *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, Boston, Massachusetts, USA, June 23-25, 1982*, J. R. White and F. E. Allen, Eds. ACM, 1982, pp. 120–126. [Online]. Available: https://doi.org/10.1145/800230.806987

[22] L. D. Rose, B. Homer, D. Johnson, S. Kaufmann, and H. Poxon, "Cray performance analysis tools," in *Tools for High Performance Computing - Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing, July 2008, HLRS, Stuttgart*, M. M. Resch, R. Keller, V. Himmler, B. Krammer, and A. Schulz, Eds. Springer, 2008, pp. 191–199. [Online]. Available: https://doi.org/10.1007/978-3-540-68564-7_12

[23] J. Levon, P. Elie *et al.*, "Oprofile, a system-wide profiler for linux systems," *Homepage: http://oprofile. sourceforge. net*, 2008.

[24] J. Dusser, T. Piquet, and A. Seznec, "Zero-content augmented caches," in *Proceedings of the 23rd International Conference on Supercomputing*, ser. ICS '09. New York, NY, USA: ACM, 2009, pp. 46–55. [Online]. Available: http://doi.acm.org/10.1145/1542275.1542288

[25] J. Dusser and A. Seznec, "Decoupled zero-compressed memory," in *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, ser. HiPEAC '11. New York, NY, USA: ACM, 2011, pp. 77–86. [Online]. Available: http://doi.acm.org/10.1145/1944862.1944876

[26] S. S. Manohar and H. K. Kapoor, "Refresh optimised embedded-dram caches based on zero data detection," in *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, SAC 2019, Limassol, Cyprus, April 8-12, 2019*, C. Hung and G. A. Papadopoulos, Eds. ACM, 2019, pp. 635–642. [Online]. Available: https://doi.org/10.1145/3297280.3297340

[27] H. Gao, C. Wu, J. Li, and M. Guo, "Zero-chunk: An efficient cache algorithm to accelerate the I/O processing of data deduplication," in *22nd IEEE International Conference on Parallel and Distributed Systems, ICPADS 2016, Wuhan, China, December 13-16, 2016*. IEEE Computer Society, 2016, pp. 635–642. [Online]. Available: https://doi.org/10.1109/ICPADS.2016.0089

[28] J. S. Miguel, M. Badr, and N. D. E. Jerger, "Load value approximation," in *47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2014, Cambridge, United Kingdom,*

December 13-17, 2014. IEEE Computer Society, 2014, pp. 127–139. [Online]. Available: https://doi.org/10.1109/MICRO.2014.22

[29] J. S. Miguel, J. Albericio, A. Moshovos, and N. D. E. Jerger, "Doppelgänger: a cache for approximate computing," in *Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015*, M. Prvulovic, Ed. ACM, 2015, pp. 50–61. [Online]. Available: https://doi.org/10.1145/2830772.2830790

[30] A. Yazdanbakhsh, G. Pekhimenko, B. Thwaites, H. Esmaeilzadeh, O. Mutlu, and T. C. Mowry, "RFVP: rollback-free value prediction with safe-to-approximate loads," *TACO*, vol. 12, no. 4, pp. 62:1–62:26, 2016. [Online]. Available: https://doi.org/10.1145/2836168

[31] M. Burrows, Ú. Erlingsson, S. Leung, M. T. Vandevoorde, C. A. Waldspurger, K. Walker, and W. E. Weihl, "Efficient and flexible value sampling," in *ASPLOS-IX Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, MA, USA, November 12-15, 2000.*, L. Rudolph and A. Gupta, Eds. ACM Press, 2000, pp. 160–167. [Online]. Available: https://doi.org/10.1145/356989.357004

[32] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S. 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.*, vol. 15, no. 4, pp. 357–390, 1997. [Online]. Available: https://doi.org/10.1145/265924.265925

[33] S. Wen, X. Liu, J. Byrne, and M. Chabbi, "Watching for software inefficiencies with witch," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*, X. Shen, J. Tuck, R. Bianchini, and V. Sarkar, Eds. ACM, 2018, pp. 332–347. [Online]. Available: https://doi.org/10.1145/3173162.3177159

[34] B. Calder, P. Feller, and A. Eustace, "Value profiling," in *Proceedings of 30th Annual International Symposium on Microarchitecture*. IEEE, 1997, pp. 259–269.

