MemTracker: An Accelerator for Memory Debugging and Monitoring

GURU VENKATARAMANI and IOANNIS DOUDALIS
Georgia Institute of Technology
YAN SOLIHIN
North Carolina State University
and
MILOS PRVULOVIC
Georgia Institute of Technology

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. Existing software and hardware approaches for finding and identifying memory bugs have a number of drawbacks including considerable performance overheads, target only a specific type of bug, implementation cost, and inefficient use of computational resources.

This article 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 with 8 bits state per memory word, we observe performance overheads of only around 3%, on average, and 14.5% worst-case across different benchmark suites. Such low overheads allow continuous (always-on) use of MemTracker-enabled checkers, even in production runs.

This article is an extension of the article entitled "MemTracker: Efficient and Programmable Support for Memory Access Monitoring and Debugging" which appeared in the *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, in February 2007.

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.

Authors' addresses: G. Venkataramani and M. Prvulovic, College of Computing, Georgia Institute of Technology, 266 Ferst Drive, Atlanta, GA 30332-0765; email: {guru,milos}@cc.gatech.edu.

DOI 10.1145/1543753.1543754 http://doi.acm.org/10.1145/1543753.1543754

Categories and Subject Descriptors: C.1 [**Computer Systems Organization**]: Processor Architectures

General Terms: Design, Performance, Reliability

Additional Key Words and Phrases: Accelerator, memory access monitoring, debugging

ACM Reference Format:

Venkataramani, G., Doudalis, I., Solihin, Y., and Prvulovic, M. 2009. MemTracker: An accelerator for memory debugging and monitoring. ACM Trans. Architec. Code Optim. 6, 2, Article 5 (June 2009), 33 pages. DOI = 10.1145/1543753.1543754 http://doi.acm.org/10.1145/1543753.1543754

1. INTRODUCTION

Technological advances have resulted in decades-long exponential growth in computing performance, which is being exploited by increasingly complex software. As a result of this complexity, software is increasingly prone to programming errors (bugs) and many of these bugs also represent security vulnerabilities. A large number of tools and techniques have been developed by software vendors to identify bugs or to prevent them from becoming attack exploits. However, such software tools incur large performance overheads, which prohibits their use in postdeployment monitoring of live (production) runs where end users are directly affected by the overheads of the monitoring scheme. 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.

One particularly important and broad class of programming mistakes 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, and the like. Many software tools have been developed to detect some of these errors. For example, Purify [IBM Corporation 2005] and Valgrind [Seward 2004] detect memory leaks, accesses to unallocated memory, reads from uninitialized memory, and some dangling pointer and out-of-bounds accesses. Certain memory-related errors, such as buffer overflows, are also a well-known source of security vulnerabilities. As a result, security tools often focus on detection of such errors or specific error manifestation scenarios. For example, StackGuard [Cowan et al. 1998] detects buffer overflows that attempt to modify a return address on the stack.

In order to monitor memory-related bugs, a software detection tool (checker) has to *intercept* memory accesses (loads and/or stores) and perform the following set of actions:

—*State look-up*. For each access, the checker has to find the state of the memory location (e.g., determine whether the location is allocated, initialized, stores a return address).

—*State check*. Once the state is obtained, the checker has to verify if the access is allowed for a memory location with such state.

—*State update*. The state of the memory location can change as a result of the access (e.g., a write access changes an uninitialized memory location into an initialized one).

Since memory read and write instructions are executed frequently, the overhead of intercepting and checking them is very high. Slow downs of 2X to 30X (i.e., up to 30 times) have been reported for some popular software tools like Valgrind [Newsome and Song 2005; Valgrind Developers 2005].

Architectural support has been proposed to reduce performance overheads for detecting some memory-related problems [Shetty et al. 2006; Witchel et al. 2002, Zhou et al. 2004, 2005]. Many of these schemes allow loads and stores to be intercepted in hardware, 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. Previously proposed schemes disagree on whether state checks and updates should also be performed in hardware because this decision determines the tradeoff between performance and the ability to support different checkers. One approach is to hard-wire the meaning of each state for a specific checker [Crandall and Chong 2004; Zhou et al. 2004], which allows one specific checker or a family of checkers to be implemented very efficiently. Another approach is to perform interception in hardware and dynamically insert software handlers for state checks and updates [Corliss et al. 2003]. Finally, a number of existing approaches express state as access permissions or monitored regions. Such state can quickly be checked in hardware to determine whether the access can proceed without any additional checker activity. If a state update (or additional checking) is needed, a software handler is invoked [Shetty et al. 2006; Witchel et al. 2002; Zhou et al. 2004, 2005]. Overall, existing architecture support is either (i) hard-wired for a particular checker or (ii) requires software intervention for every state update. This can lead to significant performance degradation for checkers with frequent state updates.

Overall, 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 checkers. 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 problem, we propose a hardware mechanism, which we call MemTracker, that can perform interception, state checks, and state updates in hardware, while still remaining generic and not hard-wired for any particular checker [Venkataramani et al. 2007]. 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 used 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, MemTracker helps by (i) avoiding software intervention for most state updates, (ii) supporting efficient state checks and updates even when neighboring locations have different states, and

(iii) amortizing the cost of MemTracker hardware by being able to support different kinds of checkers at different times, or even at the same time.

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 together, 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 special user event instructions. Applications and runtime libraries can use the special user event instructions 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.

In this article, we make the following significant additions to MemTracker to make it more robust and increase its applicability:

—Extend the flexibility of PSTT to support multiple checkers at the same time. Note that this is different from a single checker that combines the functionality of different checkers described in our earlier proposal [Venkataramani et al. 2007]. With this added flexibility, checkers become more modular and can be combined without requiring a programmer to combine their state diagrams into a single monolithic checker.
—Optimizations to improve performance (e.g., filtering out silent state updates).
—Evaluation across multicore configurations and three different benchmark suites.
—Additional implementation details, and
—Additional sensitivity analyses.

Any highly programmable mechanism, such as MemTracker, has a wide spectrum of potential uses that cannot 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 (i) heap

memory access checking similar to that used in Purify [IBM Corporation 2005], Valgrind [Seward 2004], and HeapMon [Shetty et al. 2006]; (ii) detection of malicious or accidental overwrites of return addresses on the stack; (iii) detection of heap-based sequential buffer overflow attacks [Boletta 2002; Symantec 2002; US-CERT 2001, 2004] and errors; and (iv) simultaneous use of all three of the previously described checkers.

We find that even for the combined checker, MemTracker's performance overhead is only around 3%, on average, and 14.5% worst-case across SPEC CPU [SPEC 2006] and SPLASH-2 [Woo et al. 1995] applications, relative to a system that runs the same applications without any checking and without MemTracker support.

The rest of this article 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 generic of the previously proposed hardware mechanisms is dynamic instruction stream editing [Corliss et al. 2003] (DISE), 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 [Kuehn and Smith 1988] widens each memory location by 6 bits, 2 with hard-wired functionality, and 4 trap bits that can intercept different flavors of memory accesses. Mondrian Memory Protection [Witchel et al. 2002] has per-word permissions that can intercept read, write, or all accesses. iWatcher [Zhou et al. 2004] provides enhanced watchpoint support for multiple regions of memory and can intercept read, write, or all accesses to such a region. HeapMon [Shetty et al. 2006] 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. All four schemes only provide interception and state checking in hardware. Each state update requires software intervention to change trap, permission, watchpoint, or filter bits (in Horizon, Mondrian, iWatcher, and HeapMon, respectively). This software intervention can take the form of instrumentation at the point where the change is needed. Alternatively, accesses that require state change can result in an exception, allowing the exception handler to change the state. For example, consider a state that indicates whether a location is initialized. Initialized locations can have their trap, permission, watchpoint, or filter bits set to indicate that both reads and writes are allowed without generating exceptions. Uninitialized locations should indicate that both reads and writes require exceptions, reads to indicate an error (read from uninitialized location is detected),

and writes to allow the exception handler to change the state to "initialized." In contrast to these schemes, MemTracker can be programmed to handle state changes in hardware without software intervention. This is an important difference because, as shown in Section 6.4, in some useful checkers, state changes are numerous enough to cause significant overheads.

