Umbra: Efficient and Scalable Memory Shadowing

Qin Zhao

CSAIL
Massachusetts Institute of Technology
Cambridge, MA, USA
qin_zhao@csail.mit.edu

Derek Bruening

VMware, Inc.
bruening@vmware.com

Saman Amarasinghe

CSAIL
Massachusetts Institute of Technology
Cambridge, MA, USA
saman@csail.mit.edu

Abstract

Shadow value tools use metadata to track properties of application data at the granularity of individual machine instructions. These tools provide effective means of monitoring and analyzing the runtime behavior of applications. However, the high runtime overhead stemming from fine-grained monitoring often limits the use of such tools. Furthermore, 64-bit architectures pose a new challenge to the building of efficient memory shadowing tools. Current tools are not able to efficiently monitor the full 64-bit address space due to limitations in their shadow metadata translation.

This paper presents an efficient and scalable memory shadowing framework called *Umbra*. Employing a novel translation scheme, Umbra supports efficient mapping from application data to shadow metadata for both 32-bit and 64-bit applications. Umbra's translation scheme does not rely on any platform features and is not restricted to any specific shadow memory size. We also present several mapping optimizations and general dynamic instrumentation techniques that substantially reduce runtime overhead, and demonstrate their effectiveness on a real-world shadow value tool. We show that shadow memory translation overhead can be reduced to just 133% on average.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors – Optimization, Run-time environments

General Terms Performance

Keywords Shadow Memory, Dynamic Optimization

1. Introduction

Shadow value tools store information about every application data location accessed by an application. This information, or *shadow metadata*, is tracked at the granularity of individual instructions as the application executes. Shadow value tools have been created for a wide variety of purposes, including finding memory usage errors [21, 24], tracking tainted data [5, 18, 20], detecting race conditions [9, 12, 22, 23], and many others [3, 14, 15, 29].

Although hardware-supported shadow value frameworks have been proposed both for specific tools [7, 8, 26, 27, 30] and general tool classes [4, 6, 31], shadow value tools in use today are implemented entirely in software. This allows them to run on com-
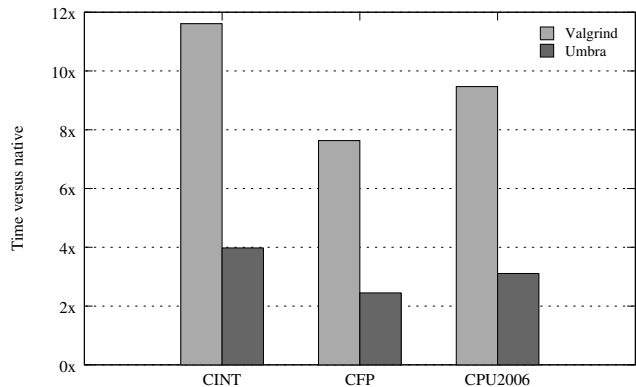
Figure 1. The performance of Umbra compared to Valgrind on 64-bit Linux on the SPEC CPU2006 benchmarks, focusing on shadow metadata mapping. Umbra is configured to use 2 shadow bits per application byte, to match the setup of the Valgrind MemCheck tool [24]. As shown in Section 4, Umbra achieves even better performance when using one shadow byte per application byte.

modity hardware, thereby broadening their reach. Software-based shadow value tools are typically implemented using a dynamic binary instrumentation system like DynamoRIO [2], Pin [13], or Valgrind [17]. By inserting additional instructions to be executed along with the application's code, a shadow value tool can update shadow metadata during program execution.

The inserted instrumentation code performs two tasks: *mapping*, which maps from an application data location to the corresponding shadow metadata location, and *updating*, which performs customized metadata updates and checks. These two tasks are also the major source of runtime overhead in software shadow value tools. In the current state-of-the-art tools, full memory shadowing results in one or two orders of magnitude slowdown. This prohibitive overhead results in infrequent deployment of shadow value tools, even though they can potentially lead to important insights about an application's behavior that in turn can be used for performance tuning or debugging. In this paper, we describe how our framework reduces the metadata mapping overhead. Figure 1 compares the mapping overhead of Umbra to the most widely-used memory shadowing framework, Valgrind. Umbra is *three times faster*.

1.1 Shadow Metadata Mapping Schemes

Shadow metadata mapping is the process of translating an application data location to its corresponding shadow metadata location. Shadow value tools typically use either a one-level or two-level

mapping scheme. In a one-level scheme, the entire application address space is shadowed with a single, contiguous shadow address space. Mapping becomes a simple offset with a scaling factor, depending on the relative sizes of the spaces. However, reserving such a large piece of address space endangers the robustness of some applications. Furthermore, operating systems often impose requirements on where certain structures are located, which constrains the deployment of a one-level scheme.

A two-level mapping scheme splits up the shadow memory into regions with the first level of translation used to determine which region is to be used for a particular memory address. All existing two-level schemes map the address space uniformly. This works well for 32-bit address spaces but cannot scale up to full 64-bit address spaces. A key insight of Umbra is to allocate and perform mappings based on the application's memory allocations.

1.2 Contributions

The following are the key contributions of this paper:

- We propose a flexible shadow memory mapping scheme that does not rely on idiosyncrasies of the operating system or underlying architecture and is not limited to specific shadow metadata sizes or semantics. To the best of our knowledge, it is the first software shadow memory mapping scheme that scales to both 32-bit and full 64-bit address spaces efficiently.

- We present several novel optimizations to improve the speed of shadow metadata mapping.

- We present a general 3-stage code layout strategy for efficient dynamic instrumentation.

- We show that shadow memory translation can be implemented with low average overhead of 133%.

- We study the trade-off between metadata space usage and metadata mapping efficiency.

- We demonstrate the usefulness of Umbra by implementing a shared data reference detection tool that is suitable for analyzing multi-threaded application data access behavior.

1.3 Outline

The rest of the paper is organized as follows: Section 2 describes the basic framework of Umbra. Section 3 then presents optimizations to improve the basic system. Section 4 evaluates the performance of Umbra. Section 5 discusses related work and Section 6 concludes the paper.

