Monitoring Data Structures using Hardware Transactional Memory
Rutgers University DCS-TR-662, December 2009

Shakeel Butt Arati Baliga Vinod Ganapathy
Rutgers University
{shakeelb,aratib,vinodg}@cs.rutgers.edu

Mihai Christodorescu
IBM TJ Watson Research Center
mihai@us.ibm.com

Abstract

The robustness of software systems is adversely affected by programming errors and security exploits that corrupt heap data structures. Such corruptions are especially common in software written in low-level languages and in extensible software systems, such as operating systems and browsers, where untrusted extensions can corrupt data structures in a software kernel. They are also hard to identify because the effect of heap data structure corruptions is often delayed.

In this paper, we present the design and implementation of TxInt, a system to detect data structure corruptions. TxInt leverages concurrency control mechanisms implemented in harware transactional memory (HTM) systems to additionally enforce programmer-specified consistency properties on data structures at runtime. TxInt incorporates several optimizations to improve the performance of runtime monitoring, thereby enabling it to be an "always on" tool for continuous runtime monitoring of data structures.

We implemented a prototype version of TxInt by extending an HTM system (LogTM-SE) and studied the feasibility of using TxInt to enforce data structure consistency properties on three benchmarks. Our study shows that TxInt is effective at monitoring data structure properties, and that with suitable optimizations, it can do so with acceptable runtime overhead.

1. Introduction

Modern software systems manage a vast amount of data on the heap, and the correctness and consistency of these data structures is crucial to the robustness of these systems. However, programming errors in such software systems may lead to data structure corruptions that adversely affect their reliability and security. These errors may either lead to data structures that violate well-accepted correctness criteria, *e.g.,* dangling pointer errors and heap metadata corruptions, or application-specific data structure consistency properties [10, 7, 33]. These errors are often hard to debug because their effect may be delayed, *e.g.,* a dangling pointer error does not result in a crash until the pointer in question is dereferenced. Consequently, data structure corruptions due to such programming errors may not manifest during testing.

For extensible software systems, such as operating systems and Web browsers, programming errors are not the only source of data structure corruptions. In such systems, the functionality of a software *kernel* is augmented by third-party extensions, which have unrestricted access to data structures of the kernel. While this feature allows good performance and easy development of extensions, it also leads to security and reliability problems. For example, buggy device drivers may corrupt data structures in the operating system kernel, thereby causing the entire system to crash. Similarly, malicious device drivers (also called *kernel-level rootkits*) can affect system security by stealthily modifying data structures of the operating system kernel [12, 43, 31, 30]. Similar problems also apply to several commodity Web browsers, which allow untrusted third-party plugins to execute in the protection domain of the browser kernel with unrestricted privileges.

In this paper, we present the design and evaluation of TxInt, a prototype system that uses *hardware transactional memory* (HTM) [39, 40] to detect data structure corruptions. HTM systems (*e.g.,* [24, 21, 23, 22]) provide a set of mechanisms in hardware and software to support *memory transactions*, and can potentially ease the development of concurrent programs. To use transactional memory for concurrency control, programmers demarcate critical sections in a program as transactions using instructions provided by the hardware. The HTM system 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 can allow transactions to synchronize access to shared data structures, thereby providing a viable alternative to lock-based synchronization.

TxInt is based upon the insight that the mechanisms implemented by HTM systems to ensure atomicity and isolation can also be used to detect data structure corruptions. In particular, most HTM systems use *read/write sets* to track the set of memory locations accessed/modified by each transaction. These read/write sets are used by the HTM system to determine whether shared data structure accesses made by one transaction are in conflict with another concurrent transaction, *e.g.,* there is a race condition in the way shared data is accessed by two transactions. A transaction is committed only if it does not conflict with other transactions. Otherwise, the HTM system aborts one of the conflicting transactions.

TxInt interposes on the standard workflow of an HTM system to additionally monitor data structure consistency properties. It introspects on the HTM system's read/write sets to identify data structures that were accessed during a transaction and automatically triggers callbacks that check

the consistency of these data structures. These callbacks can include both well-accepted correctness conditions as well as application-specific assertions.

TxInt's approach to detecting data structure corruptions offers several advantages:

(**1**) **Language independence.** TxInt leverages hardware support to check data structure consistency, and can be used with any application compiled for that hardware platform. Therefore, in contrast to techniques that rely on support from garbage collectors [4] or language runtimes [15], TxInt can check data structure consistency even in applications written in low-level languages, such as C/C++. We demonstrate this feature by using TxInt to monitor data structure consistency in ClamAV, a popular antivirus tool [1], and Memcached, a distributed object caching system [2], both of which are written in C.

(**2**) **Applicability to multi-threaded programs.** TxInt can readily benefit multi-threaded applications that use transactional memory for concurrency control, where transactions serve as triggers for data structure consistency checks in addition to demarcating critical sections. The HTM system additionally ensures thread-safety of these data structure checks. TxInt also applies to single-threaded applications. For such programs, the HTM system is used solely to check data structure consistency. The benchmarks in our experimental evaluation contain both kinds of applications.

(**3**) **Extensibility.** TxInt allows programmers to specify and monitor arbitrary properties of data structures. For example, we used TxInt to check complex properties of list and trie data structures in our benchmarks. In contrast, most prior runtime techniques [4, 7, 32] can only check specific kinds of data structure consistency properties and are not as easily extensible as TxInt.

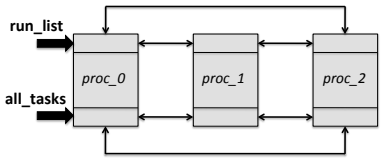
The idea of using transactional memory to check data structure consistency was first proposed by Harris and Peyton-Jones [37], who demonstrated this idea by extending a software transactional memory (STM) system for Haskell. Their work also used the STM system's read/write sets to trigger programmer-specified consistency checks on data structures accessed in a transaction.

The main contribution of our paper is in leveraging HTM machinery to enforce data structure consistency. Migrating to an HTM system is important because applications that use STM for concurrency control often suffer significant performance overheads [16]. These overheads, which arise because STM systems maintain read/write sets in software, can negate the benefits of checking data structure consistency. In contrast, HTM systems incorporate additional hardware to maintain and update read/write sets, and therefore impose lower runtime overheads.

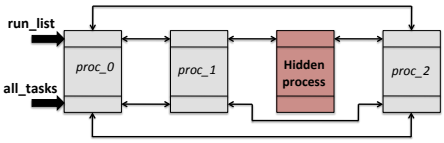
Although our overall approach is similar to that of Harris and Peyton-Jones, we encountered and addressed a number of challenges unique to HTM systems as we built TxInt.

```
(1)  task_struct *run_list;
(2)  task_struct *all_tasks;
(3)  create_new_process (task_struct *proc) {
(4)      init (proc);
(5)      add_to_list (&all_tasks, proc);
(6)  }
(7)  add_to_runlist (task_struct *proc) {
(8)      add_to_list (&run_list, proc);
(9)  }
(10) execute_process () {
(11)     task_struct *curr_proc;
(12)     curr_proc = select_elem (run_list);
(13)     execute (curr_proc);
(14)     del_from_list (&run_list, curr_proc);
(15) }
```

(a) A simple process scheduler.



