Using Hardware Transactional Memory for Data Race Detection

Shantanu Gupta
Department of EE and CS
University of Michigan
shangupt@umich.edu

Florin Sultan, Srihari Cadambi, Franjo Ivančić and Martin Rötteler
NEC Laboratories America
Princeton, NJ
{cadambi, ivancic, mroetteler}@nec-labs.com

***Abstract*—Widespread emergence of multicore processors will spur development of parallel applications, exposing programmers to degrees of hardware concurrency hitherto unavailable. Dependable multithreaded software will have to rely on the ability to dynamically detect non-deterministic and notoriously hard to reproduce synchronization bugs manifested through data races. Previous solutions to dynamic data race detection have required specialized hardware, at additional power, design and area costs. We propose RaceTM, a novel approach to data race detection that exploits hardware that will likely be present in future multiprocessors, albeit for a different purpose. In particular, we show how emerging hardware support for transactional memory can be leveraged to aid data race detection. We propose the concept of lightweight *debug transactions* that exploit the conflict detection mechanisms of transactional memory systems to perform data race detection. We present a proof-of-concept simulation prototype, and evaluate it on data races injected into applications from the SPLASH-2 suite. Our experiments show that this technique is effective at discovering data races and has low performance overhead.**

I. Introduction

Due to a variety of factors including production costs, reliability and power consumption, the computing industry is shifting to multi-core processing. Increased system performance will have to be achieved mostly through higher numbers of cores per processor rather than gains from clock speed increases. It is generally understood that the only way to continue performance improvements is by leveraging parallel programming. Some studies even indicate that while cores in future microprocessor chips will be more numerous compared to the present, they may be slower and simpler due to power density concerns. For instance, the cores may not be as deeply pipelined, or may not have sophisticated out-of-order execution mechanisms.

Parallel programming is thus necessary to continue future performance scaling. Writing correct parallel programs, however, is significantly harder than writing correct sequential programs. An insidious problem with parallel programs is data races. Data races occur when two threads of a program access a common memory location without synchronization and at least one of the accesses is a write. Some data races cause no change in the program output, but others produce incorrect results. Data races often indicate programming errors, and are hard to detect primarily due to their non-deterministic nature.

For this reason, automatic data race detection is acknowledged to be a difficult problem that has become extremely important in the light of ubiquitous parallel programming.

Data race detection methods can be classified as either static or dynamic. Static race detection tools analyze different thread interleavings to detect possible data races. However, these techniques are either not scalable or too conservative, and sometimes compromise scalability for false positives. For example, it may not be feasible to check all possible thread interleavings in a reasonable amount of time. Hence, a static analyzer may make a conservative guess, which results in data races being reported when none exist. Similarly, the absence of a lock protecting a shared variable does not necessarily imply a data race. Such false positives are an annoyance and may hamper productivity by causing the programmer to re-examine and attempt to debug portions of the code where no data races exist. On the other hand, in dynamic race detection mechanisms based on locksets [26] or happened-before [13], information about the history of an actual program execution is stored and analyzed at runtime. Such methods are faster but are restricted to discovering only data races exhibited by or close to a particular execution.

Data race detection is slow when performed entirely in software. Hardware-assisted dynamic race detection mechanisms such as HARD [32] or CORD [21] are faster but require specialized hardware, an approach that

F. Sultan was with NEC Laboratories America when this research was carried out. He is now with Telcordia Research, NJ, USA.

is not cost-effective. This paper describes RaceTM, a system that leverages transactional memory hardware, likely to exist in future parallel microprocessors, in order to perform fast and efficient dynamic data race detection. To avoid the costs associated with building specialized hardware, we propose to reuse existing hardware structures, with minimal or no changes, in order to accelerate data race detection. We demonstrate how the same hardware that supports code with transactional critical sections can be used for detecting data races in an entire program.

Transactional Memory (TM) [10] is a parallel programming model that many argue will become a standard for programming future parallel systems, especially multicore processors. Under TM, a program is written in terms of atomic transactions that are speculatively executed. Two transactions are in conflict with one another when at least one of them writes to a memory location that has been accessed by the other. When a conflict is detected between two transactions, one of them is rolled back (i.e., changes it made to memory are not committed) and attempted again. The optimistic speculative execution allows for increased concurrency, since in most cases two potentially conflicting threads will not actually conflict. The goal of TM systems is to achieve the performance of fine-grained locks while providing the relative ease of programming achieved by using coarse-grained locks.

Additional hardware is not necessary to support transactional memory, but software TM systems are estimated to be slower when compared to using fine-grained locks. Hence hardware-assisted TM systems [2], [7], [23], where specialized hardware performs conflict detection and rollback, have been proposed. As an indication of the benefits of TM systems, Sun Microsystems has included support for transactional memory in its Rock processor [28].

Although TM is gaining popularity as a programming model, not all the code of a program will be covered by transactions, for several reasons. First, I/O and certain system calls are irreversible and cannot be undone, and therefore cannot be executed speculatively and rolled-back later. Second, code fragments with large memory footprints require large amounts of storage for checkpointing. Converting such code into transactions not only affects performance, but may overrun the limited resources of the underlying TM system [14]. Third, legacy code using locks will likely transition to TM by replacing lock-protected critical sections with transactions [2], [24], [25]. This will leave large sections of the code exposed to data races.

