DeadSpy: A Tool to Pinpoint Program Inefficiencies

Milind Chabbi
Department of Computer Science
Rice University, Houston
Texas, USA
milind.chabbi@rice.edu

John Mellor-Crummey
Department of Computer Science
Rice University, Houston
Texas, USA
johnmc@rice.edu

ABSTRACT

Software systems often suffer from various kinds of performance inefficiencies resulting from data structure choice, lack of design for performance, and ineffective compiler optimization. Avoiding unnecessary operations, and in particular memory accesses, is desirable. In this paper, we describe DeadSpy — a tool that dynamically detects every dead write to memory in a given execution and provides actionable feedback to the programmer. This tool provides a methodical way to identify dead writes, which is a common symptom of performance inefficiencies. Our analysis of the SPEC CPU2006 benchmarks showed that the fraction of dead writes is surprisingly high. In fact, we observed that the SPEC CPU2006 `gcc` benchmark has 61% dead writes on average across its reference inputs. DeadSpy pinpoints source lines contributing to such inefficiencies. In several case studies with high dead writes, simple code restructuring to eliminate dead writes improved their performance significantly. For `gcc`, avoiding dead writes improved its running time by as much as 28% for some inputs and 14% on average. We recommend dead write elimination as an important step in performance tuning.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Measurement techniques, Performance attributes.

General Terms

Performance, Measurement, Algorithms

Keywords

Dead write detection, code inefficiency

1. INTRODUCTION

Software systems often suffer from various kinds of performance inefficiencies. Some inefficiencies are induced by programmers during design (e.g., poor data structure selection) and implementation (e.g., use of generic, heavy-weight programming abstractions). Sometimes performance losses are compiler induced, e.g., not inlining hot functions or moving less frequently executed code to hot regions. Software that remain in constant development tends to accumulate several such inefficiencies over time. Identifying inefficiencies in programs and eliminating them is important not only for commercial developers, but also for scientists writing computation intensive codes for simulation, analysis, or modeling. Performance analysis tools, such as `gprof` [12], HPCToolkit [1], `Xperf` [22], and `vTune` [16], attribute running time of a program to code structures at various granularities. However, such tools are incapable of identifying whether a program's execution time is *well spent*.

On modern architectures, memory accesses are costly. For many programs, exposed memory latency accounts for a significant fraction of execution time. Unnecessary memory accesses, whether cache hits or misses, lead to poor resource utilization and have a high energy cost as well [14]. In the era where processor to memory gap is widening [17, 21], gratuitous accesses to memory are a cause of inefficiency, more so if they happen on hot paths.

A *dead write* occurs when there are two successive writes to a memory location without an intervening read. Dead writes are useless operations. Performance analysis tools mentioned above lack the ability to detect inefficiencies related to dead writes. Compiler optimizations that reduce memory accesses by register allocation of scalars (e.g., [8]) or array elements (e.g., [7]) are critical to high performance. Other optimizations that eliminate redundant computations [19, 5, 9] are similarly important. However, none of these optimizations are effective at global elimination of dead writes. Analysis of dead writes is complicated by issues including aliasing, aggregate types, function boundaries, late binding, and partial deadness.

As we show in Section 4, dead writes are surprisingly frequent in complex programs. To pinpoint dead writes, we developed DeadSpy—a tool that monitors every memory access, identifies dead writes, and provides actionable feedback to guide application tuning. Using DeadSpy to analyze the reference executions of the SPEC CPU2006 benchmarks showed that the integer benchmarks had over 20% dead writes and the floating point benchmarks had over 9% dead writes. On some inputs, the SPEC CPU2006 `403.gcc` benchmark had as many as 76% dead writes.

In addition to providing a quantitative metric of dead writes, DeadSpy reports source lines and complete calling contexts involved in high frequency dead writes. Such quantitative attribution pinpoints opportunities where source code changes could significantly improve program efficiency. Various causes of inefficiency manifest themselves as dead writes at runtime. We show cases where lack of or inefficient compiler optimizations cause dead writes; we also highlight cases where programmer did not design for performance. We restructure codes that have significant fractions of dead writes and demonstrate performance improvements. To the best of our knowledge, DeadSpy is the first dynamic dead write detection tool. The proposed methodology of eliminating dead writes is a *low-cost high-yield* strategy when looking for opportunities to improve application performance.

This paper makes the following contributions:

- We identify dead writes as a symptom of inefficiency, which arise from many causes.
- We propose elimination of dead writes as an opportunity for improving performance.
- We built a tool to count dead writes in an execution and precisely attribute these counts to source lines.
- We analyzed a variety of benchmark programs and show that the fraction of dead writes in SPEC CPU2006 is surprisingly high.
- We identify several cases where dead writes result from ineffective compiler optimization.
- In case studies, we demonstrate that eliminating causes of dead writes significantly improves performance.

The rest of the paper is organized as follows. Section 2 presents a new methodology for detecting program inefficiencies via tracking dead writes. Section 3 sketches the design and implementation of DeadSpy. Section 4 evaluates benchmark programs using DeadSpy. Section 5 studies four codes to explore the causes of dead writes and the performance benefits of dead write elimination. Section 6 describes related work. Finally, Section 7 summarizes our conclusions.

2. METHODOLOGY

In this section we describe a dynamic dead write detection algorithm. The driving principle behind our tool is the invariant that *two writes to the same memory location without an intervening read operation make the first write to that memory location dead.*

To identify dead writes throughout an execution, we monitor every memory read and write operation issued during program execution. For each addressable unit of memory `M`, accessed by the program, we assign a state — `STATE(M)` as either *Read* (`R`) or *Written* (`W`) indicating whether the last operation on `M` was a read or a write respectively. The state transitions of each memory location obey the automaton shown in Figure 1, where the transition edges are labeled with `<instruction/action>` pairs. Every memory location starts in a *Virgin*(`V`) state. An instruction that reads `M` updates `STATE(M)` to `R`. An instruction that writes `M` updates

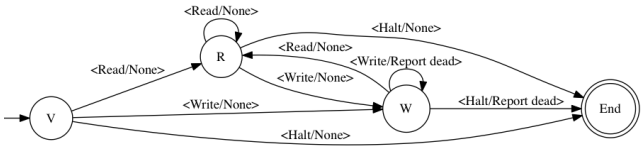

Figure 1: State transition diagram.

`STATE(M)` to `W`. A state transition from `W` to `W` corresponds to a dead write. When a dead write is detected, information is recorded for later reporting. Similarly, a final write to `M` without a subsequent read qualifies as a dead write. All other state transitions have no actions associated with them. A `halt` instruction transitions the automaton to the terminating state. For a terminating program, our technique guarantees termination. Since the automaton considers the effect of each memory operation executed in the program from start to finish, there can be no false-positives or false-negatives. The approach is sound even in the event of asynchronous control transfers.