2. Base System

We designed Umbra with the following goals in mind:

Flexibility. Umbra should be a general-purpose memory shadowing framework and not be tied to any particular use case. It should support a wide range of shadow metadata sizes a tool might need, from a single shadow bit per application byte to several shadow bytes per application byte. Many prior systems were built with only one application in mind and only work well with certain predetermined metadata sizes.

Platform independence. Umbra should be platform independent and not rely on features of a particular platform in order to achieve efficiency. This makes the system easy to port.

Efficiency. Umbra's runtime overhead should be as low as possible.

2.1 Base System Overview

There are two basic components to Umbra. The *shadow metadata manager* is in charge of shadow memory management, including allocating and de-allocating shadow metadata, as well as maintaining the mapping information between application data and shadow metadata. The *instrumenter* instruments the program during its execution. The inserted instructions perform metadata mapping and updating for each instruction in the application.

2.2 Shadow Metadata Manager

An application's data is stored either in registers or in memory. These are dealt with differently.

2.2.1 Metadata Management for Registers

Most modern processors have a fixed and limited number of registers that can be used by a program. Furthermore, the registers used by every instruction can be determined by inspection. Thus we are able to statically allocate shadow metadata for every register and bind it accordingly. If an instruction uses any registers, we can insert metadata updating code to update or check the corresponding metadata directly without any metadata mapping code.

2.2.2 Metadata Management for Memory

Unlike registers, a single instruction can access dynamically varying memory locations. Thus, the shadow metadata for application memory must be managed dynamically. The Shadow Metadata Manager dynamically allocates and frees shadow metadata and must perform metadata mapping to locate the appropriate shadow metadata before that metadata can be updated. In Umbra, metadata mapping is achieved via a *shadow memory mapping table*.

We observe that the application memory is organized into a number of memory modules, including stacks, heaps, data and code segments of executables and shared libraries, etc. This observation inspires us to use a simple yet novel shadow memory mapping scheme that uses the application memory module as a mapping unit: for each such module the Shadow Metadata Manger allocates a shadow memory module and associates it with the application memory module. For example, Table 1 shows that the simple program `HelloWorld` running on Linux has five application memory modules. We simply associate each module with one shadow memory module.

Module	Application Memory	Shadow Memory
HelloWorld	08048000-0804b000	28048000-2804b000
[heap]	097a3000-097c4000	297a3000-297c4000
libc.so	b7e20000-b7f7f000	57e20000-57f7f000
ld.so	b7f8b000-b7fab000	57f8b000-57fab000
[stack]	bfb95000-bfbaa000	5fb95000-5fbaa000

Table 1. Application memory modules for a simple application.

By focusing on allocated memory regions rather than on the entire address space, this approach scales to large 64-bit address spaces without requiring multiple translation steps or extremely large tables: the table scales with the application's memory use, independently of the maximum size of the address space.

The module level mapping can be further improved by moving to a more coarse-grained mapping: address space unit mapping. The idea is to virtually split the whole process address space into two address spaces: the application address space and the shadow address space, as was implemented by TaintTrace [5]. However, unlike TaintTrace, which splits the space into two equally-size pieces, we carve up the address space in a much more flexible and efficient manner.

We treat the whole process address space as a collection of address space units. Each address space unit has three possible states:

- An *application address space unit* is used for hosting application modules. The size of each application address space unit is fixed, e.g., 256MB for 32-bit architectures and 4GB for 64-bit architectures, and its start address must be unit-aligned. This restriction is to enable a fast check to determine whether a memory address is in an application unit.

- A *shadow address space unit* is reserved for storing shadow memory metadata. The size of the shadow units depends on the size of the shadow metadata. For example, when using one shadow bit per application byte, the shadow units are one-eighth the size of the application units.

- An *unused unit* is memory that is not yet used.

At the start of application execution, we first obtain information about all of the application memory modules. We assign those address space units that contain application modules as application address space units. Then we reserve new shadow address space units for each application unit. If an application memory module spans multiple units, we reserve multiple contiguous shadow units for its metadata. Table 2 shows the address space units and the modules inside each for the HelloWorld example of Table 1.

	Application Memory	**Shadow Memory**
Unit	00000000–10000000	20000000–30000000
Module	08048000–0804b000	28048000–2804b000
	097a3000–097c4000	297a3000–297c4000
Unit	b0000000–c0000000	50000000–60000000
Module	b7e20000–b7f7f000	57e20000–57f7f000
	b7f8b000–b7fab000	57f8b000–57fab000
	bfb95000–bfbaa000	5fb95000–5fbaa000

Table 2. Address space units and the modules inside each for the HelloWorld example of Table 1.

When the application requests a new piece of memory from the operating system via a system call, the Shadow Metadata Manager intercepts the the system call and ensures that the application receives that new memory from either an application address space unit or an unused unit that then becomes an application unit. The Manager then adjusts the shadow units to maintain the corresponding shadow memory. If the application resizes an existing piece of memory, the Manager performs the same actions on the shadow memory. This may require relocating the shadow memory if there is a conflict between application memory and shadow memory. To detect and handle cases where the application obtains new memory without using system calls, e.g., during stack expansion, we use a signal handler to catch and resolve the access violation signal raised by accessing the unallocated corresponding shadow memory.

This simple mapping scheme allow us to use a small shadow memory mapping table to maintain the translation information between application and shadow memory modules. Table 3 shows the mapping table of the HelloWorld example. Each application address space unit has one entry in the mapping table, which stores two values:

- $base_{app}$ is the start address of the application address space unit, which is used for table entry lookup.

- *offset* is an offset value for address translation.

When translating from an application address $addr_{app}$ to its corresponding shadow address $addr_{shd}$, we first identify in which application unit $addr_{app}$ is contained by using a mask to calculate $addr_{app}$ aligned to the unit size. We compare that value with the $base_{app}$ of each table entry, and then calculate $addr_{shd}$ using

$base_{app}$	$Offset$
0x00000000	0x20000000
0xb0000000	-0x60000000

Table 3. Shadow memory mapping table for HelloWorld example.

