MemTracker: Efficient and Programmable Support for Memory Access Monitoring and Debugging *

Guru Venkataramani
Georgia Tech
guru@cc.gatech.edu

Brandyn Roemer
Georgia Tech
gtg538y@mail.gatech.edu

Yan Solihin
North Carolina State University
solihin@ece.ncsu.edu

Milos Prvulovic
Georgia Tech
milos@cc.gatech.edu

Abstract

Memory bugs are a broad class of bugs that is becoming increasingly common with increasing software complexity, and many of these bugs are also security vulnerabilities. Unfortunately, existing software and even hardware approaches for finding and identifying memory bugs have considerable performance overheads, target only a narrow class of bugs, are costly to implement, or use computational resources inefficiently.

This paper describes MemTracker, a new hardware support mechanism that can be configured to perform different kinds of memory access monitoring tasks. MemTracker associates each word of data in memory with a few bits of state, and uses a programmable state transition table to react to different events that can affect this state. The number of state bits per word, the events to which MemTracker reacts, and the transition table are all fully programmable. MemTracker's rich set of states, events, and transitions can be used to implement different monitoring and debugging checkers with minimal performance overheads, even when frequent state updates are needed. To evaluate MemTracker, we map three different checkers onto it, as well as a checker that combines all three. For the most demanding (combined) checker, we observe performance overheads of only 2.7% on average and 4.8% worst-case on SPEC 2000 applications. Such low overheads allow continuous (always-on) use of MemTracker-enabled checkers even in production runs.

1. Introduction

Technological advancement has resulted in a decades-long exponential growth in computing speeds, which are being utilized by increasingly complex software. As a result, software is increasingly prone to programming errors (bugs) and many of these bugs also create security vulnerabilities. Unfortunately, architectural support for software monitoring and debugging has not kept pace with software complexity, and programmers and users still rely on software-only tools for many critical monitoring and debugging tasks. For many of these tasks, performance overheads of such tools are prohibitive, especially in post-deployment monitoring of live (production) runs where end users are directly affected by the overheads of the monitoring scheme.

One particularly important and broad class of programming errors is erroneous use or management of memory (memory bugs). This class of errors includes pointer arithmetic errors, use of dangling pointers, reads from uninitialized locations, out-of-bounds accesses (e.g. buffer overflows), memory leaks, etc. Many software tools have been developed to detect some of these errors. For example, Purify [8] and Valgrind [13] detect memory leaks, accesses to unallocated memory, reads from uninitialized memory, and some dangling pointer and out-of-bounds accesses. Some memory-related errors, such as buffer overflows, can also be security vulnerabilities. As a result, security tools often focus on detection of such errors or specific error manifestation scenarios. For example, StackGuard [3] detects buffer overflows that attempt to modify a return address on the stack.

Unfortunately, software-only detection of memory errors typically incurs a very high performance overhead, because the detection tool has to *intercept* memory accesses (loads and/or stores). For each access, the checker must find the status of accessed memory location (*state lookup*), e.g. to find whether the location is allocated, initialized, stores a return address, etc. The checker then verifies if the access is allowed by the location's status (*state check*), and possibly changes the status of the memory location for future accesses (*state update*), for example to mark an uninitialized location as initialized if the access is a write. Since memory read and write instructions are executed frequently, the overhead of intercepting and checking them is very high. For example, slowdowns of 2X to 30X (i.e., up to 30 *times*) have been reported for Valgrind [10, 19].

Architectural support has been proposed to reduce performance overheads for detecting some memory-related problems [4, 11, 14, 21, 22, 23, 24]. In general, these schemes allow loads and stores to be intercepted directly without inserting *instrumentation* instructions around them. After an access is intercepted, a checker still needs to perform a *state check* and possibly a *state update* caused by the access. While load/store interception can easily be done by hardware, there is less agreement on whether state checks and updates should be performed in hardware. The main dif-

*This work was supported, in part, by the National Science Foundation under grants CCF-0429802, CCF-0447783, CCF-0541080, CCF-034725, CCF-0541108. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

ficulty is how to perform checks and updates in hardware while not binding the hardware design to a specific meaning (semantics) of each state. One existing approach to state checks and updates is to hardwire the meaning of each state for a specific checker [4, 22]. Another approach is to perform only interception in hardware, and make software handle the state checks and updates [2]. Finally, a number of existing approaches express state as *access permissions* or *monitored regions*. Such state can be quickly checked in hardware to determine whether the access can be allowed to proceed without any additional checker activity. If a state update (or additional checking) is needed, a software handler is invoked [14, 21, 23, 24]. Overall, existing architecture support is either 1) hardwired for a particular checker, or 2) requires software intervention for *every state update*. This can lead to significant performance degradation when state updates are frequent in some checkers.

As a consequence, there is a *dilemma* in designing architecture support for tracking memory access behavior: to sacrifice performance by designing hardware support that is not checker specific, or to sacrifice generality by hard-wiring specific checks for performance. Unfortunately, both alternatives are unappealing: users are reluctant to enable memory access tracking mechanisms that have significant performance overheads, while architecture designers are unwilling to implement hardware that is checker-specific. To overcome this, this paper proposes a new hardware mechanism that can *perform interception, state checks, and state updates in hardware*, but still remains *generic* and is *not hard-wired* for any particular checker. We show that some very useful checkers require frequent and fine-grain state updates, which benefit greatly from hardware state checks and updates. For example, the checkers we use in our experiments need frequent fine-grain state updates to keep track of which locations are initialized and which locations contain return addresses. For such checkers, a scheme is needed that can 1) avoid software intervention for most state updates, 2) support efficient state checks and updates even when neighboring locations have different states, and 3) be programmable enough to support different kinds of checks.

Our hardware mechanism, which we call MemTracker, is essentially a programmable state machine. It associates each memory word with a *state* and treats each memory action as an *event*. States of data memory locations in the application's virtual address space are stored as an array in a separate and protected region of the application's virtual address space. Each event results in looking up the state of the target memory location, checking if the event is allowed given the current state of the location, and possibly raising an exception or changing the state of the location. To control state checking and updates, MemTracker uses a *programmable state transition table* (PSTT).

The PSTT acts as an interface that allows a checker, or even multiple checkers combined, to specify how states should be managed by the hardware. When an event targets a memory word, the word's current state and the type of event (read, write, etc.) are used to look up a PSTT entry. Each entry in the PSTT specifies the next state for the word and indicates whether or not an exception should be raised. MemTracker events are load/store instructions and and special *user event* instructions that applications and runtime libraries can use to inform MemTracker of high-level actions such as allocations, deallocations, or other changes in how a location will or should be used. By changing the contents of the PSTT, the meaning of each user event and state can be customized for a specific checker. As a result, the same MemTracker hardware can efficiently support different detection, monitoring, and debugging tasks.

Any highly programmable mechanism, such as MemTracker, has a wide spectrum of potential uses that can not be evaluated or tested exhaustively. Instead, such mechanisms are typically evaluated using a small set of familiar and popular uses that are thought to be representative examples of the broader universe of potential uses.

We follow the same evaluation approach with MemTracker, and use a set of familiar and well-known detection, monitoring, and debugging tasks to represent potential uses of our mechanism. These tasks include 1) heap memory access checking similar to that used in Purify [8], Valgrind [13] and HeapMon [14], 2) detection of malicious or accidental overwrites of return addresses on the stack, and 3) detection of heap-based sequential buffer overflow attacks [1, 16, 17, 18] and errors, and 4) a checker that combines the functionality of all three checkers listed above.