To avoid hard-wiring the number of bits for each state, but still provide efficient checks and updates, MemTracker uses a flat (array) state structure in memory. In contrast, Mondrian [Witchel et al. 2002] uses a sophisticated trie structure to minimize state storage overheads for coarse-grain states, at the cost of more complex fine-grain state updates. iWatcher [Zhou et al. 2004] keeps track of ranges of data locations with the same state, which also complicates fine-grain updates. Horizon [Kuehn and Smith 1988] simplifies state look-ups by keeping state in 6 extra bits added to each memory location, which requires nonstandard memory modules and adds a state storage cost even when no checks are actually needed. In contrast, MemTracker keeps state information separately in memory and uses only as much state as needed. In particular, when checking is not used, there is no memory allocated for MemTracker state.

Since MemTracker has originally been introduced, two mechanisms have adopted MemTracker's approach of decoupling data and state information: FlexiTaint [Venkataramani et al. 2008] and HardBound [Devietti et al. 2008]. FlexiTaint targets a different problem domain: hardware acceleration of dynamic taint propagation, which associates a "taint" with each value (in memory or registers) and propagates taints as values are moved or used in computation. A key optimization in FlexiTaint is that most of the time, taints are simply copied from source to destination, which allows the use of special filter tables to minimize the number of taint checks. In contrast, MemTracker is designed for memory checkers, which associate their state with each memory location (rather than its value). This prevents the use of FlexiTaint-like optimizations that filter out checks, but allows a less costly implementation because MemTracker only needs to check memory accesses (instead of all instructions). HardBound is hard-wired for a particular scheme that provides spatial safety guarantees by storing bounds information for pointers separately and keeping it transparent to the underlying hardware. Since the metadata is stored in the pointer itself, it is difficult to find accesses that use stale or dangling pointers. In contrast, MemTracker associates state with memory words themselves. After memory is freed, the state associated with those memory words would reflect that the memory is "free" and an access to that memory through a dangling pointer can be detected correctly.

Nethercote and Seward [2007] have done detailed studies for shadow memory management needed by software memory checkers, such as Valgrind. Their design allocates a byte of meta-data for every byte in memory and proposes techniques for optimizing the memory overhead. We expect the number of state bits for every memory word to be typically small (maximum of 8 or 16 state bits per word), as evidenced in our evaluation (see Section 6). Hence, we do not anticipate the amount of virtual memory needed for state to be a limiting

problem especially for larger address spaces as we move from 32-bit to 64-bit addresses. Also, the optimizations, such as compression employed by Nethercote and Seward [2007], can be adopted by MemTracker to support more sophisticated checkers that require larger percentage of virtual memory space.

Other related work includes AccMon [Zhou et al. 2004], Dynamic information flow tracking [Suh et al. 2004], Minos [Crandall and Chong 2004], Memory centric security [Shi et al. 2007], LIFT [Qin et al. 2006], WIT [Akritidis et al. 2008], LBA [Chen et al. 2008], and SafeMem [Qin et al. 2005]. 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. WIT uses points-to analysis at compile time and inserts guards around objects to detect buffer overflow attacks. Memory Centric architecture associates security attributes to memory instead of individual user process. Most of the previously described 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. LBA adopts some of the key contributions of MemTracker, such as hardware support for accelerating memory checkers and programmability to suport multiple checkers. Additionally, it proposes sophisticated (and more complex) address translation mechanisms for state. MemTracker relies on simple indexing functions for state look-ups to avoid performance loss. Also, LBA implements variable state granularity (e.g., per-byte, per-word) and supports state semantics (e.g., encoding lock-set information for memory words).

Overall, MemTracker is unique in that it can efficiently support different memory checkers, even those that require simple but frequent state updates automatically without software intervention. It should be noted, however, that MemTracker can only automatically handle state checks and updates that can be expressed as a state transition table, and other checks and state updates would still require software intervention. However, many useful memory checkers can be expressed in terms of state transitions, and we expect that, in other checkers, MemTracker's state machine can be used as a sophisticated filter to minimize the number of software interventions.

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.

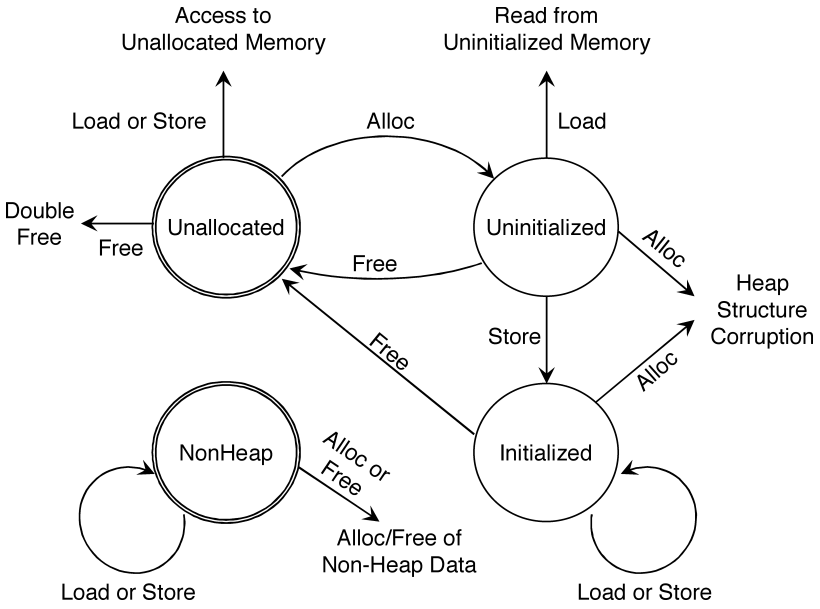


Fig. 1. State transition diagram for the HeapData checker.

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 [IBM Corporation 2005] and Valgrind [Seward 2004], and has also been used to evaluate hardware support for runtime checking of memory accesses in HeapMon [Shetty et al. 2006]. 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.

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 [Shetty et al. 2006], 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 [Shetty et al. 2006] to identify only heap accesses. The NonHeap state permits loads and stores but prohibits heap allocations and deallocations. The Operating system can identify which pages are for the heap because the heap segment is typically grown through the BRK system call. Therefore, it can initialize heap memory locations to Unalloc and nonheap locations to NonHeap.

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 article 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 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. Whenever a new physical page is mapped for data, the operating system has to determine whether a physical page exists for the corresponding state addresses. If not, a new physical page is mapped for state information. This operation needs to be performed for statically mapped pages at the time of loading the program as well as for dynamically mapped pages in the heap section of the program where pages are mapped on demand. In order to prevent accidental or malicious overwrites to pages containing state information, we provide an extra "state permission bit" apart from the normal Read, Write, and Execute permission bits. Pages containing "state" information have state permission bit set and the other permission bits are turned off. This permits MemTracker to access the "state" pages safely and restricts normal load and store instructions from accessing the state information. This approach avoids custom storage for state and benefits from existing address translation and virtual memory mechanisms. Also, we can use existing structures like translation lookaside buffer (TLB) to perform virtual address translation for state addresses. We provide a "state" permission to TLB in addition to the read, write, execute permission bits. If regular load/store accesses a "state" page directly, an exception is raised.