$addr_{app}$ and *offset* from the matched entry based on the equation below:

$$addr_{shd} = addr_{app} \times scale + offset \qquad (1)$$

$Scale$ is the scale factor from application memory to shadow memory, and it is 1 in the HelloWorld example for one shadow byte per application byte mapping. If we restrict the shadow metadata unit size to be a power of 2, this equation can be optimized using a shift as shown below, which is a faster operation than multiply on most architectures:

$$addr_{shd} = addr_{app} \ll scale_{shift} + offset \qquad (2)$$

2.3 Instrumenter

The *instrumenter* inserts the *metadata tracking* (i.e., metadata mapping and updating) code into the application code stream. Metadata updating code varies depending on the shadow value tool. Here we focus on the metadata mapping code. In particular, we focus on code for application memory accesses, since metadata for registers is statically bound. For each application memory reference, the instrumented code performs a sequence of operations as shown in Figure 2:

Metadata Tracking(Instr)
1: Save application context
2: Calculate $addr_{app}$ from Instr and saved context
3: Search mapping table for the correct entry
4: Calculate $addr_{shd}$
5: Update metadata at $addr_{shd}$
6: Restore application context

Figure 2. Pseudocode for metadata tracking instrumentation.

Steps 1 and 6 preserve the application's context. Step 2 calculates the memory address $addr_{app}$ from the instruction and the application's context (e.g., the register value used as a base register in the instruction's address operand). Step 3 walks the mapping table to find the containing application address space unit and its translation information. Step 4 then calculates the corresponding shadow memory address $addr_{shd}$ using $addr_{app}$ and the translation information found in Step 3. Step 5 performs metadata update operations, which are specified by the Umbra client.

2.4 Client Interface

Umbra's memory shadowing framework is combined with a *client* to produce a complete shadow value tool. Umbra provides a simple interface that allows the tool developer to concentrate on inserting code for the client's metadata updates (Step 5 in Figure 2) without worrying about the details of mapping between application data and shadow metadata.

The interface includes a data structure `umbra_client_t` and a list of event callback hooks. The `umbra_client_t` structure, shown in Figure 3, allows a client to specify the parameters of the desired shadow memory mapping, such as the number of registers to be used, the unit size of application data and of shadow metadata, and the events of interest. In theory, Umbra could allow the application data unit size (`app_size`) and shadow metadata size (`shd_size`) to be any value. In our current version we restrict the value to a power of two in order to simplify the implementation and

```
struct umbra_client_t {
  /* shadow memory specification */
  int num_regs;    /* number of registers to use */
  int app_size;    /* application data unit size */
  int shd_size;    /* shadow metadata unit size  */
  /* event callback hooks */
  ...
}
```

Figure 3. A client specifies desired parameters of Umbra's memory shadowing framework using the `umbra_client_t` data structure.

provide better performance. Most tools desire a power-of-two size regardless.

An Umbra client must export an initialization function named `umbra_client_init`, which is called by Umbra at application start time. The function fills in the fields of `umbra_client_t` and registers event hooks to point at client-provided callback functions. Umbra then sets up the shadow memory and instruments the application code according to the client specifications. Umbra also calls the provided callback functions when the indicated events occur. Examples of commonly used event hooks are listed in Table 4.

Event Hooks	Description
`client_exit`	Process exit
`client_thread_init`	Thread initialization
`client_thread_exit`	Thread exit
`shadow_memory_create`	Shadow memory creation
`shadow_memory_delete`	Shadow memory deletion
`instrument_update`	Insert metadata update code

Table 4. Umbra client event callback hooks.

The `instrument_update` event is the most important callback function to be implemented by a client. Umbra passes needed information to the client via callback function arguments, including the memory reference instruction to be instrumented and an array of registers whose first register will contain the address of the corresponding shadow metadata during execution of the instruction. The client-implemented callback function inserts metadata update code, which will be executed immediately prior to the application's memory reference instruction each time that instruction is invoked. The `shadow_memory_create` and `shadow_memory_delete` events allow a client to perform metadata initialization and collect or report results, respectively, while the thread and process events allow the client to update any bookkeeping it maintains.

3. Optimization

The framework described in Section 2 works correctly, but it incurs large time and space overheads. We can significantly reduce these overheads using a number of techniques. Performance is improved in two different ways:

- We present several mapping improvements that speed up or even avoid walking the mapping table during translation.

- We optimize the inserted instrumentation itself to reduce overhead.

3.1 Translation Optimizations

We use a number of caching strategies to eliminate translation overhead.

3.1.1 Thread-Private Mapping Table (O1)

To support multi-threaded applications, any query or update of the shadow memory mapping table must be guarded by a lock. This incurs locking overhead and may suffer from lock contention. We use a thread-private mapping table to reduce such overhead. The thread-private table caches the information from the global table. Any application memory update by a thread is immediately pushed to the global mapping table. The thread-private table only pulls updates from the global table when necessary. The rest of the data structures described in this section are thread-private, thereby avoiding the complexity and overhead of synchronization.

3.1.2 Metadata Lookup Hashtable (O2)

Traversing the mapping table for every memory reference can cause large overheads. A *metadata lookup hashtable* is introduced to improve lookup speed. This table serves a similar role as the translation lookaside buffer (TLB) does for virtual page table lookup. The lookup hashtable has a fixed number of slots that store pointers to thread-private mapping table entries. It uses a unit-aligned application address as search key, and returns the mapping table entry pointer if the requested address is present in the hashtable. If the address is absent, a mapping table traversal is performed, and the hashtable is updated with the newly found pointer.

3.1.3 Last Unit (Memoization) Check (O3)

Each thread also stores the memory mapping found in the previous translation lookup. Before performing any lookup, we first check if it is the memory unit we found last time. This optimization takes advantage of the reference locality of the overall application execution.

3.1.4 Reference Cache (O4)

Our final translation optimization tries to avoid the mapping table lookup by taking advantage of each individual instruction's reference locality: an instruction typically accesses memory at the same location or at a nearby location on each subsequent execution.

A *reference cache* is a software data structure containing the same information as the mapping table entry:

```
struct reference_cache_t {
    void *base;
    void *offset;
}
```

`base` is a unit-aligned application memory address while `offset` holds the corresponding mapping information to its shadow memory.

We associate each memory reference instruction with a reference cache that stores the memory reference and translation information from the instruction's previous execution. When translating a memory address, we first check the reference cache to see if it accesses the same unit as its previous execution. If it matches, we use the stored `offset` directly. Otherwise, the translation proceeds to the lookup, and the reference cache is updated with the new mapping information. Because the total number of static application instructions that are executed in any one run is small, the total memory usage for the reference cache is small as well, only a few kilobytes for most applications.