We find that, even for the combined checker, MemTracker's performance overhead is only 2.7% on average and 4.8% worst-case across SPEC CPU 2000 [15] applications, relative to a system that runs the same applications without any checking and without MemTracker support.

The rest of this paper is organized as follows: Section 2 discusses related work, Section 3 presents an overview of our MemTracker mechanism, Section 4 presents some hardware implementation details, Section 5 presents the setup for our experimental evaluation, Section 6 presents our experimental results, and Section 7 summarizes our findings.

2 Related Work

The most general of previously proposed hardware mechanisms is DISE [2] (Dynamic Instruction Stream Editing), which pattern-matches decoded instructions against templates and can replace these instructions with parameterized code. For memory access checking, DISE provides efficient interception that allows instrumentation to be injected into the processor's instruction stream. In contrast to DISE, MemTracker does not modify the performance-critical front-end of the pipeline, and it performs load/store checks without adding dynamic instructions to execution.

Horizon [9] widens each memory location by six bits, two with hard-wired functionality and four trap bits that can intercept different flavors of memory accesses. Mondrian Mem-

ory Protection [21] has per-word permissions that can intercept read, write, or all accesses. iWatcher [23] provides enhanced watchpoint support for multiple regions of memory and can intercept read, write, or all accesses to such a region. HeapMon [14] intercepts accesses to a region of memory and uses word-granularity filter bits to specify locations whose accesses should be ignored, with the checker implemented as a helper thread. In all these schemes, state updates (and many state checks) require software intervention. As shown in Section 6.4, in some useful checkers such state changes are numerous enough to to cause significant overheads.

To avoid hard-wiring of the states but still provide efficient checks and updates, MemTracker uses a flat (array) state structure in memory. In contrast, Mondrian [21] uses a sophisticated trie structure to minimize state storage overheads for coarse-grain state, at the cost of more complex fine-grain state updates. iWatcher [23] keeps track of ranges of data locations with the same state, which also complicates fine-grain updates. Horizon [9] simplifies state lookups by keeping state in six extra bits added to each memory location, which requires non-standard memory modules and adds a state storage cost even when no checks are actually needed. In contrast, MemTracker keeps state information separately in ordinary memory, and uses only as much state as needed. In particular, when checking is not used, there is no memory allocated for MemTracker state.

Other related work includes AccMon [22], Dynamic information flow tracking [6], Minos [4], Memory centric security [20], LIFT [5], and SafeMem [11]. AccMon uses "golden" (correct) runs to learn which instructions should access which locations, then checks this at runtime using Bloom filters to avoid unnecessary invocations of checker code; Dynamic information flow tracking and Minos add one *integrity* bit to each location to track whether the location's value is from an untrusted source; SafeMem scrambles existing ECC bits to trigger exceptions when un-allocated locations are accessed and to help garbage-collection. LIFT does dynamic software instrumentation and relies heavily on optimized code to be inserted for checks. Memory Centric architecture associates security attributes to memory instead of individual user process. Most of the above mechanisms are designed with specific checks in mind: in AccMon, much of the hardware is specific to its heuristic-based checking scheme; in Dynamic information flow tracking and Minos, the extra bit tracks only the integrity status of the location; SafeMem cannot track per-word state and can only intercept accesses to blocks with no useful values – a block with useful values needs a valid ECC to protect its values from errors.

Overall, unlike prior mechanisms MemTracker is unique in that it can efficiently support different checkers, even those that require simple but frequent state updates automatically without software intervention. It should be noted, however, that MemTracker cannot automatically handle state checks and updates that cannot be expressed in its programmable state transition table. Such checks still require software intervention even in MemTracker. However, MemTracker may still provide a performance advantage because its state machine can act as a more sophisticated filter for triggering software interventions only when needed.

3. MemTracker Overview

To provide context and a motivating example for our discussion of MemTracker, we first describe an example checker. We then describe our MemTracker mechanism and how it can be used to implement the example checker.

3.1. HeapData: an Example Memory Access Checker

Many typical programming mistakes, such as use of dangling pointers or neglecting to initialize a variable, are manifested through accesses to unallocated memory locations or to loads from uninitialized locations. Detection of such accesses is one of the key benefits of tools such as Purify [8] and Valgrind [13], and has also been used to evaluate hardware support for runtime checking of memory accesses in HeapMon [14]. To help explain our new MemTracker support, we will use *HeapData*, an example checker that is functionally similar to the checker used in HeapMon. This checker tracks the allocation and initialization status of each word in the heap region using three states: *Unallocated*, *Uninitialized*, and *Initialized*. The state transitions for this checker is shown in Figure 1. All words in the heap area start in the *Unallocated* state. When a block of *Unallocated* memory words is allocated (e.g. through `malloc()`), the state of each word changes to *Uninitialized*. The first write (e.g. using a store instruction) to an *Uninitialized* word changes its state to *Initialized*. The word then remains in the *Initialized* state until it is deallocated (e.g. through `free()`), at which time it changes back to the *Unallocated* state.

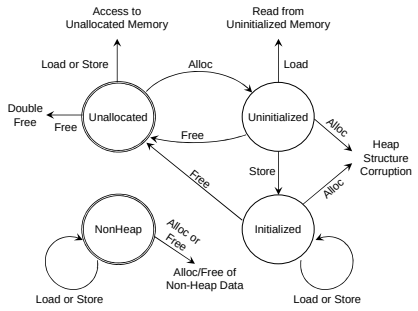

Figure 1. Transition diagram for the HeapData checker.

Only an *Initialized* word can be read, and writes are allowed only to *Uninitialized* or *Initialized* words. Memory allocation is valid only if all allocated words are in the *Unallocated* state, and deallocation is valid only if all deallocated words are either *Uninitialized* or *Initialized*. All other reads, writes, allocations, and deallocations are treated as errors and should result in invoking a software error handler.

In addition to these three states from HeapMon [14], our *HeapData* checker has a *NonHeap* state which is used for

all data outside the heap region. This new state allows us to treat all memory accesses in the same way, without using an address-range filter [14] to identify only heap accesses. The *NonHeap* state permits loads and stores, but prohibits heap allocations and deallocations.

Overall, our example *HeapData* checker reacts to four different kinds of events (loads, stores, allocations, and deallocations), and keeps each word in the application's data memory in one of four different states.

We note that HeapData and other checkers in this paper are used only to illustrate how MemTracker can be used and to evaluate MemTracker's performance. We do not consider the checkers themselves as our contribution, but rather as familiar and useful examples that help us demonstrate and evaluate our real contribution – the MemTracker mechanism.

3.2. MemTracker Functionality

For each word of data, MemTracker keeps an entry that consists of a few bits of state. These state entries are kept as a simple packed array in main memory, where consecutive state entries correspond to consecutive words of data. This packed array of state entries is stored in the application's virtual address space, but in a separately mapped (e.g. via `mmap`) region. This approach avoids custom storage for state and benefits from existing address translation and virtual memory mechanisms. Keeping state separate from other regions allows us to change its access permissions (e.g. via `mprotect`) so that only MemTracker mechanisms can access (e.g. update) state entries. Regular (data) load/store instructions can not access this area, which prevents accidental (or malicious) modifications of the state array using ordinary store instructions.