(b) Normal operation of `all_tasks` and `run_list`.



(c) Example of a hidden process in `run_list`.

Figure 1. Motivating example.

First, HTM systems reason about data structures in terms of low-level memory addresses. TxInt must therefore express consistency checks using memory addresses of data structures and trigger them based upon whether these memory addresses appear in the HTM system's read/write sets. In contrast, Harris and Peyton-Jones' work relies in a key way on Haskell data types to specify and enforce data structure consistency. Second, HTM systems use hardware to implement read/write sets (*e.g.,* as Bloom filters). Each read/write set can store only a limited number of entries and therefore over-approximates the memory locations accessed in a transaction. Because TxInt relies on read/write sets to identify data structures accessed in a transaction, it will likely trigger more runtime checks than necessary, thereby resulting in a performance overhead. This overhead turns out to be significant, necessitating us to incorporate several optimizations to make TxInt practical as an "always on" tool for consistency checking. This problem does not arise in STM systems, because they support read/write sets of unbounded size in software.

We implemented a prototype of TxInt by extending the LogTM-SE HTM system [24]. Our implementation only required minor modifications to the HTM system. We applied it to monitor data structure consistency properties on three

benchmarks, including a microbenchmark that we developed, and two real-world applications—ClamAV and Memcached. Our experimental evaluation shows that TxInt is effective at monitoring complex properties and that it can be optimized to ensure an acceptable runtime overhead.

2. Motivation and System Overview

We use the example in Figure 1(a) to motivate the key requirements that a data structure monitor must satisfy, and then illustrate how TxInt satisfies these requirements. The snippet of code in Figure 1(a) is drawn from the microbenchmark that we used in our experiments, and represents a simplified version of the process management code in Linux.[1] Each process is represented by a C `struct` of type `task_struct`. This code manages processes in two circular, doubly-linked lists called `all_tasks` and `run_list`. The `all_tasks` list is a list of all processes on the system, and is populated when a new process is created in the function `create_new_process`. As processes become ready for execution, the corresponding `task_struct` structures are placed on `run_list` (by the function `add_to_runlist`). The scheduler, which is denoted in Figure 1(a) by `execute_process`, periodically selects processes from `run_list` and schedules them for execution; upon completion, it removes them from `run_list`. Figure 1(b) shows an example in which three processes are created, all of which are scheduled for execution.

In Linux, user-space process accounting utilities, such as `ps`, consult the `all_tasks` list to identify the list of processes on the system. In contrast, the scheduler uses the `run_list`, as shown in `execute_process` in Figure 1(a). During normal execution, a process that is an element of `run_list` will also have a corresponding entry in the `all_tasks` list. Consequently, all running processes will also be displayed in the output of the `ps` command.

However, prior work [30] shows that the discrepancy between the scheduler's view and the user-space view of processes can be exploited to hide malicious user-space processes in Linux (and other commodity operating systems). In particular, commodity operating systems support an extensible architecture in which kernel modules and device drivers have unfettered access to read and write kernel data structures. Petroni *et al.* [30] used this feature to construct a malicious loadable kernel module (also called a *rootkit*) that created a new process and inserted it into `run_list` but not into `all_tasks` (Figure 1(c)). This process is scheduled for execution but is not visible to user-space accounting utilities, and can therefore stealthily perform malicious activities, *e.g.,* logging keystrokes or serving as a backdoor for future attacks on the system.

This example illustrates that data structure properties can be subtly violated to compromise system security. In this example, the property violated was a data structure invariant, `run_list` $\subseteq$ `all_tasks`, *i.e.,* all elements of `run_list` must also be elements of the `all_tasks` list. Data structure corruptions can similarly lead to reliability problems. For instance, a programming error in the `del_from_list` function, which deletes an element from the doubly-linked `run_list`, may fail to modify the `next` and `prev` pointers of elements of the list appropriately, in turn leading to a dangling pointer. This error may result in a crash in a future invocation of `execute_process`, when `select_elem` traverses `run_list` to choose another process for execution. Such security and reliability problems can be detected by monitoring data structure properties.

The above example motivates five design requirements that a data structure monitor must satisfy:

(**1**) **Ability to monitor complex data structures.** Verifying properties of a complex data structure may require traversing the data structure, and possibly other related data structures. For instance, in the example above, ensuring that `run_list` $\subseteq$ `all_tasks` involves traversing both `run_list` and `all_tasks` to ensure that all elements of the former are also elements of the latter.

(**2**) **Extensibility.** In the above example, a programmer may additionally wish to verify the circularity of both `run_list` and `all_tasks`. The monitor must therefore be extensible, and allow the programmer to supply a checker for this property.

(**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**) **Ability to monitor all data accesses.** Modern applications manage a large number of data structures, most of which are critical to their correct operation. Failure to monitor all data structures accessed during the execution of the application may result in security and reliability holes. For example, a rootkit can evade detection by a monitor that fails to check the `run_list` data structure during the execution of `execute_process`.

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

Simultaneously satisfying all five requirements, and in particular the last two, is challenging. As we discuss in Section 4, the basic design of TxInt satisfies the first three requirements. However, the limitations imposed by existing HTM systems required us to employ several optimizations in TxInt, which result in a tradeoff between performance and the ability to monitor all data accesses. Nevertheless, TxInt is tunable, and programmers can choose to execute some portions of the program without incurring the overhead of

[1] Although we use this example from Linux to motivate our work, we have only applied the TxInt prototype to user-space applications to date.

```
(1)   task_struct *run_list;
(2)   task_struct *all_tasks;
(3)   register_ds (run_list, check_contain);
(4)   register_ds (all_tasks, check_contain);
(5)   create_new_process (task_struct *proc) {
(6)       transaction (txint_entry) {
(7)           init (proc);
(8)           add_to_list (&all_tasks, proc);
(9)       }
(10)  }
(11)  add_to_runlist (task_struct *proc) {
(12)      transaction (txint_entry) {
(13)          add_to_list (&run_list, proc);
(14)      }
(15)  }
(16)  execute_process () {
(17)      task_struct *curr_proc;
(18)      transaction (txint_entry) {
(19)          curr_proc = select_elem (run_list);
(20)          execute (curr_proc);
(21)          del_from_list (&run_list, curr_proc);
(22)      }
(23)  }
```

The TxInt monitor (implemented in software)

```
(m1)  bool txint_entry() {
(m2)      rset = Get read set from hardware registers;
(m3)      wset = Get write set from hardware registers;
(m4)      for (each ds ∈ registered data structures) {
(m5)          if (map(ds)∩(rset∪wset) ≠ ∅) {
(m6)              invoke callback associated with ds;
(m7)      }
(m8)  }
(m9)  void register_ds(void *dsptr, void *fptr) {
(m10)     //register fptr as checking callback for dsptr
(m11) }
(m12) bool check_contain () {
(m13)     for (each p ∈ run_list)
(m14)         if (p ∉ all_tasks) return false;
(m15)     update address maps of run_list and all_tasks;
(m16)     return true;
(m17) }
```

Figure 2. Process scheduler from Figure 1(a) modified to use TxInt to monitor `run_list` and `all_tasks`. Code in bold-faced font must be added by a programmer to use TxInt.

runtime checks, while fully monitoring all data accesses in other portions of the program. We defer a detailed discussion of performance to Section 4.