As stack memory references in one thread all access the same memory unit, they all share one reference cache. If the application swaps stacks, only one reference cache miss will occur followed by a series of hits once the new stack's information is in the cache.

3.2 Instrumentation Optimizations

In addition to improving the performance of the metadata mapping scheme, we also apply several general optimizations to our inserted instrumentation.

3.2.1 Context Switch Reduction (O5)

Previous work [5, 20, 29] proposed optimization to reduce context switch overhead by analyzing register usage and utilizing dead registers whenever possible. We further extend this optimization. In the case that we have to save and restore a register for stealing, we expand the register steal range as far as possible. Typically, more than one application memory reference falls in the range, allowing us to share the save and restore cost across multiple shadow memory translations. Careful attention must be paid to fault handling, where the register's value may need to be restored even when there is no explicit use in the regular execution path.

3.2.2 Reference Group (O6)

We observe that it is often the case that, in the same basic block, several instructions reference memory close to each other: e.g., function local variables, or different fields of the same object. If we statically know that two memory references access the same application address space unit or two contiguous units, we cluster these two instructions into one *reference group*. All the memory references in a reference group share the same reference cache. In addition, only the first reference need perform a mapping lookup. All subsequent references can use the translation information from that first lookup.

A *trace* is a code fragment with a single entry but multiple exits. DynamoRIO builds traces from several frequently executed basic blocks. Reference group optimization can be extended over multiple basic blocks of a trace due to the single entry property.

This optimization assumes that shadow memory is allocated contiguously if its application memory is allocated together, which is guaranteed to be true in Umbra's mapping scheme.

3.2.3 3-Stage Code Layout

The metadata mapping pseudocode from Figure 2 is updated in Figure 4 with the addition of the optimizations presented in Section 3.1.

If we inlined all 27 steps for every memory reference instruction, the code size expansion would be prohibitive, causing poor performance in both the software code cache and hardware instruction cache. Instead, we split the instrumentation into three parts, resulting in a *3-stage code layout*:

- The first stage (*inline stub*) is inlined for fast execution at a small space cost and minimal context switch; this stage includes the address check in steps 3–4.

- The second stage (*lean procedure*) is invoked if the inlined check of the first stage misses. It uses shared code with a fast function call protocol to execute the code and return with small context switch overhead. This stage is used for steps 5–19. The fast function call protocol includes only a partial context switch and uses a memory store and jump instructions to perform a call and return without requiring a stack. The callee cannot use the stack but has several saved registers available for scratch space.

- The third stage (*callout*) performs a full context switch and invokes shared code that is implemented in C rather than hand-coded in machine instructions. This stage is invoked only if the second stage lookup fails; it covers step 20–24.

In this way, we are able to strike a balance between performance and space requirements, reducing the size of instrumented code

Metadata Tracking(Instr)
1:    Save application context
2:    Calculate $addr_{app}$ from Instr and saved context
Inline stub:
3:    (O4) Check Instr's Reference Cache
4:    Jump to 25 if hits
Lean procedure:
5:    (O3) Check Last Unit
6:    Jump to 9 if no match
7:    Update Instr's Reference Cache
8:    Jump to 25
9:    (O2) Search Metadata Lookup Hashtable
10: Jump to 14 if not found
11: Update Instr's Reference Cache
12: Update Last Unit
13: Jump to 25
14: (O1) Search Thread Private Mapping Table
15: Jump to 20 if not found
16: Update Hashtable
17: Update Last Unit
18: Update Instr's Reference Cache
19: Jump to 25
Callout:
20: Search Global Mapping Table
21: Update Thread-Private Mapping Table
22: Update Hashtable
23: Update Last Unit
24: Update Instr's Reference Cache
$addr_{shd}$ *calculation*:
25: Calculate $addr_{shd}$
26: Update metadata at $addr_{shd}$
27: Restore application context

Figure 4. Pseudocode for optimized shadow metadata tracking instrumentation.

without compromising the optimizations. In most cases, only the first stage is executed, allowing us to avoid large overheads due to context switches. This 3-stage code layout strategy can also be applied to general dynamic instrumentation tasks for better performance without sacrificing functionality, where an inline stub performs simple common-case actions and a lean procedure and callout are used for less common and more complex situations.

3.3 Mapping Table Update

Although the performance of mapping table lookups is improved, the multiple levels of cache increase the complexity of updating the shadow memory mapping table when the application memory layout changes.

Adding a new application address space unit is normally cheap, requiring only a new entry in the global mapping table. The new information will be propagated into every level of cache lazily as the application accesses the newly allocated memory.

In contrast, removing an entry is much more expensive, requiring that we suspend all threads while we update every level of cache in every thread. We try to delay such updates on the mapping table for better performance. For example, if the application de-allocates (unmaps) a piece of memory, we delete the corresponding shadow memory, but do not change the mapping table even if there is no application memory in the application address space unit. If the application later allocates memory from that same application address space unit, the same shadow address space unit and mapping table entry are used.

In some cases an expensive update is unavoidable. For example, if an application requests memory from a fixed address that was re-

served for shadow memory; or an application expands its memory across a unit boundary and causes a conflict with shadow memory. For such cases, we suspend all the threads, move the shadow memory, relabel the address space units, and update every level of cache in all threads. These are extremely rare events with our large mapping units and as such they have negligible impact on overall performance. In contrast, if we used a finer-grained module-level mapping, we would have to update all of the cached mapping information on every module change, including each call to `mremap`.

3.4 Discussion

This mapping scheme and optimization works well even for large applications with complex memory usage. It avoids updating the mapping table when an application repeatedly allocates and de-allocates memory in the same address space unit, and it is flexible enough to handle rare conflicts by relocating the shadow memory and updating the mapping table. In a 64-bit architecture, the available address space is much larger than current applications use and even larger than current hardware's physical capacity, so our scheme can easily handle large applications without any problem. In contrast, it is possible that a 32-bit address space might be exhausted by the application and Umbra together. However, this possibility is present for any shadow value framework, including the widely used MemCheck [24]. Umbra's shadow memory layout can be configured to match Memcheck's second-level shadow memory layout, and Umbra's small mapping table occupies less space than MemCheck's first-level table. Thus, Umbra should be able to operate on any application that runs successfully under MemCheck.