When an event (e.g. a load) targets a memory location, the state entry for that location can be found using simple logic (see Section 4.2). Our MemTracker mechanism reads the current state of the memory location and uses it, together with the type of the event, to find the corresponding transition entry in the *programmable state transition table* (PSTT). Each entry in PSTT specifies the new state for the memory location, and also indicates whether to trigger an exception. Transition entries in the PSTT can be modified by software, allowing MemTracker to implement different checkers.

3.3. MemTracker Events

MemTracker treats existing load and store instructions as events that trigger state lookups, checks, and updates. Other events, such as memory allocation and deallocation in HeapData, should also be able to affect MemTracker state. However, these high-level events are difficult to identify at the level of the hardware and differ from checker to checker. To support these events effectively, we extend the ISA with a small number of *user event* instructions. User event instructions can be used in the code to "annotate" high-level activity in the application and library code. The sole purpose of these instructions is to be MemTracker events. These instructions only result in MemTracker state lookups, checks, and updates, and do not actually access the data associated with that state. The number of different user level instructions supported in the ISA is a tradeoff between PSTT size (which grows in proportion to the number of events) and sophistication of checkers that can be implemented (more user event instructions allow more sophisticated checkers). In this paper, we model MemTracker with 32 user event instructions, which is considerably more than what is actually needed for the checkers used in our experimental evaluation.

In terms of the number of affected memory locations, MemTracker must deal with two kinds of events: constant-footprint and variable-footprint events. An example of a constant-footprint event is a load from memory, where the number of accessed bytes is determined by the instruction's opcode. The handling of these events is straightforward: the state for the affected word or set of words (e.g. for a double-word load) is looked up, the transition table is accessed, and the state is updated if there is a change. An example of a variable-footprint event is memory allocation, in which a variable and potentially large number of locations can be affected. We note that variable-footprint events can be dynamically replaced by loops of constant-footprint events, either through binary rewriting or in the processor itself in a way similar to converting x86 string instructions into μops [7].

In this paper we use a RISC processor without variable-footprint (e.g. string) load and store instructions, but we provide support for variable-footprint user events to simplify implementation of checkers. Instructions for variable-footprint user events have two source register operands, one for the base address and the other for the size of the target memory address range. This implementation allows simpler checker implementations, and avoids code size increase and software overheads of looping. However, in our simulation, a variable-footprint event is treated as a series of constant-footprint events with each accessing one word at a time. Hence, the overheads due to variable-footprint events are fully accounted. In our MemTracker implementation for this paper, 16 of our 32 user event instructions are variable-footprint, and the rest are word-sized constant-footprint events (sub-word events are discussed in Section 3.5). Note that we only need two variable-footprint and five constant-footprint user events to simultaneously support all checkers used in our evaluation. The remaining user events are there to provide support for more sophisticated checkers in the future.

3.4. MemTracker Setup for the HeapData Checker

To implement the *HeapData* checker from Section 3.1, we use two variable-footprint user event instructions, UEVT0 for allocations and UEVT1 for deallocations. We instrument the user-level memory allocation library to execute UEVT0 at the end of the allocation function, and UEVT1 at the start of the deallocation function. Figure 2 shows the PSTT configuration for this checker, which is a tabular equivalent of

states and transitions described in Section 3.1, except for the sub-word LD/ST events, which we discuss in Section 3.5.

State \ Event	UEVT0 (Alloc)	UEVT1 (Free)	LD	ST	Sub-word LD	Sub-word ST
0 (NonHeap)	0 E	0 E	0	0	0	0
1 (Unalloc)	2	1 E	1 E	1 E	1 E	1 E
2 (Uninit)	2 E	1	2 E	3	2 E	2 or 3
3 (Init)	3 E	1	3	3	3	3

Figure 2. State transition table for our example HeapData checker. Entries with "E" trigger exceptions.

It should be noted that the need to modify the allocation library is not specific to MemTracker – all tools or mechanisms that track allocation status of memory locations require some instrumentation of the memory management library to capture allocation and deallocation activity. Compared to software-only tools that perform such checks, MemTracker-based implementation has the advantage of eliminating the instrumentation of load/store memory accesses and the associated performance overhead. Even for applications that frequently perform memory allocations and deallocations, the number of loads/stores still easily exceeds the number of allocations and deallocations[1], and hence even such applications benefit considerably from MemTracker.

3.5. Dealing with Sub-Word Accesses

MemTracker keeps state only for entire words, so sub-word memory accesses represent a problem. For example, consider the shaded transition entry in Figure 2, which corresponds to a sub-word store to an *Uninitialized* word. Since the access actually initializes only part (e.g. the first byte) of the word, we could leave the word in state 2 (*Uninitialized*). However, a read from the same part of the word (first byte) is then falsely detected as a read from uninitialized memory. Conversely, if we change the state of the word to 3 (*Initialized*), a read from another part of the word (e.g. the last byte) is not detected as a problem, although it is in fact reading an uninitialized memory location.

In this tradeoff between detecting problems and avoiding false problems, the right choice depends on the circumstances and the checker. To allow flexibility in implementing checkers, sub-word load/stores are treated as separate event types in the PSTT. This allows us to achieve the desired behavior for sub-word accesses. For example, during debugging we can program the PSTT of the HeapData checker (Figure 2) such that sub-word stores to uninitialized data leave the state of the word as uninitialized to detect all reads from uninitialized locations. In live runs, we can avoid false positives by programming the PSTT to treat a sub-word write as an initialization of the entire word.

[1]In fact, several load/store instructions are executed during each heap allocation and deallocation

3.6. MemTracker Event Masking

It may be difficult to anticipate at compile time which checkers will be needed when the application is used. Therefore, it would be very useful to be able to switch different checkers on and off, depending on the situation in which the application runs. To achieve that, we can generate code with user events for several different checkers, and then provide a way to efficiently ignore events used for disabled checkers. This would also eliminate nearly all overheads when all checking is disabled.

To ignore load and store MemTracker events when checking is not used, we can set up the PSTT to generate no exceptions and no state changes. User events can be similarly neutralized, e.g. to turn the checker off without removing instrumentation for it. However, state and PSTT lookups would still affect performance and consume energy. To minimize this effect, we add an *event mask register* that has one bit for each type of event. If load and/or store events are masked out, loads and stores are performed without state and PSTT lookups. A masked-out user event instruction becomes a no-op, consuming only fetch and decode bandwidth.

4. MemTracker Implementation

4.1. Programmable State Transition Table (PSTT)

The size of the entire PSTT depends on the maximum number of states and events that are supported. The maximum number of states and events supported reflect a tradeoff between hardware complexity and checker sophistication. In this paper we describe a simple implementation of MemTracker, which supports up to 16 states and up to 36 events (load, store, sub-word load, sub-word store, and 32 user events). The resulting PSTT has 576 entries with 5 bits in each entry (4 next-state bits and one exception bit), for a total PSTT size of 360 bytes. This small PSTT easily fits on-chip, allows one-cycle lookups, and consumes little energy.

The combination of all checkers in our experiments only needs seven states, so providing 16 states is more than enough. However, if more states and/or event types are needed in the future, the PSTT may become too large to be kept on-chip, or too slow to be accessed with direct lookups. To accommodate such a large PSTT, we can accommodate frequently used entries in a small on-chip PSTT cache, while keeping other entries in memory. However, such a design is beyond the scope of this paper.