For multi-threaded programs, if two consecutive writes to a location, without an intervening read, are from two different threads, this may indicate a source of non-determinism or a data race. In this work, our focus is to identify program inefficiencies for a single thread of execution; for multi-threaded executions, we treat such writes to a memory location simply as dead writes.

3. DESIGN AND IMPLEMENTATION

We have implemented DeadSpy using Intel's cross platform dynamic binary instrumentation tool *Pin* [20] to monitor every read and write operation. We use *shadow memory* [23] to remember the state of each memory location. We instrument each function's `CALL` and `RETURN` instructions to build a dynamic *calling context tree (CCT)* [2] and record the program contexts involved in dead writes. Each interior node in our CCT represents a function invocation; and each leaf node represents a write instruction. We apportion deadness to pairs of CCT contexts that are involved in dead writes and present them to the user at program termination. In our implementation, we decided not to include a scan at program termination to recognize each final `W` $\rightarrow$ `End` transition as dead; its contribution to overhead is negligible and in practice it may be infeasible to eliminate.

In the following subsections, we first introduce terminology used in the rest of the paper and briefly describe Pin. Then, we describe our implementation of DeadSpy, including its use of shadow memory and CCT construction, along with its strategies for recording and reporting dead writes. Next, we present the challenges involved in attributing dead writes to source lines and then we sketch the details of our solution. We conclude this section with the details of accounting and attributing deadness.

3.1 Terminology

In the context of this paper, we define the following terms:

- *Read* is an instruction that has the side effect of loading a value from memory.

- *Write* is an instruction that has the side effect of storing a value into memory.
- *Operation* represents a dynamic instance of an instruction.

For a `W` → `W` state transition at location `M`, we define:

- *Dead context* as the program context in which the first write happened; this context wrote an unread value and hence is a candidate for optimization.
- *Killing context* as the program context which overwrote the previously written location without an intervening read of the location.

3.2 Introduction to Pin

Pin is a dynamic binary instrumentation tool which provides a rich set of high-level APIs to instrument a program with *analysis routines* at different granularities including module, function, trace, basic block and instruction. A *trace* in Pin jargon, is a single entry multiple exit code sequence — for example, a branch starts a new trace at the target, and a function call, return, or a jump ends the trace. Instrumentation occurs immediately before a code sequence is executed for the first time. Using Pin, we instrument every read and write instruction to update the state of each byte of memory affected by the operation. Pin also provides APIs to intercept system calls, which we use to update the shadow memory to account for side effects of system calls.

3.3 Maintaining memory state information

In our implementation, we maintain the current state of each memory location in a *shadow memory*, analogous to *Memcheck*, a Valgrind tool [23]. Each memory byte `M` has a shadow byte to hold its current state `STATE(M)`. The shadow byte of `M` can be accessed by using `M`'s address to index a two-level page table. We create chunks of 64KB shadow memory pages on demand. On 64-bit Linux machines, only the lower 48 bits are used in user address space. With 64KB shadow pages, the lower 16 bits of address provide an offset into a shadow page where the memory status is stored. The higher 20 bits of 48 bits, provide an offset into the first-level page table which holds 2^{20} entries — on a 64-bit machine it occupies 8MB space. The middle 12 bits provide an offset into a second-level page table. Each second-level page table has 2^{12} entries and occupies 32KB on a 64-bit machine. Each second-level page table is allocated on demand *iff* an address is accessed in its range. We adopt *copy-on-write* semantics so that read-only pages do not get shadowed. To maintain the context for a write operation, we store an additional pointer sized variable `CONTEXT(M)` in the shadow memory for each memory byte `M`. Thus, we have one shadow byte of state and an 8-byte context pointer for a total of nine bytes of metadata per data byte. Figure 2 shows the organization of shadow memory; numbers in dark circles represent the steps involved in address translation.

3.4 Maintaining context information

To accurately report the *dead context* and the *killing context* involved in every dead write, each instruction writing to location `M` also updates `STATE(M)` and records the information needed to recover the calling context and the instruction

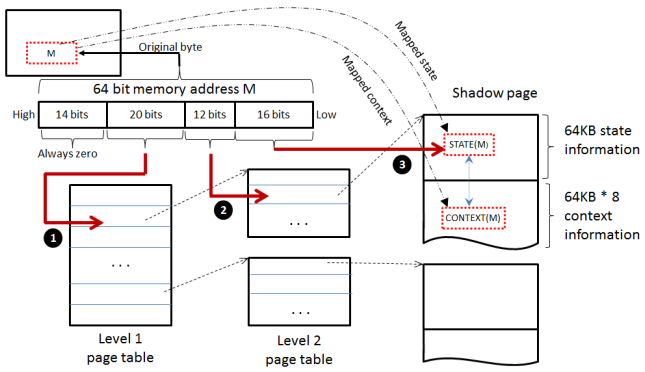

Figure 2: Shadow memory and address translation.

pointer (IP) of the write. We accomplish this by maintaining a CCT which dynamically grows as the execution unfolds. In this subsection, we provide the details of building a simple CCT; the details of including the IP information to map back to source lines are presented in Section 3.7. To build a simple CCT, we insert instrumentation before each `CALL` and `RETURN` machine instruction. We maintain a globally accessible pointer `curCtxtNodePtr` to a CCT node that represents the current function; the path from that node to the CCT root represents the call stack. Each `CALL` instruction creates a new `ContextNode` for the callee under `curCtxtNodePtr` if not already present, and sets `curCtxtNodePtr` to the callee CCT node. Each `RETURN` instruction sets the `curCtxtNodePtr` to its parent CCT node. The Pin analysis routine executed just before each write instruction updates the pointer-sized variable `CONTEXT(M)` in the shadow memory for each of the data bytes written with the location pointed to by `curCtxtNodePtr`. For multithreaded codes, there will be one CCT per thread; a spin lock ensures that the instruction and the corresponding analysis routine run atomically.

3.5 Recording dead writes

When a `W` → `W` state transition is detected, we record a 3-tuple `<`*dead context pointer, killing context pointer, frequency*`>` into a table (`DeadTable`), where each entry is uniquely identified by `<`*dead context pointer, killing context pointer*`>` ordered pair. For example, `<CONTEXT(M), curCtxtNodePtr,1>` is the 3-tuple when a dead write is observed for location `M`. If such a record is already present, its *frequency* is incremented. The *frequency* accumulates the number of bytes dead for a given pair of contexts. We discuss the details of assessing and attributing dead writes in Section 3.8.

3.6 Reporting dead and killing contexts

At program termination, all 3-tuples are retrieved from the `DeadTable`, and sorted by decreasing *frequency* of each record. For each tuple, the full call chain representing its *dead context* is obtained by traversing parent links in the CCT starting from the *dead context pointer*[1]. Similarly, we

[1]For library calls where the target of a call is to a trampoline, we disassemble target address of the jump to extract the correct function name.

