Monitoring Data Structures Using Hardware Transactional Memory*

Shakeel Butt[1], Vinod Ganapathy[1], Arati Baliga[2], and Mihai Christodorescu[3]

[1] Rutgers University, Piscataway, New Jersey, USA
{shakeelb,vinodg}@cs.rutgers.edu
[2] AT&T Security Research Center, Middletown, New Jersey, USA
arati.baliga@att.com
[3] IBM TJ Watson Research Center, Hawthorne, New York, USA
mihai@us.ibm.com

Abstract. The robustness of software systems is adversely affected by programming errors and security exploits that corrupt heap data structures. In this paper, we present the design and implementation of TxMon, a system to detect such data structure corruptions. TxMon leverages the concurrency control machinery implemented by hardware transactional memory (HTM) systems to additionally enforce programmer-specified consistency properties on data structures at runtime. We implemented a prototype version of TxMon using an HTM system (LogTM-SE) and studied the feasibility of applying TxMon to enforce data structure consistency properties on several benchmarks. Our experiments show that TxMon is effective at monitoring data structure properties, imposing tolerable runtime performance overheads.

Keywords: Data structure properties, Hardware transactional memory.

1 Introduction

Modern software systems manage a vast amount of data on the heap. Programming errors in such software systems can lead to data structure corruptions that adversely affect their robustness. These errors may result in data structures that violate well-accepted correctness criteria (*e.g.,* dangling pointer errors and heap metadata corruptions) or application-specific data structure consistency properties. Such programming errors are often hard to debug because their effect is delayed, *e.g.,* a dangling pointer error does not result in a crash until the pointer in question is dereferenced.

In this paper, we present the design and evaluation of TxMon, a system that uses hardware transactional memory (HTM) [18,20] to detect data structure corruptions. HTM systems (*e.g.,* [26,23,16,9,10,14]) provide a set of mechanisms in hardware and software to support *memory transactions*, and have been proposed as a mechanism to ease the development of parallel programs. To use transactional memory for concurrency control, programmers use instructions provided by the hardware to demarcate critical sections in a program. The HTM system speculatively executes transactions,

* Funded in part by NSF CNS grants 0728937, 0831268, 0915394 and 0952128. Most of this work was done when A. Baliga was affiliated with Rutgers University.

and ensures that the memory operations performed within these transactions are *atomic*, *i.e.,* they appear to execute in their entirety or not at all, and *isolated*, *i.e.,* their effects are not visible to other concurrently-executing threads until the transaction completes. By ensuring these properties, the HTM system allows transactions to synchronize access to shared data structures.

TxMon is based upon the insight that the mechanisms in HTM systems to implement transactions can also be used to detect data structure corruptions. HTM systems maintain bookkeeping information to track the set of memory locations accessed by each transaction. For example, in the LogTM-SE HTM system [26], speculative values computed by the transaction are written to memory, and the original values at these memory locations are stored in a per-thread *transaction log*, which is used to restore the contents of memory if the transaction aborts as the result of a race condition. Similar bookkeeping information is also available in other HTM systems [23,16,9,14,10].

TxMon interposes on the standard workflow of an HTM system to monitor data structure properties. It inspects the HTM system's bookkeeping information to identify data structures that were modified during a transaction and automatically triggers callbacks that check properties of these data structures, which can include both well-accepted correctness conditions as well as application-specific assertions. We show that in HTM systems that expose their bookkeeping information to software, *e.g.,* LogTM-SE and Rock [10][4], TxMon can be implemented with no hardware modifications. This ensures that applications on these platforms can readily benefit from TxMon.

Contributions. To sum up, this paper makes the following contributions:

(**1**) *Design of TxMon.* We present the design of TxMon, which uses the concurrency control machinery implemented in HTM systems to monitor data structure properties. Among the key features in the design of TxMon are *address maps*, which are a representation of complex data structures. We also present a novel technique to update address maps as the data structures that they represent are modified.

(**2**) *Implementation in LogTM-SE.* We implemented a prototype of TxMon by leveraging the LogTM-SE HTM system. Because LogTM-SE exposes transaction logs to software, our implementation required *no* modifications to the HTM system.

(**3**) *Evaluation of TxMon.* We used TxMon to monitor data structure properties on multi-threaded benchmarks from the Splash-2 suite [25], and two real-world applications, namely ClamAV and Memcached. Our evaluation shows that TxMon is effective at monitoring complex properties and that it imposes an acceptable runtime overhead.

More broadly, TxMon demonstrates that transactional memory hardware can provide additional benefits beyond providing concurrency control. There is still a debate in the community about the correct abstraction to ease parallel programming. In the long term, additional benefits of HTM systems as demonstrated by TxMon and similar recent and ongoing efforts [11,12,7,17,19] can serve as the catalyst that will lead to more research on transactions and their adoption by hardware and software vendors.