4.2. Finding State Information

Different checkers that use MemTracker support need different numbers of state bits per word, so we provide support to select the number of state bits at runtime. In particular, we add a *MemTracker Control Register* (MTCR), which allows selection of the number of state bits per word. MTCR specifies the number of bits of state corresponding to each word of data, and only power-of-two state sizes (1, 2, or 4) are supported so that the state-lookup logic is simple.

State is stored in memory starting at the virtual address specified in the *State Base Address Register* (SBAR), and address of the state of a given data address can be found quickly by adding the SBAR with selected bits of the data address using simple indexing functions. An example lookup of 2-bit state for data address 0xABCD is shown in Figure 3.

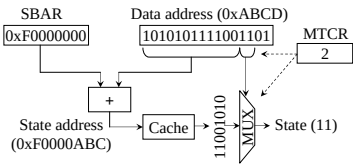

Figure 3. Lookup of 2-bit state for data location 0xABCD.

4.3. Caching State Information

There are three basic ways to cache state information: *split caching* (state blocks in a separate cache, Figure 4(a)), *shared caching* (state blocks share the same cache with data, Figure 4(b)), and *interleaved caching* (state bits stored with the cache line that has the corresponding data, Figure 4(c)). Shared caching has the advantage of using existing on-chip caches. However, in shared caching state competes with data for cache space, and lookups and updates compete with data accesses for cache bandwidth. Split caching has the advantage that it leaves the data cache unmodified, but adds extra cost for the separate state cache. Finally, interleaved caching allows one cache access to service both a data load and the lookup of the corresponding state; similarly, a data store and a state update can be performed in the same cache access. However, unlike the other two caching approaches, interleaved caching makes each cache block larger (to hold the maximum-size state for the block's data) and may slow down the cache even when state is not used.

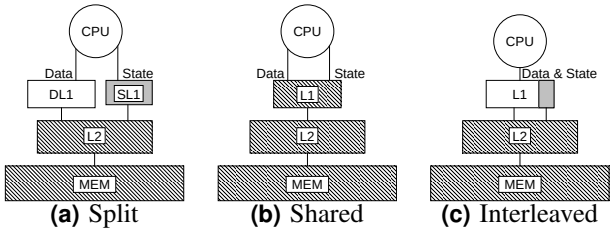

Figure 4. Caching approaches for MemTracker state in the L1 cache. For L2 and below we always use Shared.

We find that the simple and inexpensive shared caching approach works well for non-primary caches. State is considerably smaller than the corresponding data, so the relatively few state blocks cause insignificant interference with data caching in large caches (L2 and beyond). Additionally, L1 caches act as excellent "filters" for the L2 cache, so state accesses add little contention to data block accesses. As a result, the choice of caching approach mainly applies to primary (L1) caches, and all three L1 caching approaches are examined in our experimental evaluation (Section 6). We find that shared L1 caching performs poorly without expensive additional cache ports. Interleaved L1 caching also performs poorly without additional cache ports, but it simplifies some memory consistency issues (Section 4.5) and may still be good choice in chip-multiprocessors. Finally, split caching has excellent performance even with a very small and simple state cache (2KBytes in our experiments), and performance-wise is the best choice.

4.4. Processor Modifications for MemTracker

MemTracker integration into the processor pipeline is simpler for Split and Shared L1 state caching approaches, where state lookups can be delayed until the end of the pipeline. This allows MemTracker to be added as an in-order extension to the commit stage of the pipeline, avoiding any significant changes to the complex out-of-order engine of the processor (Figure 5(a)).

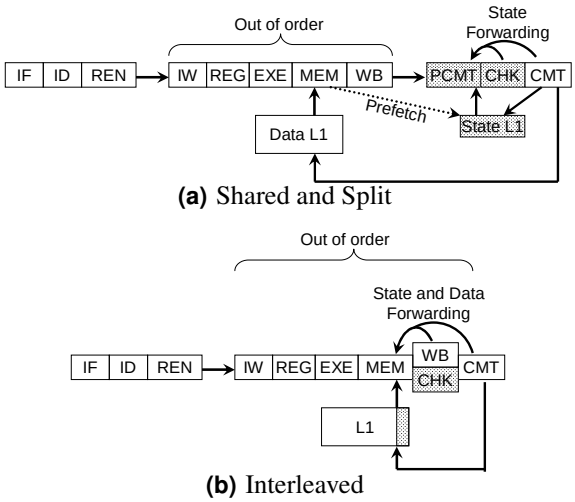

Figure 5. Processor pipeline with MemTracker support (shaded) for different L1 state caching approaches.

In a regular processor (no MemTracker support), the commit logic of the processor checks the oldest instructions in the ROB and commits the completed ones in order (from oldest to youngest). If an instruction is a store, its commit initiates the cache write access. These writes are delayed until commit because cached data reflects the architectural state of memory and should not be polluted with speculative values.

MemTracker adds two additional pipeline stages just before the commit (Figure 5(a)). The first of these stages is the *pre-commit stage* (PCMT), which checks the oldest few instructions in the ROB and lets the completed instructions to proceed in-order into the next pipeline stage. For MemTracker events (loads, stores, and user events) pre-commit also fetches MemTracker state from the state cache. In the second MemTracker pipeline stage (*check stage* or CHK), the state and the event type are used to look up the corresponding PSTT entry. If the state is not available (state cache miss),

the instruction stalls in the check stage until the state arrives. Because the pipeline from pre-commit to actual commit is in-order, such a stall prevents all other instructions from proceeding to the commit stage.

If the PSTT indicates that an exception should be raised, the processor behaves in the same way as for any other exception: the instruction and all younger ones are flushed from the pipeline and the processor begins to fetch instructions from the exception handler. If there is no exception, the instruction proceeds to the commit stage. If the new state from the PSTT is different from the current state, the state is written to the state L1 cache at commit, at the same point when stores normally deposit their data values to the data cache.

State checks in the check stage can have dependences on still-uncommitted state modifications, so a small state forwarding queue is used to correctly handle such dependence. This is similar to the "regular" store queue which forward store values to dependent loads, but our state forwarding queue is much simpler because 1) it only tracks state updates in the two stages between pre-commit and commit, so in a 4-wide processor we need at most 8 entries, and 2) all addresses are already resolved when instructions enter this queue, so dependences can always be precisely identified.

Because a state cache miss in the pre-commit stage almost directly delays the instruction commit, we initiate a state prefetch when the data address is resolved. Contention for state cache ports with state lookups in the pre-commit stage is avoided by dropping prefetches that cannot be serviced immediately. Our experiments indicate that this state prefetching is highly effective, it eliminates nearly all state misses in the pre-commit stage without the need to add any additional state cache ports.

In the interleaved state caching approach, the main advantage of interleaving is to have a single cache access read or write both data and state. As a result, state lookups are performed as soon as the data (and state) address is resolved (MEM stage in Figure 5(b)). To achieve this, MemTracker functionality is weaved into the "regular" processor pipeline and there are no additional pipeline stages. State lookups and updates are performed in much the same way as regular loads and stores: lookups when the address is resolved and updates when the instruction commits. Consequently, state must be forwarded just like data, and speculative out-of-order lookups must be squashed and replayed if they read state that is later found to be obsolete. As a result, the state lookup/update queues in this approach are nearly identical to load/store queues in functionality and implementation, but are less complex and power-hungry because state is much (by a factor of 8 or more) smaller than the corresponding data. Finally, it should be noted that, if load/store queues are replaced in the future by some other forwarding and conflict resolution mechanism(s), our state lookup/update queues can be replaced by the same forwarding and conflict resolution mechanism(s).