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. 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 look-ups, 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 look-ups, 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 trade-off 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). Note that, if binary compatibility with existing systems that do not implement MemTracker is a design requirement, existing no-op or unused opcodes in the ISA should be carefully selected to implement user events. Similarly, for new systems without MemTracker support, user event opcodes should be treated as no-ops. In this article, 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 [Hinton et al. 2001].

In this article, 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, each

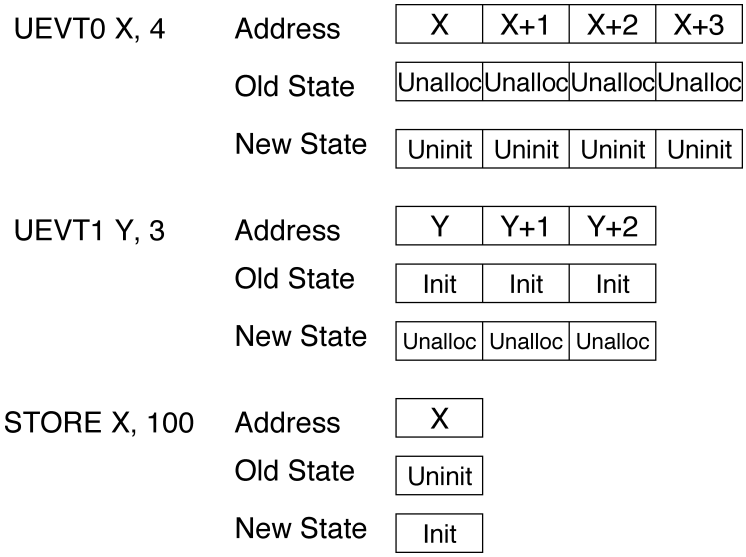


Fig. 2. Example Instructions with variable footprint (e.g., UEVT0 is used to annotate memory allocation and UEVT1 to annotate memory deallocation) and constant footprint (e.g., store).

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 for. In fact, these overheads are likely overestimated because more efficient implementations (e.g., handling a double word or an entire cache block at a time) of variable-footprint events are possible. In our MemTracker implementation, 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 gives a simple illustration for how variable-footprint instructions like UEVT0 and UEVT1 are used by our example heap checker.

As explained in Section 3.3, a variable-footprint event instruction takes the starting address and the number of affected memory words, and it performs the state look-up, check, and update for each of those memory locations. For example, UEVT0 is used as an `allocation of memory` event in the HeapData checker, and in Figure 2 we show how it changes the state of 4 words of memory at address X from Unallocated to Uninitialized. Similarly, UEVT1 is used to denote a `freeing of memory` event, and is shown in Figure 2 for 3 words at

State \ Event	UEVT0 (Alloc)	UEVT1 (Free)	LD	ST	Subword LD	Subword 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

Fig. 3. State transition table for our example HeapData checker. Entries with "E" trigger exceptions.

address Y. As an example of an instruction with constant memory footprint, we also show a `store` instruction to memory location X, which changes its state from Uninitialized to Initialized. Figure 3 shows the PSTT configuration for the heap checker, which is a tabular equivalent of states and transitions described in Section 3.1, except for the subword LD/ST events, which we discuss in Section 3.5.

It should be noted that the need to modify the memory 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 Subword Accesses

MemTracker keeps state only for entire words, so subword memory accesses represent a problem. For example, consider the shaded transition entry in Figure 3, which corresponds to a subword 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 trade-off between detecting problems and avoiding false positives, the right choice depends on the circumstances and the checker. To allow flexibility in implementing checkers, subword load/stores are treated as separate event types in the PSTT. This allows us to achieve the desired behavior for subword accesses. For example, during debugging, we can program the PSTT of the HeapData checker (Figure 3) such that subword stores to uninitialized data, leave the state of the word as uninitialized to detect all reads from uninitialized

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

locations. In live runs, we can avoid false positives by programming the PSTT to treat a subword write as an initialization of the entire word.

When treating subword accesses as accesses to the entire word, our experiments did not have any false negatives (reads of uninitialized subwords that might slip through without being noticed) because our benchmark suites, that are thoroughly tested and well structured, operate on data blocks that were multiples of word sizes. However, when subword accesses are treated in a paranoid fashion, that is, when subword access does not set the state of the entire word to Initialized, numerous false positives are observed in many benchmarks, especially in the SPECint applications, which frequently use character and short integer arrays.

Alternatively, both false positives and false negatives on subword accesses can be avoided by: (i) keeping state for each byte at a cost of increasing memory overhead needed for state (the same mechanisms described for word-level granularity still apply at byte-level tracking) or (ii) encoding the necessary semantic information as part of the state, like the solution proposed by Chen et al. [2008].

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 look-ups would still affect performance and consume energy. To minimize this effect, we add an event mask register that has 1 bit for each type of event. If load and/or store events are masked out, loads and stores are performed without state and PSTT look-ups. A masked-out user event instruction becomes a no-op, consuming only fetch and decode bandwidth.

3.7 Support for Multiple Checkers

As we have seen, MemTracker is an efficient hardware mechanism that can be programmed to implement different types of checkers. It can also be used to implement multiple checkers simultaneously. One way to do this is described in the original MemTracker proposal [Venkataramani et al. 2007]. This approach consists of combining state machines of the checkers into a combined state machine. The advantage of this approach is that the total number of state bits in the combined checker can be fewer than the sum of state bits from the original checkers. For example, the three individual checkers used in our evaluation are HeapData, HeapChunks, and RetAddr and they use 4, 2, and 3 states,

respectively (2, 1, and 2 bits of state per word). This results in a total of 5 bits of state per word (8 bit if only power-of-two state sizes are supported). However, when we design a checker to combine the functionality of these checkers, the resulting checker has only 7 states (3 bits of state per word, or 4 if only power-of-two state sizes are supported). The total number of states is reduced because some of the states can be eliminated as redundant, and the number of bits is further reduced because the individual checkers do not use all the states that can be encoded with the bits they use (e.g., the RetAddr checker uses 2 bits to encode only 3 states). This approach also has some disadvantages. In particular, we need a separate state diagram for each combination of checkers, which does not facilitate using checkers as modules, requires extensive programmer effort when combining many checkers or checkers with many states, and it is difficult to add or remove a checker at runtime. The lack of checker modularity may also prevent simultaneous use of system and user checkers: The system may impose some checkers on an application (e.g., for security), and the application itself may want to use additional checkers (e.g., to detect and report bugs). In this scenario, the application-introduced checkers should not be able to affect (e.g., subvert) system-imposed checkers. This is difficult to achieve if a programmer must design the combined checker: The system programmer cannot anticipate all the application checkers that will be needed, whereas the application programmer cannot be trusted to design the combined checker because it subsumes the functionality of system-imposed checkers.

A more modular approach to combining checkers is to leave checkers independently defined. In general, a checker is expressed as a state transition function that takes the event ID and the current state, and returns the new state and an additional bit that indicates whether or not to raise an exception. If we have two checkers A and B, defined through functions F_A and F_B and using n_A and n_B state bits, respectively, the state of the combined checker will be a concatenation of individual checker's state bits, and the combined transition function will be a simple dispatcher function that calls F_A with the first n_A bits of the combined state, calls F_B for the second n_B bits of the combined state, concatenates the resulting state into a new combined state, and performs a logical *OR* of the individual exception bits to produce the exception bits of the combined checker. This is illustrated in Figure 4.

With this approach, individual checkers can be specified as state transition functions, and the system can easily combine them without additional programmer involvement. Because each individual state transition function only sees its own part of the combined state, this also eliminates the risk of checkers interfering with each other. In particular, application-introduced checkers cannot affect the operation of system-imposed checkers.

It should be noted that the way checkers are combined affects the hardware design of MemTracker only indirectly, by making the state size larger and thus potentially requiring support for a larger maximum number of state bits. As far as MemTracker hardware is concerned, only one checker is used at a time, and the two approaches to checker combining only differ in how this one checker is constructed from component checkers.