In addition to simplifying the handling of memory map updates, address space unit mapping has other performance advantages over module level mapping. Because one application address space unit often contains several memory modules, it not only makes table traversal faster due to fewer entries, but also increases the hit ratio of the hashtable, last unit cache, and the reference cache.

We can further reduce the memory used by our mapping scheme. For example, it is possible to allocate shadow memory in a lazy way by not allocating it until its corresponding application memory is accessed and an access violation signal is raised for accessing the metadata.

4. Evaluation

In this section, we evaluate the performance of Umbra on a number of benchmarks.

4.1 Experimental Setup

We have implemented Umbra on top of DynamoRIO version 1.4.0 [1] for Linux. We used the SPLASH-2 [28] and SPEC CPU2006 suite [25] [1] with the reference input sets to evaluate Umbra. All the benchmarks are compiled as 64-bit using gcc 4.1 -O2. We ran our experiments on dual-die quad-core Intel Xeon processors with 3.16GHz clock rates, 12MB L2 cache on each die, and 8GB total RAM. The operating system is 64-bit Debian GNU/Linux 5.0. We configured Umbra to use 4GB as the address space unit size.

4.2 Performance Evaluation

In the first experiment, we assess the translation overhead. For every memory reference performed by the application we calculate the corresponding shadow memory address without any further operation being done on the shadow memory. The resulting performance normalized to native execution is shown in Table 5. The second column (DR) shows that the DynamoRIO core has an average

slowdown of 14%. The third (1B-1B) and fourth (1B-2b) columns list the performance of Umbra mapping every byte of application memory into 1-byte and 2-bit shadow memory, respectively. The slowdown varies from about 10% to 6x, and the benchmarks that run slower in DynamoRIO usually suffer more runtime overhead under Umbra, implying that the sources of overhead are similar in both Umbra and DynamoRIO.

Benchmark	DR	Umbra		Valgrind	
		1B-1B	1B-2b	base	map
400.perlbench	1.76	4.57	6.12	10.00	19.20
401.bzip2	1.07	2.75	3.75	6.43	10.06
403.gcc	1.20	2.24	2.80	4.23	7.40
429.mcf	1.08	1.75	1.92	2.40	2.78
445.gobmk	1.59	5.01	6.82	10.93	15.96
456.hmmer	1.01	2.85	3.79	5.31	8.82
458.sjeng	1.50	4.91	6.63	10.81	14.85
462.libquantum	0.98	1.07	1.11	2.55	3.02
464.h264ref	1.29	4.00	5.57	8.47	37.04
471.omnetpp	1.20	2.44	3.41	3.68	6.84
473.astar	1.05	2.15	2.62	3.89	5.55
483.xalancbmk	1.28	2.57	3.25	4.60	7.76
CINT Average	**1.25**	**3.03**	**3.98**	**6.11**	**11.61**
410.bwaves	1.04	1.52	1.89	3.64	5.67
416.gamess	0.96	2.33	3.21	4.63	8.37
433.milc	1.00	1.23	1.38	2.04	3.05
434.zeusmp	0.99	1.36	1.66	—	—
435.gromacs	1.03	1.84	2.77	8.20	12.31
436.cactusADM	1.00	2.04	4.49	3.79	8.62
437.leslie3d	1.00	1.51	1.99	3.20	5.91
444.namd	1.00	1.11	1.37	3.59	5.53
447.dealII	1.18	2.98	3.77	—	—
450.soplex	1.02	1.46	1.64	2.85	3.88
453.povray	1.38	3.51	4.74	7.44	13.05
454.calculix	1.00	1.33	1.80	3.26	5.51
459.GemsFDTD	1.01	1.39	1.70	2.36	4.55
465.tonto	1.19	2.34	3.21	5.41	12.75
470.lbm	1.00	1.05	1.12	1.90	2.60
482.sphinx3	1.04	1.98	2.45	12.38	15.01
CFP Avg	**1.05**	**1.81**	**2.45**	**4.62**	**7.63**
SPEC Avg	**1.14**	**2.33**	**3.11**	**5.31**	**9.47**

Table 5. Performance summary on the SPEC CPU2006 benchmarks for DynamoRIO (DR), Umbra configured for byte-to-byte shadowing (1B-1B), Umbra configured for byte-to-2-bit shadowing (1B-2b), Valgrind base, and Valgrind performing shadow metadata mapping (map). Valgrind's shadow mapping is byte-to-2-bit. 434.zeusmp and 447.dealII fail to run under Valgrind.

We also measure the running time of the Valgrind base and of Valgrind's MemCheck tool [24] modified to only perform address mapping (byte-to-2-bit). The resulting data are presented in the fifth (base) and sixth (map) columns of Table 5. The Valgrind base performs extra operations targeted at shadow value tools that the DynamoRIO core does not and which Umbra must perform itself, making the core-to-core comparison less meaningful than the Umbra-to-Valgrind-with-mapping comparison. Valgrind's mapping overheads are much higher than Umbra's, ranging from 2x to over 30x, with an average 8.47x slowdown [2].

---

[1] wrf is excluded because it cannot be compiled by gcc 4.1

[2] 434.zeusmp and 447.dealII are not included as they fail to run under Valgrind

4.3 Optimization Impact

We next conduct a set of experiments to quantify the impact of our optimizations described in Section 3. These experiments perform translation for a 1-byte to 1-byte mapping. Figure 5 shows the performance normalized to native execution of base DynamoRIO and of each optimization added to the set of prior optimizations.
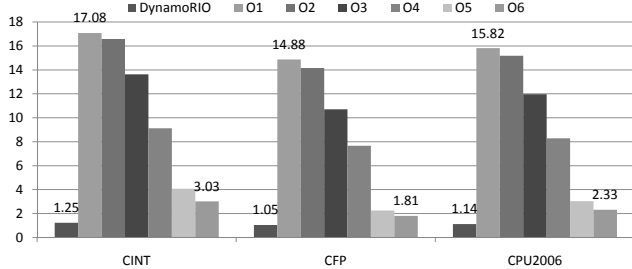

Figure 5. Impact of optimizations from Section 3, applied cumulatively: O1 (Thread Private Mapping Table), O2 (Hashtable), O3 (Last Unit Check), O4 (Reference Cache), O5 (Context Switch Reduction), and O6 (Reference Cache Group).