4.5. Multiprocessor Consistency Issues

MemTracker states are treated just like any other data outside the processor and its L1 cache, so state is automatically kept coherent in a multiprocessor system. Hence, we focus our attention on memory consistency issues. We use the strictest consistency model (sequential consistency) in our implementation of MemTracker. We also briefly explain how to support processor consistency, on which many current machines are based. We note that other consistency models can also be supported, but they are too numerous to address in this paper.

Because MemTracker stores state separately from data in memory and L2 caches, the ordering of data accesses themselves is not affected. The ordering of state accesses can be kept consistent using the same mechanisms that enforce data consistency. However, MemTracker introduces a new problem of ensuring that the ordering between data and state accesses is consistent. In particular, even in a RISC processor, a single load instruction could result in a data read, a state lookup, and a state update; similarly, a store instruction could result in a state lookup, a data write, and a state update. In a sequentially consistent implementation, data and state accesses from a single instruction must appear atomic. This creates three separate issues: atomicity of state and data writes in store instructions, atomicity of state and data reads in load instructions, and atomicity of state reads and writes in all event instructions.

Atomicity of state and data writes in a store instruction is easily maintained in interleaved caching because the same cache access writes both data and state, hence the two writes are actually simultaneous. In split and shared caching, we force both writes to be performed in the same cycle. If one suffers a cache miss, the other is delayed until both are cache hits and can be performed in the same cycle.

Atomicity of the state lookup and the data read in a load instruction is also easily maintained in interleaved caching, because they are again part of the same cache access. In split and shared caching, the instruction must be replayed if the data value has (or may have) changed before the state is read by the same instruction. Depending on how load-load ordering is kept consistent by the processor, this involves replaying the instruction when an invalidation for the data is received before it commits, or when the data value at commit is not the same as the data value originally loaded from the cache.

Finally, atomicity of the state lookup and update can be maintained by replaying the instruction if the state it has read has changed before it writes the new state. We achieve this by detecting state invalidations that affect uncommitted event instructions, but it can also be done by checking if the state value to be overwritten is the same as the state value originally read by the instruction.

In consistency models that allow write buffers (e.g. processor consistency), the simplest way to ensure correct behavior is to flush the write buffer when there is a state update and do the write directly to the cache. This approach should

work well when state updates are much less frequent than data writes, as we show in Section 6. Additional optimizations are possible, but are beyond the scope of this paper.

Overall, MemTracker support can be correctly implemented in sequentially and processor-consistent multiprocessors in a relatively straightforward way, by extending existing approaches that maintain data consistency. We note that any mechanism that maintains state (e.g. fine-grain protection) separately from data would have similar issues and demand similar solutions.

4.6. Setup and Context-Switching

We consider MemTracker-related processor state (PSTT, MTCR, event mask register) to be a part of process state, which is saved and restored on context switches. In our current implementation, the total amount of MemTracker state to be saved/restored in this manner is less than 400 bytes, which is comparable to the rest of process state (general purpose, floating point, control, and status registers) and would cause negligible overheads. We note that per-word states need not be saved/restored on context switches. Instead, they are only cached on-chip and move on- and off-chip as a result of cache fetch and replacement policy.

5. Evaluation Setup

5.1. Memory Problem Detectors

To demonstrate the generality of our MemTracker support and evaluate its performance more thoroughly, we use four checkers designed to detect different memory-related problems. One of these checkers is the *HeapData* checker used as an example in Section 3.1.

State \ Event	UEVT30 (SetDelimit)	UEVT31 (ClrDelimit)	LD	ST	Sub-word LD	Sub-word ST
0 (Normal)	1	0	0	0	0	0
1 (Delimit)	1 E	0	1 E	1 E	1 E	1 E

Figure 6. PSTT setup for the *HeapChunks checker.*

The second checker is *HeapChunks*, which detects heap buffer overflows from sequential accesses. For this checker, the memory allocation library is modified to surround each allocated data block with delimiter words, whose state is changed (using event UEVT30) to *Delimit*, while all other words remain in the *Normal* state. Any access to a *Delimit* word is an error. When the block is freed, the state of its delimiter words is changed back to *Normal* (using UEVT31). Note that the standard GNU heap library implementation keeps meta-data for each block (block length, allocation status, etc.) right before the data area of the block. As a result, we do not need to allocate additional delimiter words, but rather simply use these meta-data words as delimiters.

This HeapChunks checker uses one-bit state per word, and the PSTT for it is shown in Figure 6. Note that HeapChunks is intended as an example of a very simple checker with one-bit state, and that it does not provide full protection from heap block overflows. For example, it would not detect out-of bounds accesses due to integer overflow or strided access patterns. However, it does detect the most prevalent kind of heap-based attacks (sequential buffer overrun).

State \ Event	UEVT24 (RAwr)	UEVT25 (RArd)	UEVT26 (RAfree)	LD	ST	Sub-word LD	Sub-word ST
0 (NotRA)	1	0 E	0 E	0	0	0	0
1 (GoodRA)	1 E	1	0	1	2	1	2
2 (BadRA)	1	2 E	0	2	2	2	2

Figure 7. PSTT setup for the *RetAddr checker.*

The third checker is *RetAddr* (Figure 7), which detects when a return address on the stack is modified. This checker detects stack smashing attacks that attempt to redirect program control flow by overwriting a return address on the stack. This checker keeps each word in the stack region in one of three states: *NotRA*, *GoodRA*, and *BadRA*. All stack words start in the *NotRA* state, which indicates that no return address is stored there. When a return address is stored in a stack location, its state changes to *GoodRA*. An ordinary store changes this state to *BadRA*. When a return address is loaded, we check the state of the location and trigger an exception if the location is not in the *GoodRA* state. Our simulations use the MIPS ISA, which uses ordinary load and store instructions to save/restore the return address of a function, which is otherwise kept in a general-purpose register. To expose return address activity to our checker, we insert UEVT24 (*RAwr*) after each valid return address store, UEVT25 (*RArd*) before each valid return address load, and UEVT26 (*RAfree*) before the return address stack location goes out of scope (when the activation record for the function is deallocated). All these user events target the intended location for the return address. For our experiments, this event insertion is done by a modified GCC code generator, but it would be relatively simple to achieve the same goal through binary rewriting. For CISC processors (e.g. x86), return address pushes and pops are done as part of function call/return instructions, so it is trivial to identify them. The *RetAddr* checker is another example of a useful checker that can benefit from MemTracker due to frequent state updates: each function call/return will result in at least three state updates to the return address' state.

The fourth checker combines all three checkers described above. This *Combined* checker uses seven different states, and configures MemTracker to use four-bit states. This is the most demanding of the four checkers in terms of the number of user events that must be executed and in terms of state memory and on-chip storage requirements, so we use it as the "default" checker in our evaluation and use the three component checkers to evaluate the sensitivity of MemTracker's performance to different checkers and state sizes.