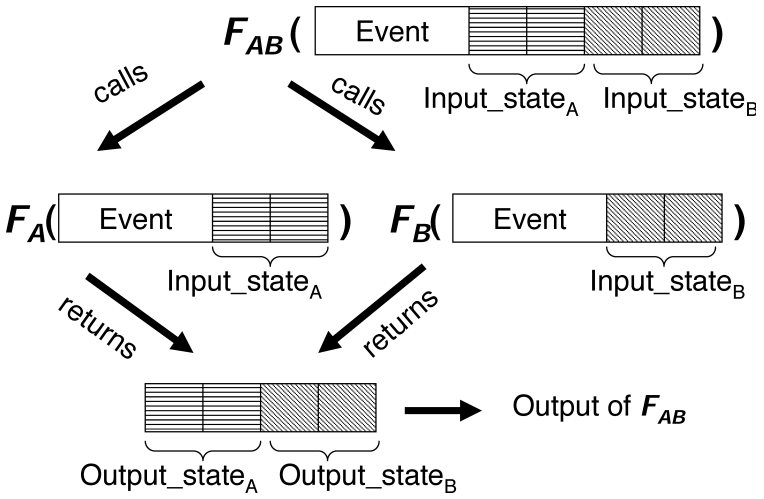


Fig. 4. A dispatcher function F_{AB} calling F_A and F_B to determine the output of checkers A and B, respectively.

Since the modular approach to checker combining can increase the number of state bits, we need to consider whether the PSTT structure is capable of handling significantly larger numbers of states. With a maximum of 4 state bits (16 states) described in the original MemTracker proposal, the number of PSTT entries is 576 (16 states and 36 events) and the entire PSTT is 360 bytes in size (576 entries, each with 4 bits of state and 1 exception bit). However, with the new dispatcher implementation, we need a larger maximum allowable number of state bits (e.g., 8 or even 16 bits). With 8-bit state (256 states), the PSTT has 9,216 entries and each entry is 9 bits in size (8+1), for a total PSTT size of slightly over 10-KBytes. With 16 bit state, the PSTT size grows to nearly 5-MB. While the original 360-byte PSTT can easily be stored on-chip and a state look-up can be performed quickly without hurting performance, this is unlikely to be the case with the larger PSTT needed to support more state bits.

To address this problem, we change the way the PSTT is presented to MemTracker hardware. Instead of keeping the entire PSTT on-chip, we cache a few recently used PSTT entries on-chip, and use a software miss handler to service misses in this cache. This miss handler is effectively the state transition function F discussed earlier in this section—given the current state and event, it computes the new state and the exception bit and places this result in the PSTT cache to help avoid future PSTT cache misses. This approach also facilitates checker modularity and fast context switching—to change the current checker, the system must only invalidate the on-chip PSTT cache and change the address of the PSTT cache miss handler, instead of having to load the new content of the entire PSTT. This approach is similar to the technique used in FlexiTaint [Venkataramani et al. 2008] for its taint propagation look-ups.

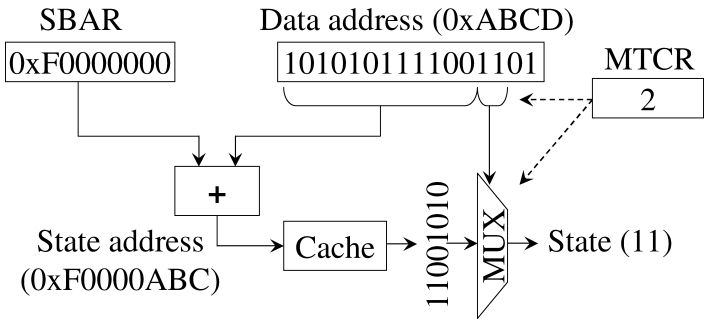


Fig. 5. Look-up of 2-bit state for data location 0xABCD.

4. MEMTRACKER IMPLEMENTATION

4.1 Programmable State Transition Table (PSTT)

Different checkers that use MemTracker 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 specifies the number of state bits per word. Note that the number of state bits is determined by the checker and not by MemTracker. Hence, the amount of virtual memory needed for storing state is also determined by MTCR. In our current implementation, we only allow power-of-two numbers, to simplify state look-ups and updates. We also limit the maximum to 8 bits because the largest checker used in our evaluation (the modular combined checker) only uses 5 bits. Note that the number of supported state bits can be easily changed to 16 or even 32, at the cost of widening the MemTracker logic and PSTT cache entries. With the new PSTT caching approach, we do not explicitly store the entire PSTT (it is expressed as a set of software handlers), so a larger number of state bits does not result in exponential growth in storage needed to store the PSTT. In our current implementation, we use a small 256-entry PSTT cache structure. This small PSTT-cache is direct mapped and is addressed using a concatenation of the event ID (6 bits to encode 36 events) and current state (8 bits in our implementation). Each entry consists of a tag (6 bits, 14 bits minus the 8 index bits), a new state (8 bits), and an exception bit (1 bit), for a total size of 480 bytes.

4.2 Finding State Information

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 look-up of 2-bit state for data address 0xABCD is shown in Figure 5.

4.3 Caching State Information

There are three basic ways to cache state information: split caching (state blocks in a separate cache, Figure 6(a)), shared caching (state blocks share the same

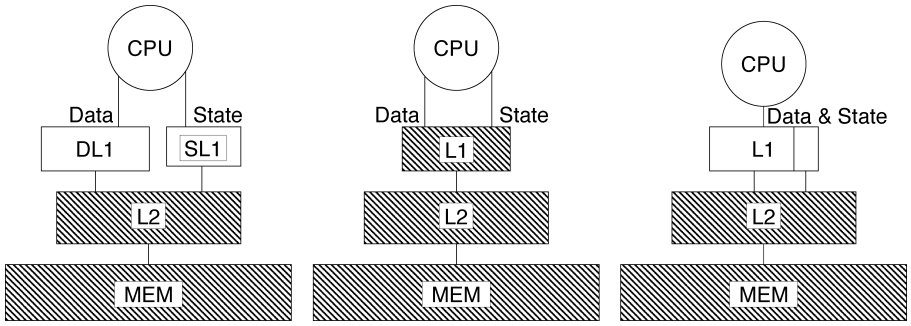
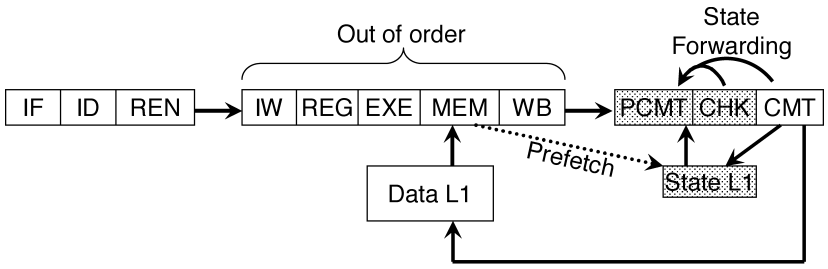


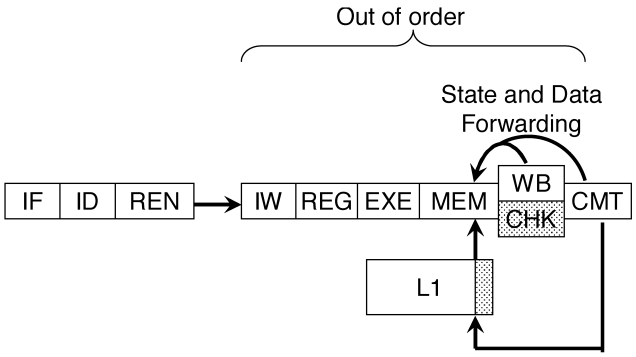
Fig. 6. Caching approaches for MemTracker state in the L1 cache. For L2 and below, we always use Shared.

cache with data, Figure 6(b)), and interleaved caching (state bits stored with the cache line that has the corresponding data, Figure 6(c)). Shared caching has the advantage of using existing on-chip caches. However, in shared caching, state competes with data for cache space, and look-ups 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. Also, the state is usually much smaller than data. So, multiple states can fit in a single memory word and state accesses that exhibit good locality have higher hit rate in the state cache. Finally, interleaved caching allows one cache access to service both a data load and the look-up of the corresponding state; similarly, a data store and a state update can be performed in the same cache access. Conceptually, interleaved caching proves to be a logical organization since the state information is held as part of the data. For multiprocessors, maintaining atomicity of state and data is easier because the state and data updates can be performed together. 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.

