

But How Do We *Really* Debug Transactional Memory Programs?

Justin Gottschlich Rob Knauerhase Gilles Pokam

Intel Labs

Abstract

With recent announcements of hardware transactional memory (HTM) systems from IBM and Intel, HTM will soon be available for widescale adoption. Such platforms, combined with tested and stable software transactional memory systems, are likely to make real transactional memory (TM) systems available for the first time, which promises to be a more attractive alternative than lock-based parallel programming in terms of programmability and performance.

With these first-ever real systems come several open questions. Perhaps one of the most obvious is, “how does one debug a TM program that uses real hardware?” While prior research in this area exists, there are, to the best of our knowledge, no commercially-available TM debuggers and only a handful of research projects exploring such possibilities, many of which use simulated HTMs that may utilize unrealistic hardware. In this paper, we motivate the need for more than traditional and ad hoc debugging support. We then propose a novel record-and-replay system, based on existing research prototypes and demonstrate its usefulness by reviewing several use cases in TM programming, restricting use to real features in IBM and Intel’s HTMs.

1. Introduction

The advent of multicore processors and their far reaching use in everything from servers to cellphones has made parallel programming possible across a range of devices [1]. Yet, current parallel programming techniques using locks have inherent scalability limitations and are notoriously challenging to program correctly and efficiently, even for the best programmers [5, 9].

To combat this, researchers have spent the last two decades exploring transactional memory (TM), which promises to avoid many of the correctness issues associated with locks (e.g., deadlocks, priority inversion) and hopes to be as or more efficient. A myriad of advances in the past few years has led us to where we are today, with first generation commodity hardware-based TM (HTM) systems available or on the horizon from IBM and Intel [10, 20] and a dedicated community of experts exploring standardization of transactional language constructs for the Standard C++ Programming Language [2]. While these advances move us toward a standardized programming model that may lead to more efficient programs that are easier to reason about, they also

move us into a space where we have limited prior experience, raising the question, “how do we debug TM programs that use *real* hardware?”

Prior research has explored how to debug TM systems, but much of this TM work has been placed squarely in the software-only (STM) space [7, 12, 22, 23]. While HTM debugging research has been investigated, to the best of our knowledge, it has been limited to simulated HTMs [3, 14], where only hardware transactions are debugged. It is not clear if such research is sufficient for real HTM systems, because all of the existing commercial HTMs that we are aware of are best-effort [4]; that is, they cannot guarantee forward progress, requiring an STM fallback mechanism to ensure forward progress. As such, correctness and performance debuggers for real HTMs require the ability to handle the complications that arise when *both* hardware and software transactions execute in the same program.¹ For this reason, debugging the concurrent execution of hardware and software transactions is critical for real systems.

In this paper we show how traditional debugging techniques fail to properly capture and reproduce correctness and performance defects in TM programs. We then describe, at a high-level, an industrial prototype [15, 16] hardware-assisted record and replay (R&R) with minor modifications that enable comprehensive correctness and performance debugging for programs using real TM hardware. We examine the system’s usefulness by analyzing use cases where hardware, software, and hardware and software transactions execute concurrently.

2. Problems with Traditional Debugging

A number of complications arise when applying traditional debugging techniques to parallel programs. Perhaps the most well-known is caused by the nondeterministic nature of multithreaded software. That is, a multithreaded program may yield different outcomes when using identical input because the program’s threads may be interleaved differently across multiple executions. Therefore, an execution may emit a bug during a production run, but may not when later debugged even with identical input. Because of this

¹ IBM’s System z HTM supports constrained transactions that do guarantee forward progress, but, as the name implies, these transactions have strict restrictions and are unlikely to be used as general purpose transactions.

limitation, traditional debugging techniques that work well for sequential codes, such as using breakpoints and single-stepping through software, are inappropriate for parallel software.