5.2. Benchmark Applications

We use all applications from the SPEC CPU 2000 [15] benchmark suite, and omit only Fortran 90 applications be-

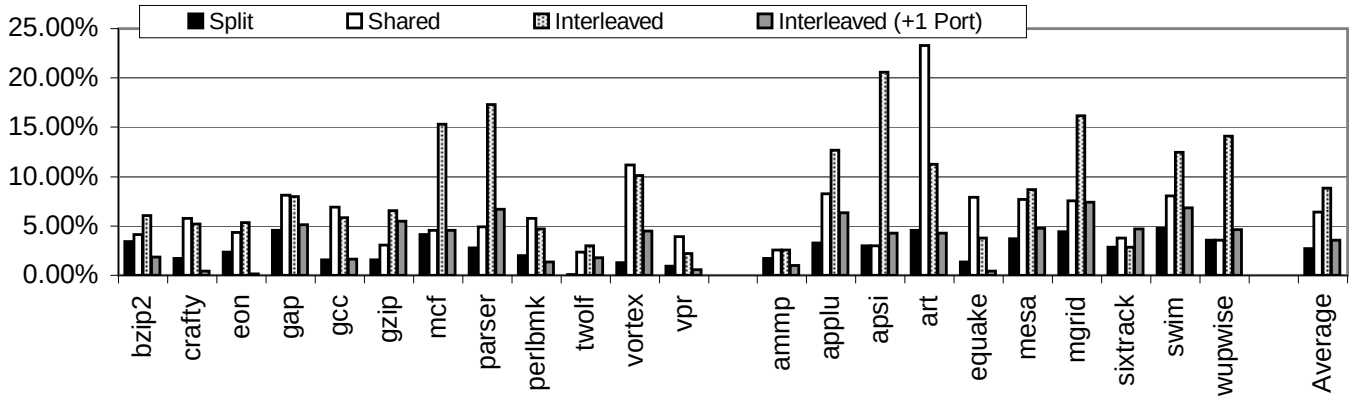

Figure 8. Effect of shared, split, and interleaved caching of state and data in L1 caches.

cause this language is not yet supported by the compiler infrastructure for our simulator (GCC 3.4.0 toolchain). For each application, we use the reference input set in which we fast-forward through the first five billion instructions to skip initialization phases and simulate the next two billion instructions in detail. We note that our fast-forward must still model all MemTracker state updates to keep the checker's state correct. If we ignore allocations, initializations, and return address save/restore while fast-forwarding, when we enter detailed simulation, our checkers would trigger numerous exceptions due to falsely detected problems (e.g. reads from locations whose allocation we skipped).

5.3. Simulation Environment and Configuration

We use SESC [12], an open-source execution-driven simulator, to simulate a near-future out-of-order superscalar processor running at 5GHz. The L1 data cache we model is 16KBytes in size, two-way set associative, dual-ported, with 32-byte blocks. The L2 cache is 2MB in size, four-way set associative, single-ported, and also with 32-byte blocks. The processor-memory bus is 128 bits wide and operates at 500MHz, and the round-trip main memory latency is 320 cycles when there is no contention.

Our default MemTracker configuration (shown in black in all charts) uses split state caching in the L1 cache (Figure 4(a)), with 2KBytes of state cache, which is two-way set-associative, dual-ported, and with 32-byte blocks.

6. Evaluation

6.1. Effect of L1 Caching Approaches

As described in Section 4.3, we examine three different approaches to caching state in primary caches: *Split*, *Shared*, and *Interleaved*. Figure 8 shows execution time overheads for these approaches on the most demanding *Combined* checker, relative to a system without any checking and without MemTracker support. We observe that the *Split* configuration has the lowest overhead, both on average (2.7%) and worst-case (4.8%), with a small 2KByte L1 state cache. The lower-cost *Shared* approach has more overhead on average (6.4%), and its overhead also varies considerably across benchmarks, with the worst case of 23.3% (in art). This higher overhead is caused by contention between state and data for both L1 cache space and bandwidth. Since the only advantage of the *Shared* approach is reduced cost due to using the existing L1 cache, it makes little sense to add L1 ports or capacity to reduce this overhead - an additional port in a 16KByte L1 data costs more than the entire 2KByte L1 state cache in our *Split* configuration.

Finally, the *Interleaved* approach also has higher overhead on average (8.9%), and worst-case (20.6%, in apsi). This configuration has dedicated space for state in each L1 line, so the additional overhead comes mainly from contention for bandwidth. With an additional L1 port, the overhead is reduced to an average of 3.6% and worst-case of 7.4%.

Overall, the *Split* configuration has the best average performance and the best worst-case performance. It is also relatively easy to integrate into the processor pipeline using the in-order pre-commit implementation described in Section 4.4. The additional 2KByte L1 state cache in this configuration is eight times smaller than the L1 data cache, and adds little to the overall real-estate of a modern processor. Hence, we use this *Split* configuration as the default MemTracker setup in the rest of our experiments. However, we note that the *Interleaved* approach has some advantages in terms of multiprocessor implementation (Section 4.5) and, with additional L1 bandwidth, has similar (but a bit worse) performance to the *Split* configuration. Consequently, the *Interleaved* approach with added L1 cache bandwidth may also be a good choice if the goal is to simplify support for multiprocessor consistency.

6.2. Performance with Different Checkers

Figure 9 shows that the overhead mostly depends on the number of state bits per word used by a checker. The one-bit *HeapChunks* checker has the lowest overhead – 1.4% on average and 2.4% worst-case. Both two-bit checkers have overheads of 1.9% on average and 4.3% worst-case. We note that these checkers have different user events – the *HeapData* checker uses variable-footprint user event instructions

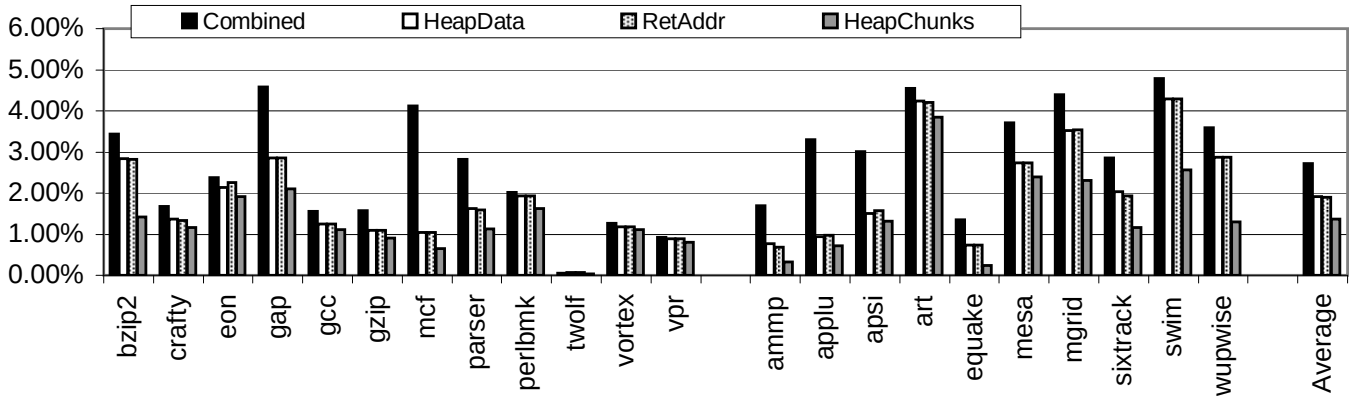

Figure 9. Overhead of different checkers in MemTracker, with Split L1 state caching using a 2KByte L1 state cache.

to identify heap memory allocation and deallocation, while the *RetAddr* checker uses word-sized user events to identify when the return address is saved and restored. However, after application's initialization phase, *HeapData*'s user events are rare, while each of *RetAddr*'s more frequent events requires little processing. As a result, in both checkers user events contribute little to the overall execution time.

