Spindle: Informed Memory Access Monitoring

Haojie Wang, *Tsinghua University, Qatar Computing Research Institute;*
Jidong Zhai, *Tsinghua University;* Xiongchao Tang, *Tsinghua University, Qatar Computing Research Institute;* Bowen Yu, *Tsinghua University;* Xiaosong Ma, *Qatar Computing Research Institute;* Wenguang Chen, *Tsinghua University*

https://www.usenix.org/conference/atc18/presentation/wang-haojie

This paper is included in the Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18).

July 11–13, 2018 • Boston, MA, USA

ISBN 978-1-939133-02-1

Spindle: Informed Memory Access Monitoring

Haojie Wang[*†], Jidong Zhai[*], Xiongchao Tang[*†], Bowen Yu[*], Xiaosong Ma[†], Wenguang Chen[*]

Abstract

Memory monitoring is of critical use in understanding applications and evaluating systems. Due to the dynamic nature in programs' memory accesses, common practice today leaves large amounts of address examination and data recording at runtime, at the cost of substantial performance overhead (and large storage time/space consumption if memory traces are collected).

Recognizing the memory access patterns available at compile time and redundancy in runtime checks, we propose a novel memory access monitoring and analysis framework, Spindle. Unlike methods delaying all checks to runtime or performing task-specific optimization at compile time, Spindle performs *common static analysis* to identify predictable memory access patterns into a compact program structure summary. Custom memory monitoring tools can then be developed on top of Spindle, leveraging the structural information extracted to dramatically reduce the amount of instrumentation that incurs heavy runtime memory address examination or recording. We implement Spindle in the popular LLVM compiler, supporting both single-thread and multi-threaded programs. Our evaluation demonstrated the effectiveness of two Spindle-based tools, performing memory bug detection and trace collection respectively, with a variety of programs. Results show that these tools are able to aggressively prune online memory monitoring processing, fulfilling desired tasks with performance overhead significantly reduced (2.54× on average for memory bug detection and over 200× on average for access tracing, over state-of-the-art solutions).

1 Introduction

Memory access behavior is crucial to understand applications and evaluate systems. They are widely monitored in system and architecture research, for memory bug or race condition detection [21, 27, 31], information flow tracking [16, 30], large-scale system optimization [35, 36, 42], and memory system design [14, 17, 20].

Memory access monitoring and tracing need to obtain and check/record memory addresses visited by a program and this process is quite expensive. Even given complete source-level information, much of the relevant information regarding locations to be accessed at runtime is not available at compile time. For example, it is common that during static analysis, we see a heap object accessed repeately in a loop, but does not have any of the parameters needed to perform our desired examination or tracing: where the object is allocated, how large it is, or how many iterations there are in a particular execution of the loop. As a result, existing memory checking tools mostly delay the checking/transcribing of such memory addresses to execution time, with associated instructions instrumented to perform task-specific processing. Such runtime processing brings substantial performance overhead (typically bringing 2× or more application slowdown [5, 33] for online memory access checking and much higher for memory trace collection [6, 22, 26]).

However, there are important information not well utilized at compile time. Even with actual locations, sizes, branch taken decisions, or loop iteration counts unknown, we still see patterns in memory accesses. In particular, accesses to large objects are not isolated events that have to be verified or recorded individually at runtime. Instead, they form groups with highly similar (often identical) behaviors and *relative* displacement in locations visited given plainly in the code. The processing tasks that are delayed to execution time often perform the same checking or recording on individual members of such large groups of highly homogeneous accesses. In addition, the memory access patterns recognizable during static analysis summarize *common structural information* useful to many memory checking/tracing tasks.

Based on these observations, we propose Spindle,

[*]Department of Computer Science and Technology, Tsinghua University {wanghaoj15, txc13, yubw15}@mails.tsinghua.edu.cn, {zhaijidong, cwg}@mail.tsinghua.edu.cn
[†]Qatar Computing Research Institute, HBKU {whaojie, txiongchao, xma}@qf.org.qa

a new platform that facilitates hybrid static+dynamic analysis for efficient memory monitoring. It leverages common static analysis to identify from the target program the source of redundancy in runtime memory address examination. By summarizing groups of memory accesses with statically identified program structures, such compact intermediate analysis results can be passed to Spindle-based tools, to further perform task-specific analysis and code instrumentation. The regular/predictable patterns contained in Spindle-distilled structural information allow diverse types of memory access checking more efficiently: by ***computing*** rather than ***collecting*** memory accesses whenever possible, even when certain examination has to be conducted at runtime, it can be elevated from instruction to object granularity, with the amount of instrumentation dramatically pruned.

We implement Spindle on top of the open-source LLVM compiler infrastructure [10]. On top of it, we implement two proof-of-concept custom tools, a memory bug detector (S-Detector) and a memory trace collector (S-Tracer), that leverage the common structural information extracted by Spindle to optimize their specific memory access monitoring tasks.

We evaluated Spindle and the aforementioned custom tools with popular benchmarks (NPB, SPEC CPU2006, Graph500, and PARSEC) and open-source applications covering areas such as machine learning, key-value store, and text processing. Results show that S-Detector can reduce the amount of instrumentation by 64% on average using Spindle static analysis results, allowing runtime overhead reduction of up to 30.25× (2.54× on average) over the Google AddressSanitizer [33]. S-Tracer, meanwhile, reduces the trace collection time overhead by up to over 500× (228× on average) over the polular PIN tool [22], and cuts the trace storage space overhead by up to over 10000× (248× on average).

Spindle is publicly available at https://github.com/thu-pacman/Spindle.

2 Overview

2.1 Spindle Framework

Spindle is designed as a hybrid memory monitoring framework. Its main module performs static analysis to extract program structures relevant to memory accesses. Such structural information allows Spindle to obtain regular or predictable patterns in memory accesses. Different Spindle-based tools utilize these patterns in different ways, with the common goal of reducing the amount of instrumentation that leads to costly runtime check or information collection.

Figure 1 gives the overall structure of Spindle, along with sample memory monitoring tools implemented on top of it. To use Spindle-based tools, end-users only have to compile their application source code with the

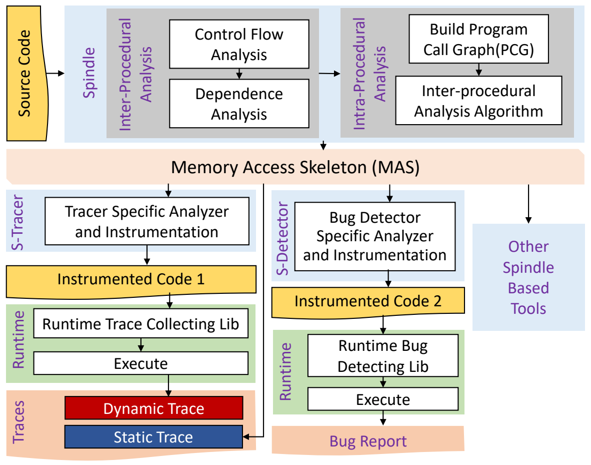

Source Code
Spindle
Inter-Procedural Analysis
Control Flow Analysis
Dependence Analysis
Intra-Procedural Analysis
Build Program Call Graph(PCG)
Inter-procedural Analysis Algorithm
Memory Access Skeleton (MAS)
S-Tracer
Tracer Specific Analyzer and Instrumentation
Instrumented Code 1
Runtime
Runtime Trace Collecting Lib
Execute
Traces
Dynamic Trace
Static Trace
S-Detector
Bug Detector Specific Analyzer and Instrumentation
Instrumented Code 2
Runtime
Runtime Bug Detecting Lib
Execute
Bug Report
Other Spindle Based Tools

Figure 1: Spindle overview

Spindle-enhanced LLVM modules, whose output then goes through tool-specific analysis and instrumentation. More specifically, the common static analysis performed by Spindle will generate a highly compact *Memory Access Skeleton (MAS)*, describing the structured, predictable memory access components.