[4] We reference Rock although it is abandoned now (for economic reasons) because its design substantially resembles LogTM-SE, which we used for our prototype implementation.

```
(1)  struct item { ...
(2)    rel_time_t time;    //last access time
(3)    unsigned refcount; //reference count
(4)    struct item *next, *prev; ...
(5)  };
(6)  struct item *heads[255], *tails[255];
(7)  process_get_command (key) { ...
(8)    struct item *it = search(key);
(9)    if (it) {
(10)     it->refcount++;
(11)     it->time = curr_time;
(12)     move_to_head(it);
(13)   } ...
(14)   return it;
(15) }
(16) process_add_command (key, value) { ...
(17)   struct item *it = alloc_item(key, value);
(18)   ...
(19) }
```

Fig. 1. Motivating example. This figure shows a simplified code snippet from Memcached.

2 Motivation and Overview

We use the example in Figure 1 to motivate the key requirements that a data structure monitor must satisfy, and then illustrate how TxMon satisfies these requirements. The snippet in Figure 1 is a simplified version of code drawn from Memcached [2], a distributed object caching server that has been adapted by Web services such as Livejournal, Slashdot and Wikipedia.

Memcached is a multi-threaded server that stores key/value pairs. Clients can invoke commands on the server to perform a variety of functionalities, such as fetching the value corresponding to a key, adding a new key/value pair, deleting an existing pair, and so on. Figure 1 shows snippets from the implementation of two such commands that fetch and add key/value pairs. Each key/value pair is stored in exactly one of 255 doubly-linked lists depending upon the size of the value. The arrays `heads` and `tails` store pointers to the heads and tails of these lists. Each element of these lists includes a timestamp field, which denotes the last access time (get or set) of an item, and a reference count field, which stores the number of active clients that are currently accessing that item.

As an object caching system, Memcached employs several complex policies to decide which key/value pairs to cache on the server, and how to organize these pairs in its linked lists. The code of `process_get_command` in Figure 1 depicts one such policy, which ensures that the most recently accessed object is placed at the head of the corresponding list. This property allows Memcached to employ a variant of the LRU algorithm[5] to evict items from the cache.

Even for this simple eviction policy to work correctly, several data structure properties must hold. First, all modifications to the linked lists must ensure that the items of each list are sorted in order of their access times. Second, it must ensure that the heads

[5] The actual eviction policy considers reference counts, access times, and other fields of `item` objects to decide upon a victim.

of linked lists are reachable from tails, and vice versa. Failure to ensure these data structure properties can result in incorrect operation. In particular, Memcached searches for victims from the tails of linked lists. If the first property fails to hold, the choice of the resulting victim will violate the LRU policy. Similarly, if the second property fails to hold, the eviction algorithm may not explore all elements in a linked list.

However, Memcached has a large code base and could contain programming errors; indeed the defect history of Memcached [1] shows over a hundred reports since October 2008. These programming errors could corrupt Memcached's data structures, which in turn may cause the server to malfunction. These errors may not manifest during testing, and deployed code can malfunction when these errors are encountered in the field. In fact, a recent version of Memcached contained an error that failed to decrement reference counts of items properly, thereby leading to memory leaks under rare conditions (because objects with non-zero reference counts are not reclaimed). Such errors can be detected using a framework to monitor data structures. For example, this framework could ensure that linked lists that are modified by a client continue to satisfy the sortedness property. It could also track "old" items with non-zero reference counts, thereby identifying items that leak.

This example motivates four design requirements for a data structure monitor:

(**1**) *Ability to monitor complex data structures.* Verifying properties of a complex data structure may require traversing the data structure. For instance, in the example above, ensuring that the list at `heads[i]` is sorted involves traversing it fully.

(**2**) *Extensibility.* In Memcached, a programmer may wish to verify that the list `heads[i]` is doubly-linked, in addition to verifying sortedness of the list. The monitor must be extensible, and allow the programmer to supply a checker for additional properties.

(**3**) *Applicability to low-level code.* Data structure corruptions are common in applications written in low-level memory-unsafe languages, such as C and C++. The monitor must therefore be applicable to programs written in such languages as well.

(**4**) *Low runtime overhead.* To monitor data structure properties in deployed software, the monitor must ideally be an "always-on" tool, and must impose an acceptable runtime performance overhead.