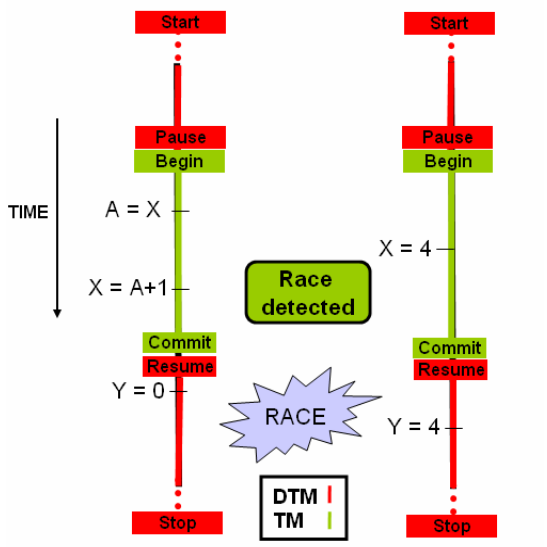

Fig. 1. Debug transactions (DTM) detect a race on Y outside regular transactions (TM).

In [6] we introduced the concept of lightweight *debug transactions* that span non-transactional code and exploit the conflict detection mechanisms of a TM system to detect data races. A programmer or compiler can use simple primitives to manipulate (start, stop, pause, resume) debug transactions during execution. Figure 1 provides a high-level view of how our system works. Data races occurring within transactions (labeled **TM**) are serialized by the TM system, while those occurring outside transaction boundaries, such as the one on variable Y, remain undetected. RaceTM automatically covers unprotected regions of code with debug transactions (labeled **DTM**), and uses the existing TM conflict detection mechanism to detect such races. In this paper, we present an implementation of RaceTM based purely on a hardware-based TM system with minimal changes to the provided hardware support and coherence protocol. This paper also provides details on the debugging support provided and presents experimental results for a subset of the SPLASH-2 [29] benchmarks.

Debug transactions differ from regular transactions in that they do not need to be rolled-back, and hence need no checkpointing support. Further, unlike regular transactions, debug transactions do not change program semantics as they do not enforce atomicity. This enables their use in sections of code not covered by transactions or where it is impossible or difficult to use TM. The concept of debug transactions does not depend on the implementation of the underlying TM system, and is independent of the type of TM conflict detection and

versioning scheme (lazy or eager, see Section II). We present a prototype of RaceTM based on LogTM [17], a hardware TM system. We demonstrate that lightweight hardware support for data race detection is possible by leveraging the same hardware that is required to support TM.

In addition to lightweight hardware support for data race detection, our prototype provides race reporting where the analysis speed can be traded-off with the information provided to the user (debugging stage vs. deployed code). We provide the option of fast analysis with limited information (from only one thread involved in the data race), or slower analysis with complete information. Advantages of our system are that it is non-intrusive (so that race detection can actually be performed during production runs) and scalable with respect to the number of processors. RaceTM is free of additional hardware cost under the assumption of a hardware-assisted TM system. Our evaluation performed with synchronization bugs injected in SPLASH-2 applications exposed no false negatives and generated few false positives.

The remainder of the paper is organized as follows. In Section II, we present an overview of data races and transactional memory. In Section III, we discuss related work. In Section IV, we present in detail our data race detection system. We present experimental results with our prototype in Section V and conclude in Section VI.

II. Background

A. Data Races

Concurrent programs exhibit complex behavior with subtle interactions between threads leading to deadlocks, data races and atomicity violations. In this paper we focus on the problem of data race detection under a transactional memory model that already guarantees freedom from deadlocks. A data race often results in unexpected memory states of shared data structures, which leads to program failures in unrelated parts of the source code. It should also be mentioned however, that many data races are *benign*, i.e., although the data race occurs, it does not cause any unexpected memory corruption or it corrupts non-essential data. An example of a benign race is a corrupted counter that is used for debugging purposes. Furthermore, synchronization primitives implemented by the user directly in the source code can cause incorrect warnings about races.

Static race detection algorithms attempt to be *sound*, i.e., they report any access that could be involved in a data race as a potential bug, which often produces too many warnings. This is amplified by language features such as pointer indirection which forces sound data race detection tools to employ *aliasing* and *points-to analyses* [11]. In contrast to static techniques, dynamic tools detect potential data races that occur at run-time, regardless of memory safety issues and without an explicit up-front pointer aliasing analysis by observing the actual memory state in the single execution trace that is being considered.

B. Transactional Memory

Transactional memory (TM) has been proposed as an alternative programming model to lock-based synchronization. Instead of waiting for a lock, a thread of a multithreaded TM program can make changes to shared memory without being concerned with other threads' actions. The underlying runtime TM system provides a mechanism for concurrency control that discovers and resolves conflicting accesses to shared memory. A TM system presents the abstraction of a set of shared memory locations which a programmer-defined sequence of reads and writes in a thread (a **transaction**) can access atomically and in isolation with respect to accesses from other threads. With TM, a programmer would use two basic system primitives to delimit transactions (`tx_begin` and `tx_end`). The system ensures that writes of a thread that originate inside a transaction occur logically at once, at the commit point of the transaction. To provide this abstraction while ensuring a high degree of concurrency between transactions, the TM system performs two tasks: *conflict detection* and *version management*.

Conflict detection discovers conflicting accesses originating in concurrent transactions. Most TM systems track the write-set (locations written) and read-set (locations read) for each transaction, and signal a conflict if a transaction's write-set overlaps with the read/write-set of other concurrent transactions. Conflict detection can be either eager (at the moment of the access that triggers the conflict) or lazy (deferred to the commit point). On a conflict, the system forces all but one of the transactions involved in it to *abort*; the winning transaction may proceed and eventually *commit* its changes atomically to memory.

Aborting a transaction requires rollback of execution and memory state to the point of the `tx_begin` call. Version management is responsible for storing both the new data (to merge with the memory state on commit) and the old data (to restore the initial memory state in case of abort). Version management can be *eager* or *lazy*. With eager management, memory updates are performed immediately and old values are stored in an undo log. With lazy management, memory updates are buffered until the commit point. There are a

number of proposed TM implementations: *(i)* software-TM (STM), which implements TM primitives entirely in software [8], [9], [27]; *(ii)* hardware-TM (HTM) with full hardware support [2], [7], [17], [23] and *(iii)* hybrid-TM (HyTM), where the hardware provides some support to a STM [4], [16]. HTM is more efficient but may suffer from bounded hardware resources (e.g., write buffers, cache) used to store transactional state [2]. HyTM systems either provide minimal hardware support to speed up or augment an STM implementation [16], or build an STM that augments a best-effort HTM with an alternate software path when resources are exhausted [4].