Spindle tool developers write their own analyzer, which uses MAS to optimize their code instrumentation, aggressively pruning unnecessary or redundant runtime checks or monitoring data collection. In general, such task-specific tools enable *computing* groups of memory addresses visited before or after program executions, to avoid *examining* individual memory accesses at runtime. As illustrated in Figure 1, each of such Spindle-based tools (the memory bug detector S-Detector and memory trace collector S-Tracer in this case) will generate its own instrumented application code. As our results will show, for typical applications, the majority of memory accesses are computable given a small amount of runtime information, leading to dramatic reduction of instrumentation and runtime collection.

End-users then execute their tool-instrumented applications, with again task-specific runtime libraries linked. The instrumented code conducts runtime processing to perform the desired form of memory access monitoring, such as bug or race condition detection, security check, or memory trace collection. The runtime libraries capture dynamic information to fill in parameters (such as the starting address of an array or the actual iteration count of a loop) to instantiate the Spindle MAS and complete the memory monitoring tasks. In addition, all the "unpredictable" memory access components, identified by Spindle at compile time as input-dependent, are monitored/recorded in the traditional manner.

Spindle's static analysis workflow to produce MAS is further divided into multiple stages, performing intra-procedural analysis, inter-procedural analysis, as well as tool specific analysis and instrumentation. During the intra-procedural stage, Spindle analyzes the program control flow graph and finds out the dependence among memory access instructions. The dependence checking is then expanded across functions in inter-procedural analysis.

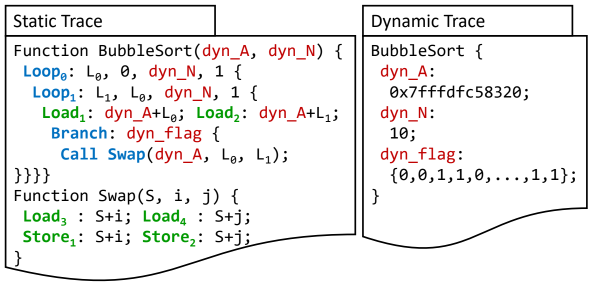
One limitation of the current Spindle framework is that it requires source level information of target programs. As this work is a proof-of-concept study, also considering the current trend of open-source software adoption [9, 41], our evaluation uses applications with source code available. Spindle can potentially work without source code though: it starts with LLVM IR and can therefore employ open-source tools such as Fcd [7] or McSema [37] to translate binary codes into IR. In our future work we are however more interested in direct static analysis, performing tasks such as loop and dependency detection on binaries.

2.2 Sample Input/Output: Memory Trace Collector

```
void BubbleSort(int *A, int N){
    for (int i = 0; i < N; ++i){
        for (int j = i+1; j < N; ++j){
            bool flag = (A[i] > A[j]);
            if (flag) {
                Swap(A, i, j);
}}}}

void Swap(int *S, int i, int j) {
    int tmp = S[i];
    S[i] = S[j];
    S[j] = tmp;
}
```

Figure 2: Sample bubble sort program


Static Trace

```
Function BubbleSort(dyn_A, dyn_N) {
 Loop_0: L_0, 0, dyn_N, 1 {
  Loop_1: L_1, L_0, dyn_N, 1 {
   Load_1: dyn_A+L_0; Load_2: dyn_A+L_1;
    Branch: dyn_flag {
     Call Swap(dyn_A, L_0, L_1);
}}}}
Function Swap(S, i, j) {
 Load_3 : S+i; Load_4 : S+j;
 Store_1: S+i; Store_2: S+j;
}
```

Dynamic Trace

```
BubbleSort {
 dyn_A:
  0x7fffdfc58320;
 dyn_N:
  10;
 dyn_flag:
  {0,0,1,1,0,...,1,1};
}
```

Figure 3: Memory traces of the bubble sort program

We take S-Tracer, our Spindle-based trace collector, as an example to give a more concrete picture of Spindle's working. Suppose the application to be monitored is the bubble sort program listed in Figure 2. S-Tracer's output, given in Figure 3, is a complete yet compressed memory access trace, consisting of its MAS (coupled with corresponding dynamic parameters) and dynamic traces collected in the conventional manner.

In the static trace, we list out the structure of the program, including the control flow, the memory accesses pattern and the call graph. There are information items that cannot be determined during static analysis, such as the base address of array A and its size N, which is also the final value of loop induction variables *i* and *j* , as well as the value of flag, which is data-dependent and determines the control flow of this program. The "Instrumented code 1" shown in Figure 1 records these missing values at executing time, which compose the dynamic trace shown on the right.

This new trace format, though slightly more complex than traditionally traces, is often orders of magnitude smaller. A straightforward post-processor can easily take S-Tracer traces and restore the traditional full traces. More practically, an S-Tracer trace driver performing similar decompression can be prepended to typical memory trace consumers, to enable fast replay without involving large trace files or slow I/O.

3 Static Analysis

3.1 Intra-procedural Analysis

During this first step, Spindle extracts a program's per-function control structure to identify memory accesses whose traces can be computed and hence can be (mostly) skipped in dynamic instrumentation.

3.1.1 Extracting Program Control Structure

A program's memory access patterns (or the lack thereof) are closely associated to its control flows. It is not surprising that it shares a similar structure with the program's control flow graph (CFG). Therefore we call this graph M-CFG. Unlike traditional control flow graphs, M-CFG records only instructions containing memory references (rather than the entire basic block), program control structures (loops and branches), and function calls. For loops and branches, we need to record related variables, such as loop boundaries and branch conditions.

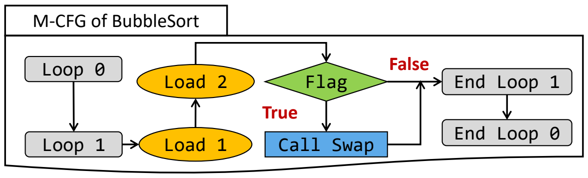


Figure 4: The M-CFG for the function BubbleSort

With M-CFG, memory access instructions are embedded within program basic control structures, as illustrated in Figure 4 for the aforementioned BubbleSort function (Figure 2). Here the M-CFG records a nested loop containing two memory accesses and a branch with a function call. Subsection 3.1.2 discusses dependence analysis regarding memory access instructions and identification of computable memory accesses, while Section 3.2 discusses as handling of function calls.

3.1.2 Building Memory Dependence Trees

In Spindle, we classify all memory accesses into either *computable* or *non-computable* types. The computable accesses can have traces computed based on the static trace, with the help of little or no dynamic information; the non-computable ones, on the other hand, need to fall back to traditional instrumentation and runtime tracing.

For such classification, we build a *memory dependence tree* for each memory access instruction. It records data dependence between a specific memory access instruction and its related variables. The tree is rooted at the memory address accessed, with non-leaf nodes denoting operators between variables such as addition or multiplication and leaf nodes denoting variables in the program. Edges hence intuitively denote dependence.