As we show in Section 4, TxMon satisfies all four requirements. To motivate how TxMon monitors data structure properties, consider an approach in which a programmer *inlines* checks at key locations in the program. For example, to ensure that the linked lists in Memcached are sorted by access time, the code snippet in Figure 1 can include inline checks to ensure this property as a post-condition of the functions `move_to_head` and `alloc_item` (which also adds elements to linked lists). Although apparently simple, an approach that inlines checks must overcome two challenges. First, appropriate data structure checks must be placed at locations where key data structures (*e.g.,* the linked lists in Memcached) are accessed. This requires the programmer to identify all such locations in the program and to identify the set of data structure checks that must be triggered at each of those locations. Identifying data structures accessed can be challenging, especially in the presence of pointer aliasing. This problem is exacerbated as software evolves, because data structure checks must also be modified to reflect changes in the program. Second, in multi-threaded software, the placement of checks must avoid time-of-check to time-of-use errors (race conditions),

```
(1)  struct item { ...
(2)    rel_time_t time;//last access time
(3)    unsigned refcount;//reference count
(4)    struct item *next, *prev; ...
(5)  };
(6)  struct item *heads[255], *tails[255];
(7)  process_get_command (key) {
(8)    transaction (txmon_entry) {  ...
(9)      struct item *it = search(key);
(10)     if (it) {
(11)       it->refcount++;
(12)       it->time = curr_time;
(13)       move_to_head(it);
(14)     } ...
(15)   }
(16)   return it;
(17) }
(18) process_add_command (key, value) {
(19)   transaction (txmon_entry) {  ...
(20)     struct item *it =
(21)     alloc_item(key, value); ...
(22)   }
(23) }
(24) //Server initialization code
(25) for (i = 0; i < 255; i++) {
(26)   register_ds(&heads[i],
         check_sort, &heads[i]);
(27)   initialize address_map(&heads[i])
(28) }
```

Implementation of TxMon's monitor

```
(m1)  txmon_entry (void) {
(m2)    retval = true;
(m3)    accset = Get accessed memory locations from HTM;
(m4)    for (each addr ∈ accset)
(m5)      for (each ds ∈ registered data structures)
(m6)        if (addr ∈ address_map(ds))
(m7)          retval &= value returned by callback for ds;
(m8)    if (!retval) invoke transaction_abort;
(m9)  }
(m10) void register_ds (void *dsptr, void *cback, ...) {
(m11)   // register cback as checking callback for dsptr
(m12)   // optionally register arguments for the callback
(m13) }
(m14) bool check_sort (struct item *hd) {
(m15)   if (list not sorted by item->time) return false;
(m16)   /* update address_map(hd) */
(m17)   for (it = hd; it ≠ NULL; it = it->next) {
(m18)     add &(it->next), &(it->prev) and &(it->time)
            to address_map(&hd);
(m19)   return true;
(m20) }
```

Fig. 2. Using TxMon. Code snippet from Figure 1 modified to use TxMon to monitor the lists headed at `heads[0]`, ..., `heads[254]`. Lines `m1`-`m13` are part of the TxMon monitor. The lines in bold-faced font show the code that a programmer must add.

in which a concurrently-executing thread may modify a data structure in the interval between property verification and use of the data structure.

The TxMon system developed in this paper eases the task of placing such data structure checks in the program. Rather than requiring a programmer to manually inline checks, TxMon instead requires code that manipulates key data structures to be embedded in *transactions*. In Figure 2 for instance, all operations on the linked lists with heads in the array `heads[]` happen within transactions. In multi-threaded code that uses transactional memory for synchronization, such transactions will naturally be placed around code that manipulates shared data structures. However, TxMon also applies to single-threaded programs. In such cases, transactions must be placed around the code where data structures are updated. In both cases, TxMon triggers property checks on data structures that were modified when the transaction completes execution.

In addition to placing transactions, the programmer has three key responsibilities:

(**1**) *Register data structures to be monitored.* The programmer must use an API supplied by TxMon to register data structures that must be monitored. In Figure 2, the `register_ds` calls placed in the initialization code of Memcached notify TxMon that the lists headed by `heads[]` are data structures that must be monitored.

(**2**) *Supply address maps.* The programmer must supply an *address map* for each registered data structure. The address map is an abstraction that stores the set of all memory

addresses associated with that data structure. We defer an overview of how address maps are computed and maintained to Section 4.3.

(**3**) *Supply checker callbacks.* The programmer specifies the properties to be verified in *checker callbacks* associated with each data structure. TxMon ensures that if a data structure is modified in a transaction, then the corresponding callback executes at the end of the transaction and verifies that the data structure's properties hold. In Figure 2, the same checker callback (`check_sort`) is associated with each of the 255 linked lists. This function checks the property that these lists are sorted by last access time.

Upon completion of a transaction, control transfers to the entrypoint of TxMon's data structure monitor. As shown in Figure 2, the entrypoint (`txmon_entry`) is a function pointer that is registered using an argument to the `transaction{...}` keyword. TxMon obtains the set of memory locations accessed by the transactions from the bookkeeping information maintained by the HTM system—in our implementation, we obtain these locations from the transaction's undo log (see Section 3). TxMon's data structure monitor determines whether the memory addresses accessed during the transaction are also contained in the address maps of any of the data structures registered with it. If so, it triggers the checker callback associated with the corresponding data structure, which verifies the properties of that data structure.

The key point to note is that unlike the approach that inlines data structure checks, *the programmer need not specify which checker callbacks must be invoked at the end of a transaction*. Rather, TxMon uses the HTM system's bookkeeping information (*i.e.,* the undo log) to infer which callbacks must be invoked.

3 HTM Systems

In this section, we provide background on hardware transactional memory, focusing on the features relevant to the design of TxMon. HTM systems typically extend hardware instruction sets with new primitives that define the start (`begin_tx`) and end of transactions (`end_tx`). They ensure atomicity and isolation for all executing transactions, but vary widely in how they do so [20]. Nevertheless, all HTM systems implement mechanisms for *conflict detection* and *version management*.