To motivate how TxInt monitors data structure properties, we consider an approach in which a programmer *inlines* checks at key locations in the program. For example, to detect the attack in Figure 1(c), the program in Figure 1(a) can include a check to ensure that the property `run_list` ⊆ `all_tasks` holds prior to the execution of the function `execute_process`. Although apparently simple, an approach that inlines checks must overcome two challenges. First, data structure checks must be placed at all locations where sensitive data structures (*e.g.,* `run_list`) are accessed (to satisfy the fourth requirement). Identifying all such locations can be challenging, especially in the presence of pointer aliasing. Second, in multi-threaded software, the placement of checks must avoid time-of-check to time-of-use errors (race conditions), in which a concurrently-executing thread may modify a data structure in the interval between property verification and use of the data structure.

The TxInt 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, TxInt instead requires code that manipulates sensitive data structures to be embedded in *transactions*, as shown in Figure 2. In this figure, all operations on the `all_tasks` and `run_list` linked lists happen within transactions. The programmer uses the `register_ds` calls in Figure 2 to notify TxInt that the `all_tasks` and `run_list` data structures must be monitored. Note that the programmer only registers pointers to the head of these data structures, and does so during program startup. The programmer also specifies the properties to be verified in *checker callbacks* associated with each data structure. In this example, the same checker callback (`check_contain`) is associated with both `all_tasks` and `run_list`. This function checks `run_list` ⊆ `all_tasks`.

Upon completion of a transaction, our modifications to the HTM system cause it to transfer control to the entrypoint of TxInt's data structure monitor. As shown in Figure 2, the entrypoint is a function pointer that is registered with the HTM system using an argument to the `transaction`{...} keyword. The HTM also passes the read/write sets to the monitor via hardware registers. Internally, the TxInt monitor maintains an *address map* for each data structure. The address map stores the set of all memory addresses associated with that data structure. For example, the address map of `run_list` would contain the memory addresses associated with each of its constituent elements. TxInt'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 the programmer need not specify which checker callbacks must be invoked at the end of a transaction. Rather, TxInt uses the HTM system's read-/write sets to infer which callbacks must be invoked.

As discussed, TxInt requires code that manipulates sensitive data structures (`run_list` and `all_tasks`) to be placed within transactions. Such transactions may naturally be placed in multi-threaded code that uses transactional memory to synchronize access to shared data structures. In such

cases, the use of TxInt additionally verifies data structure properties before transactions are committed. However, TxInt also applies to single-threaded programs. In such cases, transactions are placed in the code to simply identify program points where data structures are triggered.

3. Hardware Transactional Memory

In this section, we provide background on transactional memory, focusing on HTM systems, and the features relevant to the design of TxInt. 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 execution transactions, but vary widely in how they do so [40]. 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. Early HTM systems (*e.g.,* [39]) implemented conflict detection by piggybacking on cache coherence protocols. However, such HTM systems placed serious limitations on the length of transactions. Modern HTM systems overcome this limitation, and support transactions of unbounded length by incorporating new hardware in the form of per-transaction *read and write sets* to record the memory locations accessed/modified by a transaction. Read-/write sets can be implemented in hardware as Bloom filters (*e.g.,* [22, 24]). An HTM system can detect conflicts by intersecting the read/write sets of a transaction with those of other in-flight transactions. Bloom filters also allow for simple and efficient hardware-based mechanisms to intersect and compare read/write sets. If a conflict is detected, the HTM must abort at least one conflicting transaction; it consults a contention manager to decide which transactions to abort, and roll back the changes made by these transactions.

While read/write sets record the memory locations read/-modified by a transaction, version management mechanisms allow the HTM system to record the set of data modifications made by the 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. HTM systems vary widely in how they implement version management. For example, they can buffer all the updates made by a transaction and commit these changes at the end of a transaction (*e.g.,* [23]). An alternative approach, which is adopted by LogTM-SE, is to log the old values at memory locations modified by a transaction (in a per-transaction *undo log*), and use the undo log to restore memory if the transaction aborts.

The basic design of TxInt, which is described in the following section, is applicable to any HTM system. However, in this paper, we assume an HTM system in which read/write sets are easily accessible from software. For our prototype implementation, we chose the LogTM-SE HTM. This HTM implements read/write sets as Bloom filters (which can be exposed to software with minor HTM modifications), uses an undo log for version management, and allows transactions of unbounded length.

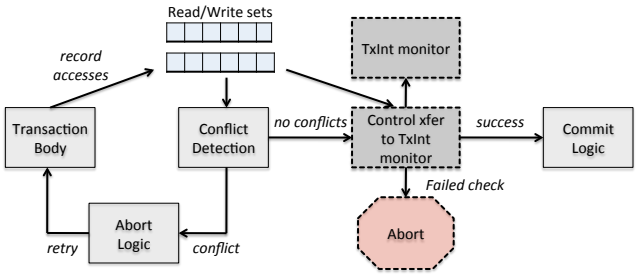


Figure 3. Workflow of a TxInt-enhanced HTM system.

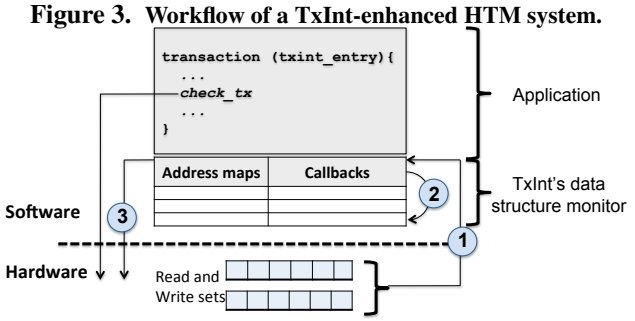


Figure 4. Flow of control in TxInt. The execution of a `check_tx` or `end_tx` instruction triggers the HTM mechanism. (1) the HTM system copies read and write sets into registers and transfers control to TxInt's entrypoint; (2) the monitor executes checker callbacks for data structures that were accessed in the transaction; and (3) upon successful completion, TxInt returns control to the hardware, which either commits the transaction or resumes the application.

4. The Design and Implementation of TxInt

TxInt enforces data structure properties by interposing on the standard workflow of an HTM system, as shown in Figure 3. As this figure shows, an HTM system detects conflicts using read/write sets. Transactions that have completed and are not in conflict with other transactions can be committed. Our modifications to the HTM system interpose on the commit logic to additionally invoke TxInt's *data structure monitor*. This monitor, which is implemented in software, verifies properties of the data structures that were accessed in the transaction. The transaction is committed only if the monitor returns successfully. While TxInt's data structure monitor is invoked at the end of transactions, it can optionally be triggered at any point during the execution of a transaction, using the `check_tx` instruction, discussed next.