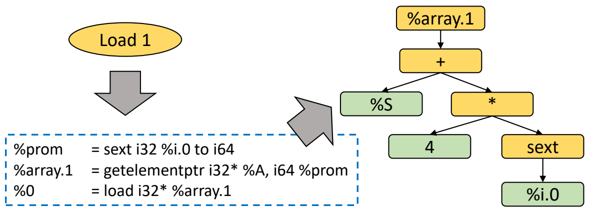
Below we list the types of leaf nodes in memory dependence trees:

- *Constant value*: value determined at compile time
- *Base memory address*: start address for continuously allocated memory region (such as an array), with value acquired at compile time for global or static variables, and at runtime for dynamically allocated variables.
- *Function parameter*: value determined at either compile time or runtime (see Section 3.2)
- *Data-dependent variable*: value dependent on data not predictable at compile time – to be collected at runtime
- *Function return value*: value collected at runtime
- *Loop induction variable*: variable regularly updated at each loop iteration, value determined at compile time or runtime

Algorithm 1 Algorithm of building memory dependence tree

```
1: input: A worklist $WL[A]$. Predefined Leaf types: $Type$
2: output: memory dependence tree: $T(A)$
3: Insert a root note $r$ to $T(A)$
4: while $WL[A] \neq \phi$ do
5:     Remove an item $v_1$ from $WL[A]$
6:     if $v_1 \notin Type$ then
7:         for $v_2 \in UD(v_1)$ do
8:             if $v_2 \in Type$ then
9:                 Insert a leaf node $v_2$
10:                Insert an edge from $v_1$ to $v_2$
11:            else
12:                Insert an operator node in $v_2$ to $T(A)$
13:                Add all variables used in $v_2$ to $WL[A]$
14:    else
15:        Insert a leaf node $v_1$ to $v_1$ to $T(A)$
16:        Insert an edge from $r$ to $v_1$ to $T(A)$
17: return $T(A)$
```

The memory dependence tree is built by performing a backward data flow analysis at compile time. Specifically, for each memory access, we start from the variable storing this memory address and traverse its use-define data structure, which describes the relation between the definition and use of each variable, to identify all the variables and operators affecting it. This traversal is an iterative process that stops when all the leaf nodes are categorized into one of the types listed above. We give the worklist algorithm (Algorithm 1) that performs such backward data flow analysis with, where we repeatedly variables storing memory addresses into the worklist $WL(A)$ and iteratively find all the related variables through the use-define structure $UD(v)$, till the worklist becomes empty.



```
%prom     = sext i32 %i.0 to i64
%array.1  = getelementptr i32* %A, i64 %prom
%0        = load i32* %array.1
```

Figure 5: Sample memory dependence tree

Figure 5 shows a group of instructions (generated from the source code in Figure 2) and the memory dependence tree corresponding to the variable `%array.1` in the last line. Here `getelementptr` is an instruction that calculates the address of an aggregate data structure (where an addition operation is implied) and does not access memory. We omit certain arguments for this instruction for simplicity. `sext` performs type casting. As to the leaf nodes, `%A` is an array base address, `4` is a constant value, and `%i.0` is a loop induction variable.

Such a dependence tree allows us to approach the central task of Spindle: ***computable memory access identification***. This is done by analyzing the types of the leaf nodes in the memory dependence tree. Intuitively, a memory access is computable if the leaf nodes of its dependence tree are either constants (trivial) or loop induction variables (computable by replicating computation performed in the original program using initial plus final values, collected at compile time or runtime). The M-CFG and the memory access dependence trees, preserving control flows, data dependencies, and operations to facilitate such replication, can be viewed as a form of *program pruning* that only retains computation relevant to memory address calculation. By replacing each memory instruction of the M-CFG with its dependence tree, we obtain a single graph representing main memory access patterns for a single function. Note that such dependence analysis naturally handles aliases.

3.2 Inter-procedural Analysis

At the end of the intra-procedural analysis, we have a memory dependence tree for every memory access within each function. Below we describe how Spindle analyzes memory address dependence across functions.

The core idea here is to propagate function arguments *plus their dependence* from the caller to the callee, and replace all the function parameters of the dependence trees in the callee with actual parameters. For this, we first build a program call graph (PCG), on which we subsequently perform top-down inter-procedural analysis. Algorithm 2 gives the detailed process.

Algorithm 2 The algorithm of inter-procedural analysis

```
 1: input: The dependence trees for each procedure p
 2: input: The program call graph (PCG)
 3: Change ← True
 4: /* Top-Down inter-procedural analysis */
 5: while (Change == True) do
 6:     Change ← False
 7:     for all procedure p in Pre-Order over PCG do
 8:         for all dependence trees d in p do
 9:             if A leaf node l of d is a function's parameter then
10:                 Replace l with its actual parameter
11:                 Change ← True
```

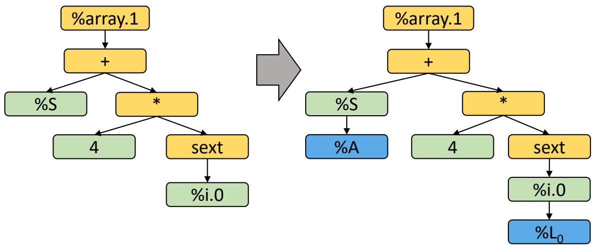


Figure 6: Transformation of dependence tree

Figure 6 illustrates the transformation a dependence tree in function `Swap` (Figure 2) undergoes during inter-procedural analysis. After intra-procedural analysis, the dependence tree for the load instruction `Load3` of function `Swap` has two leaf nodes that are function parameters, which cannot be analyzed then as the variables `%S` and `%i.0` are undetermined. Within inter-procedural analysis, these two nodes are replaced with their actual parameters, a base address `%A` and a loop induction variable `%i.0` Now the dependence tree rooted at `%array.1` is computable.

For function calls forming a loop in PCG, such as recursive calls, currently we do not perform parameter replacement for any function in this loop during our inter-procedural analysis, as when these functions terminate is typically data-dependent.

3.3 Special Cases and Complications

Index arrays If a memory dependence tree has data-dependent variables as its leaf nodes, normally we consider it non-computable. However, we still have chance to extract regular patterns. Index array is an important case of such data-dependent variables, storing "links" to other data structures, as explained below.

```
for (j=0; j<i; j++){
    for (k=0; k<m; k++)
        sum += delta * z[colidx[k]]
        //colidx is index array to z
    r[k] = d
}
```

Figure 7: NPB CG code with index array `colidx`

Figure 7 gives a simplified version of a code snippet from NPB CG [2], where the array `z` is repeatedly accessed via the index array `colidx`, which cannot be determined at compile time. However, we find that in many programs (including here) the index array itself is not modified across multiple iterations of accesses. Therefore, there is still significant room for finding repeated access patterns and removing redundancy.

To this end, Spindle performs the following extra evaluation during its static analysis. First, it compares the size of index array and its total access count. If the latter is larger, we only need to record the content of the index array and *compute* the memory accesses accordingly rather than instrumenting them at runtime. Such evaluation needs to be repeated if the content of this index array is changed, of course. This is the case with the example given in Figure 7, where the total memory access count for the index array `colidx` is `i*m` and greater than the size of `colidx`. Thus at runtime we only need to record its content at the beginning of this nested loop and the base address of array `z`. Combining such information and memory dependence tree, we can compute all the memory access locations.