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 (see Section 6). We find that shared L1 caching performs poorly without expensive additional cache ports. Interleaved L1 caching performs slightly better at an added cost of access latency and area, but it simplifies some memory consistency issues (see Section 4.6) and may still be good choice in chip multiprocessors. Finally, split caching has good performance even with a smaller and simpler state cache, and has a good performance-cost-power trade-off.



(a) Shared and Split



(b) Interleaved

Fig. 7. Processor pipeline with MemTracker support (shaded) for different L1 state caching approaches.

4.4 Processor Modifications for MemTracker

MemTracker integration into the processor pipeline is simpler for Split and Shared L1 state caching approaches, where state look-ups 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 7(a)).

In a conventional 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 7(a)). The first of these stages is the precommit 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), precommit also fetches MemTracker state from the state cache. In the event of state look-up resulting in a cache miss, the pipeline is stalled until the miss is serviced. Because MemTracker processes instructions in order, a cache miss here will stall the commit of instructions,

which can be costly. We alleviate this problem by issuing a nonbinding state prefetch when the data address is resolved. This prefetch is dropped if no cache port is available to avoid contention between these prefetches and state look-ups (or with data accesses in the shared configuration). Our experiments indicate that this state prefetching is highly effective, it eliminates nearly all state misses in the precommit stage without the need to add any additional cache ports.

In the second MemTracker pipeline stage (check stage or CHK), the state and the event type are used to look up the corresponding PSTT cache entry. If the state is not available (state cache miss), the resulting exception is treated as any other exception: it causes the instruction (and all instructions fetched after it) to be squashed, the miss handler is called (it computes the PSTT entry and updates the PSTT cache), and after the handler completes, the squashed instruction is re-executed (it now finds the PSTT entry in the PSTT cache and completes). An exception also occurs if the PSTT indicates that an exception should be raised: the current instruction and all younger instructions are squashed 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 dependencies on still-uncommitted state modifications, so a small state forwarding queue is used to correctly handle such dependence. This is similar to the "conventional" store queue, which forwards store values to dependent loads, but our state forwarding queue is much simpler because (i) it only tracks state updates in the two stages between precommit and commit, so in a 4-wide processor, we need at most 8 entries; and (ii) all addresses are already resolved when instructions enter this queue, so dependencies can always be precisely identified.

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 look-ups are performed as soon as the data (and state) address is resolved (MEM stage in Figure 7(b)). To achieve this, MemTracker functionality is weaved into the "conventional" processor pipeline and there are no additional pipeline stages. State look-ups and updates are performed in much the same way as data loads and stores: look-ups when the address is resolved and updates when the instruction commits. Consequently, state must be forwarded just like data, and speculative out-of-order look-ups 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 4 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 look-up/update queues can be replaced by the same forwarding and conflict resolution mechanism(s).

4.5 Filtering of Silent State Writes

A store instruction, in addition to performing data write, now has to fetch the state and possibly update the state. Hence, every store instruction could have up to three memory accesses. This is expensive, especially for the state cache that could have two accesses (one read and one write) every time a store is performed. Because every state update is preceded by a state look-up, silent updates can easily be detected (the new state is the same as the old one) and eliminated. This saves contention for cache ports and reduces extra energy consumption by avoiding unnecessary cache accesses. In multiprocessor configurations, the elimination of a state write also eliminates the need to enforce write atomicity (this will be explained in Section 4.6). We observe in our experiments that this optimization eliminates 98.5% of state writes, on average, across different benchmark suites.

4.6 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, which many current machines are based on. We note that other consistency models can also be supported, but they are too numerous to address in this article.

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 look-up, and a state update; similarly, a store instruction could result in a state look-up, 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 look-up 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. For this, we can use the processor's existing mechanism for enforcing load-load ordering. In most processors, this involves replaying the instruction when an invalidation for the data block is received before the instruction commits. Other processors can have a different mechanism to trigger load-load replay mechanism (e.g., check if the data value read at commit differs from the one originally loaded from the cache). If the state access triggers a page fault, it is treated as any other page fault: the instruction is canceled, the page fault is serviced, and the execution of the application starts by re-executing the faulting instruction.

Finally, atomicity of the state look-up and update can be maintained using the same replay mechanism—the instruction is replayed if the state read by the instruction changes before the instruction commits its state change to the cache.

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 article.

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.

4.7 Setup and Context-Switching

The integrity of MemTracker-related information such as PSTT-cache or PSTT (in our original implementation) can be compromised if adequate steps are not taken to prevent users from corrupting the data in the transition table. We maintain PSTT-related information as part of the process context that cannot be modified directly by the application. Initializing the PSTT-cache miss handler register or the PSTT (in our prior implementation) is performed through a system call before loading the application. The operating system is responsible for initializing the PSTT structures in privileged mode and keeping the PSTT in system memory that is not mapped into the application's address space. This prevents the users from directly modifying the PSTT.

As we consider MemTracker-related processor state (PSTT-cache, PSTT cache miss handler exception address register, MTCR, event mask register) to be a part of process state, this information is saved and restored on context switches. In our original implementation, the total amount of MemTracker state to be saved/restored in this manner was nearly 300 bytes because the entire PSTT was saved and restored. In our new implementation, the entire PSTT is not actually stored anywhere—its entries are dynamically generated as needed. Thus, the PSTT cache can simply be invalidated on a context switch, and the saved/restored MemTracker state consists of only the three registers

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

Fig. 8. PSTT setup for the HeapChunks checker.

State \ Event	UEVT24 (RAwr)	UEVT25 (RArd)	UEVT26 (RAfree)	LD	ST	Subword LD	Subword 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

Fig. 9. PSTT setup for the RetAddr checker.

listed previously. 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.

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 metadata words as delimiters.

This HeapChunks checker uses 1-bit state per word, and the PSTT for it is shown in Figure 8. Note that HeapChunks is intended as an example of a very simple checker with 1-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).