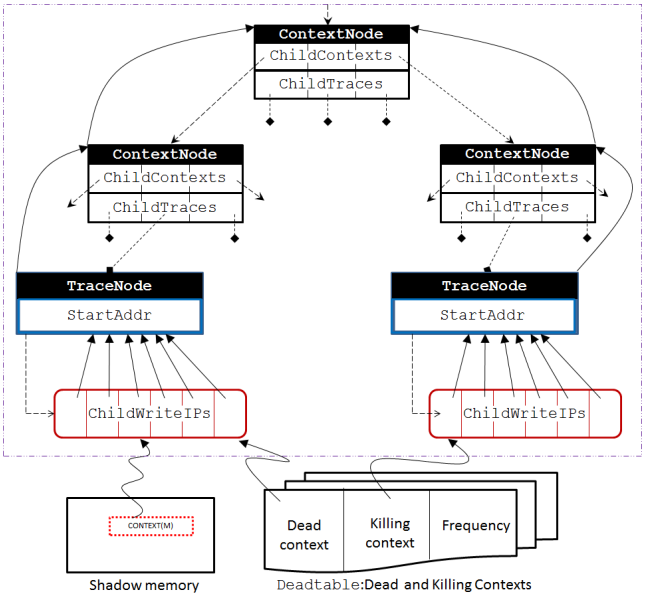

Figure 3: Calling context tree.

can retrieve the *killing context* by traversing parent links in the CCT starting from the *killing context pointer*.

3.7 Attributing to source lines

Sometimes having only a chain of function names as a context may not suffice; source line numbers involved in the dead and killing writes may be needed. Tracking this information requires recording the IP in addition to the enclosing function for each write operation. Naively recording additional pointer sized IP values bloats shadow memory and adds excessive overhead. We provide a solution which avoids both. One could consider adding write IPs as leaf nodes in the CCT; these nodes would represent writes within a parent function. However, every time a write operation is executed, recording the context would involve looking up the corresponding CCT node for the write IP; this would inflate the cost of monitoring dramatically. Instead, one could consider representing the set of write instructions in a function as an array and assigning a slot index to each. Using Pin, one can assign a unique slot index to each write instruction during JIT translation; with this approach, the slot index for each write instruction is available to an analysis routine during execution in constant time.

Pin makes information about function boundaries available to instrumentation at run time. This information could be used to scan each function and assign a slot index to each write instruction. A problem with this approach is that it requires precise knowledge of function boundaries. In practice, program disassembly is imprecise [26] and discovering function boundaries relies upon compiler-generated symbol information, which is often incomplete and sometimes incorrect [29]. The approach of associating write instructions in a function with slots would fail when execution transfers to an arbitrary code region where precise function bounds are unavailable. This is not uncommon; even the startup code executed before reaching the `main()` function exercises such corner cases.

Providing a perfect solution to track IPs of write operations involves complex engineering of the CCT. To facilitate this, we developed a strategy that makes use of *traces* that Pin generates at JIT time. While the function bounds can be incorrect, the instructions identified in a trace are always correct since trace extraction happens at runtime. We modify the aforementioned simple CCT such that each `ContextNode` has several child traces (`ChildTraces`), each one represented by a `TraceNode`. Each `TraceNode` has an array of child IPs `(ChildWriteIPs)` where each array element corresponds to a write instruction under that trace, as shown in Figure 3.

We use Pin to instrument each trace entry. We also use Pin to add instrumentation before each `CALL` instruction to set a flag. On entering a trace, if the flag is set, we know that we have just entered a new function. If the flag is set when entering a trace, we inspect the children of the current CCT node `curCtxtNodePtr` looking for a child `ContextNode` representing the current IP; if none exists, we create and insert one. We update `curCtxtNodePtr` to the appropriate child and reset the flag. Whether the flag was set or not, we next lookup a child `TraceNode` under `curCtxtNodePtr` (creating and inserting a new one if necessary), and set `curTraceNodePtr` to point to the current trace. *Tail calls* to known functions are handled by having Pin instrument function prologues to adjust `curCtxtNodePtr` to point to a sibling node, and then adjust `curTraceNodePtr`. A tail call to an instruction sequence not known to be a function ends up being associated with a trace under the current function.

Using the aforementioned CCT structure, let's consider how we maintain the shadow information for a write operation. If a write instruction W_i is numbered as the 42^{nd} write while JIT-ing a trace `T`; then executing W_i, which writes to location `M`, would update `CONTEXT(M)` with `&curTraceNodePtr->ChildWriteIPs[42]`. If an instance of W_i is a killing write, we record or update the 3-tuple `<CONTEXT(M), &curTraceNodePtr->ChildWriteIPs[42], frequency>` in its `DeadTable`.

To report dead and killing contexts with source line information, we maintain a map that enables us to recover the IP of each write instruction in a trace. To recover the IP of a write instruction in a trace, we use the address of the first instruction in the trace, `StartAddr`, as the key in a map that yields an array; this array maps the slot index of each write instruction in a trace to its corresponding IP. That, combined with the call chain implied by the path through the CCT from a `TraceNode` to the root, identifies the complete calling context.

3.8 Assessing deadness

Read and write operations happen at different byte-level granularities, for example 1, 2, 4, 8, 16, etc.. We define $\overline{AverageWriteSize}$ in an execution as the ratio of the total number of bytes written to the total number of write operations performed.

$$\overline{AverageWriteSize} = \frac{NumBytesWritten}{NumWriteOps}$$

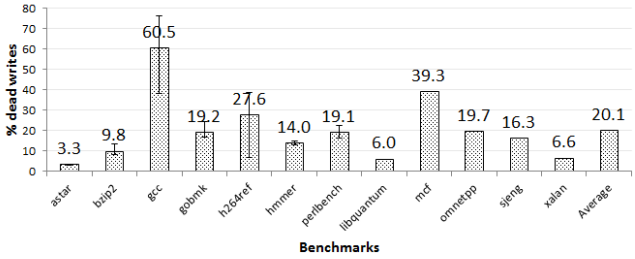

Figure 4: Deadness in SPEC CPU2006-INT.

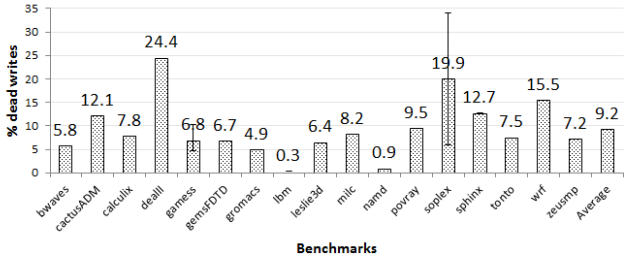

Figure 5: Deadness in SPEC CPU2006-FP.