Conflict detection mechanisms allow the HTM system to detect race conditions between concurrently executing transactions. An HTM system detect conflicts by intersecting the memory locations read/written by a transaction with those of other in-flight transactions. If a conflict is detected, the HTM must abort at least one conflicting transaction. Version management mechanisms allow the HTM system to record the set of data modifications made by a transaction. When a transaction is committed (or aborted), the HTM system consults the version manager to commit (or discard) the changes made by the transaction. For instance, LogTM-SE logs the old values of the memory locations modified by a transaction (in a per-transaction log), and uses the log to restore memory if the transaction aborts.

Both the conflict detection and version management mechanisms of an HTM system thus maintain a record of the memory locations modified by a transaction. TxMon can use the information from either mechanism to trigger data structure checks. However, in this paper, we focus on a system design that obtains memory access information from

the version management mechanism of an HTM system. Our choice was motivated by the observation that HTM systems often store more precise version management information than they do conflict detection information. This is because conflict detection mechanisms need only conservatively determine if two transactions are in conflict. A false conflict wrongly aborts a transaction, but does not affect the correctness of the system. In turn, this allows HTM systems to maintain read/write sets using fixed-size hardware structures, such as Bloom filters, which over-approximate the set of memory locations accessed by a transaction.

In contrast, version management information is used to determine the values of memory locations at the end of a transaction (either upon a commit or an abort), and must therefore be precise. Precision is important because it affects the performance of TxMon. If TxMon leverages imprecise read/write sets to identify the set of memory locations accessed by a transaction, it may trigger checks on data structures that were not otherwise accessed within the transaction. In turn, the execution of these additional checks may result in poor performance and spurious reports of failed data structure checks. Indeed, a preliminary design of TxMon that relied on a Bloom filter implementation of read/write sets had overheads in excess of 800% on the benchmarks reported in Section 5.

Our prototype implementation of TxMon, described in the following section, uses LogTM-SE [26]. This HTM system implements read/write sets as Bloom filters, uses a software-accessible undo log for version management, and allows transactions of unbounded length. Because version management information is accessible from software, LogTM-SE offers the additional benefit of allowing TxMon to be implemented *without any hardware modifications*.

4 Design and Implementation

TxMon enforces data structure properties by interposing on the standard workflow of an HTM system, as shown in Figure 3. In a standard HTM system, a transaction that has completed execution is passed to the conflict detection module, which determines whether to commit or abort the transaction. Our modifications to the HTM system's workflow ensure that TxMon's *data structure monitor* is first invoked upon the completion of the transaction.

The data structure monitor, which is implemented in software, verifies properties of the data structures that were accessed in the transaction. To identify the data structures accessed in the transaction, the monitor consults the transaction's undo log, which is stored in software. If the monitor returns successfully, it passes control to the conflict detection module, which then proceeds as before. If the monitor detects a data structure that violates a programmer-specified property, it invokes the HTM's mechanisms to abort the transaction. Thus, a transaction is committed only if it does not conflict with other transactions *and* the data structures that it modifies satisfy all programmer-specified properties. While TxMon's data structure monitor is invoked at the end of transactions by default, it can optionally be triggered at any point during the execution of a transaction using a function call, *e.g.,* a call to `txmon_entry`. We modified the `transaction{...}` construct to additionally accept a parameter, which specifies the entrypoint of TxMon's data structure monitor, *e.g.,* as shown in Figure 2.

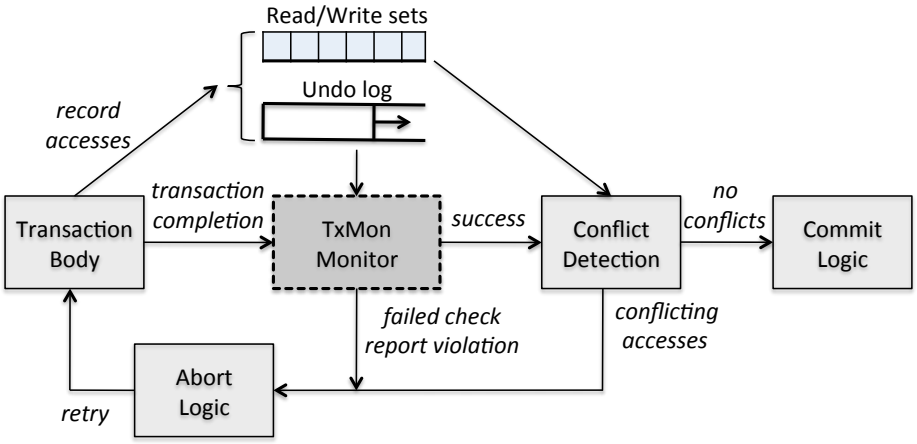

Fig. 3. Workflow of a TxMon-enhanced HTM.