Dynamic Data Race Detection. TM mechanisms resolve conflicting accesses between concurrent *transactions*. However, non-transactional code is still prone to data races, and despite recent research into how to efficiently enlarge the scope of transactions [16], it is likely that the vast majority of code (either legacy lock-based code ported to TM or new TM-based code) will use small transactions for reasons related to both performance and program semantics [2]. Since it is largely envisioned that lock-based legacy code will most likely be "transactified" (at least in a first approximation) by replacing locks with transactional code sections (see for example [2], [24], [25]), it is likely that dormant data race bugs will silently propagate into the transactionalized code. Worse, such bugs may be introduced in newly written transactional code, as programmers become accustomed to the idea that TM is easy enough to use, while eliding the danger of simple but subtle bugs such as an access to a shared variable slipping out of a transaction's scope. For example, a programmer of low-level code (e.g., a library) may assume that his code is called from an outer scope that is transactional, while the caller ignores this assumption.

We therefore believe that in TM systems data races will continue to be important. We advocate an efficient and cost-effective solution to cope with data races in transactional programs. We make the observation that basic data race detection mechanisms are already present in TM systems, where they are used for detecting conflicting *transactions*. Moreover, in a transactional program these mechanisms are unused for large sections of the code, and those are exactly the places where potential data races are to be found. We propose RaceTM, a novel approach to reusing the conflict detection machinery of a HTM system to provide efficient, accurate and non-intrusive dynamic race detection.

III. Related Work

Dynamic Data Race Detection. Two algorithms have been commonly used for data race detection: lockset and happened-before. Lockset [26] detects races indirectly by reporting violations of a *locking discipline* which states that accesses to a given shared variable must be protected by a common lock. It tracks a per-thread lockset (set of locks held) and, for all shared variables, a per-variable candidate set (the set of locks that *any* thread held at the time it accessed the shared variable). If the candidate set becomes empty, i.e., the variable is not protected by at least one common lock across all accessing threads, then a *potential* race is reported. Happened-before algorithms [19], [21], [22] use the happened-before [13] relation induced by synchronization events between threads of a program to establish a partial order on events in the program (in particular, on memory accesses). If two conflicting accesses cannot be temporally ordered according to the happened-before relation, i.e., they are concurrent, this means they took place with no intervening synchronization operation and are thus involved in a race. Hybrid systems [18], [20], [31] have combined lockset and happened-before. They are all software-based and exploit higher-level abstractions and interfaces at the language level.

Specialized Hardware Support. HARD [32] implements lockset-based detection using Bloom filters to represent locksets and candidate sets. It adds 16-bit fields to each L1/L2 cache line for the candidate set, computed as a hash (Bloom filter vector) over addresses of all locks protecting each shared variable mapped to that cache line. The thread lockset is also stored as a Bloom filter in a special Lock Register in the CPU. In HARD, false positives can occur due to false sharing of a candidate set by variables mapped to the same cache line. A large number of locks may cause collisions in the Bloom filters, yielding non-empty candidate sets for unprotected accesses, and thus false negatives (missed races). HARD requires specialized support from the CPU and the cache hierarchy (special registers and cache fields) and mandates changes to the cache coherency protocol to propagate candidate sets and to broadcast accesses that cause a variable's candidate set to change.

CORD [21] uses a scalar timestamp per cache line along with per-word R/W access bits to *approximate* an happened-before ordering. The timestamp of a cache line is compared against a thread's logical clock at the time of an access to determine whether the latest and the current access are properly ordered. For practical reasons, it deviates from Lamport's happened-before in two ways: *(i)* it forcefully synchronizes a thread's clock even when it detects a data race (potentially obscuring some future races); *(ii)* it may cause threads to falsely synchronize via (unrelated) accesses to main memory,

since it approximates the lost timestamp of an evicted cache line by the largest timestamp seen among all evicted lines. These inaccuracies lead to missed races, in addition to those due to the inherent inability of scalar clocks and timestamps (as opposed to vector clocks [5], [15]) to fully discriminate concurrent events originating in different threads. CORD requires dedicated hardware in the form of logic, additions to caches, the CPU, timestamp buses, and a complex protocol.

A more generalized hardware mechanism has been proposed in [3] to accelerate a range of instruction-grain monitoring tools such as memory checkers and security trackers, in addition to data race detectors. Similar to CORD, this also requires dedicated hardware.

Hardware Reuse. Solutions in this class have looked at how to reuse hardware support originally intended for other functionality to also perform data race detection. Such approaches are most cost-effective as hardware resources are not dedicated exclusively to race detection, thereby amortizing their cost. In [12] data-speculation support present in the Intel IA-64 architecture is used to detect conflicting memory accesses. The compiler inserts a speculative (advanced) load in a thread, along with a conflict check instruction at the point where the loaded value is about to be used. The scheme is limited to a pair of threads that must run on the same core. ReEnact [22] proposes several additions to hardware thread-level speculation (TLS) mechanisms to support both race detection and debugging by deterministic replay. ReEnact orders speculation epochs according to communication events between processors and local serial execution. Further, by ending epochs at synchronization events ReEnact induces an happened-before order that it uses to detect data races.