We approximate the number of dead write operations in an execution as the ratio of the total number of bytes dead to its $\overline{AverageWriteSize}$.

$$\widehat{NumDeadOps} = \frac{NumBytesDead}{\overline{AverageWriteSize}}$$

We define *Deadness* in an execution as the percentage of the total dead write operations out of the total write operations performed, which is same as the percentage of the total number of bytes dead out of the total number of bytes written.

$$\begin{aligned} Deadness &= \frac{\widehat{NumDeadOps}}{NumWriteOps} \times 100 \\ &= \frac{NumBytesDead}{NumBytesWritten} \times 100 \end{aligned}$$

DeadSpy accumulates the total bytes written and the total operations performed in an execution. As stated in Section 3.5, each record C_{ij} in a `DeadTable` is associated with a *frequency* $F(C_{ij})$, representing the total bytes dead in a *dead context* i due to a *killing context* j. We compute total *Deadness* as:

$$Deadness = \frac{\sum_i \sum_j F(C_{ij})}{NumBytesWritten} \times 100$$

We apportion *Deadness* among contributing pair of contexts C_{pq} as:

$$Deadness(C_{pq}) = \frac{F(C_{pq})}{\sum_i \sum_j F(C_{ij})} \times 100$$

An alternative metric to *Deadness* is *Killness* — the ratio of the total write operations killing previously written values to the total write operations in an execution. In practice, we found that both *Deadness* and *Killness* values for programs are almost the same.

4. ANALYSIS OF BENCHMARKS

Experimental setup. For our experiments, we used a quad-socket system with four `AMD Opteron 6168` processors clocked at `1.9 GHz` with 128GB of 1333MHz DDR3 running `CentOS 5.5`. We used the `GNU 4.1.2` [11] compiler tool chain with `-O2` optimization.

4.1 SPEC benchmarks

Deadness in SPEC CPU2006. We measured the deadness of each of the SPEC CPU2006 integer and floating point reference benchmarks; and the results are shown in Figures 4

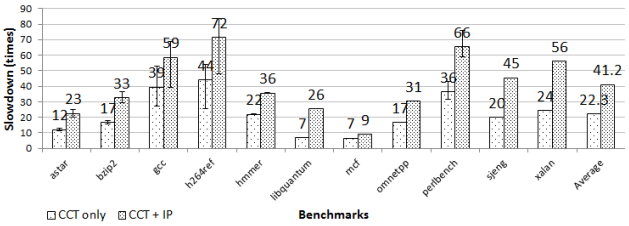

Figure 6: DeadSpy overhead for SPEC CPU2006-INT.

and 5 respectively. Several benchmarks execute multiple times, each time with a different input; each column in Figures 4 and 5 shows the measurements averaged over different inputs for the same benchmark. For a benchmark with multiple inputs, error bars represent the lowest and the highest deadness observed on its different inputs. Average deadness for integer benchmarks is 20.1% with the highest of 76.2% for `gcc` on the input `c-typeck.i`, and the lowest of 3.2% for `astar` on the input `rivers.cfg`. Average deadness for floating point benchmarks is 9.2% with the highest of 33.9% for `soplex` on the input `pds-50.mps`, and the lowest of 0.3% for `lbm`. Average difference between *Killness* and *Deadness* is 2.3% for integer benchmarks and 0.5% for floating point benchmarks.

These measurements indicate that for several of these codes, large fractions of memory access operations are dead. In many cases, just a few pairs of contexts account for most of dead writes. For example, in the SPEC CPU2006 integer reference benchmarks, for the benchmark/input pair with the median deadness, its top five context pairs account for 90% of deadness, and its top 15 context pairs account for 95% of deadness. This indicates that a domain expert could optimize a handful of context pairs presented by DeadSpy and expect to eliminate most dead writes in a program.

Overhead of instrumentation. As a tool that monitors every read and write operation in a program, quite naturally, DeadSpy significantly increases execution time. Figure 6 shows the overhead of each of SPEC CPU2006 integer reference benchmarks. The overhead to obtain dead and killing contexts without line numbers is much less than when line number information is tracked as well. To track line numbers, we need to instrument each trace entry. The av-

Program	Deadness in %								
	Intel 11.1			PGI 10.5			GNU 4.1.2		
	None	Default	Max	None	Default	Max	None	Default	Max
astar	2.3	8.4	5.0	5.2	1.1	5.7	2.5	3.3	7.7
bzip2	4.7	8.6	8.9	4.9	9.9	12.8	5.1	9.8	11.9
gcc	39.3	67.8	67.2	40.8	53.3	51.9	39.7	60.5	64.5
gobmk	15.7	21.3	22.7	17.4	19.1	20.7	16.0	19.2	20.1
h264ref	14.4	28.4	38.3	15.1	24.5	26.2	15.3	27.6	27.9
hmmer	31.3	68.7	68.8	31.5	67.6	67.9	0.3	14.0	29.4
perlbench	15.3	18.0	20.0	16.7	16.5	16.8	13.2	19.1	n/a
libquantum	1.7	6.0	0.3	2.8	7.1	7.5	2.3	6.0	6.1
mcf	16.6	49.4	49.5	17.2	27.6	47.2	17.3	39.3	47.3
omnetpp	4.8	19.4	22.1	11.8	11.2	27.7	4.7	19.7	21.4
sjeng	9.6	20.4	17.8	10.3	11.4	13.9	10.0	16.3	19.7
xalan	1.5	6.6	6.7	5.1	4.7	9.4	1.7	6.6	8.4
Average	**13.1**	**26.9**	**27.3**	**14.9**	**21.2**	**25.6**	**10.7**	**20.1**	**24.0**

Table 1: Deadness in SPEC CPU2006-INT with different compilers and optimization levels.

erage[2] slowdowns due to instrumentation without and with line information are 22.3x and 41.3x respectively. The maximum slowdown is seen for the `h264ref` benchmark on the input `foreman_ref_encoder_baseline.cfg` with 41.3x without line-level attribution and 83.6x with line numbers. High deadness typically comes with high overhead due to the cost of recording participant contexts on each instance of a dead write.

Deadness across compilers and optimization levels. We used `-O2` optimization as the basis for our experiments since it is often used in practice. However, our findings are not limited to a specific optimization level or a specific compiler. We found high deadness across different optimization levels and across different compilers. For completeness, we present the deadness in SPEC CPU2006 integer reference benchmarks when compiled *without* optimization, with *default* optimization, and with *highest* level of optimization on three different compiler chains viz., `Intel 11.1` [15], `PGI 10.5` [30], and `GNU 4.1.2` [11]. By reading the accompanying compiler manual pages, we concluded that the default optimization level is `-O2` on these three compilers; and the highest optimization level for `Intel 11.1` is `-fast`, for `PGI 10.5` is `-fastsse, -Mipa=fast,inline`, and for `GNU 4.1.2` is `-O3 -mtune=opteron`. We did not conduct profile guided optimizations in our experiments. The deadness found across these compilers is shown in Table 1. The `perlbench` benchmark did not finish execution when compiled with `GNU 4.1.2` at the highest optimization, even without DeadSpy attached; hence we do not have the results for the same. It is evident that high deadness is pervasive across compilers and across optimization levels. Quite intuitively, higher optimization levels, except inter-procedural analysis, offer no advantage in eliminating dead writes. Meticulous readers may observe that typically, deadness increases with increase in optimization levels on all compilers; this can be attributed to the fact that with higher optimizations, the absolute number of memory operations often reduces, but the absolute number of dead writes does not reduce proportionally.