The third checker is RetAddr (Figure 9), 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., $\times 86$), 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 in the previous text. We show two different implementations of this combined checker, as discussed in Section 3.7. Our first implementation uses seven different states that implements all three checkers in a single checker, and configures MemTracker to use 4-bit states. This implementation uses minimal number of state bits. Our second implementation uses a dispatcher to internally call the individual checkers and uses 8 state bits (2 for HeapData, 2 for RetAddr checker, and one for HeapChunks padded with 3 bits to have a power-of-two number of state bits). The dispatcher implementation of the combined checker 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 and 24 out of 29 applications for SPEC CPU 2006 application suite [SPEC 2006] benchmark suite. For each application, we use the reference input set in which we fast-forward through the first 15% of instructions to skip initialization phases and simulate the next 1 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

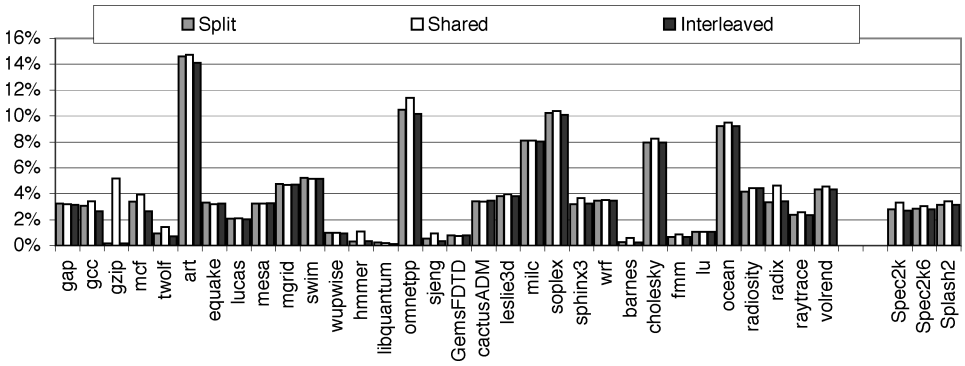


Fig. 10. Effect of shared, split, and interleaved caching of state and data in L1 caches.

whose allocation we skipped). We also evaluate Splash-2 benchmark suite [Woo et al. 1995] for our multiprocessor evaluation. We run these applications from start to finish.

5.3 Simulation Environment and Configuration

We use SESC [Renau et al. 2006], an open-source execution-driven simulator, to simulate a MIPS processor. We derive microarchitectural parameters from a modern processor (Intel Core2-like) running at 2.93 GHz. The L1 data cache we model is 32 KB in size, eight-way set associative, dual-ported, with 64-byte blocks. The L2 cache is 4 MB in size, 16-way set associative, single-ported, and also with 64-byte blocks. The processor-memory bus is 64 bits wide and operates at 1,333 MHz. We use a four-core configuration with a 32 KB private L1 and 4 MB shared L2 for multiprocessor evaluation.

Our default MemTracker configuration (shown in black in all charts) uses split state caching in the L1 cache (Figure 6(a)), with 16 KB of state cache, which is four-way set-associative, dual-ported, and with 64-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 10 shows execution time overheads for a representative set of benchmarks along with averages across entire benchmark suites on the most demanding Combined checker with 8 bits state, relative to a system without any checking and without MemTracker support. We observe that the Split configuration has a performance overhead of about an 2.8% average across SPEC benchmarks and a 3.15% average across SPLASH-2 benchmarks with the worst-case (around 14.5%) in art benchmark, with a relatively smaller 16 KB L1 state cache. The lower-cost Shared approach exhibits consistently higher overheads, on average, and its overhead also varies considerably across benchmarks, with the worst case around 14.7% (in art). The higher overheads are caused by contention between state and data for both L1

cache space and bandwidth. 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 would make 32 KB L1 data more power hungry and a larger L1 cache would increase latency.

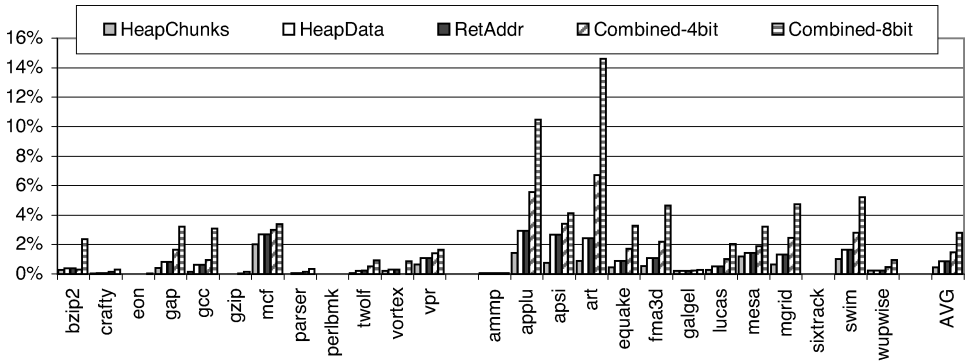
Finally, the Interleaved approach has almost similar overheads as the Split configuration. This configuration has dedicated space for state in each L1 line, so this configuration will exhibit the same performance overheads, even when MemTracker is turned off and will leave the cache slower and more power hungry due to additional bits. Further analysis results are presented in Section 6.5.

Overall, the Split configuration has the relatively better average performance-cost trade-off than other configurations. It is also comparatively easy to integrate into the processor pipeline, using the in-order precommit implementation described in Section 4.4. The additional 16 KB L1 state cache in this configuration is only half the size of 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 4.6) and, with additional L1 bandwidth, has similar 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.

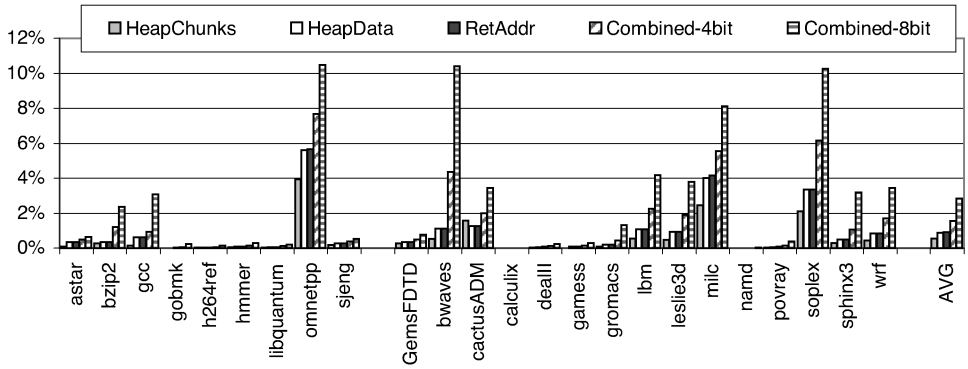
6.2 Performance with Different Checkers

Figure 11 shows that the overhead mostly depends on the number of state bits per word used by a checker. The 1-bit HeapChunks checker has the lowest overhead—around 0.5%, on average, and 2.45% worst-case in milc benchmark. Both 2-bit checkers have similar overheads of around 1%, on average, and 4% worst-case for HeapData and 4.15% for RetAddr checker both in milc benchmark. We note that these checkers have different user events—the HeapData checker uses variable-footprint user event instructions 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 not frequent, 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.

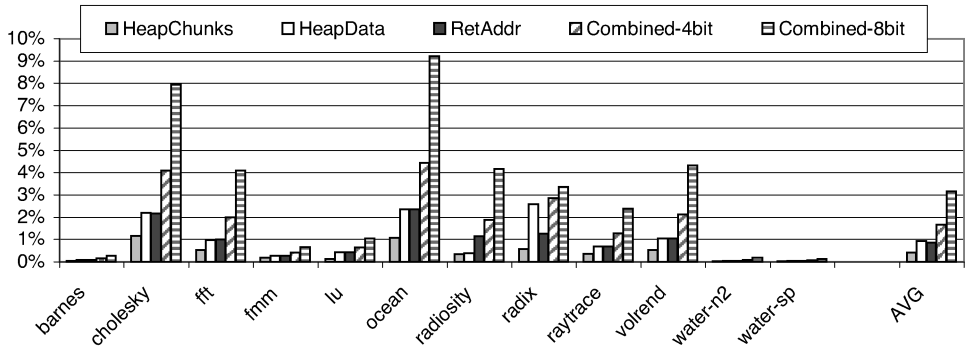
The four-bit Combined checker has an overhead of 1.5%, on average, and about 6.7% worst-case (in art). This overhead is larger than in less-demanding checkers, mainly due to larger state competing with data for L2 cache space and bus bandwidth. Still, even the "high" 6.7% worst-case overhead is low enough 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. Finally, as expected, the 8-bit Combined checker exhibits slightly worse overheads than the more efficient 4-bit combined checker. The averages are around 2.8%