The four-bit *Combined* checker has an overhead of 2.7% on average and about 4.8% worst-case (in swim). This overhead is larger than in less-demanding checkers, mainly due to larger state competing with data for L2 cache space and bus bandwidth. Still, even the "high" 4.8% worst-case overhead is low enough to allow checking even for "live" performance-critical runs. Also, note that the overhead of the combined checker is significantly lower than the sum of overheads for its component checkers. This is mainly because the combined checker does not simply do three different checks – they are combined into a single check.

6.3. Sensitivity Analysis

We performed additional experiments with 1KByte, with 4KByte, and with 16KByte L1 state caches in the *Split* configuration. We find that state caches larger than our default of 2KBytes bring negligible performance improvements (<0.5%) in all applications and on average, which indicates that the 2KByte cache is large enough to capture most of the first working set for MemTracker state. The smaller 1KByte cache still shows an overhead of only 3.2% on average relative to the no-checking baseline. However, the worst-case overhead is higher in some applications: in mcf the overhead jumps from 4.1% with the 2KByte state cache to 9.2% with a 1KByte state cache. The reason for this is that the smaller state cache "covers" less memory than the L1 data cache for the 4-bit Combined checker used in these experiments, which puts a larger number of state L1 cache misses on the critical path. Additionally, line size in the state cache is the same as in the data cache (32 bytes in our experiments) although state is smaller than the corresponding data. This puts smaller state caches at a disadvantage in applications with less spatial locality.

We also conducted experiments in which we disable state prefetches (see Figure 5(a)) in the *Split* configuration. We find that the average overhead increases from 2.7% with state prefetching to 5.4% without it, and the worst-case overhead jumps from 4.8% in gap with state prefetching to 14% in equake without state prefetching. Our state prefetching mechanism is very simple to implement, and we believe its implementation is justified by the reduction in both average overhead and variation of overheads across applications.

Overall, we find that the small 2KByte state cache results in a good cost-performance tradeoff, and even a 1KByte state cache can be used to reduce cost if a wider variation in performance overhead and slightly higher average overheads are acceptable. We also find that state prefetching brings significant benefits at very little cost.

6.4. Comparison with Prior Mechanisms

To estimate the advantages of our MemTracker support, we compare its performance with checking based on an approximation of Mondrian Memory Protection [21] and with an approximation of checking based on software-only instrumentation. We do not actually implement these schemes, but rather make optimistic estimates of the cost for key checking activities. As a result, these estimates are likely to *underestimate* the actual advantages of MemTracker over prior schemes, but even so they serve to highlight the key benefits of our mechanism.

In a Mondrian-based implementation of the Combined checker, Mondrian's fine-grain permissions must be set to only allow accesses that are known to be free of exceptions and state changes. Examples of such accesses are load/store to already initialized data, load/store to non-return-address stack locations, and loads from unmodified return address locations. We assume zero overhead for these accesses, which makes Mondrian permission fetches and checks "free". For permissions changes on allocation, deallocation, or return address load/stores, we model only a penalty of 30 cycles for raising an exception. We note that this penalty is optimistic, because it subsumes a pipeline flush for the exception, the jump to the exception handler, the actual execution of the ex-

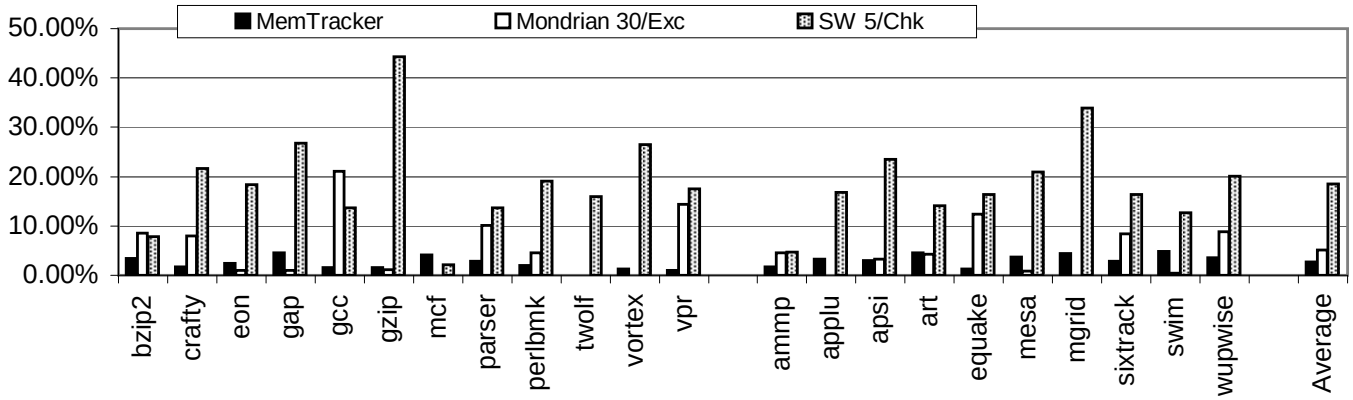

Figure 10. Effect of state changes in software handlers.

ception handler (which must update Mondrian's permissions trie structure), and the return to the exception site.

For software-only checking, we only model a five-cycle penalty for each check that must be performed for a load/store. This check must read the state for the target memory location, determine if a state change is needed or if an error is indicated, and finally update the state. The 5-cycle penalty is added to the execution time of the unmodified application, so the penalty includes all effects of instrumentation, including any misses in the instruction cache, misses in the data cache when looking up state, conditional branches when checking the state, and actual execution of instrumentation instructions. We note that this 5-cycle penalty is very optimistic; for example, in HeapMon [14] the actual reported average duration of a (highly optimized) load or store check is 18 to 480 cycles, depending on the application.

The results of these experiments are shown in Figure 10. For MemTracker, we use a Split L1 caching configuration with a 2KBytes L1 state cache. We find that our MemTracker mechanism (with all overheads accounted for) outperforms both Mondrian-based checking and software-only checking. The average overhead for software-only checking is 18.5%, with a worst-case of 44.4%. Due to our optimistic assumptions for software-only checking, this overhead is much lower than previously reported numbers [10, 19] for such checkers, but it is still too high to allow always-on checking in production runs.

The average overhead for Mondrian-based checking is 5.1%, compared to 2.7% for MemTracker. The worst case for Mondrian is 21.1% in gcc, compared to MemTracker's 4.8% in swim. We note that Mondrian requires complex hardware to look up and manage its trie permissions structures and uses several kinds of on-chip caching to speed up its permissions checks. As a result, Mondrian implementation is unlikely to be less complex than MemTracker, so MemTracker's higher performance and lower performance variation across applications is a definite advantage. It should also be noted that we fully model all overheads for MemTracker-based checking, whereas the real overheads of Mondrian-based checking could be considerably higher than our optimistic estimate.

6.5. Validation of Access Checking Functionality

We tested our checking functionality by injecting bugs and attacks into several applications as they are running with our *Combined* checker. All instructions of the applications are simulated from the start of execution until either a bug/attack is detected, or until they complete execution, in which case we check the program's results for correctness.