4.2 OpenMP NAS parallel benchmarks

To assess the deadness in multithreaded applications, we ran DeadSpy on the OpenMP suite of NAS parallel benchmarks version 3.3 [18] with four worker threads (i.e., `OMP_NUM_THREADS=4`). Table 2 shows the deadness found in these benchmarks. The average deadness for these applications was 5.96%, which is less than those observed for the SPEC CPU2006 serial codes. Moreover, the inter-thread deadness, which happens when a previous write by one thread is overwritten by a different thread, is negligible. For multithreaded applications tuned to maintain affinity between data and threads, there is little inter-thread deadness.

Program	Deadness in %		
	Inter-thread (A)	Intra-thread (B)	Total (A+B)
bt.S	5.28E-02	1.13E+01	1.14E+01
cg.S	6.83E-03	2.86E+00	2.87E+00
dc.S	2.26E-03	1.78E+01	1.78E+01
ep.S	9.84E-02	1.68E-01	2.66E-01
ft.S	3.35E-01	4.17E+00	4.50E+00
is.S	1.08E+00	1.06E+00	2.13E+00
lu.S	1.89E-02	6.01E+00	6.03E+00
mg.S	4.44E-01	6.10E+00	6.55E+00
sp.S	2.68E-01	3.32E+00	3.59E+00
ua.S	1.04E-01	4.43E+00	4.54E+00
Average	**2.40E-01**	**5.72E+00**	**5.96E+00**

Table 2: Deadness in OpenMP NAS parallel benchmarks.

5. CASE STUDIES

In this section, we evaluate the utility of DeadSpy for pinpointing inefficiencies in executions of four codes: the `403.gcc` and `456.hmmer` programs from SPEC CPU2006 benchmarks, the `bzip2` file compression tool [27], and a scientific application `Chombo` [3]. We investigate dead writes in executions of these applications and we apply optimizations to eliminate some of the most frequent dead writes. We present the performance gains achieved after code restructuring. Unless stated otherwise, we use the aforementioned experimental setup.

5.1 Case study 403.gcc

`403.gcc` was our top target for investigation since it showed a very high percentage of dead writes. For the `c-typeck.i` input which yielded 76% deadness, the top most pair of dead contexts accounted for 29% of the deadness. The offending code is shown in Listing 1. In this frequently-called function, `gcc` does the following:

1. On `line 3`: allocates `last_set` as an array of 16937 elements, 8 bytes each, amounting to a total of 132KB.

[2]As of this writing we do not have results for `445.gobmk` due to its large memory footprint.

```
1  void loop_regs_scan(struct loop *loop, ...){
2            ...
3    last_set = (rtx *) xcalloc(regs->num, sizeof (
         rtx));
4    /* Scan the loop, recording register usage */
5    for (each instruction in loop){
6      ...
7      if(GET_CODE (PATTERN (insn)) == SET || ...)
8        count_one_set (..., last_set,...);
9      ...
10     if (end of basic block)
11       memset(last_set,0,regs->num*sizeof(rtx));
12   }
13   ...
14 }
```

Listing 1: Dead writes in `gcc` due to an inappropriate data structure.

2. On `lines 5-12`: iterates through each instruction belonging to the incoming argument `loop`.
3. On `lines 7-8`: If `insn` matches a pattern, calls `count_one_set()` which updates `last_set` with the last instruction that set a virtual register.
4. On `lines 10-11`: if the basic block ends, `memset()` is called to reset the entire 132KB `last_set` array for reuse in the next basic block of `loop`.

The program spends a lot of time zero initializing the array `last_set`, most of which is already zero. DeadSpy detected dead writes in `memset()` with its caller as `loop_regs_scan()`. The root cause for the high amount of deadness is that the basic blocks are typically short and the number of registers used in a block is small; `gcc` allocated a maximum size array without considering this common case. Clearly, a *dense array is a poor data structure choice to represent this sparse register set*. We gathered some statistics on the usage pattern of `last_set` using the `c-typeck.i` input and found that the median use was only 2 *unique* slots with a maximum of 34 slots set between episodes of `memset()`s. Furthermore, the median of total number of writes to *non-unique* slots of `last_set` was 2 with a maximum of 63 between two episodes of `memset()`s. We found that just 22 non-unique slots were accessed on 99.6% of occasions. As a quick fix, we maintained a side array which recorded the first 22 non-unique indices accessed. If the side array does not overflow, we can simply zero at most those 22 indices of `last_set` instead of calling `memset()` on the entire 132KB array. In rare cases when the side array overflows, we can fall back to resetting the entire `last_set` array.

A poor choice of data-structures has manifested itself as dead writes. Better permanent fixes for the aforementioned problem include

- using a sparse representation for the register set, such as splay trees, or
- using a composite representation for the register set that switches from a short sparse vector, a scalable sparse set representation such as a splay tree, and the full dense set representation.

In the latter case, a competitive algorithm could switch between representations based on the number of elements in the set and the pattern of accesses.

```
1  void cselib_init (){
2            ...
3    cselib_nregs = max_reg_num ();
4    // initializes reg_values with zeros
5    VARRAY_ELT_LIST_INIT(reg_values, cselib_nregs,
         ...);
6    ...
7    clear_table (1);
8  }
9
10 void clear_table (int clear_all){
11   // sets all reg_values to zeros
12   for (i = 0; i < cselib_nregs; i++)
13     REG_VALUES (i) = 0;
14     ...
15 }
```

Listing 2: Dead reinitialization in `gcc`.

Program	Workload	%Fast	%Reduction in resource			
			L1	L2	Ops	Cyc
403.gcc	166.i	8.5	10.8	-7.4	14.4	8.7
	200.i	3.5	6.2	-0.3	3.3	3.1
	c-typeck.i	**28.1**	29.0	31.2	29.9	24.7
	cp-decl.i	15.6	14.9	-2.3	24.0	16.8
	expr.i	18.4	14.3	12.7	23.5	18.0
	expr2.i	18.7	10.6	11.8	24.3	17.4
	g23.i	10.9	9.0	10.7	15.9	10.4
	s04.i	22.8	19.2	24.1	23.2	22.2
	scilab.i	1.9	3.6	0.0	0.7	1.4
	average	**14.3**	**13.1**	**8.9**	**17.7**	**13.7**
456.hmmer	retro.hmm	15.1	2.3	1.2	0.5	15.5
	nph3.hmm	16.2	-4.1	-2.3	0.7	16.4
	average	**15.7**	**-0.9**	**-0.6**	**0.6**	**15.9**
bzip2-1.0.6	chicken.jpg	0.8	0.0	0.0	1.0	0.2
	liberty.jpg	0.7	0.0	0.0	0.7	0.5
	input.program	**14.2**	0.5	0.0	10.3	13.9
	text.html	3.5	0.0	0.0	2.1	4.7
	input.source	10.5	0.0	-0.8	7.9	9.7
	input.combined	13.2	0.0	-0.7	9.5	12.5
	average	**7.2**	**0.1**	**-0.3**	**5.2**	**6.9**
Chombo	common.input	6.6	8.2	20.3	2.4	6.0

Table 3: Performance improvements.