The figure shows that O2 (Hashtable) has only a marginal improvement over O1 (Thread-Private Mapping Table). In these benchmarks the mapping table is small (< 10 entries), making the table walk inexpensive. O2 would show more improvement over O1 with a larger mapping table. O3 (Last Unit Check) and O4 (Reference Cache) take advantage of application reference locality on the overall application as well as individual instructions, which improves performance significantly: they halve the overall running time on average. O5 (Context Switch Reduction) has the biggest impact and further halves the runtime overhead. The context switch overhead is expensive, as register saving and restoring requires memory load and store operations. Adding several extra memory operations, especially stores, for every application memory reference can easily saturate the memory bus and cause long delays. O6 (Reference Cache Group) removes another 20% of running time by further taking advantage of reference locality over basic blocks and avoiding redundant runtime checks via static analysis. When all optimizations are applied, the overall average runtime overhead is reduced to 133% over native execution.

To better understand the quality of these optimizations, we collect a number of statistics about the benchmark characteristics and the optimization effects. Table 6 presents the ratio of these statistics relative to the total number of application instructions executed.

The flags stolen and registers stolen ratios show the effect of the Context Switch Reduction optimization (O5). The flags stolen ratio is reduced from 41.79% (save and restore on every memory reference) to 2.55% and the register save and restore is reduced from 41.79% to 8.20%. This significantly reduces the pressure on the memory bus and thus reduces runtime overhead, and explains why context switch reduction has the biggest improvement. The reference cache check ratio shows that the Reference Cache Group optimization (O6) effectively removes 19% (41.79% - 22.76%) of redundant reference cache checks.

We also collect the hit ratio of our cache lookups, which are presented in Table 7.

We expected the Last Unit Check to experience some thrashing when an application accesses alternating memory units, e.g., interleaving stack and heap accesses. In contrast, each instruction usually accesses the same type of memory and thus the same memory unit. Table 7 confirmed our expectations. The Reference Cache

Metric	CINT	CFP	All
memory references	40.34%	42.88%	41.79%
flags stolen	3.17%	2.07%	2.55%
registers stolen	11.44%	5.77%	8.20%
ref cache checks	25.70%	20.56%	22.76%

Table 6. Optimization statistics normalized to the total number of application instructions executed.

Metric	CINT	CFP	All
ref cache hit ratio	99.91%	99.94%	99.93%
last check hit ratio	66.26%	69.98%	68.93%

Table 7. Hit ratio of the Reference Cache (O4) and the Last Unit Check (O3).

hit ratio is extremely high ($> 99.9\%$), while the Last Unit Check hit ratio is much lower.

4.4 Impact of Shadow Metadata Size

The shadow metadata size chosen can significantly impact the mapping overhead. To evaluate that impact we measure the following shadow sizes:

1B-to-1B maps 1 byte of application memory to 1 byte of shadow memory. This is the fastest mapping because only a simple offset is required.

1B-to-4B maps 1 byte of application memory to 4 bytes of shadow memory. This requires one left shift and one addition, as shown in Section 2.

4B-to-1B maps 4 bytes of application memory to 1 byte of shadow memory. It is similar to the 1B-to-4B mapping but uses a right shift.

1B-to-2b maps 1 byte of application memory to 2 bits of shadow memory. It first performs a 4B-to-1B mapping and then uses an extra register to hold the bit position. This incurs additional overhead for register stealing and bit position calculation.
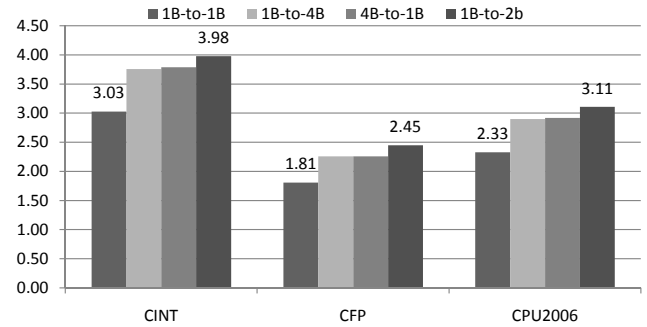

Figure 6. Performance of different shadow metadata sizes, listed as *application size*-to-*shadow size* where *B* is *byte* and *b* is *bit*.

Figure 6 shows the normalized performance. As expected, the 1B-to-1B mapping has the best performance, and 1B-to-2b has the worst, a 30% slowdown compared to 1B-to-1B.

4.5 Code Cache Expansion

In the experiments above, the instrumented code is organized into three stages (Section 3.2.3). The Reference Cache check (O4) is

inlined and only one register is stolen for use. The Thread-Private Table walk (O1), Hashtable search (O2), and Last Unit Check (O3) are implemented as a shared lean procedure, where two registers are used. The global mapping table lookup is implemented as a C function, where a full context switch is required. Umbra's code cache size on average is about 4 times that of the base DynamoRIO. In contrast, inlining all instrumented code instead of using a 3-stage layout would result in a more than 100 times code expansion.

4.6 Example Tool

We used Umbra to build a sample shadow value tool called *SDRD*, or Shared Data Reference Detector. This tool identifies which memory is referenced by which threads. We designed the shadow metadata as a bitmap, representing each thread with a different bit. On every application memory access, Umbra translates the application memory address into a shadow memory address and passes it to SDRD. By setting the appropriate bit for the current thread in the shadow metadata, SDRD is able to tell which threads have accessed this data. We use a 4-byte application memory to 4-byte shadow metadata mapping scheme (4B-to-4B), so we are able to keep track of up to 32 threads per 32-bit word accessed. If the application access size is smaller than 4 bytes we align it to 4 bytes, resulting in a single-word granularity.

