Brief Announcement: RaceTM – Detecting Data Races Using Transactional Memory

Shantanu Gupta
Dept. of Electrical Eng. and Computer Science
University of Michigan, Ann Arbor, MI
shangupt@umich.edu

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

ABSTRACT

Widespread emergence of multicore processors will spur development of parallel applications, exposing programmers to more hardware concurrency. Dependable multithreaded software will have to rely on the ability to dynamically detect data races, which are non-deterministic and notoriously hard to reproduce symptoms of synchronization bugs. In this paper, we propose RaceTM, a novel approach that exploits transactional memory support to detect data races. We introduce the concept of lightweight *debug transactions* that exploit the conflict detection mechanisms of transactional memory systems to perform data race detection. Debug transactions differ from regular transactions in that they do not need to be rolled back, and therefore require no versioning or checkpointing support. Debug transactions do not overlap with a regular transaction, thus providing a transparent mechanism to leverage existing transactional memory support for data race detection.

Categories and Subject Descriptors: D.1.3 [**Software**]: Programming Techniques, Concurrent Programming; D.2.5 [**Software**]: Software Engineering, Testing and Debugging

General Terms: Design, Reliability

Keywords: Transactional memory, Data race detection, Software dependability, Multicore, Parallel Programming

1. INTRODUCTION

Due to a variety of factors including production costs, reliability and power consumption, the computing industry is shifting to multicore processing. Increased system performance will have to be achieved mostly through higher numbers of cores per processor rather than the traditional approach of performance gains from clock speed increases. It is generally understood that the only way to continue performance improvements is by leveraging parallel programming in software development.

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 reproduce 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 actually 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 [7] or happened-before [3], 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. Thus, coverage of all data races in the program is not guaranteed when using dynamic race detection.

This paper proposes RaceTM, a system that leverages transactional memory in order to perform efficient dynamic data race detection. Under transactional memory, 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. We observe the similarity between transactional conflicts and data races, and leverage it for data race detection.

Additional hardware is not strictly 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 [1, 2], 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 [8].

RaceTM, by design, is not constrained to any particular implementation of the transactional memory system and is amenable to both software as well as hardware versions. However, given the fact that hardware transactional memory is likely to exist in future parallel microprocessors, the deployment of RaceTM in hardware presents itself as an attractive opportunity.

Although TM is gaining popularity as a programming model, not all the code of a program can be covered by transactions. First, I/O and certain system calls are irreversible and cannot be executed speculatively and rolled-back later. Second, converting code fragments with large memory footprints into transactions not only affects performance, but may overrun the limited resources of the underlying TM system [4]. Third, legacy code using locks will likely transition to TM by replacing lock-protected critical sections with transactions [1,5,6]. This will leave large sections of the code exposed to potential data races.

In this paper, we introduce 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. Data races occurring within regular transactions are serialized by the TM system, but those occurring outside transaction boundaries remain undetected. As a solution, RaceTM covers unprotected regions of code with debug transactions and uses the existing TM conflict detection mechanism to detect such races.

Debug transactions differ from regular transactions in that they do not need to be rolled-back, and hence need no versioning and 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 (hardware or software), and is independent of the type of TM conflict detection and versioning scheme (lazy or eager). The next section presents some details of the RaceTM technique, showing how the concept introduced here can be effectively employed for data race detection.

2. RACETM

An instance of a data race is shown in Figure 1(a), where the memory location referenced by variable `X` is subject to a data race. Typically, synchronization primitives, such as locks, are used to prevent data races (Figure 1(b)).

In transactional memory systems, locked critical regions are converted into transactions (Figure 1(c)) to indicate that the memory accesses within those regions must be performed *atomically* and in *isolation*. Transactions partition code into two categories, one that is covered by transactions and the other that is not. We denote the portions of code covered by user-specified transactions as **TX**, and portions not covered by transactions as **NoTX**. TX sections of the code are protected by the semantics of transactional memory. In other words, accesses to shared memory locations within transactions do not lead to data races. This is a result of conflict detection (and subsequent rollback). However, any code outside transactions remains vulnerable to data races. Thus, improper deployment of transactions, in the same way as with locks, can cause programming errors leading to data races. Figure 2(a) illustrates this condition where the variable `Y` is undergoing a data race.