4.1 HTM Changes to Support TxInt

To support the above changes to the workflow of an HTM system, we made three key changes to the implementation of its programming interface:

(**1**) **Registering an entrypoint using `begin_tx`.** We modified `begin_tx`, the instruction that signifies that start of a transaction, to accept an address <addr> as an argument, *i.e.,* the programming interface changes to `begin_tx` <addr>. This address must point to a valid region of code in the application's address space, and is meant to denote the entrypoint into TxInt's data structure monitor. In Figure 2, the address of `txint_entry` is passed as the argument to `begin_tx` (the language-level `transaction`{...} construct is compiled into a pair of `begin_tx`/`end_tx` instructions).

(**2**) **New instruction: `check_tx`.** We added a new instruction that allows programmers to trigger TxInt within a transaction. Execution of this instruction triggers three steps (see Figure 4). First, it copies the transaction's read and write sets into hardware registers. These registers can be accessed by TxInt's data structure monitor. Second, it transfers control to <addr>, the entrypoint registered using the most recently encountered `begin_tx` instruction. This step invokes TxInt's data structure monitor. Third, once the monitor returns successfully (*i.e.,* returns `true`), hardware resumes execution in the instruction following the `check_tx` instruction. If the monitor detects a data structure corruption, it triggers the HTM's transaction abort logic.

(**3**) **Modifying the functionality of `end_tx`.** In an HTM system, the `end_tx` instruction triggers the conflict detection mechanism. If no conflicts are found, it triggers the transaction commit logic. We modified the implementation of `end_tx` to trigger TxInt's monitor (in a manner akin to `check_tx`) once a transaction successfully passes conflict detection. The main difference is that once the monitor returns successfully, `end_tx` automatically triggers the transaction commit logic.

4.2 Implementation in LogTM-SE

We implemented the above changes by modifying the LogTM-SE HTM system. This system is built for the SPARC architecture 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. LogTM-SE also supports strong atomicity [40], *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, it implements read/write sets using Bloom filters (these Bloom filters are called a transaction's read/write *signatures*). Bloom filters are fixed-size structures, and can easily be made accessible to software.

In our implementation, `check_tx` and `end_tx` copy the contents of read/write Bloom filters into hardware registers. However, it is also well-known that Bloom filters can only approximate a set of elements (in this case, memory addresses). In particular, a membership check for a memory address can falsely determine that the address is a member of the read (or write) set represented by the Bloom filter. This problem is particularly pronounced when a Bloom filter is used to represent a large set, *e.g.,* if a 1024-bit Bloom filter is used to represent a set of addresses drawn from a 32-bit address-space, as in our implementation. Section 4.4 describes the implications of this problem and the optimizations that TxInt uses to address it.

The LogTM-SE HTM system provides a mechanism called *escape actions* that allows code in a transaction to execute outside the purview of the HTM system, *i.e.,* memory accesses within an escape action are not recorded in that transaction's read/write sets. In our implementation, the code of the TxInt data structure monitor executes in an escape action. This feature ensures that any accesses to the data structure within the monitor itself (*e.g.,* in a checker callback) are not recorded in the read/write set. We synchronize access to the monitor's own data structures by acquiring a global lock before executing the code of the monitor and releasing the lock upon completion.

Overall, we added about 50 lines of C++ and SPARC code to the LogTM-SE simulator to implement the changes to its programming interface, showing that TxInt can be implemented with minimal modifications to an existing HTM system.

4.3 TxInt's Data Structure Monitor

TxInt's data structure monitor is implemented in software, and is loaded into the application's address space. Its main responsibility 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 property `run_list` $\subseteq$ `all_tasks`), the address map of `all_tasks`

(and run_list) should contain the memory locations of all next and prev fields of each task_struct node in the list. This is because any attempt to violate the property run_list ⊆ all_tasks must modify these fields.

In our implementation, address maps are represented as Bloom filters that are of the same size as the read/write Bloom filters (1024 bits). We also use the same function as implemented by the HTM system to hash a memory address to a location in the Bloom filter. This design allows the address map of a data structure to be efficiently compared against a read (or write) Bloom filter for intersection.

When the application invokes the TxInt monitor (via a check_tx or end_tx instruction), the monitor traverses the address map table to determine data structures that were accessed by the application, and triggers the callbacks associated with those data structures. Note that the monitor must intersect read/write Bloom filters with *all* the address maps registered with it. This can be done by sequentially traversing the address map table, requiring an O(n) operation each time the monitor is invoked, where n is the size of the address map table. However, this operation can be inefficient if the transaction only accesses a small number of data structures, *i.e.,* only a few address maps intersect the read/write Bloom filters. To optimize for this case, we implemented a tree-structured index for the address map table. The leaves of this tree store the address maps in the table. Each node in the tree stores a Bloom filter that is the logical-or of its children (equivalent to a set union), resulting in a tree of height $\lg n$. Locating the k address maps that intersect non-trivially with a read/write Bloom filter only requires $k \lg n$ operations. The tree-based index is therefore more efficient for small values of k.

• ***Computing and Maintaining Address Maps.*** Because the monitor detects accesses to a data structure by comparing read/write sets 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 run_list must appropriately modify its address map in the TxInt 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 element proc when it is created in create_new_process. However, this approach is impractical for large code bases, because it requires the programmer to manually insert code 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 TxInt monitor, to indicate which data structures must be monitored. For example, in Figure 1, the programmer only registers pointers to the heads of all_tasks and run_list. 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 property verification. We can therefore piggyback address map creation with data structure 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 all_tasks and run_list is supplied in line $m15$. As this callback executes, TxInt can update its address map for the data structures monitored by the callback.

We illustrate this idea using the data structure in Figure 1(b). The address map for run_list contains the next and prev pointers of each of the three nodes in the linked list. Suppose that a call to add_to_runlist in Figure 1 adds a fourth node to run_list. This operation adds the addresses of the next and prev pointers that were modified to the read/write Bloom filters of the transaction. In turn, this triggers the TxInt monitor, which invokes the check_contain callback. When this callback executes, it traverses all the nodes in run_list to verify the property run_list ⊆ all_tasks. In doing so, it computes a new address map for run_list containing the addresses of the next and prev fields of the newly-added node. The code of the monitor replaces the address map for run_list with the newly computed one, thereby ensuring that address maps accurately reflect modifications to run_list.

• ***Classes of Properties Checked.*** The architecture of TxInt allows arbitrary functions to be registered as callbacks for a data structure. However, our experience suggests that two kinds of properties are particularly useful to express data structure properties, namely *global invariants* and *context-sensitive invariants*, as discussed below.

(**1**) **Global invariants** specify data structure properties that must hold for the duration of program execution. The property run_list ⊆ all_tasks is an example of a global invariant. Other examples include checking the circularity of a linked list and checking ranges of scalar values. Our prototype supports an API to ease the specification of checker callbacks for a variety of global invariants. We have implemented a library that allows programmers to easily create checker callbacks for invariants inferred by Daikon [25].