Multi-threaded programs The discussion so far has been focused on analyzing single-thread programs. However, Spindle's methodology can also be easily applied to multi-threaded applications. Spindle is thread-safe and we perform the same static analysis as for single-thread programs, except that we also mark the point where a new thread is created and record relevant parameter values. With parallel executions, during dynamic memory monitoring (discussed in the next section), the current thread ID would be easily fetched along with information such as loop iteration count and branch taken, which allows us to distinguish runtime information collected by different threads. Note that certain techniques need to be augmented to handle multi-threaded executions. E.g., the array index technique (Section 3.3) needs to be protected by additional check, as an array could be modified by another thread.

Again, with addresses or values that cannot be determined at compile time, such as shared objects or branches affected by other threads, we fall back to runtime instrumentation. So typical SPMD codes will share the same static MAS, to be supplemented by per-thread or even per-process runtime information, making Spindle even more appealing in efficiency and scalability. If significant amount of output is generated, such as with memory trace collection, Spindle allows users to have the option to look at a single-thread's memory accesses or correlating accesses from all threads (though trace interleaving is a separate topic that requires further study.)

For example, with *pthread*, Spindle instruments `pthread_create` to record where a new thread is created. During multi-threaded execution, the appropriate thread ID is recorded for each function. Thus we know which thread the dynamic information collected by Spindle belongs to, therefore can apply per-thread static analysis, similar to that in single-thread executions.

4 Spindle-based Runtime Monitoring

This section illustrates how Spindle's static analysis results can be used to reduce runtime instrumentation. We first describe common runtime information to be obtained through instrumentation, then present two samples of Spindle-based tool design, for memory bug detection and memory trace collection, respectively.

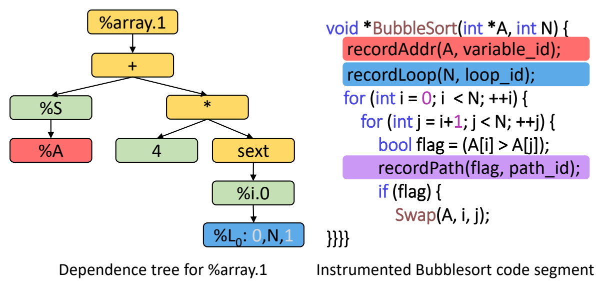
4.1 Runtime information collection

During program runs, Spindle's static memory access skeleton is supplemented by information not available at compile time. Generally, three cases require instrumentation: control structures, input-dependent variables, and non-computable memory accesses:

Control structures Spindle needs to record the initial values of all the loop induction variables and the loop iteration count if they are unknown at compilation time. Moreover, for a loop with multiple exit points, we need to instrument each exit point to track where the loop exits. Similarly, for conditional branches in MAS, we need to record their taken statuses to track taken paths.

Input dependent variables For input dependent variables, runtime information is necessary but certain static analysis can indeed reduce runtime overhead. For instance, the address of a dynamically allocated memory region can be obtained at runtime by collecting actual values. An optimization in Spindle is that we do not instrument every instruction that references input dependent variables, but only where they are defined, initialized, or updated. E.g., for a global variable needed by the analysis, it leverages static analysis to only record its initial value at the beginning of the program, and then again upon its updates.

Non-computable memory accesses For non-computable memory accesses (as mentioned in Subsection 3.1.2), we fall back to conventional dynamic monitoring/instrumentation.


%array.1
+
%S
*
%A
4
sext
%i.0
%L$_0$: 0,N,1
Dependence tree for %array.1
Instrumented Bubblesort code segment

```
void *BubbleSort(int *A, int N) {
  recordAddr(A, variable_id);
  recordLoop(N, loop_id);
  for (int i = 0; i  < N; ++i) {
    for (int j = i+1; j < N; ++j) {
      bool flag = (A[i] > A[j]);
      recordPath(flag, path_id);
      if (flag) {
        Swap(A, i, j);
}}}}
```

Figure 8: Sample runtime information collection

Figure 8 shows an example of runtime information collection for the `BubbleSort` routine discussed earlier in Section 2.2. The left side gives the dependence tree of the variable `%array.1` in function `Swap`, where undetermined address `%A` and loop L_0's induction variable `%N` need to be collected at runtime. Note that L_0's initial index value (0) and increment (1) can be determined at compile time. The right side lists the instrumented `BubbleSort` code. Here Spindle automatically instruments three memory accesses by inserting the highlighted statements (for `A`, `N`, and the branch related `flag`, which falls out of the dependence tree shown). `variable_id`, `loop_id`, and `path_id` are also automatically generated by Spindle for its runtime library to find the appropriate static structures.

4.2 Spindle-based tool developing

Spindle's performs automatic code instrumentation for runtime information collection, based on its static analysis. To build a memory monitoring tool on top of Spindle, users only need to supply additional codes using its API to perform custom analysis, as to be illustrated below. Our two sample tools, S-Detector and S-Tracer, each takes under 500 lines of code to implement both compile-time analysis and runtime library.

4.2.1 Memory Bug Detector (S-Detector)

Memory bugs, such as buffer overflow, use after free, and use before initialization, may cause severe runtime errors or failures, especially with programming languages like C and C++. There have been a series of tools, software- or hardware-based, developed to detect memory bugs at compile-time or runtime. Among them, Memchecker [39] uses hardware support for memory access monitoring and debugging and is therefore fast (only 2.7% performance overhead for SPECCPU 2000). Such special-purpose hardware is nevertheless not yet adopted by general processors. ARCHER [43] relies on static analysis only, so is faced with the difficult trade-off between accuracy (false positives) and soundness (false negatives), like other static tools. A recent, state-of-the-art tool is AddressSanitizer (ASan) [33], an industrial-strength memory bug detection tool developed by Google and now built into the LLVM compiler. ASan inserts memory checking instructions (such as out-of-bound array accesses) into programs at compile time, then uses shadow memory [25] for fast runtime checking. Despite well implemented and highly tuned, ASan still introduces 2–3$\times$ slowdown to SPEC programs.

In this work, we present S-Detector, a memory bug detector that leverages Spindle-gathered static information to eliminate unnecessary instrumentation to facilitate efficient online memory checking. Our proof-of-concept implementation of S-Detector can currently detect invalid accesses (e.g., out-of-bound array access and use after free) and memory leaks (dynamically allocated objects remaining unfreed upon program termination).

With Spindle's MAS, S-Detector is aware of a program's groups of memory accesses and therefore able to perform checking at a coarser granule. E.g., with dynamically allocated arrays, even when neither the starting address (*base*) or size (*bound*) is known at compile time, its accesses are given as relative to these two values and can therefore be checked for out-of-bound bugs at compile time. With existing tools like ASan, however, such checks are delayed till runtime and repeated at every memory acesses.

Therefore, S-Detector performs aggressive *memory check pruning* by proactively conducting compile-time access analysis and replacing instruction-level checks by object-level ones. Only for accesses labeled "non-computable" by Spindle, S-Detector falls back to traditional instrumentation. Below, we illustrate S-Detector's memory check pruning with two sample scenarios, both contained in the same code snippet from SPEC CPU2006 `mcf` (Figure 9).

```
1  while (pos - 1 && red_cost >
2             (cost_t)new[pos/2-1].flow){
3      new[pos-1].tail = new[pos/2-1].tail;
4      new[pos-1].head = new[pos/2-1].head;
5      // Three more accesses to struct members
6      // of new[pos-1] and new[pos/2-1].
7      pos = pos/2;
8      new[pos-1].tail = tail;
9      // Four more accesses to struct members
10     // of new[pos-1].
11 }
```

Figure 9: Sample code from SPEC CPU 2006 `mcf`

In-structure accesses This sample code references an array of structures (`new`), issuing multiple accesses to members of its elements. In this case, assisted with Spindle-extracted MAS, all access targets can be represented as `addr = struct_base + constant_offset`. Once S-Detector finds that the constant `offset` is valid for this struct, i.e., `offset<struct_size`, it only needs to determine if this structure element itself is valid at runtime, i.e., the memory range `[struct_base, struct_base + struct_size)` is a valid range. This groups the per-member access checks to per-element checks (validating structure elements like `new[pos-1]` and `new[pox/2-1]`) and significantly reduces the amount of instrumentation.