IV. Transactional Memory-based Data Race Detection

A. RaceTM: General Technique

Transactions partition a given piece of code into two categories, one that is encompassed by transactions and the other that is not. We denote the regions of code covered by user-specified transactions as **TM**, and those not covered as **NoTM**. The TM regions are protected by transactional memory semantics: conflict detection (and subsequent rollback) ensures that accesses to shared memory within transactions do not lead to data races. However, any code outside transactions remains vulnerable to data races. Improper deployment of transactions, in the same way as with locks, can cause programming error leading to data races. Figure 2(a) illustrates this condition where the variable `Y` is undergoing a data race.

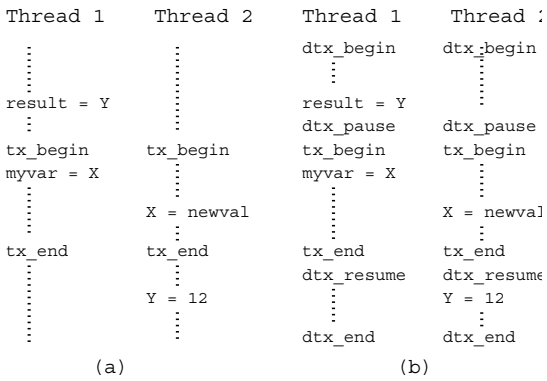

Fig. 2. (a) Code accessing `X` protected by transaction, but the code accessing `Y` vulnerable to a data race, (b) Code accessing `X` protected by transaction. Code accessing `Y` vulnerable to data race which will be detected by the debug transaction.

The key insight of RaceTM [6] is to use the transactional memory conflict detection capability to detect data races, such as the one for variable `Y` in Figure 2(a), that are caused by shared memory accesses in the NoTM portion of the code. It is evident that such races can occur due to programmer mistakes, but they are semantically no different from the ones encountered while missing locks in the code. As a solution, we propose the deployment of *debug transactions* (DTX) that can span regions of code outside regular transactions. Debug transactions behave the same as transactions except that they do not support rollback. They are lightweight transactions capable of performing conflict detection on memory accesses. Rollback support accounts for most of the TM-related overhead since it requires state checkpointing and version management. With DTXs in place, the entire code can be covered by the transactional memory conflict detection mechanisms, and any potential bug in accessing shared memory can be reported. RaceTM provides control primitives for starting/stopping and for pausing/resuming a DTX (explained in detail in Section IV-B2). Figure 2(b) shows a code sample that employs DTXs (specified by calls to primitives prefixed by `dtx`) in the regions of code left out by regular transactions, which makes it possible to detect the data race on `Y`.

The presence in the code of synchronization primitives such as locks and barriers, of which a TM system is not aware, may interfere with debug transactions. For example, locks are expected to still be used in transactional memory programs to serialize accesses to I/O devices (since I/O operations cannot be replayed). Similarly, legacy code in the OS and runtime libraries may also use locks. A DTX can still cover code that uses

these primitives, but at the expense of flagging false data races and requiring additional postprocessing to filter them out. To avoid false race reports from such sources, RaceTM provides support for DTXs to be temporarily paused across lock-protected regions of code, across barriers, as well as, conservatively, for system and library calls that use lock-based synchronization.

Barriers define points in the code where all program threads synchronize. Consequently no data race can occur for memory accesses across barriers. In order to make DTXs cognizant of barrier semantics and avoid signaling races for conflicting accesses separated by barriers, we *reset* the state of all DTXs at program barriers (in addition to pausing DTXs across barrier primitives). Conflict detection in TM systems relies on maintaining read/write-sets either in hardware or software. *Resetting* a DTX means that read/write-sets used for race detection are cleared for all threads when they reach a barrier, thereby avoiding reporting false data races.

Addition of DTXs partitions the code in three categories: **TM**, **DTM** and **NoTM**. The response of a RaceTM system that monitors conflicts between these sections of the code is summarized in Table IV-B1. The behavior is different from that of a conventional transactional memory system only when one of the participating sections is a DTM. The *strong isolation*[1] property of transactional memory can guarantee that interactions of a TM region with DTM and NoTM do not lead to data races. However, to help with debugging transactional programs, RaceTM can easily flag the conflicts between TM and DTM sections. Discovering such conflicts can be of crucial importance, for example when the underlying transactional memory implementation does not support strong isolation. Finally, the conflict between two NoTM sections of code cannot be captured because no record of memory accesses is maintained for those regions.

B. Implementation Details

1) LogTM: We have built a RaceTM prototype on top of an existing hardware HTM (simulated) implementation, LogTM [17]. LogTM distinguishes itself from other transactional memory proposals by integrating TM semantics in a directory cache coherence protocol to enable eager conflict detection and fast commit (by performing eager versioning). Furthermore, LogTM uses software to handle aborts with very little performance penalty. Next, we provide a brief description of the LogTM's version management and conflict detection technique.

[1]Strong isolation implies that transactions are atomic and isolated from all other threads whether or not they are in transactions.

Section 1	Section 2	System response
TM	TM	Rollback one of them
TM	NoTM	-
NoTM	NoTM	-
TM	DTM	Report potential bug
DTM	DTM	Report data race
DTM	NoTM	Report data race

TABLE I
SYSTEM RESPONSES TO MEMORY ACCESS CONFLICTS BETWEEN TWO SECTIONS OF CODE OF A MULTI-THREADED PROGRAM

Version management: Eager version management employed by LogTM stores the new values "in place" and logs old values separately. Each thread is allocated a cacheable virtual memory area for maintaining an undo log of old values. On every store within a transaction, the new value replaces the old value present in the cache. The cache line holding the old value and its address are appended to the log associated with the thread. To avoid redundant log updates, the updated cache line is marked with a write *W* bit (see Figure 4). As discussed next, this W bit is also useful for performing conflict detection between concurrent transactions. The advantage of eager versioning (by replacing old values in the cache) is fast commits. Although aborts are costlier, [17] shows a net gain when commits are more common than aborts.