The key insight of RaceTM is to use 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 NoTX portion of the code. 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, which in regular transactions incurs most of the overhead since it requires state checkpointing and version management. Debug transactions are thus *lightweight* transactions capable of performing conflict detection on memory accesses. 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. 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, and therefore makes it possible to detect the conflict and data race on variable `Y`.

The presence 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 rolled back or 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 post-processing 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, barriers, as well as, conservatively, for system and library calls that use lock-based synchronization.

Barriers are used in the code as a point 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. Conflict detection for transactional memory systems relies on maintaining a list of memory locations accessed either in hardware or software. Resetting a DTX means that this list of past memory accesses is cleared for all threads when they reach a barrier.

With the addition of DTXs, RaceTM partitions the code into three categories: **TX**, **DTX** and **NoTX**. The response of a RaceTM system that monitors conflicts between these sections of the code is summarized in Table 1. The behavior is different from that of a conventional transactional memory system only when one of the participating sections is a

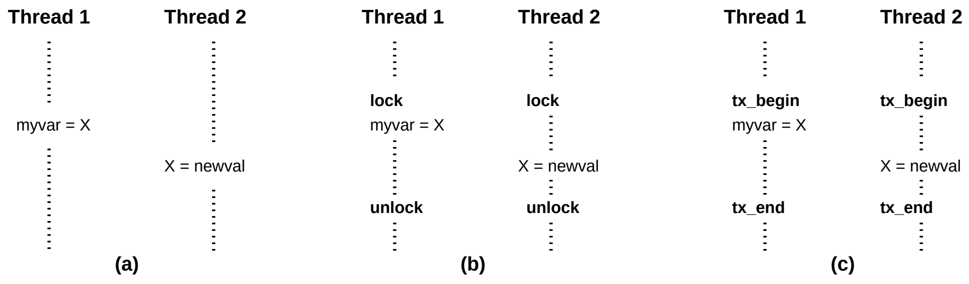

Figure 1: (a) Data race on variable `X`. (b) Critical section involving `X` protected by a lock. (c) Critical section involving `X` protected by a transaction.

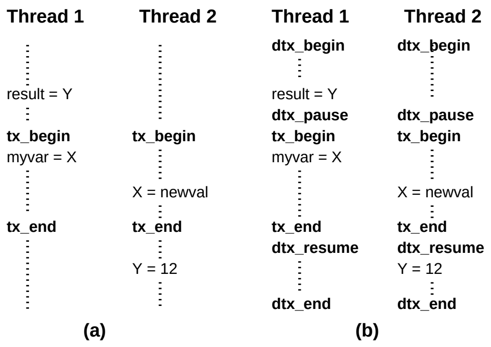

Figure 2: (a) Code accessing `X` protected by transaction, but the code accessing `Y` vulnerable to a data race, (b) Code accessing `X` protected by transaction. Code accessing `Y` vulnerable to data race which will be detected by the debug transaction.

DTX. The *strong isolation*[1] property of transactional memory guarantees that interactions of a TX region with DTX and NoTX do not lead to data races. However, to help with debugging transactional programs, RaceTM can easily flag the conflicts between TX and DTX 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 NoTX sections of code cannot be captured because no record of memory accesses is maintained for those regions.

Section 1	Section 2	System response
TX	TX	Rollback one of them
TX	NoTX	-
NoTX	NoTX	-
TX	DTX	Report potential bug
DTX	DTX	Report data race
DTX	NoTX	Report data race

Table 1: System responses to memory access conflicts between two sections of code of a multi-threaded program

Between eager and lazy detection-based TM systems, RaceTM is better suited to the former. Eager detection flags the memory access conflicts as and when they occur in the program execution. This is a more efficient model for reporting data races than lazy detection, which enumerates all conflicts when a transaction ends. For instance, detecting a race eagerly can allow the program to be stopped immediately and avoid any side effects of the race.

[1]Strong isolation implies that transactional memory accesses are isolated from concurrent nontransactional accesses.

3. CONCLUSIONS

We have described RaceTM, a dynamic data race detection technique that exploits support for transactional memory likely to be present in future chip-level multiprocessors. We have proposed the concept of lightweight debug transactions that uses the conflict detection mechanism of a (hardware or software) transactional memory system to perform data race detection.

4. REFERENCES

[1] 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.

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] M. Tremblay and S. Chaudhry. A Third-Generation 65nm 16-Core 32-Thread Plus 32-Scout-Thread CMT SPARC Processor. In *Proceedings of the 2008 IEEE International Solid State Circuits Conference.* IEEE, 2008.