Global invariants may temporarily be violated when a data structure is being modified. For example, the circularity of run_list may temporarily be violated when a node is added to the list, *e.g.,* in add_to_list. Therefore, these invariants must be checked only when the data structure is well-formed. TxInt allows programmers to specify when checks must be triggered using the check_tx and end_tx instructions, thereby avoiding cases where checks may fail because a data structure is not well-formed.

TxInt supports a particularly important class of global invariants, namely, those that check the constancy of data values stored in a region of memory. Such invariants can be used to implement fine-grained software fault isolation (SFI) [35].

In turn, SFI has a number of applications. For example, it can be used to detect heap metadata corruptions by registering the region of memory that stores heap metadata with the TxInt monitor. SFI can also be used to *protect the state of the TxInt monitor itself* from data corruptions, *e.g.,* the address map table. This feature ensures that TxInt can detect modifications to its own data structures in spite of executing in the same address-space as the application that it monitors.

(**2**) **Context-sensitive invariants** specify data structure properties that hold at specific points during program execution. For example, consider the property `run_list` ≠ NULL. This property must hold when `add_to_runlist` (in Figure 1) terminates, but need not hold when `execute_process` terminates. The TxInt monitor supports context-sensitive invariants by allowing the programmer to modify the checker callback associated with a data structure at different points in the program. For example, the programmer could register the property `run_list` ≠ NULL before invoking `add_to_runlist`, and unregister it after the function returns.

4.4 Design Enhancements to Improve Performance

The basic design of TxInt described so far satisfies the first four criteria outlined in Section 2. The use of an HTM system also ensures that read/write sets can be recorded efficiently. However, as we explain below, the use of Bloom filters to represent read/write sets and address maps often results in a system that imposes unacceptably high performance overheads. For example, TxInt imposes overheads in excess of 8× when used with Memcached. In this section, we explain why the use of Bloom filters is a performance bottleneck, and describe two optimizations that we developed to make TxInt practical for "always on" use.

Recall that a Bloom filter is an approximate representation of a set of elements. In the TxInt prototype, the Bloom filter contains 1024 entries that store a set of memory addresses, which could have up to 2^{32} elements. When the set of memory addresses is large, the corresponding Bloom filter "saturates," *i.e.,* most of its bits will be set. In our setting, Bloom filter saturation leads to two problems:

(**1**) **Read/write set saturation.** Bloom filters representing read/write sets may saturate for large transactions, *i.e.,* transactions that access a large number of data structures. In software that uses HTMs for concurrency control, such saturation can result in *false conflicts*, *i.e.,* situations where two concurrent transactions access different data elements, but appear to conflict because the intersection of their read/write Bloom filters is non-empty. This kind of saturation is germane to the design of the HTM itself, and has been recognized as a problem by the HTM community [17]. False conflicts can abort transactions that could have otherwise completed successfully. Because aborted transactions must be re-executed, this situation results in runtime performance overhead.

Read/write set saturation is particularly problematic for TxInt. Even if a transaction successfully passes conflict detection, saturated Bloom filters give the illusion that a large number of data structures were accessed in the transaction. In turn, TxInt will trigger the callbacks associated with each of these data structures, even if they were not actually accessed in the transaction, thereby resulting in further overheads.

(**2**) **Address map saturation.** Large data structures will typically have saturated address maps. This situation is particularly problematic for linked lists, arrays and other complex data structures that contain several elements. A saturated address map will intersect non-trivially with most read-/write Bloom filters, even if the read/write Bloom filters are not saturated. In turn, this will trigger the callbacks associated with that address map, even if the corresponding data structure was not accessed within the transaction.

One way to avoid executing a large number of falsely-triggered data structure checks is to avail of the HTM system's undo logs to determine whether the data structure was accessed in a transaction. The undo log of a transaction stores the list of memory addresses that were modified by the transaction, which TxInt can use to verify that a memory address was indeed accessed. While this approach ensures that data structures will be checked only if they are accessed, walking the log is an expensive operation, and can result in performance overheads as well. We therefore incorporated two optimizations in the design of TxInt to address poor performance that results from Bloom filter saturation.

• ***Optimization 1: Creating address maps using sampling.*** As discussed above, address maps of large data structures can be saturated. This optimization targets address map saturation and creates address maps by sampling the data structure's memory locations. That is, each memory location that was previously stored in the address map of the data structure is now included in the address map with a probability p_1. As a consequence, only a fraction p_1 of the data structure's memory locations appear in its address map. We periodically resample the data structure to produce new address maps. In our current implementation, we resample when we update the address maps to reflect changes to the data structure.

This optimization ensures that address maps will be less saturated, thereby triggering fewer checks of the data structure. While this optimization reduces the runtime performance overhead, it comes at a cost—a data structure check may not be triggered even if the data structure was accessed (a *false negative*), thereby potentially failing to detect data structure corruptions.

• ***Optimization 2: Probabilistically triggering data structure checks.*** This optimization targets read/write Bloom filter saturation. Saturated read/write Bloom filters intersect non-trivially with a large fraction of address maps in the

TxInt monitor (even if the address maps are not saturated), thereby triggering a large number of data structure checks.

We handle this problem in TxInt by setting a threshold to detect whether a read/write Bloom filter is saturated; a Bloom filter is said to be saturated if the number of bits set in the Bloom filter exceeds that threshold. When a saturated read/write Bloom filter intersects non-trivially with an address map, we trigger the associated callback with probability p_2. Consequently, only a fraction p_2 of address maps that intersect with the read/write Bloom filter will trigger data structure checks. As with optimization 1, this optimization can also potentially result in false negatives.

Both these optimizations offer a practical way to trade performance against the ability to monitor all data accesses. The total number of data structure checks triggered can be reduced by choosing suitable values for probabilities p_1 and p_2, thereby reducing the runtime overhead of TxInt. These optimizations are independent, and may possibly be used in conjunction with each other to achieve good performance.

5. Evaluation

We evaluated TxInt using the default configuration of Simics, which simulates an UltraSPARC-III-plus processor running at 75MHz, with 256MB RAM, 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 TxInt used 1024-bit Bloom filters. On this platform, the `check_tx` instruction takes 300 nanoseconds to copy Bloom filters into hardware registers, and transfer control from the HTM system to the TxInt monitor; this time is comparable to the cost of making a function call on this architecture. We evaluated the effectiveness and performance of TxInt using two macrobenchmarks (ClamAV and Memcached) and one microbenchmark.

5.1 ClamAV

We evaluated TxInt's ability to monitor complex data structure properties by using it with 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. As it operates, Clamscan internally constructs and maintains several data structures to represent the virus definition database. Because ClamAV is written in C, it may potentially contain vulnerabilities that can be exploited by malware to hijack its execution and evade detection (*e.g.,* [3]). It is therefore critical to protect the integrity of Clamscan's data structures, such as those that represent the virus definition database, from malware.

For the experiments reported below, we used Clamscan version-0.95.2 with its default virus definition database. We ran Clamscan on a TxInt-enhanced LogTM-SE system, 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. Clamscan is a single-threaded benchmark; transactions placed in its code serve the sole purpose of triggering TxInt's data structure monitor. This example illustrates how TxInt can benefit single-threaded applications as well.