4.1 Implementation in LogTM-SE

We implemented TxMon using the LogTM-SE HTM system. This system is built for the SPARC architecture, and the HTM hardware has been simulated using the Virtutech Simics full system simulator. LogTM-SE employs eager conflict detection, *i.e.,* conflicts between transactions are detected as soon as they happen, and supports nested transactions. LogTM-SE also supports strong atomicity [20], *i.e.,* it can detect conflicting data accesses even if one of them was generated by non-transactional code.

Our choice of LogTM-SE as the implementation platform was motivated by three reasons. First, as a practical matter, LogTM-SE is a mature, freely-available, state of the art HTM system. Rather than building a new HTM system from scratch, using LogTM-SE allowed us to evaluate what changes would be necessary to an existing HTM system to monitor data structure properties. Second, LogTM-SE supports transactions of unbounded length. This feature is important for real-world applications, such as ClamAV and Memcached, in which data structures are modified by complex functions. Third, and most significant, LogTM-SE implements version management using a software-accessible undo log. Applications that use LogTM-SE transactions allocate memory for the log in their address space during startup. During execution, LogTM-SE eagerly updates memory locations modified within each transaction with speculative values, and checkpoints the original values at these locations within a per-transaction undo log. The log itself is stored in software, but is updated by the HTM hardware. TxMon's data structure monitor, which is implemented in software and is loaded into the application's address space, can also access the undo log to obtain the set of memory locations accessed by the transaction. As a result, we were able to implement TxMon with *no modifications* to the proposed LogTM-SE hardware.

While the design of LogTM-SE eased the implementation of TxMon, it may also be possible to design TxMon-like monitors for HTM systems that use alternative designs.

For instance, some HTM systems (*e.g.,* [3]) buffer speculative updates and commit them at the end of the transaction (if there are no conflicts). In such systems, it suffices to expose the buffer that stores these updates to the data structure monitor. In some cases, hardware changes may be necessary to expose such state to the monitor (*e.g.,* the addition of new instructions to the ISA), but we expect that such changes will be relatively minor.

4.2 TxMon's Data Structure Monitor

The main responsibility of TxMon's data structure monitor is to trigger checks to verify the properties of all data structures accessed by a transaction. At the heart of the monitor is a table that stores *address maps* of data structures to be monitored, and the checking callback associated with each data structure. Programmers can register/unregister data structures to be monitored using an API exported by the monitor (*e.g.,* the function `register_ds` shown in Figure 2). The address map of a data structure contains the set of all memory locations of the data structure that are relevant to the property to be checked. The programmer must also supply the address map of each data structure (or specify how the address map must be computed) when he registers the data structure. For the example considered in Section 2 (*i.e.,* verifying the sortedness of the lists `heads[0]`, ..., `heads[254]`) the address map of each of these lists should at least contain the memory locations of all `next`, `prev` and `time` fields of each `struct item` node in the list. This is because any code that mistakenly violates the sortedness property must modify at least one of these fields of a `struct item` node.

In our implementation, address maps are implemented using hash tables that store the set of memory locations in a data structure. When the application invokes the TxMon monitor, the monitor fetches the set of memory locations accessed by the transaction from its undo log. It then queries address maps to determine data structures that were accessed by the application and triggers the callbacks associated with those data structures. The address map table also stores the arguments to be passed to the callback, *e.g.,* the argument `head[0]` is passed to the `check_sort` callback when triggered on the first linked list in Memcached.

Recall that TxMon's data structure monitor is triggered via a function call at the end of a transaction. As a result, the monitor and all the data structure checks that it triggers execute in the context of the transaction. This feature is useful for:

(**1**) *Detecting concurrent data structure modifications.* The property checks triggered by the monitor may need to traverse the data structures being monitored. In multi-threaded software, the monitored data structures may be shared, and may be modified by concurrently executing threads as the property checker traverses them. If the checker traverses a data structure when it is temporarily in an inconsistent state, it will report spurious property violations. By executing the property checks in the context of the transaction itself, TxMon ensures that any concurrent modifications of monitored data structures by other transactions will conflict with the current transaction. These conflicts are detected automatically by the HTM system's machinery, which will then abort one of the conflicting transactions. The LogTM-SE HTM system can also detect conflicting accesses to monitored data structures from *non-transactional* code because it offers strong atomicity.

(**2**) *Protecting monitor state.* The monitor itself stores address maps, which are shared data structures. As explained below, address maps may be updated as data structures evolve. Executing TxMon's monitor in the context of the transaction ensures that concurrent modifications of monitor state will be identified by the HTM system's conflict detection machinery.

4.3 Computing and Maintaining Address Maps

Because the monitor detects accesses to a data structure by comparing the entries in the undo log to its address map, this map must be updated periodically to reflect changes to the data structure. For example, the addition of a new node or the deletion of an existing node in `heads[0]` must appropriately modify its address map in the TxMon monitor. One way to achieve this goal is to register/unregister elements of a data structure as they are allocated/destroyed. In the example shown in Figure 1, this would require the programmer to register each new item `it` when it is created in `process_add_command`. However, this approach is impractical for large code bases, because it requires the programmer to update address maps at several locations in the code.