Multithreaded debugging complexity increases when programs use transactions. First, TMs can introduce complexity when their execution is speculative. This generally results in two values for every transactional write, a global (or real) value and a speculative value. Detecting inconsistencies between real and speculative values that can emerge from undetected data races is useful for debugging, especially in weakly isolated systems [18]. Second, unlike traditional concurrency mechanisms, like locks, transactions can abort and be retried and yield different behavior on the retried execution. Ensuring this repeatable abort-and-retry ordering can be vital to reproduce cold-path transaction bugs. Third, the conflict management system for an HTM, STM, or hybrid TM system (HyTM), which uses both hardware and software transactions, can be highly complex [17, 19]. Understanding how conflict management schemes affect a transaction’s execution, such as stalling or aborting, is crucial for understanding why a TM program behaves in a specific fashion. For example, certain TMs may abort transactions when they perform uncontended reads to ensure *opacity*, a correctness criterion that prevents illegal TM memory accesses from causing negative program side-effects [8].

Traditional debugger support is insufficient to handle these cases. Even something as simple as extending a debugger to provide the user with real and speculative transaction values is not easily achievable in production HTMs. This is because many of these systems provide limited or no support for *escape actions*, which allow certain tagged instructions within a transaction to execute non-transactionally [14]. Without seeing real and speculative values, programmers are likely to have difficulty identifying certain correctness bugs. Furthermore, extending debuggers so they keep track of transactional bookkeeping information during a debugged execution may be impractical due to the potential *probe effect* – that is, the overhead introduced by analyzing a system – when used with HTMs or HyTMs [7, 22, 23]. Still, even if such probe effects were minimized, universal transactional bookkeeping would not be possible for all real HTMs due to limited or no escape action support. Yet, without some schematic view of an execution, locating and fixing TM performance bugs will continue to be a daunting challenge.

Ad hoc parallel program debugging techniques, like using `printf` to capture the interleaved operations between threads, also do not transition well, or at all, to TM programs. This is because many transactions cannot contain I/O because such instructions are irrevocable (non-abortable) [6, 21]. Furthermore, even in cases where such ad hoc techniques are allowed, they often have the side-effect of chang-

ing the contention signature of the program, because, generally speaking, two or more irrevocable transactions cannot execute at the same time. Therefore, one of the key behaviors of transactions is lost when such an approach is used; that is, speculative execution is not fully realized, likely causing certain TM bugs to lie dormant when such ad hoc debugging techniques are used.

3. Record and Replay for TM

One approach to debugging multithreaded programs is to record enough information during a program execution so that, later on, the same program can be re-executed deterministically. Using this approach, a program can be deterministically replayed many times, which is often necessary for programmers to first analyze and then fix complex bugs. This technique, called record-and-replay (RnR), has provided developers with an avenue to fix both performance and correctness bugs in multithreaded programs that would otherwise be challenging, or impossible, to fix without some form of determinism.

For RnR systems to provide deterministic replay, they generally capture two sources of non-determinism: memory non-determinism and input non-determinism. Memory non-determinism deals with capturing the order of shared memory accesses between threads, while input non-determinism captures non-determinism generally found in system calls, such as the unique identifier returned by `rdtsc` (i.e., read timestamp counter). The main source of overhead in RnR systems is usually in capturing memory non-determinism. Researchers have found that using hardware to track memory non-determinism is an effective way to mitigate such overhead [13, 16].

Our approach to managing this overhead in QuickRec [15] is to use a hardware mechanism that divides a program execution into an ordered sequence of chunks, where a chunk in each thread of execution represents a consecutive sequence of instructions that executes without an intervening shared memory conflict or system event. With this mechanism, the threads’ interleaved shared-memory instructions are captured by the order of the chunks. Input non-determinism, on the other hand, where non-deterministic program inputs originate from either the application itself or the interactions with the surrounding system, are captured by the operating system that runs on top of the recording hardware. While we believe that QuickRec is well-suited to assist in the debugging of many types of parallel programming problems, capturing and reproducing the specific events that occur in programs that use TM present unique challenges.

3.1 Open Challenges in TM RnR

For QuickRec, a chunk is described in terms of number of retired instructions. However, the commercially available

HTM systems considered in this study all enforce strong isolation where the side-effect of an instruction execution in a transaction is shielded until the transaction commits. This creates new challenges in terms of interfacing an RnR system with an HTM core. Adjacent to this issue is the fact that input data is not visible until a transaction commits, making it difficult to collect non-deterministic input data using software-only approaches. In a flat nested TM space, we need not worry about nesting level of transactions, but we expect that such information would be helpful for performance debugging, as well as correctness debugging. We are actively researching solutions to these issues.