Version	Time in seconds
Unmodified	10.95
Ported to LogTM-SE (no TxInt)	10.99 (0%)
With TxInt enabled	11.35 (3.7%)

Figure 5. Performance of Clamscan with TxInt.

We studied the source code of Clamscan to identify data structures that were critical to its correct and secure operation. We identified one data structure, called `engine` (of type `struct cl_engine`), that was particularly critical. 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. We wrote checkers for a total of 22 properties for this data structure. In each case, we also injected synthetic faults into Clamscan that violated these properties, and verified that TxInt was able to detect the violation. Below, we explain one of these invariants, namely, the Aho-Corasick trie property.

To ensure that files can be efficiently matched against virus definitions, Clamscan internally stores these definitions as a trie, as specified by the Aho-Corasick algorithm [5]. The Aho-Corasick algorithm also requires each node in this trie to also contain a link to another node in the trie, denoting its longest proper suffix; this link is called a *failure link*. In all, Clamscan stores pointers to nine such tries, for various kinds of file formats, in an array called `engine->root`. An attack that corrupts the integrity of any of these tries (*e.g.,* by modifying failure links) can easily misguide the detection algorithm.

To protect these tries, we registered the memory addresses of nodes in these tries with the TxInt monitor. We verified two properties: (a) failure links point to existing nodes in the trie and that they are not modified after Clamscan is initialized; and (b) the trie is free of cycles. While these properties are not complete (*i.e.,* they do not completely specify the trie data structure), they increase the difficulty of corrupting the data structure and misleading the scanner.

Figure 5 reports the overall performance of Clamscan as it scanned a directory containing 356 files. All numbers are averaged over 10 runs of Clamscan. The first row shows the time taken by an unmodified version of Clamscan executing in our simulation environment. The second row reports the performance of a version of Clamscan that contains transactions to enforce the properties discussed above. However, we did not enable TxInt; this row therefore reports the performance of porting Clamscan to LogTM-SE. The third row

Version	Operations/second
Unmodified	22,200
Ported to LogTM-SE (no TxInt)	21,962 (1.01×)
With TxInt enabled	2,764 (8.03×)

Figure 6. Performance of Memcached with TxInt.

shows the performance of Clamscan with TxInt enabled to check the properties discussed above. We configured TxInt to use the tree-structured index (described in Section 4.3) to locate address maps that intersect read/write sets. In all cases, we used the Solaris `gethrvtime` function to measure the time taken by Clamscan.

As this figure shows, TxInt imposes an overhead of 3.7% on Clamscan, which shows that it can be adopted as an "always on" tool to monitor the integrity of Clamscan's data structures. Because the runtime performance overhead was manageable, we did not enable the optimizations discussed in Section 4.4.

5.2 Memcached

Memcached is a distributed object caching system that has been adapted by Web services such as Livejournal, Slashdot and Wikipedia. Unlike Clamscan, Memcached is multithreaded, and currently uses locks for synchronization. It stores key/value pairs and supports operations such as inserting new key/value pairs and querying the value associated with a key. Internally, it maintains 255 *slabs* to store key/value pairs corresponding to values of different sizes. Each of these slabs is implemented as a doubly-linked list, and serves as an LRU cache for items stored in the slab.

We converted Memcached to use transactional memory for synchronization by converting each client request to execute as a single transaction, and registered these linked lists with the TxInt 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 time that they were inserted.

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. Figure 7 reports the performance of (i) the unmodified version of Memcached, (ii) a version that uses transactions for synchronization alone; and (iii) version that uses TxInt to monitor data structures.

As Figure 7 shows, TxInt imposes a significant (8.03×) overhead to enforce data structure properties. We found that this was because transactions in Memcached saturated the read/write Bloom filters of LogTM-SE, which in turn triggered checks for several linked lists that were not accessed in transactions. We therefore enabled optimization 2 (Section 4.4) to probabilistically trigger data structure property checks if the number of bits set in a read/write Bloom filter exceeded 90%. Figure 7 reports the impact of this optimization for various values of p_2. As this figure shows, reducing the probability of checks improves the performance of Memcached. However, as described in Section 4.4, this performance improvement comes at the cost of failing to check data structures that were accessed in transactions. We quantify the impact of such *false negatives* using a microbenchmark, which we describe next.

Prob. (p_2)	Operations/second
100%	2,764 (8.03×)
75%	3,598 (6.17×)
50%	4,700 (4.70×)
25%	7,017 (3.16×)
10%	14,207 (1.56×)
1%	19,703 (1.12×)

Figure 7. Impact of optimization 2 on memcached performance.

5.3 Process Scheduling Microbenchmark

In addition to evaluating the effectiveness of TxInt with Clamscan and Memcached, we designed a microbenchmark to dissect how TxInt's performance can be affected by the optimizations discussed in Section 4.4. This microbenchmark is based upon the process scheduling example in Section 2. The microbenchmark begins execution by initializing 300 doubly-linked lists, named $\texttt{run\_list}_1$, ..., $\texttt{run\_list}_{300}$. It also initializes one `all_tasks` list. Each linked list is initialized to contain five nodes; each node stores an integer, representing the process identifier, and pointers to the previous and next nodes in the list.

We registered all these linked lists with the TxInt monitor; the address map of each linked list stores the memory addresses of all fields of each of the five nodes in the list. For each list $\texttt{run\_list}_i$, we registered a checking callback to ensure that $\texttt{run\_list}_i \subseteq \texttt{all\_tasks}$, *i.e.,* that if a node with a process identifier p exists in $\texttt{run\_list}_i$, then a corresponding node with the same process identifier exists in `all_tasks`. The main body of the microbenchmark is a CPU-bound loop that accesses nodes in $\texttt{run\_list}_1$, ..., $\texttt{run\_list}_m$ ($1 \leq m \leq 300$) within a single transaction. Thus, we can vary the length of the transaction by modifying the value of m. Because our main goal was to evaluate performance, we ensured that all the invariants $\texttt{run\_list}_i \subseteq \texttt{all\_tasks}$ were satisfied for the duration of the experiment, *i.e.,* all $\texttt{run\_list}_i$'s and `all_tasks` had nodes with the same set of process identifiers.

Figure 8 plots the impact of optimization 1 on the performance of the microbenchmark. The x-axis shows the value of p_1, the sampling rate, while the y-axis shows the ratio of the time taken by a TxInt-enhanced version to the native version of the microbenchmark. We ran the microbenchmark with a number of different values for m, which is the number

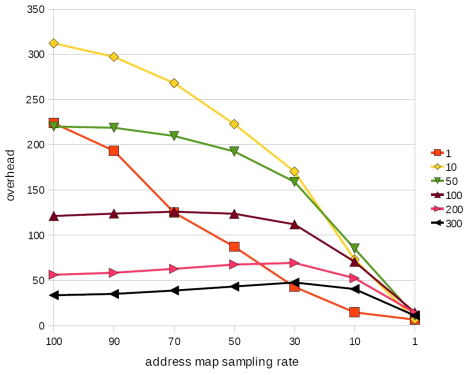


Figure 8. Performance impact of Optimization 1.