We alleviate this problem by automating the creation of address maps. During program startup, we only require that the *heads* of data structures be registered with the TxMon monitor, to indicate which data structures must be monitored. For example, in Figure 1, the programmer only registers pointers to the heads of each of the 255 linked lists `heads[0]`, . . ., `heads[254]`.

To create address maps, we leverage the insight that the callback associated with each data structure *must* access all its memory addresses that are relevant for the verification of that property. We can therefore piggyback address map creation with data structure property verification. To do so, we require the programmer to specify how the address map of a data structure must be updated within the callback of that data structure. In Figure 2, code that updates the address maps of the list passed as an argument to `check_sort` is supplied in lines `m16`-`m18`. As this callback executes, TxMon can update its address map for the data structures visited. The size of the address map is proportional to the size of the data structure.

The architecture of TxMon allows arbitrary C functions to be registered as callbacks for a data structure. The programmer can check the data structure specific properties in these functions e.g. the sortedness property of the linked lists `heads[0]`,. . ., `heads[254]`. We have also implemented a library that allows programmers to easily create checker callbacks for properties inferred by Daikon [13].

5 Evaluation

We evaluated TxMon using two macrobenchmarks, namely Memcached and ClamAV, and three microbenchmarks from the Splash-2 suite. Figure 4 summarizes the benchmarks and workloads, and the corresponding data structures and properties monitored. Our experiments used the default configuration of Simics, which simulates an n-core UltraSPARC-III-plus processor running at 75MHz (we varied n for different benchmarks, as described in the subsections that follow), with $n\times$256MB RAM (*i.e.,* we

	Benchmark	Workload	#Tx-S	#DS	#Prop	#Tx-D
(1)	Memcached-1.4.0	Insert/query 100 pairs	7	255 linked lists	3	500
(2)	Clamscan-0.95.2	Scan 356 files	23	`engine`	22	374
(3)	Barnes	16K particles	3	Octree	1	68,819
(4)	Radiosity	batch	44	Task queues	1	239,949
(5)	Raytrace	teapot (small image)	10	Task queues	1	47,751

Fig. 4. Summary of benchmarks, workloads, data structures and number of properties monitored. The "#Tx-S" column shows the number of transactions added to the code of the benchmark, "#DS" shows the number of data structures monitored, "#Prop" shows the number of properties enforced, while "#Tx-D" shows the number of transactions executed at runtime.

	Version	Ops/sec
(1)	Unmodified (baseline)	7052
(2)	Ported to LogTM-SE (no TxMon)	7066 (1×)
(3)	With TxMon enabled	5937 (1.18×)
	.....	.....
(4)	TxMon/Walking log only	6,615 (1.06×)

Fig. 5. Performance of Memcached.

scaled memory proportional to the number of available cores), a 32KB instruction cache, 64KB data cache, and an 8MB L2 cache, running a Solaris 10 operating system. We extended Simics with the Wisconsin GEMS suite (version 2.1) to simulate a LogTM-SE HTM system. Our implementation of TxMon used 1024-bit Bloom filters to store read/write sets for conflict detection.

5.1 Memcached

This section presents a detailed performance evaluation of TxMon on Memcached. As discussed earlier, Memcached stores key/value pairs and supports operations such as inserting new key/value pairs and querying the value associated with a key. We converted Memcached to use transactional memory for synchronization by replacing each use of lock-based synchronization to use transactions instead. We registered each of the 255 linked lists that it uses to store key/value pairs with the TxMon monitor. We wrote checkers to enforce the following properties: (a) the tail of each list is reachable from the head by following `next` fields; (b) the head of each list is reachable from the tail by following `prev` fields; (c) items in each list are stored sorted in decreasing order of the last access time.

We used Memcached version 1.4.0 for our experiments and ran a workload that inserted 100 key/value pairs, and then queried Memcached for the values corresponding to each of the 100 keys that were just inserted. We measured average performance of Memcached as it performed these 200 operations. For this benchmark, we used an 8-core configuration of our Simics testbed, with 4 Memcached server threads processing requests received from a client thread.

As Figure 5 shows, TxMon imposed a moderate (1.18×) overhead as it enforced data structure properties. We conducted another experiment to better understand the source of this overhead. We modified TxMon's data structure monitor to walk the undo

log and fetch the set of memory locations accessed, but did not trigger any data structure checks using this information. That is, using the monitor shown in Figure 2 as an example, lines `m5`-`m7` did not execute. Entry (4) of Figure 5 show the performance of Memcached of this experiment. As this figure shows, the operation of walking the undo log to fetch addresses imposed an overhead of $1.06\times$. TxMon cannot avoid this overhead, because it must read the undo log to decide which data structure checks to trigger.