(a) Spec2000 benchmarks



(b) Spec2006 benchmarks



(c) Splash-2 benchmarks

Fig. 11. Overhead of different checkers in MemTracker, with Split L1 state caching using a 16 KB L1 state cache.

for SPEC benchmarks and 3.15% for SPLASH-2 benchmarks with the worst-case of around 14.5% in art benchmark. These higher overheads are largely due to higher contention for capacity and bandwidth for L2 cache shared by both data and state. We use 256 entry PSTT-cache in our experiments. We did

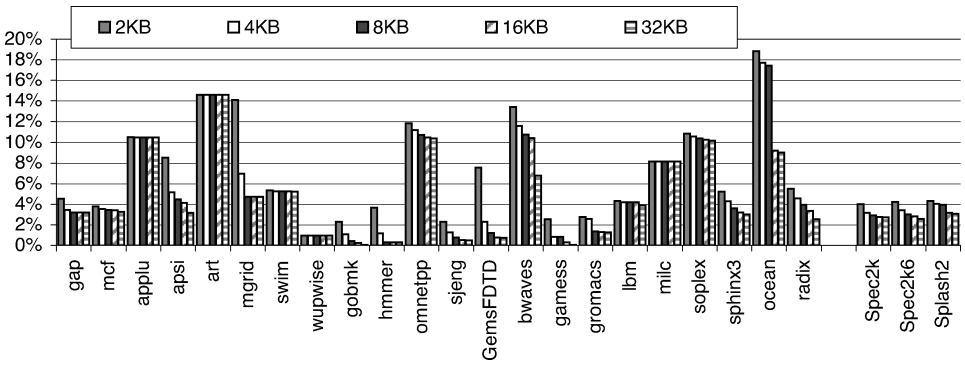


Fig. 12. Performance overhead variation due to different sizes of L1 state cache.

not notice any significant overheads due to misses in PSTT cache beyond the initial cold misses. However, we note that, as more sophisticated checkers are implemented, there may be an increased demand for PSTT-cache entries and hence, a larger PSTT might be needed.

6.3 Sensitivity Analysis

We performed additional experiments with 2 KB, with 4 KB, with 8 KB, and with 32 KB (all with 4-way associativity and 64-byte line size) state L1 state caches in the *Split* configuration in addition to our default 16 KB L1 state cache used in our experiments. Figure 12 shows the results of these experiments for a subset of the benchmarks along with averages across entire benchmark suites. We find that state caches larger than our default of 16 KBs bring negligible performance improvements ($<$0.5%) in all applications and on average, which indicates that the 16 KB cache is large enough to capture most of the first working set for MemTracker state. The smaller 8 KB cache still shows almost similar overheads for all benchmarks except ocean benchmark where an 8 KB cache has higher miss rate than the 16 KB. For smaller cache sizes, namely 4 KB and 2 KB, the overheads progressively worsen due to higher contention for cache capacity. The reason for this is that the smaller state cache "covers" less memory than the L1 data cache for the 8-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 (64 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. While some applications like mgrid, bwaves, sphinx3, and ocean show that performance can be progressively improved with higher capacity state caches, certain applications like applu, art, and soplex are agnostic to increasing cache sizes. On further investigation, we find that these applications have increased contention on L2 bandwidth caused by the sharing between data and state information. However, on average, we find that caches with higher capacity tend to improve performance overheads across benchmarks.

We also conducted experiments in which we disable state prefetches (see Figure 7(a)) in the Split configuration. We find that the average overhead

increases by around 2.7% 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 16 KB state cache results in a good cost-performance tradeoff, and even though smaller state caches 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 [Witchel et al. 2002] and with an approximation of checking based on software-only instrumentation. It should be noted that Mondrian was not intended to be a general-purpose tagging scheme, and our experiments do not show how Mondrian performs poorly for its intended purpose (fine-grain protection). Instead, we use Mondrian to show that fine-grain memory protection cannot be efficiently used to emulate MemTracker. 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 nonreturn-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 exception handler (which must update Mondrian's permissions trie structure), and the return to the exception site.

For software-only checking, we only model a 5-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 [Shetty et al. 2006] the actual reported average duration of a (highly optimized) load or store check is 18 to 480 cycles, depending on the application.

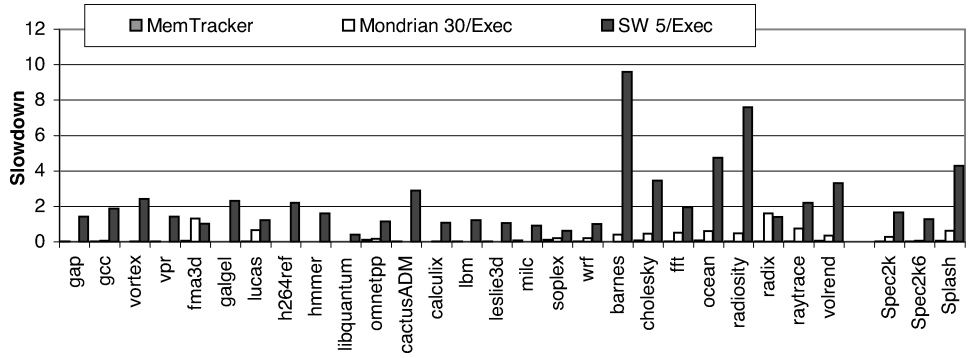


Fig. 13. Effect of state changes in software handlers.

The results of these experiments are shown in Figure 13. For MemTracker, we use a Split L1 caching configuration with a 16 KBs 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 slowdown for software-only checking is 1.6X (times), on average, across SPEC-2006 benchmarks, 1.3X across SPEC-2000 benchmarks, and 4.3X for SPLASH-2 applications, with a worst-case of 9.6X for barnes. Due to our optimistic assumptions for software-only checking, this overhead is lower than previously reported numbers [Newsome and Song 2005; Valgrind Developers 2005] 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 29% average across SPEC-2006 applications, 6% average across SPEC-2000 benchmarks, and 63% across SPLASH-2 applications. Many applications in the Mondrian-based scheme behave similar to MemTracker as the number of state changes in these applications are relatively few. However, certain applications like fma3d, lucas, barnes, cholesky, and radix have much higher overheads due to higher frequency of state changes. Since MemTracker can perform such state changes automatically in hardware while a Mondrian-based scheme would need to invoke a software handler (assumed to have a 30-cycle penalty), they result in large overheads for some benchmarks. 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 Latency, Area and, Power Overheads

We perform experiments to determine the latency, area, and power overheads due to different caching configurations namely Split, Shared, and Interleaved. We use Cacti 4.2 [Jouppi et al. 2006], an integrated access time, area, and

	Latency	Area	Power
Split	0.0%	56.9 %	21.9%
Shared	0.0%	0.0%	101.5%
Interleaved	21.4%	25.1%	43.0%

Fig. 14. Latency, area, and power overheads, expressed as a fraction of the latency, area, and power of the unmodified 32KB L1 cache.

dynamic power model to model overheads, due to our different caches. The results are shown in Figure 14.

The Split configuration stores the state information in a separate but smaller cache. Hence, the latency of accessing the state can be hidden by the longer latency of the L1 data cache. It is slightly expensive in terms of area; however, it occupies only 57% (roughly) of the size of already small L1 data cache. Modern processors estimate approximately 2.9% of total on-chip area dedicated to L1 data cache [McDonald et al. 2005]. Hence, for applications that need minimum performance overheads and are highly sensitive to L1 data cache misses, this small addition is needed and can easily be accommodated. The Shared cache configuration does not add latency or area to the L1 data cache; however, every memory access involves an extra access for state information (which doubles the energy cost) and a small percentage of memory access have a third access to the data cache to update the state information. Hence, overall, the power "more than doubles" for the L1 data caches that stores both data and state. For Interleaved caching configuration, the cache lines are extended to accommodate the 8-bit states corresponding to each memory word. Hence, the L1 cache access latency increases unnecessarily by 21.4%, even when MemTracker is turned off. Also, the area of the L1 data caches increases by a quarter of its original size. While all loads will fetch the state information along with data, stores have to perform state check before actually performing the write to memory. Additionally, there will be a small percentage of time when a state update is needed. Hence, the dynamic power cost grows by almost 43% for the interleaved configuation.