Another dead write context was found in the `cselib_init()` function shown in Listing 2. Looking at the macro `VARRAY_ELT_LIST_INIT` revealed that it was allocating and zero initializing the array `reg_values`. Then, without any further reads from the array, the call to `clear_table(1)` was again resetting all elements of `reg_values` to zeros. The dead write symptom highlights losses due to the use of a *generic, heavyweight API on a hot path where slim APIs are needed*. We fixed this by simply calling a specialized version of `clear_table()` that did not initialize `reg_values`.

We identified and fixed two more top dead contexts in `403.gcc`, both related to repeated zero initialization of a dense data structure where the usage pattern was sparse. In general, finding the root causes of performance issues was straightforward once the dead and killing contexts were presented. Optimizing the top four pairs of dead contexts resulted in improving `gcc`'s running time by 28% for the `c-typeck.i` input. The average speedup across all inputs was 14.3%. Table 3 shows the performance improvements in terms of percentage speedup (%Fast column), reduction in L1 data-cache misses (`L1` column), L2 data-cache misses (`L2` column), operations completed (`Ops` column), and processor cycle counts (`Cyc` column) for each of the input files of `gcc` compared to the baseline. The performance improvements come from both reduced cache miss rates and reduced

```
1       ic[k] = mpp[k] + tpmi[k];
2       if ((sc = ip[k] + tpii[k]) > ic[k])
3         ic[k] = sc;
```

Listing 3: Dead writes in `403.hmmer`.

```
1       int icTmp = mpp[k] + tpmi[k];
2       if ((sc = ip[k] + tpii[k]) > icTmp)
3         ic[k] = sc;
4       else
5         ic[k] = icTmp;
```

Listing 4: Avoiding dead writes in `403.hmmer`.

operation counts. Occasionally, there are slightly more L2 misses (represented by negative numbers) which are offset by improvements in other areas. We ran the modified `gcc` on a version of the SPEC'89 `fpppp` code, which we converted from Fortran to C code for our experiments. This benchmark is marked by very long basic blocks and stresses register usage. This stresses the code segments we modified. Despite being a pathological case, our modified `gcc` showed a 2% speedup when compiling `fpppp`.

5.2 Case study 456.hmmer

`456.hmmer` benchmark is a computationally intensive program. It uses profile hidden markov models of multiple sequence alignments, which are used in computational biology to search for patterns in DNA sequences.

It was surprising to see that `456.hmmer` had only 0.3% deadness in the unoptimized case, whereas it had 30% deadness in the optimized case for the `GNU` compiler (see Table 1). In fact, the absolute number of dead writes increased by 54 times from the unoptimized to the highest optimized code for the `nph3.hmm` workload. `Intel` and `PGI` compilers also showed disproportionate rise in deadness for optimized cases.

Listing 3 shows the code snippet where DeadSpy identified high-frequency dead writes for the default (`-O2`) optimized code. This code appears in a two-level nested loop, and DeadSpy reported that the write in `line 3` overwrote the write in `line 1`. In the unoptimized code, the two writes to `ic[k]` one each on `line 1` and `line 3` are separated by a read in the conditional expression on `line 2`, thus making them non dead. In the optimized code, the value of `ic[k]` computed on `line 1` is held in a register, which is reused during the comparison on `line 2`; however, the write to memory on `line 1` is not eliminated, since the compiler cannot guarantee that the arrays `ip, tpii` and `ic` do not alias each other. Thus in the optimized code, if `line 3` executes, it kills the previous write to `ic[k]` on `line 1`.

On inspecting the surrounding code, we inferred that the three pointers always point to different regions of memory and never alias each other, thus making them valid candidates for declaring as `restrict` pointers in `C` language. However, we found that the `gcc 4.1.2` compiler does not fully respect the `restrict` keyword. Hence we hand optimized the code as shown in Listing 4. The optimization improved the running time by more than 15% on average. Table 3 shows the reduction in other resources.

```
1  Bool mainGtU ( UInt32  i1, UInt32  i2, UChar*
       block, ...) {
2    Int32  k; UChar  c1, c2; UInt16 s1, s2;
3    /* 1 */
4    c1 = block[i1]; c2 = block[i2];
5    if (c1 != c2) return (c1 > c2);
6    /* 2 */
7    i1++; i2++; c1 = block[i1]; c2 = block[i2];
8    if (c1 != c2) return (c1 > c2);
9    /* 3 */
10   i1++; i2++; c1 = block[i1]; c2 = block[i2];
11   if (c1 != c2) return (c1 > c2);
12   ... 12 such checks ...
13   ... rest of the function ...
14 }
```

Listing 5: Dead writes in `bzip2` at `mainGtU()`.

```
1  leal (%r11,%rcx), %r8d   #%r8d contains i1
2  leal 1(%r8), %ebx        #compute (i1+1) in %ebx
3  leal 2(%r8), %r9d        #compute (i1+2) in %r9d
4  movq %rbx, 360(%rsp)     #spill (i1+1) to stack
5  movq %r9, 352(%rsp)      #spill (i1+2) to stack
6  #... (i1+3) to (i1+11) are computed and spilled
       ...
7  #... first check of if(c1 != c2)
8  cmpb %al, (%r15,%rdx) #if (c1 != c2)
9  ...
10 #... second check of if(c1 != c2)
11 cmpb %al, (%r15,%rdx) #if (c1 != c2)
12 ...
```

Listing 6: Hoisting and spilling in `bzip2`.