The cost of performing checks depends on a number of factors, such as the type of workload, the number and size of the data structures being monitored, and the time-complexity of performing checks. For instance, the overheads of performing data structure checks in Memcached increased to $1.37\times$ when the workload was modified to insert and query 200 key/value pairs and to $1.73\times$ for 500 key/value pairs. As another example, it may well be that a workload consists of a number of $O(1)$ operations on Memcached's linked lists (*e.g.,* modifications to the heads of the lists), but that each of these operations triggers data structure checks that cost $O(n)$ (*e.g.,* traversal of the entire list). In such cases, the overhead of TxMon will be significantly higher if data structure checks are triggered naïvely. One way to reduce the overhead is to trigger checks with a probability $1/p$ at the end of trasaction, thus data structure checks will only be triggered once every p modifications to the data structure.

5.2 ClamAV

We evaluated TxMon's ability to monitor complex data structure properties on Clamscan, a command line version of ClamAV. Clamscan uses a virus definition database to scan a set of input files, and determines if any of these files contain patterns in the database. Clamscan maintains several data structures to represent the virus definition database. Clamscan may potentially contain vulnerabilities that can be exploited by malware to hijack its execution and evade detection (*e.g.,* [4]). It is therefore critical to protect the integrity of Clamscan's data structures, such as those that represent the virus definition database.

We used Clamscan version-0.95.2 with its default virus definition database. We ran Clamscan on a uniprocessor configuration of our Simics testbed, and used it to scan the contents of a directory containing 356 files. We modified Clamscan so that code that accesses critical data structures is embedded in transactions. In all, we modified the code to place 23 transactions. We identified one critical data structure, called `engine` (of type `struct cl_engine`). This data structure has several fields, which store scan settings, file types to be scanned, and a pointer to an internal representation of the virus database, stored as a trie, as specified by the Aho-Corasick algorithm [6]. We wrote checkers for a total of 22 properties for this data structure.

As the Figure 6 shows, TxMon imposes an overhead of $1.03\times$ on Clamscan. Although Clamscan manages several complex data structures that can be expensive to traverse, the low runtime overhead observed is because most of these data structures are not modified during normal execution of Clamscan. However, a memory corruption bug or a security exploit that modifies monitored data structures will trigger TxMon to traverse those data structures. This experiment shows that TxMon can be adopted as an "always on" tool to monitor the integrity of Clamscan's data structures.

	Version	Time
(1)	Unmodified (baseline)	10.95s
(2)	Ported to LogTM-SE (no TxMon)	10.99s (1×)
(3)	With TxMon enabled	11.30s (1.03×)

Fig. 6. Performance of Clamscan.

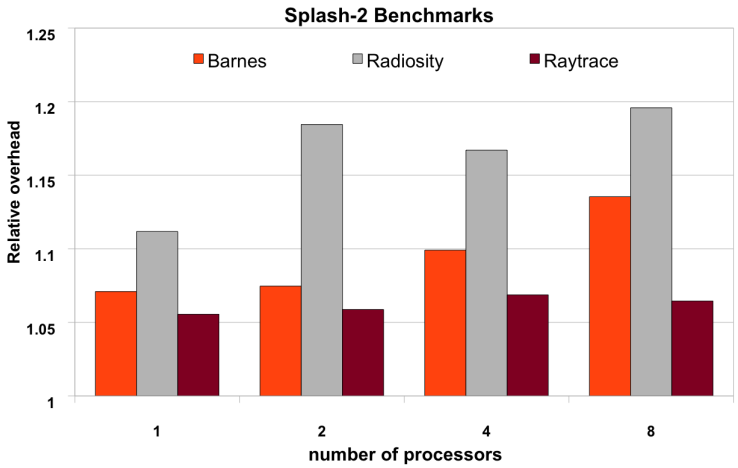

Fig. 7. Performance of Splash-2 benchmarks. This figure shows the overhead of a transactionalized TxMon-enhanced benchmark relative to one that does not employ data structure checks.

5.3 Splash-2 Benchmarks

The Splash-2 suite [25] contains several multi-threaded benchmarks that have previously been used in transactional memory research. We used three benchmarks from this suite, namely Barnes, Radiosity and Raytrace, which were converted in prior work [26,23] to use transactions for synchronization. For each benchmark, we identified a complex data structure used by the benchmark (see Figure 4). The checking callback for each data structure simply traversed the data structure and updated its address map.

Figure 7 shows the overheads that TxMon imposed on the execution of each of these benchmarks on various testbed configurations (in which we varied the number of cores). For each benchmark, we calculated overheads relative to a version that employed transactions only for concurrency control, *i.e.,* TxMon was disabled, so no data structure checks were performed. The overheads ranged from about $1.05\times$ for Raytrace to about $1.20\times$ for Radiosity. These experiments again show that the overheads imposed by TxMon are tolerable even if it is configured to be an "always-on" monitoring tool.

6 Related Work

The use of transactions to monitor data integrity was first suggested by the relational database community [24,15]. The idea was to use the transaction machinery implemented by database systems to additionally check data consistency when the database is modified. Recent work has adapted these ideas to isolate and recover from faults in software by creating custom implementations of transactions and speculative execution mechanisms [21,22].