6.6 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 redirect 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 crafty, 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 article. 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 article 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 two different implementations of a checker that combines all three. Even for the most demanding combined checker that maintains 8 bits of state for every memory word, we observe performance overheads of around 3%, on average, and 14.5% worst-case across SPEC and SPLASH-2 applications.

We examine several approaches to caching MemTracker state in the on-chip L1 cache and find that a small dedicated state L1 cache offers good performance benefit at a low cost. In the L2 cache and memory, MemTracker state is stored just like any other data without any extra support. We also describe how to implement MemTracker by adding two in-order stages to the back end of the processor pipeline and by avoiding significant changes to most of the processor pipeline. With its low performance overhead, simple implementation

and improved programmability, we believe that MemTracker is one right step toward hardware support mechanisms that will be needed by developers to continuously monitor the highly complex applications of the future.

REFERENCES

AKRITIDIS, P., CADAR, C., RAICIU, C., COSTA, M., AND CASTRO, M. 2008. Preventing memory error exploits with wit. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'08)*. IEEE Computer Society, 263–277.

BOLETTA, J. 2002. Security Focus Newsletter #172. http://citadelle.intrinsec.com/mailing/current/HTML/ml_securityfocus_news/0067.html.

CHEN, S., KOZUCH, M., STRIGKOS, T., FALSAFI, B., GIBBONS, P. B., MOWRY, T. C., RAMACHANDRAN, V., RUWASE, O., RYAN, M., AND VLACHOS, E. 2008. Flexible hardware acceleration for instruction-grain program monitoring. In *Proceedings of the 35th International Symposium on Computer Architecture (ISCA'08)*. IEEE Computer Society, 377–388.

CORLISS, M. L., LEWIS, E. C., AND ROTH, A. 2003. Dise: A programmable macro engine for customizing applications. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA'03)*. ACM Press, 362–373.

COWAN, C., PU, C., MAIER, D., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., ZHANG, Q., AND HINTON, H. 1998. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Conference*. USENIX Association, 63–78.

CRANDALL, J. R. AND CHONG, F. T. 2004. Minos: control data attack prevention orthogonal to memory model. In *Proceedings of the 37th International Symposium on Micro-architecture (MICRO-37)*. IEEE Computer Society, 221–232.

DEVIETTI, J., BLUNDELL, C., MARTIN, M. M. K., AND ZDANCEWIC, S. 2008. Hardbound: Architectural support for spatial safety of the c programming language. *SIGARCH Comput. Archit. News 36*, 1, 103–114.

HINTON, G., SAGER, D., UPTON, M., BOGGS, D., CARMEAN, D., KYKER, A., AND ROUSSEL, P. 2001. The micro-architecture of the Pentium 4 Processor. *Intel. Tech. J.* First Quarter.

IBM CORPORATION. 2005. IBM Rational Purify. http://www.ibm.com/software/awdtools/purify/.

JOUPPI, N. P. ET AL. 2006. Cacti 4.2. http://quid.hpl.hp.com:9081/cacti/

KUEHN, J. T. AND SMITH, B. J. 1988. The horizon super-computing system: Architecture and software. In *Proceedings of the ACM/IEEE Conference on Super-computing*. IEEE, 28–34.

MCDONALD, R. G., BURGER, D., AND KECKLER, S. 2005. The design and implementation of the TRIPS prototype chip. http://www.hotchips.org/archives/hc17.

NETHERCOTE, N. AND SEWARD, J. 2007. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd International Conference on Virtual Execution Environments (VEE'07)*. ACM, 65–74.

NEWSOME, J. AND SONG, D. 2005. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS)*. Internet Society.

QIN, F., LU, S., AND ZHOU, Y. 2005. SafeMem: Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs. In *Proceedings of the 11th International Symposium on High-performance Computer Architecture*. IEEE Computer Society, 291–302.

QIN, F., QIN, F.,WANG, C., LI, Z., SEOP KIM, H., ZHOU, Y.,WU, Y., AND WU, Y. 2006. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Micro-architecture (MICRO-39)*. IEEE Computer Society, 135–148.

RENAU, J. ET AL. 2006. SESC. http://sesc.sourceforge.net.

SEWARD, J. 2004. Valgrind: An open-source memory debugger for 86-GNU/Linux. http://valgrind.kde.org/.

SHETTY, R., KHARBUTLI, M., SOLIHIN, Y., AND PRVULOVIC, M. 2006. Heapmon: A helper-thread approach to programmable, automatic, and low-overhead memory bug detection. *IBM J. Res. Dev. 50*, 2/3, 261–275.

SHI, W., LU, C., AND LEE, H.-H. S. 2007. Memory centric security architecture. *Trans. High-perform. Embed. Archit. Compilers I 4050*, 1, 95–115.

SPEC. 2006. Standard performance evaluation corporation benchmarks. http://www.spec.org.

SUH, G. E., LEE, J. W., ZHANG, D., AND DEVADAS, S. 2004. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XI)*. ACM, 85–96.

SYMANTEC. 2002. Microsoft IIS HTR chunked encoding heap overflow allows arbitrary code. http://securityresponse.symantec.com/avcenter/security/Content/2033.html.

US-CERT. 2001. 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.

US-CERT. 2004. Buffer overflow in Microsoft Internet Explorer.
http://www.us-cert.gov/cas/techalerts/TA04315A.html.

VALGRIND DEVELOPERS. 2005. *The Valgrind Quick Start Guide*.
http://valgrind.org/docs/manual/quickstart.html.

VENKATARAMANI, G., DOUDALIS, I., SOLIHIN, Y., AND PRVULOVIC, M. 2008. Flexitaint: A programmable accelerator for dynamic taint propagation. In *Proceedings of the IEEE 14th International Symposium on High-performance Computer Architecture (HPCA'08)*. IEEE Computer Society, 173–184.

VENKATARAMANI, G., ROEMER, B., SOLIHIN, Y., AND PRVULOVIC, M. 2007. Memtracker: Efficient and programmable support for memory access monitoring and debugging. In *Proceedings of the IEEE 13th International Symposium on High-performance Computer Architecture (HPCA'07)*. IEEE Computer Society, 273–284.

WITCHEL, E., CATES, J., AND ASANOVIC, K. 2002. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*. ACM Press, 304–316.

WOO, S. C., OHARA, M., TORRIE, E., SINGH, J. P., AND GUPTA, A. 1995. The splash-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA'95)*. ACM, 24–36.

ZHOU, P., LIU, W., FEI, L., LU, S., QIN, F., ZHOU, Y., MIDKIFF, S., AND TORRELLAS, J. 2004. Accmon: Automatically detecting memory-related bugs via program counter-based invariants. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Micro-architecture (MICRO-37)*. IEEE Computer Society, 269–280.

ZHOU, P., QIN, F., LIU, W., ZHOU, Y., AND TORRELLAS, J. 2004. iwatcher: Efficient architectural support for software debugging. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA'04)*. IEEE Computer Society, 224.

ZHOU, Y., ZHOU, P., QIN, F., LIU, W., AND TORRELLAS, J. 2005. Efficient and flexible architectural support for dynamic monitoring. *ACM Trans. Archit. Code Optim. 2*, 1, 3–33.

Received July 2008; revised December 2008; accepted December 2008