This pattern of deadness repeated several times in the same function. `Intel 11.1` compiler at its default optimization level honors the `restrict` keyword, hence we used it for comparison. We observed that the enclosing function had 13 non-aliased pointers. Declaring these 13 pointers as `restrict` improved the running time by more than 40% on average (`L1` misses, `L2` misses, instructions executed, and cycle count reduced respectively by 34%, 47%, 45%, and 40%). The Intel compiler performed dramatically better because of efficient SIMD vectorization via SSE instructions once the pointers were guaranteed to not alias each other. On disabling vectorization we observed 16% speedup; nevertheless DeadSpy *pinpointed optimization limiting code regions, indicating opportunities for performance improvement.*

5.3 Case study bzip2-1.0.6

`bzip2` is a widely used compression tool. For our experiments on `bzip2`, we used the most recent publicly available version `1.0.6` with the same workload files as SPEC CPU2006.[3]

DeadSpy reported frequent dead writes in the inlined function `mainGtU()` shown in Listing 5. In this function, 12 conditions are successive checked each of which accesses the array elements `block[i1]`$\cdots$`block[i1+11]` and `block[i2]`$\cdots$`block[i2+11]`. The function returns if any one of the checks fail. Corresponding x86 assembly in Listing 6 shows that `gcc 4.1.2` hoists the computation of indices `(i1+1)`$\cdots$`(i1+11)` ahead of the first conditional on `line 8`. `Lines 2` and `3` in Listing 6 show sample instruc-

[3]We did not use `401.bzip2` from SPEC CPU2006 since inlining is disabled in that version, presumably for code portability reasons. Enabling inlining on `401.bzip2` will produce the same issue as discussed here.

```
1  /* value unknown at compile time */
2 extern int gDisableAggressiveSched;
3 Bool mainGtU (...) {
4    ...
5    switch(gDisableAggressiveSched){
6    case 1: // always taken
7      c1 = block[i1]; c2 = block[i2];
8      if (c1 != c2) return (c1 > c2);
9    case 2: // Fall through
10     i1++; i2++; c1 = block[i1]; c2 = block[i2];
11     if (c1 != c2) return (c1 > c2);
12   case 3: // Fall through
13     i1++; i2++; c1 = block[i1]; c2 = block[i2];
14     if (c1 != c2) return (c1 > c2);
15   ... 12 such checks ...
16   } // end switch
17   ... rest of the function ...
18 }
```

Listing 7: `bzip2` modified to eliminate dead writes arising due to hoisting and spilling.

tions computing `(i1+1)` and `(i1+2)`. On the register starved x86 architecture, the precomputed values could not be kept in registers and hence they are all spilled. `Lines 4` and `5` in Listing 6 show sample spill code for `(i1+1)` and `(i1+2)`. The compute and spill pattern repeats unconditionally 12 times before the first conditional test. During execution of the `mainGtU()` function, based on the data present in `block`, often the code fails one of the early conditional tests. In this case, all of the pre-computed values and spills are unused. DeadSpy detects that these spill slots are written repeatedly and unread. Since `mainGtU()` happens to be at the heart of `bzip2`'s compute kernel, the effect of hoisting index computations into the hot code region has a negative impact on performance. This exposes *the lack of cooperation between the instruction scheduling and register allocation phases of* `gcc`, which results in poor generated code.

To suppress `gcc`'s over-aggressive scheduling and register spilling, we overlaid a switch statement over the control flow—a technique analogous to Duff's device [10]. Listing 7 shows restructured code where the value of `gDisableAggressiveSched` is always `1` at runtime making the `case 1` arm of the `switch` statement to be the always taken branch.

Table 3 shows the improvements obtained using our workaround. `bzip2` shows an average speedup of 7.2% with the maximum of 14.2% for the `input.program` workload. While the cache misses remained almost the same before and after the fix, the operations performed and the cycle counts reduced by a proportion commensurate with the observed performance gains. This is justifiable since the code changes eliminated dead writes to on-stack temporaries which are typically cached. Execution time was improved by avoiding the unnecessary index computation and spilling.

5.4 Case study Chombo's amrGodunov3d

We ran DeadSpy on `amrGodunov3d`, a standard benchmark program that uses `Chombo` [3], which is a framework for solving partial differential equations in parallel using block-structured, adaptively refined grids. For detecting node-level inefficiency, we ran `amrGodunov3d` on a single node[4] and detected 34% deadness.

[4]We used Intel Xeon E5530 processor and Intel compiler tools version 12.0.0.

```
1 Wgdnv(i,j,k,0) = x
2 if (spout.le.(0.0d0)) then
3   Wgdnv(i,j,k,0) = y
4 endif
5 if (spin.gt.(0.0d0)) then
6   Wgdnv(i,j,k,0)= z
7 endif
```

Listing 8: Dead writes in a `Chombo` Riemann solver.

```
1 if (spin.gt.(0.0d0)) then
2   Wgdnv(i,j,k,0)= z
3 else if (spout.le.(0.0d0)) then
4   Wgdnv(i,j,k,0) = y
5 else
6   Wgdnv(i,j,k,0) = x
7 endif
```

Listing 9: Avoiding dead writes in Riemann solver.

The Chombo framework lacks *design for performance*; it is a hybrid code which has a `C++` driver with computational kernel written in `Fortran`. We discuss two high frequency dead write scenarios where developer did not pay enough attention to performance while designing the framework.

First, the code snippet shown in Listing 8 appears in a 3-level nested loop of the computationally intensive `Riemann solver` kernel which works on a 4D array of 8 byte `real` numbers. DeadSpy reported that the write in `line 1` was killed by the write in `line 3` as well as the write in `line 6`; and the write in `line 3` was killed by the write in `line 6`. To fix the problem, we applied a trivial code restructuring by using `else if` nesting as shown in Listing 9.

Second, the code snippet shown in Listing 10 appears on a hot path with 2-level nested loop. The call to construct `FArrayBox` objects — `WTempMinus` and `WTempPlus`, on `lines 2, 3` respectively, zero initialize a 4D-box data structure member in `FArrayBox`. The call to the `copy()` member function on `lines 6,7` fully overwrites the previously initialized 4D-box data structures with new values. Similarly, 4D-box `AdWdx` constructed and zero initialized in `line 4` is fully overwritten inside the function `quasilinearUpdate()` called from `line 8`. These dead writes together contribute to a deadness of 20% in the program. The dead writes result from *using a non specialized constructor*. We remedied the problem by overloading the constructor with a specialized leaner version which did not initialize the 4D box inside `FArrayBox`.

In `Chombo`, the dead writes due to initialization followed by the overwrite pattern was pervasive. In fact, we observed this pattern to be pervasive in SPEC CPU2006 benchmarks, we omit the details for brevity. By using slimmer versions of constructors in five more similar contexts, we were able to improve `Chombo`'s running time by 6.6%. Table 3 shows reductions in cache misses and total instructions as well.

6. RELATED WORK

Several compiler optimizations focus on reducing avoidable operations. In some way, every execution speed related optimization tries to reduce operation count and/or memory accesses. An exhaustive review of compiler optimizations is beyond the scope of this paper. We highlight some prior art

```
1 // Temporary primitive variables
2 FArrayBox WTempMinus(WMinus[dir1].box() ,...);
3 FArrayBox WTempPlus(WPlus [dir1].box() ,...);
4 FArrayBox AdWdx(WPlus[dir1].box() ,...);
5 // Copy data for in place modification
6 WTempMinus.copy(WMinus[dir1]);
7 WTempPlus.copy(WPlus [dir1]);
8 m_gdnvPhysics->quasilinearUpdate(AdWdx,...);
```

Listing 10: `Chombo` dead writes in C++ constructors.

related to our work.