In-loop accesses Given the `while` loop in the same sample code, Spindle records the following information for its loop induction variable `pos`: its initial and final values (denoted here as `pos_init` and `pos_final`), as well as the operation used to update it across iterations (divided by 2 at Line 7). Based on the MAS, S-Detector can easily infer the offset range of array `new`'s access to be within [`pos_end/2-1`, `pos_init-1`]. In addition, it records array `new`'s size in bytes (`new_size`) and the size of `new`'s elements (`struct_size`). Aside from quick checks to ensure that the object has been allocated and not freed yet, S-Detector verifies that

$$(\texttt{pos\_init}-1)*\texttt{struct\_size}<\texttt{new\_size} \quad (1)$$

and

$$\texttt{pos\_end}/2-1\geq 0 \quad (2)$$

Actually inequality (2) is guaranteed by the loop's exit condition, so S-Detector only needs to check (1). Even when none of these four parameter values is available at compile time, S-Detector only needs to perform a *one-time*, *object-level* check at runtime, for array object accesses within this `while` loop.

Combining the structure- and loop-level pruning described above, S-Detector can eliminate all per-instruction memory checks on accesses of the `new` object in the sample code, performing at most one single run-time check instead.

4.2.2 Memory Trace Collector (S-Tracer)

Complete, detailed memory access traces allow diverse analysis and faithful benchmarking or simulation tests. However, their colletion is expensive, both in time and space. Existing tools like PIN [22], Valgrind [26], and DynamoRIO [6] produce memory trace output of daunting sizes, due to the high frequency of memory accesses in typical program executions. It is common for several seconds' execution to generate hundreds of GBs, sometimes even over one TB, of memory traces using any of the existing tools. Large memory trace size not only introduces large overhead for underlying trace storage and various trace-based analysis tools, but also affects the performance of the original programs. For example, PIN introduces an average slowdown of 38× for SPEC INT programs to perform memory analysis [38]. In addition, large traces bring back the I/O bottleneck during replay time, slowing down trace-driven simulations. Such limitations make it less and less practical for existing memory tracing tools to measure significant portions of modern data-intensive applications.

We present S-Tracer, a memory trace collection tool based on Spindle. With the static information that provided by Spindle, S-Tracer can generate highly compressed memory access traces with much lower runtime overhead than traditional tracing tools using dynamic instrumentation. At runtime, S-Tracer couples the Spindle-extracted MAS with dynamically collected information mentioned earlier in this section. The result would be a pair of static and dynamic traces, as illustrated in Figure 2 and Figure 3.

Our discussion below focuses on specific challenges due to the limitation of using LLVM IR, where we propose several techniques to generate approximate but fairly accurate traces.

Register spilling Since Spindle performs its static analysis in the LLVM IR level, where local scalar variables are usually represented as register variables, it is difficult for our approach to capture the stack memory accesses caused by register spilling in the final binary code. Considering the small footprint of register variables even with spilling, we implement typical register allocators used in the compiler backend for Spindle at the IR level, to calculate register spilling. Based on our experiments, our approach is able to achieve the similar statistical behavior of stack accesses as by traditional tracing tools.

Implicit memory accesses with function calls Function calls can also generate stack memory operations, not explicitly described in IR and hence not captured by our intra- and inter-procedural analysis. There are two categories of such accesses. For the caller, it has to write into stack the return address, the contents of registers to be used, and function parameters (with x86_64, the first 6 parameters are put in registers while the others in stack). For the callee, upon returning it has to read from stack the return address of the caller, the content of register `EBP` (for 32-bit systems) or `RBP` (for 64-bit systems), and the content of saved registers. To handle this, we again write a simple simulator to generate these memory accesses.

Dynamically linked libraries Since Spindle performs source code analysis, for calls to functions in dynamically linked libraries, we cannot capture their memory accesses in the IR level and have to fall back again to traditional dynamic instrumentation. As an optimization, we adopt a hybrid approach, by using dynamic instrumentation to collect the relative memory traces within such functions, along with their base stack addresses within the dynamic library. When a program calls such a function, we can then calculate new memory accesses based on the new base stack address.

5 Evaluation

In this section, we demonstrate the effectiveness of Spindle with the aforementioned two sample tools built on top of its static analysis framework: S-Detector for online memory bug detection and S-Tracer for full memory access trace collection.

We compare S-Detector with the state-of-the-art memory bug detector, ASan [33] by Google. In our experiments, S-Detector and ASan do the same checks: use after free, heap buffer overflow, stack buffer overflow, global buffer overflow, and memory leaks. Note that ASan does support additional checks (use after return, use after scope, and initialization order bugs), which need to be explicitly enabled by certain compiler options. Our tests used the default compiler options and we performed extra verification to confirm that these additional checks were disabled in all of our ASan experiments.

For S-Tracer, we show that it produces orders of magnitude smaller trace output, and thus lower overhead, by omitting redundant information. To validate its correctness, we also compare its decompressed trace with trace generated by PIN, a widely used dynamic tool.

5.1 Experiment Setup

Test platform We evaluate Spindle on a server with Intel Xeon E7-8890 v3 processors (running CentOS 7.1), 128GB of DDR3 memory, and 1TB SATA-2 hard disk. For memory bug detection, the tests use mandatory options to enable ASan and DrMem. For memory trace collection, we record each memory access in a 16-byte entry, 8 bytes for memory address and another 8 bytes for access type (read/write) and access size.

Test programs Currently, Spindle fully supports C and partially supports C++ and Fortran. For memory bug detection, we follow the practice of previously published tools and use 11 C programs from SPEC CPU 2006 [1]: `400.perlbench`, `401.bzip2`, `403.gcc`, `429.mcf`, `433.milc`, `445.gobmk`, `456.hmmer`, `458.sjeng`, `464.h264ref`, `470.lbm`, and `482.sphinx3`. The program `998.specrand` is omitted as it has too few memory accesses. Using these common test programs, we not only can compare the tools' runtime overhead, but also their effectiveness of capturing known bugs.

For memory trace collection, we use the popular NPB parallel benchmark suite [2] as codes with mostly regular memory accesses, plus SPEC `429.mcf` as a memory-intensive, non-numerical program. We also sample from modern data-intensive and irregular datacenter applications: (1) the Breadth First Search (`BFS`) component of the Graph500 Benchmark [11], a representative graph application with input-dependent memory accesses, (2) a convolutional neural network for digit recognition (`MNIST`) [29], (3) `kissdb`, a key-value store [18], and (4) `Fido`, a lightweight, modular machine learning library [8]. Finally, for multi-threaded applications, we test 3 programs from the PARSEC suite [4] covering different application domains: `streamcluster` (stream processing), `freqmine` (data mining), and `blackscholes` (PDE solving), plus one MapReduce [23]-style program performing `word count`, denoted as `SC`, `FM`, `BS` and `WC` respectively.