The implementation of SDRD using Umbra is straightforward. We fill the `umbra_client_t` data structure with appropriate values as shown below:

```
num_regs = 1;  /* 1 scratch register required */
app_size = 32; /* 4-byte application data    */
shd_size = 32; /* 4-byte metadata            */
```

SDRD's `instrument_update` function inserts 3 metadata update instructions for every application memory reference, as shown in Figure 7. `reg` is the register provided by Umbra that will point to the metadata address during execution. `thread_bit` is a thread private variable that holds a bitmap with only one bit set to represent the thread itself. This bitmap can be a constant when using DynamoRIO's thread-private code caches. The first metadata update instruction is a `test` instruction [3] that checks via the metadata pointed at by `reg` whether the current thread has accessed the application data being accessed. If it has not, the metadata is updated using an atomic `or` operation. If the thread has already accessed this application data, the metata write is avoided. As shown below, the cost of the check is significantly less than the cost of performing a metadata write every time.

```
test [reg], thread_bit
jnz  skip_update
or   [reg], thread_bit => [reg]
skip_update:
    ...
```

Figure 7. Instrument metadata update code for SDRD.

In addition to `instrument_update`, SDRD also implements a callback function for the event `shadow_memory_delete` in order to report which data has been accessed by which thread when the memory is de-allocated.

We evaluate the performance of SDRD using the SPLASH-2 benchmarks with 8 threads on our 8-core system. As shown in Figure 8, Umbra works well on multi-threaded applications. Umbra by itself causes a 3x slowdown, which is consistent with the slowdown measured from the single-threaded SPEC benchmarks.

[3] The `test` instruction computes the bit-wise logical AND of two operands and sets the conditional flags according to the result. The result is then discarded.
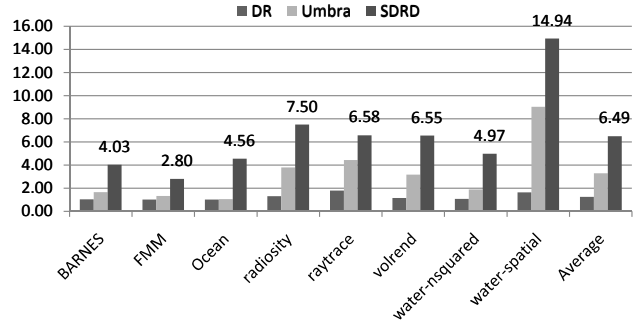

Figure 8. Performance of base DynamoRIO (DR), shadow metadata mapping (Umbra), and our Shared Data Reference Detector shadow value tool (SDRD) performing metadata updates.

The metadata updating by SDRD incurs another 3x slowdown. The water-spatial benchmark shows a larger slowdown because its running time is too short (< 0.3 second) for Umbra's initialization overhead to be amortized. We found that using the test-and-set approach shown is much faster than directly updating the metadata without testing, which incurs an average $30\times$ slowdown. This is primarily because the metadata update can easily cause expensive cache conherence maintenance and memory bus saturation.

5. Related Work

Existing shadow value tools employ shadow metadata mapping schemes consisting typically of either one or two levels of translation. When using one level of translation, the full user address space is mapped into a single shadow address space. This simplifies translation, requiring only an offset and potentially a scale if the shadow metadata size does not match its corresponding application size. However, using a single shadow region sacrifices robustness, as it requires stealing a large chunk of space from the application.

TaintTrace [5], Hobbes [3], and Eraser [23] all use one-level translation with one shadow byte per application byte. They assume a 3GB 32-bit user address space and take 1.5GB for shadow memory. Their shadow metadata mapping involves a simple offset and incurs little overhead. However, claiming a full half of the address space gives up flexibility and presents problems supporting applications that make assumptions about their address space layout. Such a design is problematic on operating systems that force various structures to live in certain parts of the address space or use different address space splits for kernel versus user space.

LIFT [20] uses one-level translation, but shadows each application byte with only one shadow bit. Consequently its mapping uses both a scale and an offset, and its shadow region only requires one-eighth of the user address space.

Several shadow value tools, like Umbra, use two-level translation schemes for flexibility. Using two levels gives up some performance but provides support for a wider range of applications and platforms. Unlike Umbra, other tools map the entire address space uniformly, rather than mapping regions based on application memory allocation.

MemCheck [24] employs a two-level translation scheme [16]. Memcheck's scheme was designed for a 32-bit address space. It splits the space into 64K regions of 64KB each. A first-level table points at the shadow memory for the 64KB region containing the address in question. Memcheck originally kept all of its shadow memory in a single contiguous region but was forced to split it up in order to support a wider range of applications and platforms,

due to the limitations discussed earlier with claiming too large of a contiguous fraction of the application address space.

Memcheck uses several optimizations to reduce overhead, but most of them are specific to Memcheck's particular metadata semantics. It saves memory and time by pointing shadow memory regions that are filled with a single metadata value to a shared shadow memory structure. For aligned memory accesses it processes all bytes in a word simultaneously. And it maintains bit-level shadowing granularity without requiring shadow bits for every application bit by compressing the shadow metadata to only use such granularity when byte-level granularity is not sufficient.

Memcheck extends its scheme to 64-bit address spaces with a larger first-level table that supports the bottom 32GB of the address space. It uses a slower translation path for addresses above 32GB, and attempts to keep as much memory as possible in the lower 32GB. The Memcheck authors report problems with their approach on other platforms and suggest it may need improvement [16]: "It is unclear how this shadow memory scheme can best be scaled to 64-bit address spaces, so this remains an open research question for the future."

The TaintCheck [18], Helgrind [12], and Redux [15] tools are all built on the same Valgrind [17] dynamic binary instrumentation platform as Memcheck. They all use the same two-level translation scheme as Memcheck.

pinSel [14] uses a two-level translation scheme similar to Memcheck's, but with 4KB shadow units rather than 64KB units. VisualThreads [9] uses 16MB units in its two-level approach.

DRD [22] uses a nine-level table to hold its shadow memory, which shadows memory accessed during each unit of time.

Commercial shadow value tools include Purify [21], Intel Parallel Inspector [11], Insure++ [19], and Third Degree [10]. Unfortunately, their shadow translation details are not published.

EDDI [29] shadows each memory page with a shadow page that stores for each application byte whether a data watchpoint has been set. A table is used to locate the shadow page for each memory page, with multiple levels used for 64-bit.

MemTracker [27] and HARD [30] propose using additional hardware to provide low-overhead shadow value tools: memory access monitoring (but not propagation) for MemTracker, and data race detection for HARD. The introduced hardware is targeted to a specific tool in each case.