To test the return address protection, we choose 15 different function calls in each of the following applications: crafty, parser, and twolf. After the return address is saved to the stack, we inject a single dynamic instance of a store instruction that overwrites it. Our checker detects all such attacks, raising an exception before the modified return address is actually used to re-direct control flow of the application.

To test the heap chunk protection, we randomly choose an allocated heap block and sequentially overwrite the block past its end. We performed a total of 60 such attacks in craft, gzip, mcf, and mesa, and our checker always detects the write that exceeds the allocated space.

To test our detection of reads from uninitialized heap locations, we randomly choose a dynamic instance of a `calloc` call and omit the initialization of the first or the last word of the block. We injected a total of 122 such errors in crafty, gzip, mcf, and mesa, and in all but one injection, reads from the uninitialized location were detected. The remaining one injection (in gzip) was not detected because the application never read the word whose initialization we omitted.

Finally, to test our detection of accesses to unallocated heap data, we intercept a randomly-chosen dynamic instance of a `malloc` call and reduce the size of the request by 4 bytes (one word). We performed a total of 183 such injections in crafty, gzip, mcf, and mesa. In 149 of these injections our checker finds a read or a write to the unallocated location. In the remaining 34 injections the last word of the allocated block is never actually accessed, so the injected error is not manifested (the application completes correctly).

Although our checkers are very effective in finding the errors and attacks they target, we note that the checkers themselves are not the focus of the paper. They are only used to

demonstrate and test our MemTracker mechanism, and problem detection abilities of these checkers are similar to other implementations of similar checkers.

7 Conclusions

This paper describes MemTracker, a new hardware support mechanism that can be set up to perform different kinds of memory access monitoring tasks. MemTracker associates each word of data in memory with a few bits of state, and uses a programmable state transition table to react to different events that can affect this state. The number of state bits per word, the events to react to, and the transition table are all fully programmable by software. The MemTracker state is kept in main memory and cached on the processor chip, and is looked up and updated by MemTracker hardware. Any state-event pair can be programmed to trigger execution of a software handler, which is used to report a problem or to handle sophisticated checks or recovery. The rich set of states, events, and transitions supported by MemTracker allows bug checks to proceed with minimal performance overheads. To evaluate our MemTracker support, we map three different checkers onto it, as well as a checker that combines all three. Even for the combined checker, we observe performance overheads of 2.7% on average and 4.8% worst-case on SPEC 2000 applications.

We examine several approaches to caching MemTracker state on-chip, and find that it is possible to implement MemTracker without significant changes to most of the processor pipeline and L1 caches, by adding two in-order stages to the back-end of the processor pipeline and by using a small (2KByte) dedicated L1 state cache. In the L2 cache and memory, MemTracker state is stored just like any other data, without any extra support. With its low performance overhead and simple implementation, we believe that MemTracker is one right step towards hardware support mechanisms that will be needed by developers to continuously monitor the highly complex applications of the future.

References

[1] J. Boletta. SecurityFocus Newsletter #172. *http://citadelle.intrinsec.com/mailing/current/HTML/ml_securityfocus_news/0067.html*, 2002.

[2] M. L. Corliss, E. C. Lewis, and A. Roth. Dise: A programmable macro engine for customizing applications. In *ISCA '03: 30th Intl. Symp. on Computer Architecture*, pages 362–373, New York, NY, USA, 2003. ACM Press.

[3] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proc. of the 7th USENIX Security Conf.*, 1998.

[4] J. R. Crandall and F. T. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *37th Intl. Symp. on Microarchitecture (MICRO)*, pages 221–232, 2004.

[5] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, Y. Wu. LIFT: A Low Overerhead Practical Information FLow Tracking System for Detecting Security Attacks. *In Proc. of the Intl. Symp. on Microarchitecture*, 2006.

[6] G. Suh, J. Lee, and S. Devadas. Secure program execution via dynamic information flow tracking. *In Proc. of the 11th Intl. Conf. on Arch. Support for Prog. Lang. and Operating Sys.. Boston, MA*, 2004.

[7] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The Microarchitecture of the Pentium® 4 Processor. *Intel Technology Journal*, 5(1), 2001.

[8] IBM Corporation. IBM Rational Purify. *http://www.ibm.com/software/awdtools/purify/*, 2005.

[9] J. T. Kuehn and B. J. Smith. The Horizon supercomputing system: architecture and software. In *Proc. of Supercomuting '88*, pages 28–34, 1988.

[10] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *12th Network and Distributed System Security Symp. (NDSS)*, 2005.

[11] F. Qin, S. Lu, and Y. Zhou. SafeMem: Exploiting ECC-Memory for Detecting Memory Leaks and Memory Corruption During Production Runs. In *Proc. of the Intl. Symp. on High Performance Computer Architecture*, 2005.

[12] J. Renau et al. SESC. *http://sesc.sourceforge.net*, 2006.

[13] J. Seward. Valgrind, An Open-Source Memory Debugger for x86-GNU/Linux. *http://valgrind.kde.org/*, 2004.

[14] R. Shetty, M. Kharbutli, Y. Solihin, and M. Prvulovic. Heapmon: A helper-thread approach to programmable, automatic, and low-overhead memory bug detection. *IBM Journal of Research and Development*, 50(2/3):261–275, 2006.

[15] Standard Performance Evaluation Corporation. SPEC Benchmarks. *http://www.spec.org*, 2000.

[16] Symantec. Microsoft IIS HTR Chunked Encoding Heap Overflow Allows Arbitrary Code. *http://securityresponse.symantec.com/avcenter/security/Content/2033.html*, 2002.

[17] US-CERT. FedCIRC Advisory FA-2001-19 "Code Red" Worm Exploiting Buffer Overflow In IIS Indexing Service DLL. *http://www.us-cert.gov/federal/archive/advisories/FA-2001-19.html*, 2001.

[18] US-CERT. Buffer Overflow in Microsoft Internet Explorer. *http://www.us-cert.gov/cas/techalerts/TA04-315A.html*, 2004.

[19] Valgrind Developers. The Valgrind Quick Start Guide. *http://valgrind.org/docs/manual/quick-start.html*, 2005.

[20] Weidong Shi, Chenghuai Lu, Hsien-Hsin S. Lee. Memory Centric Security Architecture. *In Proc. of Intl. Conf. on High Performance Embedded Architectures and Compilers*, 2005.

[21] E. Witchel, J. Cates, and K. Asanovic. Mondrian memory protection. In *ASPLOS-X: 10th international conference on Arch. Support for Prog. Lang. and Operating Sys.*, pages 304–316, New York, NY, USA, 2002. ACM Press.

[22] P. Zhou, W. Liu, L. Fei, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torellas. AccMon: Automatically Detecting Memory-related Bugs via Program Counter-Based Invariants. In *Proc. of the 37th Intl. Symp. on MicroArchitecture*, 2004.

[23] P. Zhou, F. Qin, W. Liu, Y. Zhou, and J. Torellas. iWatcher: Efficient Architectural Support for Software Debugging. In *Proc. of the 31st Intl. Symp. on Computer Architecture*, 2004.

[24] Y. Zhou, P. Zhou, F. Qin, W. Liu, and J. Torrellas. Efficient and flexible architectural support for dynamic monitoring. *ACM Trans. Archit. Code Optim.*, 2(1):3–33, 2005.