5.2 Spindle Compilation Overhead

Before we get to the tool use cases, we first assess the extra overhead brought by Spindle's static analysis. Table 1 summarizes this compilation overhead for evaluated programs, as well as their original compilation time and code size. In general, the Spindle compilation overhead only composes a small fraction of the original LLVM compilation cost (2% to 35%, average at 10%). We consider such one-time static analysis overhead negligible, considering the significant savings in the much larger runtime checking/tracing cost.

Table 1: Spindle compilation overhead

Program	Extra	Original	Code size	Program	Extra	Original	Code size
BT	0.260s	4.170s	232KB	perlbench	4.662s	23.036s	4418KB
CG	0.084s	0.651s	35KB	bzip2	0.053s	2.828s	239KB
EP	0.043s	0.493s	10KB	gcc	1.596s	66.729s	13777KB
FT	0.098s	0.908s	40KB	mcf	0.028s	0.694s	62KB
IS	0.049s	0.427s	25KB	milc	0.360s	3.899s	458KB
LU	0.225s	3.260s	244KB	gobmk	1.444s	16.921s	239KB
MG	0.161s	0.984s	43KB	hmmer	0.924s	8.773s	1126KB
SP	0.228s	2.320s	164KB	sjeng	0.270s	2.521s	298KB
BFS	0.704s	4.142s	852KB	h264ref	2.556s	15.268s	1656KB
MNIST	0.399s	1.138s	4KB	lbm	0.076s	0.906s	44KB
kissdb	0.092s	1.835s	16KB	sphinx3	0.304s	5.106s	767KB
FM	0.535s	7.760s	112KB	Fido	1.051s	9.287s	160KB
SC	0.159s	3.407s	80KB	BS	0.068s	2.250s	15KB
WC	0.054s	1.324s	19KB				

5.3 S-Detector for Memory bug detection

S-Detector runtime overhead We compare S-Detector with two popular memory bug detection tools: Google's AddressSanitizer (ASan) [33] and DynamoRIO [6]-based Dr. Memory (DrMem) [5]. To examine the benefits of instrumentation pruning based on Spindle's static analysis, we test two versions of S-Detector: *SD-All*, a baseline version that instruments all memory accesses, and *SD-Opt*, after check pruning.

On bug detection results, S-Detector captures most of the common SPEC bugs reported by DrMem and ASan, plus additional memory leaks (dynamically allocated objects not freed by program termination) that are verified by our manual code examination.

Figure 10 shows the runtime overhead of ASan, SD-All and SD-Opt, in percentage of the original program execution time. As DrMem is much heavier than others (for most programs over 10× slowdown), we omit its results from the figure for clarity. ASan is an industrial-strength tool, whose streamlined implementation delivers lower overhead than SD-All (geometric mean of overhead at 66% by the former vs. 184% by the latter), both with similar amount of instrumentation. SD-Opt, however, overcomes its slower checking implementation and brings down runtime overhead to geometric mean of 26%. Except for two out of 11 cases (`bzip2` and `h264ref`), SD-Opt reduces overhead from ASan, by up to 30.25× (`sphinx3`). We give more detailed discussion of these special cases later.

Spindle-enabled instrumentation pruning To take a closer look, we examine the amount of checks avoided by Spindle's static analysis. Figure 11 gives the percentage of eliminated memory checks, from SD-All to SD-Opt. On average, Spindle enables S-Detector to cut runtime memory checks by 64%, lowering its performance overhead consequently. The check and overhead reduction level depends on several factors, such as the amount of irregular/unpredictable memory accesses (Amdahl's Law), the overall intensiveness of memory accesses, and

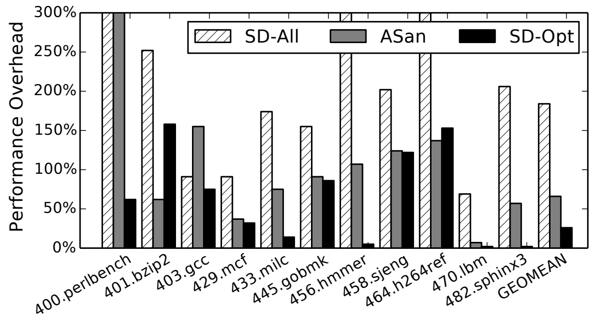


Figure 10: Overhead comparison (bars over 300% truncated)

control flow behavior. Below we give more detailed results and analysis via several case studies.

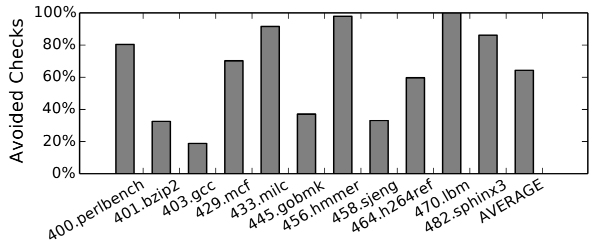


Figure 11: Reduction in runtime memory checks

`lbm`, `hmmer`, `milc`: These are the best cases among tested. Function-level profiling shows that the vast majority of their execution time and most of their memory accesses are spent within loops, where Spindle analysis allows S-Detector to apply the loop-level check presented in Section 4.2.1, replacing the per-access checks performed by ASan and DrMem. As a result, these three programs have 99%, 97%, and 91% of memory checks removed by S-Detector, respectively. Such instrumentation pruning then lowers S-Detector's runtime overhead, e.g., to 5% for `hmmer`, vs. ASan's 107%.

`gcc`: this compiler program is inherently input-dependent and as a result, has the lowest reduction by S-Detector in memory checks (19%). Interestingly, though its execution does spend most time within Spindle-identified loop structures, most of its loops are found to run only a few iterations, limiting the benefit of S-Detector's loop-level static checks. However, in this case even SD-All is faster than ASan. Follow-up measurements reveal that S-Detector's shadow memory implementation, though less efficient in general, offers better spatial locality than ASan's. With `gcc` accessed memory areas being particularly spread out, ASan's runtime check harms its locality, bringing the LLC miss rate from the original 1.3% to 5.9%, while S-Detector retains the original caching performance.

`bzip2`: this compression/decompression program is also input-dependent. Profiling reveals a performance hot-spot in sorting, with many branches whose taken status relies on input data. Even with 32% of runtime memory checks pruned, the less efficient instrumentation of

S-Detector brought overall higher overhead than ASan, 158% vs. 62%.

Despite such worst cases, the overall strong performance of S-Detector indicates that its Spindle-based static analysis, if adopted by highly-tuned, mature tools like ASan, may lead to even lower runtime overhead.

5.4 S-Tracer for Memory Trace Collection

Result Trace Verification Next, we evaluate S-Tracer, comparing it with the widely used PIN tool [22] for memory tracing. We first validate the correctness of its memory trace generation. Note that Spindle is based on compile-time instrumentation while traditional tools like PIN use runtime instrumentation. The two systems run application programs within different frameworks, each with different components (such as dynamic libraries), which may in turn alter the absolute locations of memory objects. Therefore, one would not expect them to generate identical trace sequences.

Recognizing such limitations, we first check the output trace size. We compare the size of PIN's trace with full traces recovered from Spindle's output, in the same format. The Spindle recovered trace has the similar volume to PIN's, with relative difference between 0.5% and 6% (median at 3.2%). Additional examination reveals that such discrepancies stem from the aforementioned inaccuracy caused by Spindle's approximation of stack accesses and register spilling. Though amounting for up to a few percent of the overall trace entries, affected accesses are typically localized to a very small footprint and hardly impact the overall memory access behavior.