Metadata management and propagation directly in hardware [7, 8, 26] imposes limitations on the metadata format but can reduce overheads significantly for tools that can use the supported formats. Other hardware proposals support a wider range of shadow value tools by targeting the costs of dynamic binary instrumentation [6, 31] or providing metadata support independently of the metadata structure [4].

Umbra is implemented entirely in software using the DynamoRIO [2] dynamic binary instrumentation system. It could be implemented using other binary instrumentation systems such as Pin [13] or Valgrind [17].

6. Conclusion

In this paper we presented Umbra, the first shadow memory mapping scheme that supports both 32-bit and full 64-bit address spaces efficiently. This flexible and scalable approach does not rely on any specific operating system or architectural features or specific shadow metadata sizes or semantics. We have described several novel optimizations that improve the speed of Umbra's shadow metadata mapping and detailed the contributions of each optimization.

This paper focused on efficient shadow metadata mapping. Future work includes providing a flexible interface for shadow metadata updating to allow building a wide range of tools with our framework. We are also continuing to improve the mapping performance of Umbra.

We have implemented and evaluated Umbra and shown that it is three times faster than the most widely-used shadow value framework today, Valgrind. We hope that by reducing the prohibitive overhead of shadow value tools we can increase the frequency with which these powerful tools can be deployed.

References

[1] DynamoRIO dynamic instrumentation tool platform, February 2009. http://dynamorio.org/.

[2] Derek Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, M.I.T., September 2004.

[3] Michael Burrows, Stephen N. Freund, and Janet L. Wiener. Run-time type checking for binary programs. In *Proc. of the 12th International Conference on Compiler Construction (CC '03)*, pages 90–105, 2003.

[4] Shimin Chen, Michael Kozuch, Theodoros Strigkos, Babak Falsafi, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Olatunji Ruwase, Michael Ryan, and Evangelos Vlachos. Flexible hardware acceleration for instruction-grain program monitoring. In *Proc. of the 35th International Symposium on Computer Architecture (ISCA '08)*, pages 377–388, 2008.

[5] Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. Tainttrace: Efficient flow tracing with dynamic binary rewriting. In *Proc. of the Proceedings of the 11th IEEE Symposium on Computers and Communications (ISCC '06)*, pages 749–754, 2006.

[6] Marc L. Corliss, E. Christopher Lewis, and Amir Roth. Dise: a programmable macro engine for customizing applications. In *Proc. of the 30th International Symposium on Computer Architecture (ISCA '03)*, pages 362–373, 2003.

[7] Jedidiah R. Crandall and Frederic T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Proc. of the 37th International Symposium on Microarchitecture (MICRO 37)*, pages 221–232, 2004.

[8] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: a flexible information flow architecture for software security. In *Proc. of the 34th International Symposium on Computer architecture (ISCA '07)*, pages 482–493, 2007.

[9] Jerry J. Harrow. Runtime checking of multithreaded applications with visual threads. In *Proc. of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 331–342, 2000.

[10] Hewlett-Packard. Third Degree. http://h30097.www3.hp.com/developerstoolkit/tools.html.

[11] Intel. Intel Parallel Inspector. http://software.intel.com/en-us/intel-parallel-inspector/.

[12] OpenWorks LLP. Helgrind: A data race detector, 2007. http://valgrind.org/docs/manual/hg-manual.html/.

[13] 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 *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*, pages 190–200, June 2005.

[14] Satish Narayanasamy, Cristiano Pereira, Harish Patil, Robert Cohn, and Brad Calder. Automatic logging of operating system effects to guide application-level architecture simulation. In *Proc. of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '06/Performance '06)*, pages 216–227, 2006.

[15] Nicholas Nethercote and Alan Mycroft. Redux: A dynamic dataflow tracer. In *Electronic Notes in Theoretical Computer Science*, volume 89, 2003.

[16] Nicholas Nethercote and Julian Seward. How to shadow every byte of memory used by a program. In *Proc. of the 3rd International Conference on Virtual Execution Environments (VEE '07)*, pages 65–74, June 2007.

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

[18] James Newsome. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. of the Network and Distributed System Security Symposium (NDSS 2005)*, 2005.

[19] Parasoft. Insure++. http://www.parasoft.com/jsp/products/insure.jsp?itemId=63.

[20] 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 *Proc. of the 39th International Symposium on Microarchitecture (MICRO 39)*, pages 135–148, 2006.

[21] Rational Software. Purify: Fast detection of memory leaks and access errors, 2000. http://www.rationalsoftware.com/products/whitepapers/319.jsp.

[22] Michiel Ronsse, Bastiaan Stougie, Jonas Maebe, Frank Cornelis, and Koen De Bosschere. An efficient data race detector backend for diota. In *Parallel Computing: Software Technology, Algorithms, Architectures & Applications*, volume 13, pages 39–46. Elsevier, 2 2004.

[23] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[24] Julian Seward and Nicholas Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *Proc. of the USENIX Annual Technical Conference*, pages 2–2, 2005.

[25] Standard Performance Evaluation Corporation. SPEC CPU2006 benchmark suite, 2006. http://www.spec.org/osg/cpu2006/.

[26] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *Proc. of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '04)*, pages 85–96, 2004.

[27] Guru Venkataramani, Brandyn Roemer, Yan Solihin, and Milos Prvulovic. Memtracker: Efficient and programmable support for memory access monitoring and debugging. In *Proc. of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA '07)*, pages 273–284, 2007.

[28] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proc. of the 22nd International Symposium on Computer Architecture (ISCA '95)*, pages 24–36, 1995.

[29] Qin Zhao, Rodric M. Rabbah, Saman P. Amarasinghe, Larry Rudolph, and Weng-Fai Wong. How to do a million watchpoints: Efficient debugging using dynamic instrumentation. In *Proc. of the 17th International Conference on Compiler Construction (CC '08)*, pages 147–162, 2008.

[30] Pin Zhou, Radu Teodorescu, and Yuanyuan Zhou. Hard: Hardware-assisted lockset-based race detection. In *Proc. of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA '07)*, pages 121–132, 2007.

[31] Yuanyuan Zhou, Pin Zhou, Feng Qin, Wei Liu, and Josep Torrellas. Efficient and flexible architectural support for dynamic monitoring. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2(1):3–33, 2005.