[35] B. Calder, P. Feller, A. Eustace *et al.*, "Value profiling and optimization," *Journal of Instruction Level Parallelism*, vol. 1, no. 1, pp. 1–6, 1999.

[36] P. T. Feller, "Value profiling for instructions and memory locations," Master's thesis, University of California, San Diego, 1998.

[37] S. Watterson and S. Debray, "Goal-directed value profiling," in *International Conference on Compiler Construction*. Springer, 2001, pp. 319–333.

[38] J. Tan, S. Jiao, M. Chabbi, and X. Liu, "What every scientific programmer should know about compiler optimizations?" in *Proceedings of the 34th ACM International Conference on Supercomputing*, 2020, pp. 1–12.

[39] M. Stephenson, J. Babb, and S. Amarasinghe, "Bidwidth analysis with application to silicon compilation," in *ACM SIGPLAN Notices*, vol. 35, no. 5. ACM, 2000, pp. 108–120.

[40] H. Yin, H. Cai, and H. Lu, "A new all-zero block detection algorithm for high efficiency video coding," in *2017 Data Compression Conference (DCC)*. IEEE, 2017, pp. 470–470.

[41] M. Wang, X. Xie, J. Li, H. Jia, and W. Gao, "Fast rate distortion optimized quantization method based on early detection of zero block for hevc," in *2017 IEEE Third International Conference on Multimedia Big Data (BigMM)*. IEEE, 2017, pp. 90–93.

[42] X. Yang, S. M. Blackburn, D. Frampton, J. B. Sartor, and K. S. McKinley, "Why nothing matters: the impact of zeroing," in *Acm Sigplan Notices*, vol. 46, no. 10. ACM, 2011, pp. 307–324.

[43] J. Bucek, K.-D. Lange *et al.*, "Spec cpu2017: Next-generation compute benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. ACM, 2018, pp. 41–42.

[44] D. Bailey, T. Harris, W. Saphir, R. Van Der Wijngaart, A. Woo, and M. Yarrow, "The nas parallel benchmarks 2.0," Technical Report NAS-95-020, NASA Ames Research Center, Tech. Rep., 1995.

[45] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization, IISWC 2009, October 4-6, 2009, Austin, TX, USA*. IEEE Computer Society, 2009, pp. 44–54. [Online]. Available: https://doi.org/10.1109/IISWC.2009.5306797

[46] "Coral-2 benchmarks, https://asc.llnl.gov/coral-2-benchmarks/," 2017.

[47] 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 *Acm sigplan notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.

[48] M. Chabbi, X. Liu, and J. M. Mellor-Crummey, "Call paths for pin tools," in *12th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2014, Orlando, FL, USA, February 15-19, 2014*, D. R. Kaeli and T. Moseley, Eds. ACM, 2014, p. 76. [Online]. Available: https://dl.acm.org/citation.cfm?id=2544164

[49] W. Kahan, "Ieee standard 754 for binary floating-point arithmetic," *Lecture Notes on the Status of IEEE*, vol. 754, no. 94720-1776, p. 11, 1996.

[50] D. H. Bailey, "Nas parallel benchmarks," *Encyclopedia of Parallel Computing*, pp. 1254–1259, 2011.

[51] S. Thomas, C. Gohkale, E. Tanuwidjaja, T. Chong, D. Lau, S. Garcia, and M. B. Taylor, "Cortexsuite: A synthetic brain benchmark suite," in *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2014, pp. 76–79.