Butts et al. [6] propose a hardware-based approach to detect useless operations, and speculatively eliminate instructions from future executions by maintaining a history of instructions' uselessness. In SPEC CPU2000 integer benchmarks, they found on average 8.85% useless operations, and their technique shows an average speedup of 3.6%. Their work focuses on identifying and eliminating useless computations only; useless memory operations are never eliminated since mis-prediction can lead to the violation of memory consistency. Our approach is orthogonal to this, we do not track CPU bound computations, instead we track reads and writes only. The authors do not mention if they can provide any actionable feedback on the locations of useless computations, whereas DeadSpy provides full context to take informed action.

Archambault [4] discusses an inter-procedural data-flow analysis technique to eliminate dead writes. In their patent, they suggest to merge liveness information for global variables from basic blocks to create a set of *live on exit (LOE)* data structure for procedures. They traverse the program call graph in depth first order, propagating LOE through function calls and returns. However, they do not discuss the effectiveness of their technique. Being a static analysis technique, we suspect it has limitations when dealing with aggregate types, dynamic loading and aliasing. In contrast, DeadSpy, which uses a dynamic technique, cannot distinguish between *dead for this program input* and *live for some other input*.

Gupta et al. [13] have proposed "predication-based sinking" — a cost-benefit based algorithm to move partially dead code from hot paths to infrequently executed regions. With profiling information, this technique can eliminate the intra-procedural dead write presented in the `Chombo` case study.

The shadow memory technique, presented in our implementation, is used in several well known tools such as *Eraser* [25] for data race detection, *TaintCheck* [24] for taint analysis, and *Memcheck* [28] for dangerous memory use detection. Nethercote et al. [23] present techniques for efficiently shadowing every byte of memory. Like their implementation, our shadow memory implementation also optimizes for common cases.

7. CONCLUSIONS

This paper presents DeadSpy — a tool to detect dead writes in a program. Using DeadSpy, we showed that several commonly used programs have surprisingly large fractions of dead writes, with an average of more than 20% in the SPEC CPU2006 integer benchmarks, with a maximum of 76% for `403.gcc` using a particular input. Deadness occurs due to various factors such as poor choice of data structures, lack of design for performance, and ineffective compiler optimizations. Use of appropriate data-structures, light-weight abstractions, and improved cooperation between phases of compiler optimizations can boost performance significantly. Simple code restructuring to eliminate dead writes improved performance of `gcc`, `hmmer` and `bzip` on average by 14.3%, 15.7%, and 7.2% respectively.

The pervasiveness of dead writes suggests a new opportunity for performance tuning. We recommend investigating and eliminating avoidable memory operations as an important step in performance tuning to identify opportunities for code restructuring. DeadSpy's ability to detect and pinpoint sources of inefficiencies makes it a domain expert's companion tool for performance enhancement. While DeadSpy has high runtime overhead, executing programs with small representative workloads will help uncover regions of inefficiencies quickly. In our future work, we would like reduce DeadSpy's overhead by avoiding unnecessary instrumentation using binary analysis.

8. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful feedback. We also thank Karthik Murthy, Keith Cooper, and Tim Harvey at Rice University, along with Nathan Tallent at PNNL for their invaluable inputs. This work is funded by the Defense Advanced Research Projects Agency through AFRL Contract FA8650-09-C-7915. The opinions and findings in this document do not necessarily reflect the views of either the United States Government or Rice University.

9. REFERENCES

[1] 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 and Experience*, 22:685–701, April 2010.

[2] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI '97, pages 85–96, New York, NY, USA, 1997. ACM.

[3] Applied Numerical Algorithms Group, Lawrence Berkeley National Laboratory. Chombo website. `https://seesar.lbl.gov/anag/chombo`.

[4] R. G. Archambault. Interprocedural dead store elimination. Patent, 08 2006. US 7100156.

[5] R. Bodík and R. Gupta. Partial dead code elimination using slicing transformations. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI '97, pages 159–170, New York, NY, USA, 1997. ACM.

[6] J. A. Butts and G. Sohi. Dynamic dead-instruction detection and elimination. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-X, pages 199–210, New York, NY, USA, 2002. ACM.

[7] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 53–65, New York, NY, USA, 1990. ACM.

[8] G. J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, pages 98–105, New York, NY, USA, 1982. ACM.

[9] 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.

[10] T. Duff. Tom Duff on Duff's Device. `http://www.lysator.liu.se/c/duffs-device.html`.

[11] GCC Team. The GNU Compiler Collection. `http://gcc.gnu.org/`.

[12] 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.

[13] R. Gupta, D. A. Berson, and J. Z. Fang. Path profile guided partial dead code elimination using predication. In *Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques*, PACT '97, pages 102–, Washington, DC, USA, 1997. IEEE Computer Society.

[14] P. Hicks, M. Walnock, and R. M. Owens. Analysis of power consumption in memory hierarchies. In *Proceedings of the 1997 International Symposium on Low Power Electronics and Design*, ISLPED '97, pages 239–242, New York, NY, USA, 1997. ACM.

[15] Intel Corporation. Intel Compilers. `http://software.intel.com/en-us/articles/intel-compilers/`.

[16] Intel Corporation. Intel VTune performance analyzer. `http://software.intel.com/en-us/forums/intel-vtune-performance-analyzer/`.

[17] B. L. Jacob. *The Memory system: You can't avoid it, you can't ignore it, you can't fake it.* Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2009.

[18] H. Jin, M. Frumkin, and J. Yan. The OpenMP implementation of NAS parallel benchmarks and its performance. NAS Technical Report NAS-99-011, NASA Advanced Supercomputing Division, 1999.

[19] J. Knoop, O. Rüthing, and B. Steffen. Partial dead code elimination. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 147–158, New York, NY, USA, 1994. ACM.

[20] 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.

[21] S. A. McKee. Reflections on the memory wall. In *Proceedings of the 1^{st} Conference on Computing Frontiers*, CF '04, pages 162–167, New York, NY, USA, 2004. ACM.

[22] Microsoft Corporation. Windows Performance Analysis Tools. Windows Performance Analysis Developer Center. `http://msdn.microsoft.com/en-us/performance/cc825801`.

[23] N. Nethercote and J. Seward. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 65–74, New York, NY, USA, 2007. ACM.

[24] J. Newsome and D. X. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, USA, 2005. The Internet Society.

[25] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15:391–411, November 1997.

[26] B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited. In *Proceedings of the Ninth Working Conference on Reverse Rngineering (WCRE'02)*, pages 45–54, Washington, DC, USA, 2002. IEEE Computer Society.

[27] J. Seward. bzip2. `http://www.bzip.org`.

[28] J. Seward and N. Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 2–2, Berkeley, CA, USA, 2005. USENIX Association.

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

[30] The Portland Group. PGI Optimizing Fortran, C and C++ Compilers and Tools. `http://www.pgroup.com/`.