We then validate the Spindle-generated heap memory access sequence. We examine trace fidelity by performing more detailed trace alignment and checking difference in heap access sequences. For each access on heap, we break it into a pair: (`object`, `offset`), since for each execution the dynamically allocated `object`'s `base` is different but the `offset` remains constant. We use `Linux diff` tool to compare S-Detector's heap trace and PIN's and find that overall, S-Tracer generates heap traces close to PIN's (relative difference ratio between 0.0% and 4.7%, median at 1.5%).

In the worst case, S-Tracer could generate an overall 5.9% difference in total trace size and 4.7% difference ratio on heap accesses, mostly attributed to stack accesses (more influenced by register allocation) and register spilling. Below we test this worst case, `BFS`, using a cache simulator, to (1) demonstrate a use case of our fast and large-capacity memory tracing and (2) provide a validation for trace fidelity. The test uses a simple trace-driven tool that simulates an 8-way set-associative cache with 64-byte cache line, and two replacement algorithms (LRU and FIFO). We validate simulation results using S-Tracer traces against that using PIN's, at varied cache sizes (including typical L2 and LLC sizes). Figure 12 shows that S-Tracer output achieves almost identical outcome as the PIN trace in miss ratio, across different combinations of cache size and replacement strategies.

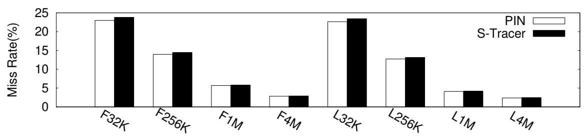


Figure 12: The cache miss rate of `BFS` in a trace-driven simulator. F means FIFO algorithm, L means LRU algorithm. The size means the cache size we simulate.

Trace Size Reduction Next we assess S-Tracer's gain in tracing time/space efficiency. Figure 13 shows a comparison of the trace size generated by S-Tracer and PIN, *in log scale*, for 13 single-thread and 4 multi-threaded programs. Truncated bars are from programs whose PIN traces exceed our 1TB storage capacity (`BT`, `EP`, `LU`, `SP` of Class A). For S-Tracer, the trace size includes both the static and dynamic components.

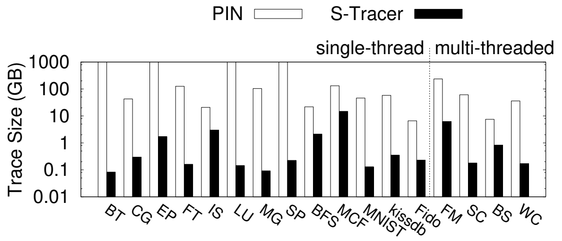


Figure 13: Trace size comparison

As expected, S-Tracer achieves orders of magnitude reduction in trace size from the PIN baseline. For programs dominated by regular memory accesses, like most of the programs in NPB benchmark, `MNIST`, `kissdb`, `streamcluster`, and `wordcount`, it reduces trace size by more than 100×. For the four NPB benchmarks where PIN exceeds the 1TB storage space, S-Tracer generates traces sized at 85MB-1.71GB. Even for the less regular programs, such as `BFS` and `freqmine`, Spindle brings considerable trace size reduction. In the worst case (`IS`, integer sorting), a 6.93× reduction is achieved.

We also evaluated compressing PIN's trace with a naive alternative, `gzip`, which ended up producing orders of magnitude larger traces than S-Tracer does. Besides, generating then compressing traces is much more expensive than Spindle-based approach, online or offline.

Runtime Tracing Overhead Reduction To evaluate the runtime overhead of trace collection, Figure 14 shows the slowdown factor (left axis, in *log scale*), calculated by dividing the execution time with tracing by the original time, for S-Tracer and PIN.

As expected, the online overhead difference is dramatic. In the 13 programs that PIN can complete tracing (full trace size under 1TB disk space), the average slowdown is 502× (and up to over 2000×), while S-

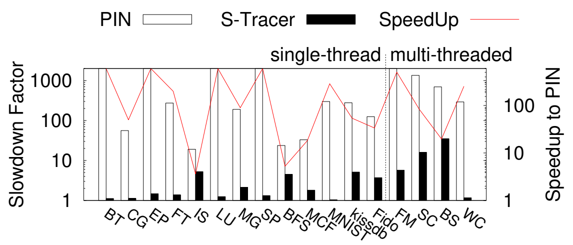


Figure 14: Application slowdown by S-Tracer and PIN with I/O (left) and S-Tracer speedup over PIN (right)

Tracer brings that of 6.5× on average (and up to 35.2×), making full trace collection/storage much more affordable. Across the applications, S-Tracer reduces slowdown from PIN by a factor of 61× on average.

Though we do not have space to show the no-I/O results, the savings there are still significant. For the 17 test programs, PIN introduces an average slowdown of 70.1× (and up to 384×), while S-Tracer brings that of 4.5× on average (and up to 33×). Across the applications, S-Tracer reduces slowdown from PIN by a factor of 17.9× on average. The reason is that Spindle allows S-Tracer to perform far less dynamic instrumentation, and an application's relative time overhead is highly correlated to its dynamic trace generation rate.

6 Related Work

Using Static Analysis to Assist Runtime Checking This group of work is closest to Spindle in approach. In particular, GreenArray [24] is an LLVM-based tool that analyzes the value range of index variables as well as the boundary of memory regions at compile time, to eliminate unnecessary runtime memory check. Spindle is different in that (1) its static analysis performs much more than inferring variables' value range, allowing complete computation of their value by iteration and full trace collection, and (2) the static skeleton it produces enables more types of and much more aggressive pruning in runtime checking, judging by reported GreenArray performance relative to AddressSanitizer.

Abstract Execution (AE) [19] produces a target-event-specific program slice, to be coupled by a "schema compiler" with runtime collected information and executed again for analysis or trace collection. Spindle, instead, records static trace at compile time, which is directly utilized during the target programs (production) execution.

On utilizing static information to assist trace collection, Cypress [44] uses hybrid static-dynamic analysis for parallel programs' communication trace compression. There are also techniques that perform static binary rewriting/instrumentation [32] or regular-expression-based memory access pattern construction for memory layout transformation [15]. However, none of these approaches is able to gather enough static structural information to enable versatile runtime monitoring/tracing as Spindle does.

Also, logical connectives proposed for relational analyses between input and output memory states [13] may be used by Spindle to further reduce instrumentation.

Monitoring/Tracing overhead reduction Prior work has explored reducing monitoring or tracing overhead in other ways. MemTrace [28] performs lightweight memory tracing of unmodified binary applications by translating 32-bit codes to 64-bit codes, which is fast but limits its application to running 32-bit programs on 64-bit machines. Among sampling-based methods, Vetter [40] evaluates techniques for analyzing communication activity in large-scale distributed applications. RACEZ [34] uses hardware performance monitoring units to sample memory accesses at runtime, and then uses the collected memory access trace for offline data-race detection. However, such low-overhead methods lose important information, such as temporal order of operations, or miss detection targets.

Finally, Bao et al. [3, 12] adopt a DIMM-snooping hardware mechanism to collect virtual memory reference traces. This hardware solution indeed minimizes collection overhead, but is rather costly and only catches memory accesses missed by on-chip caches.

7 Conclusion and Future Work

This paper presents Spindle, a versatile memory monitoring framework that performs detailed static analysis to extract program structures, allowing different types of static and dynamic techniques to *compute* rather than *collect* memory accesses whenever possible. Our development and experiments confirm that there are abundant redundancy and regularity in memory accesses, even for applications perceived as more irregular and data-dependent. By identifying predictable memory access behaviors at compile time and supplementing statically obtained memory access skeletons with runtime information, we can dramatically reduce the amount of online checking (for purposes like bug or race detection) or data collection (for purposes like memory access pattern analysis or memory tracing).