[52] W. Li, J. Carrete, N. A. Katcho, and N. Mingo, "Shengbte: A solver of the boltzmann transport equation for phonons," *Computer Physics Communications*, vol. 185, no. 6, pp. 1747 – 1758, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0010465514000484

[53] S. Sardashti and D. A. Wood, "Decoupled compressed cache: Exploiting spatial locality for energy-optimized compressed caching," in *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2013, pp. 62–73.

[54] Y. Zhong and W. Chang, "Sampling-based program locality approximation," in *Proceedings of the 7th international symposium on Memory management*, 2008, pp. 91–100.

[55] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "Rapl: memory power estimation and capping," in *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*. IEEE, 2010, pp. 189–194.

[56] G. S. Alexis Michael Tourapis, Karsten Shring, "H.264/14496-10 avc reference software manual," 2009. [Online]. Available: http://iphome.hhi.de/suehring/tml/

[57] M. Valiev, E. J. Bylaska, N. Govind, K. Kowalski, T. P. Straatsma, H. J. Van Dam, D. Wang, J. Nieplocha, E. Apra, T. L. Windus *et al.*, "Nwchem: A comprehensive and scalable open-source solution for large scale molecular simulations," *Computer Physics Communications*, vol. 181, no. 9, pp. 1477–1489, 2010.

[58] A. Joulin and T. Mikolov, "Inferring algorithmic patterns with stack-augmented recurrent nets," in *Advances in neural information processing systems*, 2015, pp. 190–198.

[59] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. ACM, 2016, pp. 785–794.

[60] T. Jones, A. Brown, I. Bush, and S. C. Benjamin, "Quest and high performance simulation of quantum computers," *Scientific reports*, vol. 9, no. 1, pp. 1–11, 2019.

[61] D. Bruening, E. Duesterwald, and S. Amarasinghe, "Design and implementation of a dynamic optimization framework for windows," in *4th ACM workshop on feedback-directed and dynamic optimization (FDDO-4)*, 2001.

[62] "Zerospy." [Online]. Available: https://github.com/JerryYouxin/zerospy_for_SC20

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

We evaluate ZeroSpy on an Intel Xeon E5-2680v4@2.40GHz with 14 cores. To evaluate ZeroSpy and report redundant zeros, we choose SPEC CPU2017, CORAL-2, NPB v3.3, Rodinia v3.1, CortexSuite and a production software package ShengBTE v1.1. For SPEC CPU2017 benchmarks, we all use ref inputs. For CORAL-2, NPB, CortexSuite and Rodinia, we use the default input dataset given in the benchmark suites. For ShengBTE, we use official QE input released in the software package, which spends most of the time in computation. For all benchmarks, we use GCC v9.2.0 with -O3 optimization for compilation. We run each parallel program with 14 threads (1 CPU) and collect the proposed metrics (on average) representing redundant zeros. We use OpenMPI v4.0.3 for compiling ShengBTE to support execution with multiple processes.

ARTIFACT AVAILABILITY

Software Artifact Availability: All author-created software artifacts are maintained in a public repository under an OSI-approved license.

Hardware Artifact Availability: There are no author-created hardware artifacts.

Data Artifact Availability: There are no author-created data artifacts.

Proprietary Artifacts: None of the associated artifacts, author-created or otherwise, are proprietary.

Author-Created or Modified Artifacts:

```
Persistent ID: https://github.com/JerryYouxin/zerosp⌋
↪  y\_for\_SC20;
↪  https://doi.org/10.5281/zenodo.3865337
Artifact name: zerospy
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: Intel Xeon E5-2680v4@2.40GHz with 14 cores

Operating systems and versions: CentOS 7.6 running Linux kernel 3.10.0

Compilers and versions: GCC v9.2.0

Applications and versions: SPEC CPU2017, CORAL-2, NPB v3.3, Rodinia v3.1, CortexSuite and a production software package ShengBTE v1.1.

Libraries and versions: OpenMPI v4.0.3

URL to output from scripts that gathers execution environment information.

```
https://github.com/JerryYouxin/zerospy\_for\_SC20/bl⌋
↪  ob/master/execution-environment.txt
```