Conflict detection: Conflict detection is done eagerly and is handled at the coherence protocol level [17]. First, the requesting processor sends out a coherence request. Then, the directory responds or forwards the request to another processor. The responding directory or processor examines the local state to *detect* a conflict and *acks* (no conflict) or *nacks* (conflict) the request. Finally, the requesting processor *resolves* any conflict.

To enable detection, LogTM augments each cache line with a read (R) bit and the write (W) bit (see Figure 4). These bits are set when the cache lines are accessed during a transaction, and flash cleared when the transaction commits or aborts. The detection logic (in the responding processor) flags a conflict when a line with its W bit set gets a remote read or write request, or a line with its R bit set gets a remote write request. When a cache line with its R/W bit set is replaced, the LogTM system sets a per processor overflow bit and subsequently flags conflicts for every remote request forwarded to this processor. The directory state for the replaced cache line is left unchanged so that future requests get forwarded to the responding processor (this feature is called *sticky-bit* in [17]). Thus, LogTM conservatively detects conflicts with slightly-augmented coherence hardware. As we will show later, RaceTM also uses cache line bits and exploits the coherence protocol to detect races. Conflicts are

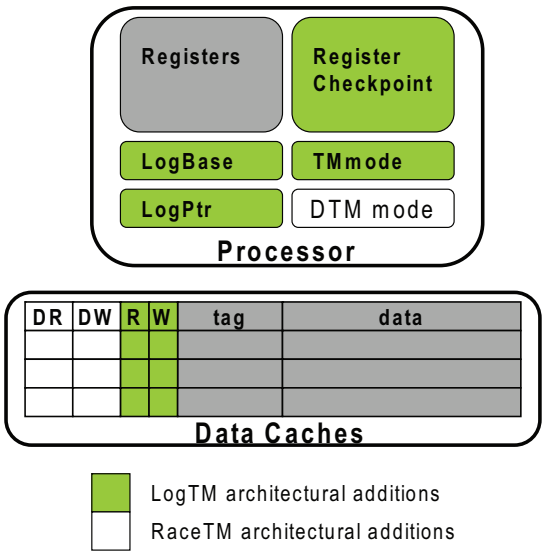

Fig. 4. Architectural additions highlighted for the LogTM and RaceTM. Areas of the processor and cache shaded in green are required by LogTM. Our RaceTM extensions are shaded in white.

resolved by aborting a transaction (when the conflict is between two transactions) or requiring the processor executing a NoTM section to wait and retry (when the conflict is between a transaction and a NoTM section). LogTM handles aborts using software; in RaceTM, we leverage this feature for reporting data races.

2) RaceTM: To keep hardware costs low, we implemented the RaceTM prototype as minimal extensions to the base LogTM coherence protocol. Conceptually, the RaceTM protocol helps in detecting data races as conflicts involving at least one DTX, captured by the existing LogTM hardware support. Note that DTXs need no rollback support from LogTM, i.e., has no checkpointing and logging overheads.

The architectural state added for RaceTM consists of a processor mode bit (DTM mode) and debug read (DR) / debug write (DW) bits for every cache line (see Figure 4). The DTM mode bit is set while the processor runs a debug transaction. The DR and DW bits of a cache line are set when the line is accessed for a read and write, respectively, and the DTM mode bit is set.

Recall that RaceTM divides a program into TM, DTM and NoTM regions. As a processor executes a DTM region of code, its DTM mode bit is set and all memory accesses set DR/DW bits in the respective cache lines. Memory access conflicts between different categories of code are tracked using the LogTM conflict detection mechanism. Essentially, the same technique is used but on a larger set of bits (R, W, DR and DW). A subset of these conflicts qualify as data races, in particular a conflict between two DTMs and a conflict between DTM and NoTM (see Table IV-B1). The coherence protocol messages are tagged with a field that specifies the code category executing on the requesting processor at the time of the access. RaceTM adds no extra messages to the base coherence protocol.

The data race detection proceeds as follows. The requesting processor sends a coherence request tagged with its code category. The directory then responds or forwards the request to another processor. The responding processor examines the local state to detect a conflict. A data race is reported in the following scenarios: (a) A remote read or write request from a processor running DTM or NoTM, and responding processor has DW bit set; or (b) a remote write request from a processor running DTM or NoTM, responding processor has DR bit set. Figure 3 shows the example from Section IV-A and how RaceTM detects the data race.

The eviction of a cache line with either of its DR/DW bits set could result in false negatives, i.e., missing a dynamic data race instance. These events would affect mainly long-distance races, which tend to be rare [22]. Moreover, they will not affect races that occur within close temporal proximity; these are the truly dangerous races since they can actually lead to arbitrary program behavior, e.g., due to randomness in scheduling [22]. Even if RaceTM does miss a dynamic instance of a data race, it is likely that the program bug will eventually be exposed by future data race instances. Because of these reasons, no special support was added to compensate for evicted cache lines.

RaceTM provides the following primitives:

- `dtx_begin`: Starts the DTX: sets the DTM mode bit in the processor. Usually used at the start of a program.
- `dtx_end`: Stops the DTX: clears the DR/DW bits and the DTM mode bit. Usually used at the end of a program.
- `dtx_pause`: Pauses the DTX: clears the DTM mode bit but keeps the DR/DW bits intact. Typically used to skip sections of code that do not need to be covered by a DTX, such as transactions, lock-protected code, etc.
- `dtx_resume`: Resumes a paused DTX: sets the DTM mode bit in the processor.
- `dtx_clear`: Clears the DR/DW bits to reset the DTX state. This is accomplished by ”flash” clearing these bits in the cache, similar to [17]. In our simulator, `dtx_clear` is modeled as a single instruction.