Acknowledgment

We thank all reviewers for their insightful comments and our shepherd Samira Khan for her timely guidance. We also thank colleagues from both the Tsinghua University PACMAN group and the QCRI Distributed Systems group, for their valuable feedback and suggestions. This work is supported in part by the National Key R&D Program of China (Grant No. 2017YFA0604500), National Natural Science Foundation of China (Grant No. 61722208, 61472201), Tsinghua University Initiative Scientific Research Program (20151080407). Jidong Zhai is the corresponding author of this paper.

References

[1] SPEC CPU 2006. https://www.spec.org/cpu2006/.

[2] D. Bailey, T. Harris, W. Saphir, R. V. D. Wijngaart, A. Woo, and M. Yarrow. *The NAS Parallel Benchmarks 2.0*. NAS Systems Division, NASA Ames Research Center, Moffett Field, CA, 1995.

[3] Yungang Bao, Mingyu Chen, Yuan Ruan, Li Liu, Jianping Fan, Qingbo Yuan, Bo Song, and Jianwei Xu. Hmtt: A platform independent full-system memory trace monitoring system. In *Proceedings of the 2008 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '08, pages 229–240. ACM, 2008.

[4] The PARSEC benchmark. http://parsec.cs.princeton.edu/.

[5] Derek Bruening and Qin Zhao. Practical memory checking with dr. memory. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization*, pages 213–223, Los Alamitos, CA, USA, 2011.

[6] Derek L Bruening. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Massachusetts Institute of Technology, 2004.

[7] The fcd tool. https://github.com/zneak/fcd/.

[8] The fido library. http://fidoproject.github.io/.

[9] Brian Fitzgerald, Jay P Kesan, Barbara Russo, Maha Shaikh, and Giancarlo Succi. *Adopting Open Source Software*. MIT Press, 2011.

[10] The LLVM Compiler Framework. http://llvm.org.

[11] Graph500. http://www.graph500.org/.

[12] Yongbing Huang, Licheng Chen, Zehan Cui, Yuan Ruan, Yungang Bao, Mingyu Chen, and Ninghui Sun. Hmtt: A hybrid hardware/software tracing system for bridging the dram access trace's semantic gap. *ACM Trans. Archit. Code Optim.*, 11(1):7:1–7:25, 2014.

[13] Hugo Illous, Matthieu Lemerre, and Xavier Rival. A relational shape abstract domain. In *NASA Formal Methods Symposium*, pages 212–229. Springer, 2017.

[14] Ravi Iyer, Li Zhao, Fei Guo, Ramesh Illikkal, Srihari Makineni, Don Newell, Yan Solihin, Lisa Hsu, and Steve Reinhardt. Qos policies and architecture for cache/memory in cmp platforms. In *SIGMETRICS'07*, pages 25–36. ACM, 2007.

[15] Jinseong Jeon, Keoncheol Shin, and Hwansoo Han. Layout transformations for heap objects using static access patterns. In *Proceedings of the 16th International Conference on Compiler Construction*, CC'07, pages 187–201, 2007.

[16] Vasileios P Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D Keromytis. libdft: Practical dynamic data flow tracking for commodity systems. In *ACM SIGPLAN Notices*, volume 47, pages 121–132. ACM, 2012.

[17] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *Micro*, pages 65–76, 2010.

[18] The kissdb program. https://github.com/adamierymenko/kissdb.git.

[19] J. R. Larus. Abstract execution: A technique for efficiently tracing programs. *Software Practice Experience*, 20(12):1241–1258, November 1990.

[20] Lei Liu, Zehan Cui, Mingjie Xing, Yungang Bao, Mingyu Chen, and Chengyong Wu. A software memory partition approach for eliminating bank-level interference in multicore systems. In *PACT'12*, pages 367–376. ACM, 2012.

[21] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. Avio: Detecting atomicity violations via access interleaving invariants. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 37–48. ACM, 2006.

[22] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200. ACM, 2005.

[23] The mapreduce program. https://github.com/sysprog21/mapreduce.git.

[24] Henrique Nazaré, Izabela Maffra, Willer Santos, Leonardo Barbosa, Laure Gonnord, and Fernando Magno Quintão Pereira. Validation of memory accesses through symbolic analyses. In *ACM SIGPLAN Notices*, volume 49, pages 791–809. ACM, 2014.

[25] Nicholas Nethercote and Julian 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.

[26] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100. ACM, 2007.

[27] Soyeon Park, Shan Lu, and Yuanyuan Zhou. Ctrigger: Exposing atomicity violation bugs from their hiding places. In *ASPLOS*, pages 25–36. ACM, 2009.

[28] Mathias Payer, Enrico Kravina, and Thomas R Gross. Lightweight memory tracing. In *USENIX Annual Technical Conference*, pages 115–126, 2013.

[29] The CNN program. `https://github.com/preimmortal/CNN.git`.

[30] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *ACM Sigplan Notices*, pages 245–258. ACM, 2009.

[31] Paruj Ratanaworabhan, Martin Burtscher, Darko Kirovski, Benjamin Zorn, Rahul Nagpal, and Karthik Pattabiraman. Detecting and tolerating asymmetric races. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 173–184. ACM, 2009.

[32] Amitabha Roy, Steven Hand, and Tim Harris. Hybrid binary rewriting for memory access instrumentation. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '11, pages 227–238. ACM, 2011.

[33] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, pages 309–318, 2012.

[34] Tianwei Sheng, Neil Vachharajani, Stephane Eranian, and Robert Hundt. Racez: A lightweight and non-invasive race detection tool for production applications. In *ICSE*, pages 401–410, 2011.

[35] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. pages 45–57, 2002.

[36] Allan Snavely, Laura Carrington, Nicole Wolter, Jesus Labarta, Rosa Badia, and Avi Purkayastha. A framework for performance modeling and prediction. In *SC*, pages 1–17, 2002.

[37] The McSema tool. `https://github.com/trailofbits/mcsema/`.

[38] Gang-Ryung Uh, Robert Cohn, Bharadwaj Yadavalli, Ramesh Peri, and Ravi Ayyagari. Analyzing dynamic binary instrumentation overhead. In *WBIA Workshop at ASPLOS*, 2006.

[39] Guru Venkataramani, Brandyn Roemer, Yan Solihin, and Milos Prvulovic. Memtracker: Efficient and programmable support for memory access monitoring and debugging. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 273–284. IEEE, 2007.

[40] Jeffrey Vetter. Dynamic statistical profiling of communication activity in distributed applications. In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '02, pages 240–250. ACM, 2002.

[41] Georg Von Krogh and Eric Von Hippel. The promise of research on open source software. *Management science*, 52(7):975–983, 2006.

[42] Shasha Wen, Milind Chabbi, and Xu Liu. Redspy: Exploring value locality in software. In *ASPLOS*, pages 47–61. ACM, 2017.

[43] Yichen Xie, Andy Chou, and Dawson Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. *ACM SIGSOFT Software Engineering Notes*, 28(5):327–336, 2003.

[44] Jidong Zhai, Jianfei Hu, Xiongchao Tang, Xiaosong Ma, and Wenguang Chen. Cypress: Combining static and dynamic analysis for top-down communication trace compression. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'14, pages 143–153, 2014.