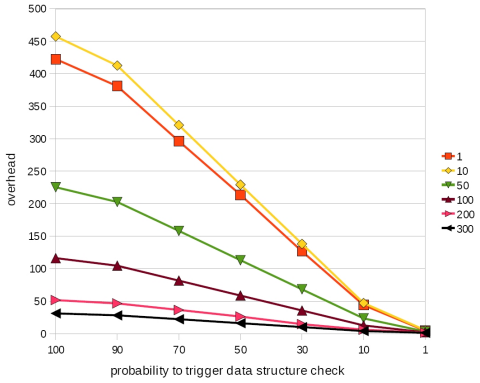


Figure 9. Performance impact of Optimization 2.

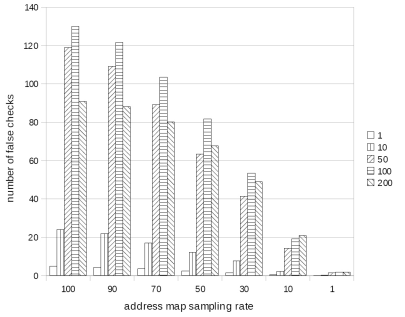


Figure 10. Number of false checks.

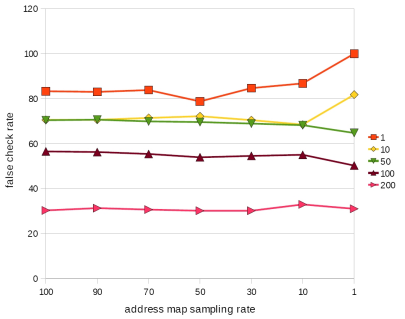


Figure 11. False check rate.

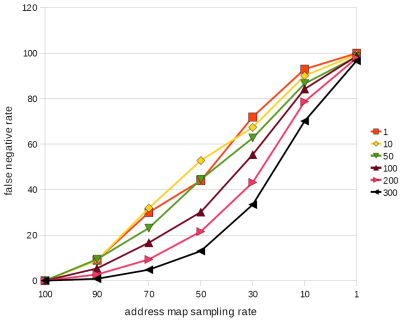


Figure 12. False negative rate.

of `run_lists` accessed within the transaction, in the main loop of the microbenchmark. Figure 8 shows that as the sampling rate p_1 is reduced, the performance overhead imposed by TxInt also reduces.

Note that the overheads observed for larger values of m (*i.e.,* larger transactions) is *smaller* than the overheads observed for smaller values of m, *e.g.,* an overhead of 40× for m=300 versus 310× for m=10. This is because the microbenchmark is CPU-bound, and only performs a few simple memory operations within the transaction. When TxInt is triggered at the end of each transaction, it compares the read/write Bloom filters against the address maps of all registered data structures. This traversal incurs a cost, which is amortized as the value of m increases, thereby resulting in lower overheads.

To evaluate the effect of optimization 2, we fixed the value of p_1 to 100%, and ran the microbenchmark with different values for p_2, which is the probability with which a data structure check is triggered when the read/write Bloom filter intersects its address map. Figure 9 presents the results of this experiment. The x-axis of this figure shows the value of p_2 (as a percentage), while the y-axis shows overhead, which reduces linearly with p_2.

Figure 10 explains why these optimizations improve performance. This figure shows how the number of "false" data structure checks triggered (shown on the y-axis) varies with p_1, which is plotted on the x-axis.[2] A false data structure check is defined as one that is triggered even if the data structure was not accessed in the transaction. By definition, the number of false data structure checks for $m = 300$ is zero. As Figure 10 indicates, the number of false checks reduces as p_1 is reduced. In turn, fewer data structure checks are triggered, thereby improving performance.

Although the *number* of false checks triggered reduces with p_1, we found that the false check *rate*, *i.e.,* the ratio of false checks to the total number of checks triggered, remains relatively constant as p_1 is reduced (see Figure 11). However, performance overhead is determined by the number of checks, not the false check rate. Consequently, reducing p_1 improves the overall performance of the microbenchmark.

While reducing p_1 (and p_2) improves performance, it does so at the cost of potentially failing to trigger checks on data structures that were accessed in a transaction. We call each such check that *should* have been triggered (but was not) a false negative. The false negative rate is the ratio of false negatives to the total number of triggered checks. Figure 12 plots the false negative rate as a function of p_1 for different values of m. This figure shows that reducing the value of p_1 increases the false negative rate.

[2] We observed similar trends for optimization 2 by varying p_2, but omit those charts because of space constraints.

6. Related Work

We focus our discussion to four categories of related work on runtime techniques for data monitoring.

• ***Applications of Transactions and Transactional Memory.*** The use of transactions to monitor data integrity was first suggested by the relational database community [44, 36]. 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 [41, 29].

With advances in transactional memory, researchers have begun to explore similar applications using hardware and software support for transactional memory [37, 14, 26, 19, 18]. Harris and Peyton Jones [37] and Birgisson *et al.* [14] explored the use of STM systems to monitor data structure accesses. Harris and Peyton Jones used an STM system for Haskell to monitor programmer-specified data invariants, while Birgisson *et al.* used an STM for Java to enforce authorization policies on data accesses. Both these systems rely on STM extensions for specific languages and are inapplicable to the general case of monitoring applications written in low-level languages. Although compiler support for STMs may make these techniques applicable to low-level languages (*e.g.,* [11]), 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 [16]. TxInt addresses this problem by migrating to HTM systems, which mitigate the overhead of maintaining and updating read/write sets. However, as discussed in the body of this paper, leveraging HTM hardware to monitor data structures involves overcoming several challenges unique to HTM systems.

Researchers have also made the case for deconstructing HTM systems, and reusing HTM hardware for applications beyond concurrency control [26, 19]. In particular, the position paper Hill *et al.* [26] 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 TxInt, our work explores the challenges of building a data structure integrity monitor using the basic watchpoint framework outlined in that paper.

• ***Runtime Instrumentation and Monitoring Environments.*** A number of debugging aids use specialized runtime environments or program instrumentation to localize the source of programming errors that lead to corrupted data structures [34, 7, 42, 28, 38, 8]. The most popular of these tools is Valgrind [42], which provides a flexible framework for heavyweight dynamic analysis. Valgrind is ideally suited to detect common errors such as memory leaks and dangling pointers. However, it uses dynamic binary recompilation and therefore imposes significant performance overheads, making it impractical for use as an "always-on" tool. It is also not easy to extend Valgrind to monitor programmer-specified data structure properties, of the kind discussed in this paper. Other tools such as HeapMD [7], Purify [38] and SWAT [8] instrument the program being analyzed to monitor the heap for specialized classes of data structure corruptions, such as memory leaks and dangling pointers. Although effective at achieving their goal, these tools cannot readily be extended to monitor arbitrary data structure properties. Moreover, with the exception of SWAT, the instrumentation introduced by these tools generally imposes a significant runtime overhead. SWAT, in particular, is noteworthy in its use of a sampling framework [6] to reduce the overhead introduced by instrumentation. As discussed in this paper, TxInt also employs sampling to reduce runtime overheads that result from the limitations of HTM systems.