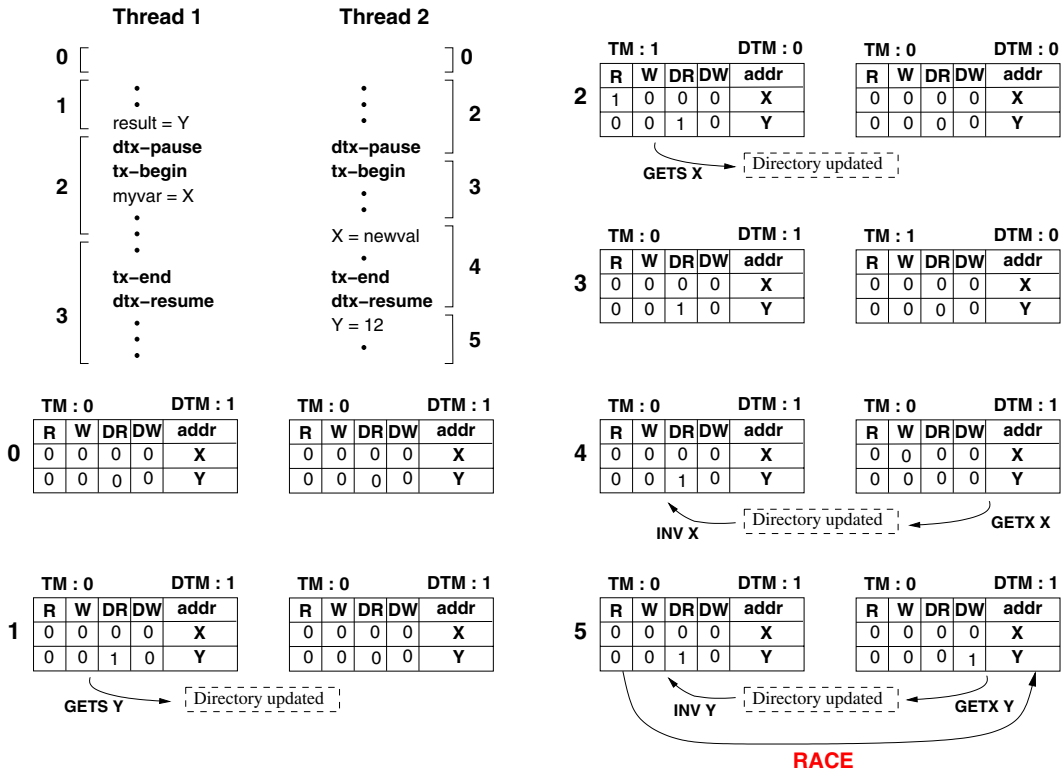

Fig. 3. *RaceTM in action*: This figure shows two parallel threads (top left) running using RaceTM. The number marked next to the code shows the time step at which that portion gets executed. Each grid box pair represents the system state at a given time step. The system state consists of processor's TM and DTM mode bits and R, W, DR, DW bits for two cache lines. (0) System is in the initial state, no bit is set for either of the cache lines and both threads are running debug transactions. (1) Thread 1 sends a read request for address Y, and DR bit gets set. (2) Thread 1 pauses the debug transaction, begins a transaction, and sends an exclusive request for address X; Thread 2 pauses the debug transaction. (3) Thread 1 commits the transaction - consequently the R bit for address X gets cleared - and resumes the debug transaction; Thread 2 begins a transaction. (4) Thread 2 sends an exclusive request for address X, commits the transaction and resumes the debug transaction. (5) Thread 2 sends an exclusive request for address Y, responding processor examines local state and finds the DR bit set for address Y. This is a conflict that qualifies as a data race: Thread 1 responds with a special RACE message; Thread 2 can raise an exception and log the race in software.

Used when all threads synchronize, e.g., at a barrier.

Inserting calls to these primitives into a program is straightforward using the compiler pre-processor. For coarse grained debugging, `dtx_start` and `dtx_stop` can easily encapsulate functions that can potentially run in parallel (i.e., functions that are assigned to threads). Calls to `dtx_pause` and `dtx_resume` can be used to bracket regular transactions, lock protected regions, or any region of code that does not need to be covered by a DTX. `dtx_clear` is called from a barrier to clear the DTX state as described in Section IV-A.

Debugging support. RaceTM can provide feedback to a programmer in the form of race reports. Our prototype uses LogTM's software trap on conflict detection as a medium to report data races. Upon detecting a data race, RaceTM provides the *conflict address* (address of shared memory location triggering the data race), *instruction addresses* (the program counter values for the threads involved when the race was detected) and the *conflict type* (e.g., DTM read conflicting with DW bit).

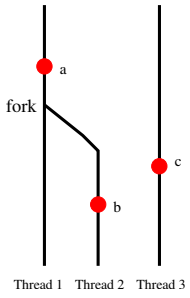

Fig. 5. Trading off false positives and false negatives: The three lines represent the threads' execution timeline. Red dots indicate shared memory accesses.

C. Limitations

Handling of fork/join. The current design of RaceTM

does not provide an elegant way to handle arbitrary `forks` in the program. As an example, in Figure 5 the conflict between accesses *a* and *b* will be flagged as a data race by RaceTM. However, this is a false positive because the fork orders the two accesses. This situation can be avoided by adding a `dtx_clear` at the point of fork, but then the data race between accesses *a* and *c* will be missed, resulting in a false negative. One possible solution to this problem could be the use of timestamps [13], [21] for debug transactions. Note that a common fork-join parallel computation pattern is for a master thread to fork and join all the other threads in the program. RaceTM can easily handle this case by treating the global fork/join as a barrier and clearing the DTX state (we use this feature in our benchmarks of Section V).

Implementation-related Limitations. LogTM uses cache lines to store conflict detection metadata. The effects of this on RaceTM are:

- *Thread migration:* A thread cannot be migrated because its transactional state (including debug state) reside in the cache lines.
- *False sharing conflicts:* Multiple program variables may map to the same cache line, forcing conflict detection bits (R, W, DR and DW) to be shared between them. This can lead to false positives in race detection, an effect that we quantify in Section V.
- *False negatives from evictions:* When a cache line is evicted, debug state is lost, potentially leading to false negatives in detecting conflicts. However, this is not a big problem in practice (see Section IV-B1).