With advances in transactional memory, researchers have begun to explore similar applications using hardware and software support for transactional memory [17,19,11,7]. Harris and Peyton Jones [17] first explored the use of STM systems to monitor data structure accesses. Their work used an STM system for Haskell to monitor programmer-specified data invariants. Because it relies on STM extensions for a specific language, it is not applicable to the general case of monitoring applications written in low-level languages. Although compiler support for STMs may make their technique applicable to low-level languages (*e.g.,* [5]), prior work suggests that STMs impose significant runtime overheads, suggesting that an STM-based approach may not be a practical option to build an "always-on" data structure monitor [8]. TxMon addresses this problem by migrating to HTM systems, which mitigate the overhead of maintaining and updating transactional bookkeeping information in software.

Researchers have made the case for deconstructing HTM systems, and reusing HTM hardware for applications beyond concurrency control [19,11]. In particular, the position paper by Hill *et al.* [19] describes the use of HTM machinery to implement a data watchpoint framework. Although the ideas outlined in that paper are similar to those adopted by TxMon, our work explores the challenges of building a data structure integrity monitor using the basic watchpoint framework outlined in that paper.

7 Summary

This paper presented TxMon, a system that uses hardware transactional memory to monitor data structure properties. TxMon applies to software written in low-level languages, and can ensure that complex data structures satisfy a rich set of correctness properties. Experiments with both microbenchmarks and application benchmarks show that TxMon imposes tolerable runtime overheads.

References

1. Issues – memcached – project hosting on Google code. `http://code.google.com/p/memcached/issues`.
2. Memcached: A distributed memory object caching system. `http://memcached.org`.
3. Stanford transactional coherence and consistency project. `http://tcc.stanford.edu`.
4. ClamAV multiple vulnerabilities, December 2007. Secunia Advisories – SA28117.
5. A. Adi-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, and B. Saha. Compiler and runtime support for efficient software transactional memory. In *ACM Conf. on Prog. Lang. Design & Impl.*, June 2006.

6. A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. In *Comm. ACM*, 1975.
7. J. Bobba, W. Xiong, L. Yen, M. D. Hill, and D. A. Wood. StealthTest: Low overhead online software testing using transactional memory. In *Intl. Symp. Parallel Architectures and Compilation Techniques*, September 2009.
8. C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, Stefanie Chiras, and S. Chatterjee. Software transactional memory: Why is it only a research toy? *Comm. ACM*, 2008.
9. L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas. Bulk disambiguation of speculative threads in multiprocessors. In *Intl. Symp. Comp. Arch.*, June 2006.
10. S. Chaudhry. Rock: A third generation 65nm, 16-core, 32 thread + 32 scout-threads CMT SPARC processor. In *HotChips*, 2008.
11. J. Chung, W. Baek, N. G. Bronson, J. Seo, C. Kozyrakis, and K. Olukotun. ASeD: Availability, security, and debugging using transactional memory (poster). In *SPAA*, June 2008.
12. J. Chung, M. Dalton, H. Kannan, and C. Kozyrakis. Thread-safe dynamic binary translation using transactional memory. In *IEEE Symp. High Perf. Comp. Arch.*, February 2008.
13. M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comp. Prog.*, December 2007.
14. B. Goetz. Optimistic thread concurrency: Breaking the scale barrier, 2009. Azul Systems Technical Whitepaper.
15. M. Hammer and D. McLeod. A framework for data base semantic integrity. In *Intl. Conf. Soft. Engg.*, 1976.
16. L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Intl. Symp. Comp. Arch.*, June 2004.
17. T. Harris and S. Peyton-Jones. Transactional memory with data invariants. In *TRANSACT*, June 2006.
18. M. Herlihy and J. E. B. Moss. Transactional support for lock free data structures. In *Intl. Symp. Comp. Arch.*, 1993.
19. M. D. Hill, D. Hower, K. E. Moore, M. M. Swift, H. Volos, and D. A. Wood. A case for deconstructing hardware transactional memory systems. In *UW-Madison Computer Sciences Technical Report CS-TR-2007-1594*, 2007.
20. J. R. Larus and R. Rajwar. *Transactional Memory*. Synthesis Lectures on Comp. Arch. Morgan Claypool, 2006.
21. A. Lenharth, V. Adve, and S. T. King. Recovery domains: An organizing principle for recoverable operating systems. In *ACM Conf. Architectural Support for Programming Languages and Operating Systems*, March 2009.
22. M. E. Locasto, A. Stavrou, G. F. Cretu, and A. D. Keromytis. From STEM to SEAD: Speculative execution for automated defense. In *USENIX Annual Tech. Conf.*, June 2007.
23. K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *IEEE Symp. High Perf. Comp. Arch.*, February 2006.
24. M. Stonebraker. Implementation of integrity constraints and views by query modification. In *ACM SIGMOD*, 1975.
25. S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Intl. Symp. on Comp. Arch.*, June 1995.
26. L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. In *IEEE Symp. High Perf. Comp. Arch.*, February 2007.