Recent work has proposed the use of the garbage collector to monitor data structure properties [4]. This framework, called GC-Assertions, piggybacks on the garbage collector to traverse the heap and enforce programmer-specified properties on heap data structures with low runtime overhead. Unlike TxInt, however, the GC-Assertions framework does not allow a programmer to specify *when* data structure checks must be triggered; rather they are triggered automatically during garbage collection. As a result, GC-Assertions may be unsuitable to monitor data structure invariants, which may temporarily be violated as a data structure is modified. GC-Assertions is also not applicable to low-level languages, such as C/C++, which use manual memory management.

• ***Hardware-assisted Monitoring.*** A number of recent systems have proposed hardware extensions to detect data structure corruptions, particularly memory errors that compromise security (*e.g.,* [33, 32, 9, 27, 20]). Among these, the iWatcher system [33] is particularly noteworthy, because it allows a programmer to specify watchpoints on data structures and efficiently monitor properties on these data structures. Like TxInt, iWatcher also leverages hardware designed for an entirely different purpose (thread-level speculation). TxInt and iWatcher may possibly be used in conjunction on a platform that supports both thread-level speculation and transactional memory.

• ***Rootkit Detection Tools.*** Rootkits have recently evolved to achieve malicious goals by affecting the integrity of kernel data structures; Figure 1(c) presented an example of one such rootkit. To counter such rootkits, researchers have proposed detection tools that periodically (and asynchronously) scan the contents of kernel memory, and verify that kernel data structures satisfy consistency properties or invariants [43, 13, 31, 30]. Like GC-Assertions, these tools do not allow a programmer to specify when data structures must be checked. They may thus report false positives when a data structure is in a temporary state of flux as it is being modified. Although we have only applied TxInt to user-space applications thus far, we plan to investigate whether the use of

TxInt to kernel-mode code can detect rootkits without the above shortcoming.

References

[1] Clam antivirus. `http://www.clamav.net`.

[2] Memcached: A distributed memory object caching system. `http://www.danga.com/memcached`.

[3] ClamAV multiple vulnerabilities, December 2007. Secunia Advisories – SA28117.

[4] E. Aftandilian and S. Z. Guyer. GC Assertions: Using the garbage collector to check heap properties. In *PLDI*, 2009.

[5] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. In *Comm. ACM*, 1975.

[6] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *PLDI*, 2001.

[7] T. Chilimbi and V. Ganapathy. HeapMD: Identifying heap-based bugs using anomaly detection. In *ASPLOS*, 2006.

[8] T. Chilimbi and M. Hauswirth. Low-overhead memory leak detection using adaptive statistical profiling. In *ASPLOS*, 2004.

[9] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *ISCA*, 2004.

[10] B. Demsky and M. Rinard. Automatic detection and repair of errors in data structures. In *OOPSLA*, 2003.

[11] A. Adi-Tabatabai *et al.*. Compiler and runtime support for efficient software transactional memory. In *PLDI*, 2006.

[12] A. Baliga *et al.*. Lurking in the shadows: Identifying systemic threats to kernel data. In *IEEE Oakland*, 2007.

[13] A. Baliga *et al.*. Automatic inference and enforcement of kernel data structure invariants. In *ACSAC*, 2008.

[14] A. Birgisson *et al.*. Enforcing authorization policies using transactional memory introspection. In *CCS*, 2008.

[15] A. Yip *et al.*. Improving application security with data flow assertions. In *SOSP*, 2009.

[16] C. Cascaval *et al.*. Software transactional memory: Why is it only a research toy? *Comm. of the ACM*, 2008.

[17] J. Bobba *et al.*. TokenTM: Efficient execution of large transactions with hardware transactional memory. In *ISCA*, 2008.

[18] J. Bobba *et al.*. StealthTest: Low overhead online software testing using transactional memory. In *PACT*, 2009.

[19] J. Chung *et al.*. ASeD: Availability, security, and debugging using transactional memory (poster). In *SPAA*, 2008.

[20] J. Devietti *et al.*. Hardbound: Architectural support for spatial safety of the C programming language. In *ASPLOS*, 2008.

[21] K. E. Moore *et al.*. LogTM: Log-based transactional memory. In *HPCA*, 2007.

[22] L. Ceze *et al.*. Bulk disambiguation of speculative threads in multiprocessors. In *ISCA*, 2006.

[23] L. Hammond *et al.*. Transactional memory coherence and consistency. In *ISCA*, 2004.

[24] L. Yen *et al.*. LogTM-SE: Decoupling hardware transactional memory from caches. In *HPCA*, 2007.

[25] M. D. Ernst *et al.*. The Daikon system for dynamic detection of likely invariants. *Sci. Comp. Prog.*, 2007.

[26] M. D. Hill *et al.*. A case for deconstructing hardware transactional memory systems. In *UW-Madison Computer Sciences Technical Report CS-TR-2007-1594*, 2007.

[27] M. Dalton *et al.*. Raksha: A flexible information flow architecture for software security. In *ISCA*, 2007.

[28] M. Kharbutli *et al.*. Comprehensively and efficiently protecting the heap. In *ASPLOS*, 2006.

[29] M. Locasto *et al.*. From STEM to SEAD: Speculative execution for automated defense. In *USENIX ATC*, 2007.

[30] N. L. Petroni *et al.*. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *USENIX Security*, 2006.

[31] N. Petroni *et al.*. Copilot: A coprocessor-based kernel runtime integrity monitor. In *USENIX Security*, 2006.

[32] P. Zhou *et al.*. AccMon: Automatically detecting memory-related bugs via program counter-based invariants. In *MICRO*, 2004.

[33] P. Zhou *et al.*. iWatcher: Efficient architectural support for software debugging. In *ISCA*, 2004.

[34] R. Shetty *et al.*. HeapMon: A low overhead, automatic and programmable memory bug detector. In *1st IBM PAC2 Conference*, 2004.

[35] R. Wahbe *et al.*. Efficient software-based fault isolation. In *SOSP*, 1993.

[36] M. Hammer and D. McLeod. A framework for data base semantic integrity. In *ICSE*, 1976.

[37] T. Harris and S. Peyton-Jones. Transactional memory with data invariants. In *TRANSACT*, 2006.

[38] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Winter USENIX Conference*, 1992.

[39] M. Herlihy and J. E. B. Moss. Transactional support for lock free data structures. In *ISCA*, 1993.

[40] J. R. Larus and R. Rajwar. *Transactional Memory*. Synthesis Lectures on Computer Architecture. Morgan Claypool, 2006.

[41] A. Lenharth, V. Adve, and S. T. King. Recovery domains: An organizing principle for recoverable operating systems. In *ASPLOS*, 2009.

[42] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[43] N. L. Petroni and M. W. Hicks. Automated detection of persistent kernel control-flow attacks. In *CCS*, 2007.

[44] M. Stonebraker. Implementation of integrity constraints and views by query modification. In *SIGMOD ICMD*, 1975.