The above implementation-related issues are addressed by a newer version of LogTM, LogTM-SE (Log-based Transactional Memory - Signature Edition) [30] (not publicly available at the time of our implementation). LogTM-SE supports transaction virtualization by decoupling transactional state from caches and enabling software to manipulate it. It uses hashed signatures (Bloom filters) to store and track fine-grained read/write-sets instead of cache line R/W bits. This solution can be also adopted in implementing RaceTM, e.g. by maintaining separate signatures for debug transactional read/write-sets, and opens avenues for future and follow-up work.

V. Experimental Results

In this section we present results of an evaluation of our LogTM+RaceTM implementation. We use a simulation framework similar to that of LogTM [17]. Table II summarizes the architectural parameters of our simulation on a 4-core multiprocessor system.

To evaluate RaceTM, we selected five applications (listed with their inputs in Table V) from the SPLASH-2 suite [29]. The original SPLASH-2 benchmarks had been modified by the LogTM authors to use transactions. We compiled them using the additional preprocessing steps discussed in Section IV-B2, in order to add calls to RaceTM primitives for enabling debug transactions. The goal of the evaluation is to quantify the performance overhead introduced by RaceTM and to determine how effective RaceTM is in exposing bugs that cause data races.

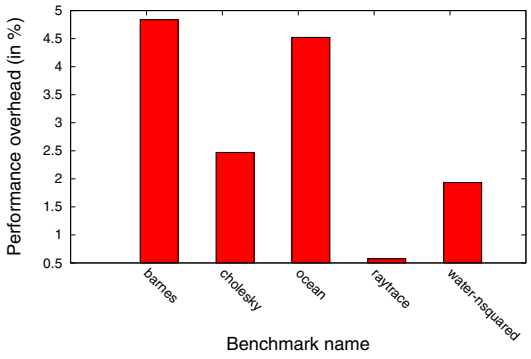

Fig. 6. Performance overhead of RaceTM working compared to the execution of LogTM only.

Performance. The performance overhead of using RaceTM over a baseline LogTM implementation is shown in Figure 6. The maximum performance overhead seen was less than 5%, at par with (and in some cases better than) other hardware based schemes for dynamic data race detection [22], [32]. The overhead of our scheme comes from the addition of calls to primitives for debug transactions. Among these, `dtx_clear` is a dominant contributor to the runtime (`dtx_clear` is called for each instance of a barrier in the code). For a quick comparison of RaceTM with software-based schemes, we also ran Barnes under `helgrind` [1], a lockset-based software race detector, observing overheads in excess of 10x.

Bug detection. Without any alteration to the code, the SPLASH-2 benchmarks reported dozens of data races when run with RaceTM. None of these result in an incorrect program execution, and have also been discussed by previous works [21], [22], [32]. We list these alarms in native code under the column *Original Code* in Table V and divide them into two major categories:

Benign races are typically caused by user-coded synchronization (such as flags) and code optimizations that do not affect program correctness. Instances of

Processors	4 cores, 1 GHz, single-issue, in-order, non-memory IPC=1
L1 Cache	16 kB, 8-way split, 32B cache line, 1-cycle latency
L2 Cache	4 MB, 8-way unified, 32B cache line, 12-cycle latency
Memory	4 GB, 80-cycle latency
Directory	Full-bit vector sharer list; migratory sharing optimization; Directory cache, 6-cycle latency
Interconnect Network	Hierarchical switch topology 14-cycle link latency

TABLE II
SYSTEM ARCHITECTURAL PARAMETERS

Application	Original Code		Bugs Injected
	Benign Raced	False Positives	
Barnes 128 particles	10	2	barrier removal (1) lock removal (2)
Cholesky wr10.O	28	13	lock removal (3)
Ocean 34 × 34 ocean	1	37	barrier removal (2) lock removal (1)
Raytrace teapot	3	2	lock removal (3)
Water-Nsquared 125 molecules	0	1	lock removal (2)

TABLE III
BENCHMARKS TAKEN FROM SPLASH-2 SUITE

Application	False Positives				
	4B	8B	16B	32B	64B
barnes	0	0	0	2	8
cholesky	0	9	7	13	12
ocean	0	0	14	37	48
raytrace	0	0	0	2	2
water-nsquared	0	0	0	1	4

TABLE IV
SENSITIVITY OF THE NUMBER OF FALSE POSITIVES DUE TO FALSE SHARING.

the former are discussed at length in the literature on dynamic data race detection [22]. We exemplify the latter with an instance that RaceTM discovered in the Barnes benchmark. The main loop in Barnes 1) calculates forces between a set of particles, 2) synchronizes threads, 3) moves the particles, and repeats. Lack of synchronization between step 3 and step 1 makes it possible for some of the threads to use old locations for some of the particles in step 1 (not yet updated by other threads still executing in the previous iteration). This presumably has a negligible impact on the final result because errors within a step are amortized over many iterations of the loop, and threads synchronize once every iteration in step 2.

False positives are due to the false sharing of cache lines by program variables, as the result of using cache lines to store read/write-sets in LogTM (see Section IV-C).

Table V shows the number of benign data races and false positives seen in the SPLASH-2 benchmarks. The false positives results were obtained for a cache line size of 32 bytes. In our knowledge, the benign races are inherent bugs in the benchmarks that a programmer introduces intentionally (or unintentionally). The numbers of benign races (per benchmark) found by RaceTM are comparable to the ones presented by [32]. In general, a hardware-only race detection technique cannot discriminate and filter out benign races. On the other hand, false positives can be tackled and their discussion follows next.

To evaluate our scheme with some actual concurrency bugs, we have performed 14 bug injection experiments by adding synchronization bugs to SPLASH-2 applications, as listed in the last column of Table V. The bugs were injected by either *(i)* removing a single instance of a lock, or *(ii)* removing a single instance of a barrier. RaceTM successfully detected all the injected bugs. Thus, the fact that we had a possibility of encountering false negatives due to cache evictions (see Section IV-C) does not end up affecting the effectiveness of RaceTM.

False positives. Table IV shows the sensitivity of the number of false positives in RaceTM with respect to the cache line size. It is evident from this table that the number of false positives can be drastically reduced when using a smaller cache line size. As expected, the number of false positives for a cache line size of 4 bytes (the smallest granularity of a variable in the code) is zero. However, a realistic multiprocessor system cannot be expected to have a 4-byte cache line, therefore a solution is needed to reduce the number of false positives in a real system. Fortunately, as mentioned in Section IV-C, RaceTM can exploit the newer version of the log-based LogTM-SE [30] to completely eliminate this artifact.

VI. CONCLUSIONS

We have described RaceTM, a dynamic data race detection technique that exploits the hardware support for

transactional memory likely to be present in future chip-level multiprocessors. We have proposed the concept of lightweight debug transactions that use the conflict detection mechanism of a (hardware or software) transactional memory system to perform data race detection. We have presented a hardware simulation prototype and evaluated it using synchronization bugs injected into SPLASH-2 benchmarks, showing that RaceTM has low overhead and is effective in uncovering the bugs by detecting data races with good accuracy.

REFERENCES

[1] Helgrind: a thread error detector (http://valgrind.org/docs/manual/hg-manual.html).

[2] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded Transactional Memory. In *Proc. 11th Symposium on High-Performance Computer Architecture*, 2005.

[3] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos. Flexible hardware acceleration for instruction-grain program monitoring. *SIGARCH Comput. Archit. News*, 36(3):377–388, 2008.

[4] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid Transactional Memory. In *Architectural Support for Programming Languages and Operating Systems*, 2006.

[5] C. Fidge. Logical Time in Distributed Computing Systems. *Computer*, 24(8):28–33, 1991.

[6] S. Gupta, F. Sultan, S. Cadambi, F. Ivančić, and M. Roetteler. RaceTM: Detecting Data Races Using Transactional Memory. In *Symposium on Parallelism in Algorithms and Architectures*, 2008. (Brief announcement).

[7] 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 *Proc. 31st International Symposium on Computer Architecture*, Jun 2004.

[8] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *Proc. 18th Conference on Object-oriented programing, systems, languages, and applications*, 2003.

[9] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer. Software Transactional Memory for Dynamic-Sized Data Structures. In *Symposium on Principles of Distributed Computing*, July 2003.

[10] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proc. 20th International Symposium on Computer Architecture*, May 1993.

[11] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *Workshop on Program Analysis For Software Tools and Engineering (PASTE)*, 2001.

[12] A. H. Karp and J.-F. C. Collard. Synchronization of Threads in a Multithreaded Computer Program. *U.S. Patent Application Publication Pub No. US 2005/0283789*, Dec. 2005.

[13] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Comm. ACM*, 21(7):558–565, 1978.

[14] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes — A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proc. 13th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2008.

[15] F. Mattern. Virtual Time and Global States of Distributed Systems. In *Proc. Intl. Workshop on Parallel and Distributed Algorithms*. 1988.

[16] C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. In *Proc. 34th International Symposium on Computer Architecture*, 2007.

[17] K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood. LogTM: log-based transactional memory. In *Proc. 12th Symposium on High-Performance Computer Architecture*, Feb. 2006.

[18] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Principles and practice of parallel programming*, 2003.

[19] D. Perkovic and P. Keleher. Online data-race detection via coherency guarantees. In *Proc. 2nd Symp. on Operating Systems Design and Implementation (OSDI'96)*, 1996.

[20] E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. In *Proc. Parallel and Distributed Processing Symposium*, Apr. 2003.

[21] M. Prvulovic. CORD: Cost-effective (and nearly overhead-free) Order-Recording and Data Race Detection. In *Proc. 12th Symposium on High-Performance Computer Architecture*, 2006.

[22] M. Prvulovic and J. Torrellas. ReEnact: Using Thread-Level Speculation Mechanisms to Debug Data Races in Multithreaded Codes. In *Symposium on Computer Architecture*, 2003.

[23] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing Transactional Memory. In *Proc. 32nd International Symposium on Computer Architecture*, 2005.

[24] H. E. Ramadan, C. J. Rossbach, D. E. Porter, O. S. Hofmann, A. Bhandari, and E. Witchel. TxLinux: Using and Managing Transactional Memory in an Operating System. 2007.

[25] C. J. Rossbach, D. E. Porter, O. S. Hofmann, H. E. Ramadan, A. Bhandari, and E. Witchel. MetaTM/TxLinux: Transactional Memory For An Operating System. 2007.

[26] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[27] N. Shavit and D. Touitou. Software Transactional Memory. In *Proc. 14th Symposium on Principles of Distributed Computing*, 1995.

[28] M. Tremblay and S. Chaudhry. A Third-Generation 65nm 16-Core 32-Thread Plus 32-Scout-Thread CMT SPARC Processor. In *Intl. Solid-State Circuits Conference*, 2008.

[29] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Symposium on Computer Architecture*, 1995.

[30] 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 *Symposium on High Performance Computer Architecture*, 2007.

[31] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. In *Proc. 20th Symposium on Operating Systems Principles*, Oct 2005.

[32] P. Zhou, R. Teodorescu, and Y. Zhou. HARD: Hardware-Assisted Lockset-based Race Detection. *Symposium on High Performance Computer Architecture*, pages 121–132, Feb 2007.