Separately, there is the integration of hardware-assisted transactions with STM systems that are often used as a fallback for failed hardware transactions. Conflicting accesses between a hardware transaction that eventually aborts a software transaction may not necessarily have the information needed to accurately point at *which* memory access was the source of the conflict. Because the hardware is unaware of STM activity, we may need to instrument STM libraries or compilers and/or portions of the operating system in order to maintain the correspondence (or lack thereof) between accesses, potentially forcing chunk terminations at certain points within the STM system. We discuss this further in the following sections.

3.2 Recording Transactions

To support replay of hardware transactions, it is important that the recording machinery ensures that each transaction execution is described using chunk information that pertains only to its execution context. If this is not done, non-transactional conflicts may cause chunk terminations leading to false positives. Furthermore, these false conflict chunk terminations may cloak true transactional conflicts because the hardware’s conflict detection mechanism is cleared at chunk termination.

To support transactional chunks, an RnR system would likely need to record the hardware transactional begin and commit events. Furthermore, we believe it will be useful to record all transactional aborts and any error information they contain so the replayer can provide richer debug utility at replay-time.

A key observation is that such an RnR system would not generate new transactional chunks for software transactions. This is done for two reasons. First, it is not clear how RnR hardware would natively support the recording of software transactions without some predefined ISA support for them. Second, as we demonstrate in Section 4, without any specialized events to record software transactions our proposed system still provides meaningful debug information when software transactions are used. However, in some cases, reductions in precision can be observed.

3.3 Replaying Transactions

To replay transactions we propose the following high-level design. First, in addition to the recording requirements presented in Section 3.2, an interface is needed to capture transactional reads and writes for software transactions. Supporting this interface should be possible by augmenting the existing STM or compiler code generation that is used for software transactions. Second, for the replayer to provide useful debug information for hardware and software transactions it likely needs to provide precise conflict detection information (as described in Section 2). To achieve this, we propose the following two data structures shown in Figure 1: AddrMap and Global Replay Data.

The Global Replay Data (GRD) is a shared memory structure that stores the currently executing thread (Active Thread) and its associated instruction pointer (IP). Just prior to executing the next instruction in the replay process, the GRD is updated with the next instruction’s information. If the next instruction is from a different thread, the GRD’s Active Thread data is also updated. Because QuickRec replay is serialized, we assume there is always only one active thread and one associated IP.

The AddrMap is a dynamic array that is used to track the memory addresses that are accessed for each transaction. Each thread is assigned its own AddrMap at replay initialization. With this information, precise transactional conflict information can be gathered during replay, which, in many cases, cannot be obtained during recording because of imprecisions that may exist in the hardware, such as cache line granularity for conflicts which do not precisely detail the specific memory address causing a transactional conflict. A single entry in an AddrMap has the following fields:

- **Address** - the memory address of the transactional access.
- **Access** - the type of access (read, write, or both) associated with the memory address.
- **Value** - the current transactional value associated with the memory address.

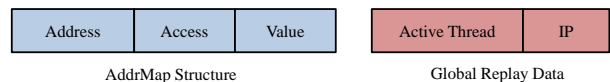


Figure 1. Data structures for QuickRec TM Replay.

During replay, each transactional read or write that is performed, both by hardware and software transactions, is captured by adding an entry to the respective thread’s AddrMap, which contains the information shown in Figure 1. In Section 4 we illustrate the strengths and weaknesses of this approach and its ability to provide meaningful transactional conflict information which is useful in the debugging of both correctness and performance bugs for TM.

4. Transactional Memory Use Cases

At the highest level, uses cases for debugging programs that use transactions generally come in two forms: (i) cases where only transactions execute concurrently and (ii) cases where a transaction executes alongside another type of concurrent access. In this paper, we focus solely on the former, where two transactions execute at the same time.² Furthermore, for this paper we assume eager conflict detection and resolution for hardware transactions, as all of the real HTMs that we are aware of, Intel’s Haswell, IBM’s Blue Gene/Q and System z [10, 11, 20], use such an approach. With this in mind, our RnR techniques are still practical for lazy HTM conflict detection and resolution, as demonstrated by Scenario #3, where a lazy conflict detection and resolution STM is used.

Scenario #1: Two Hardware Transactions³

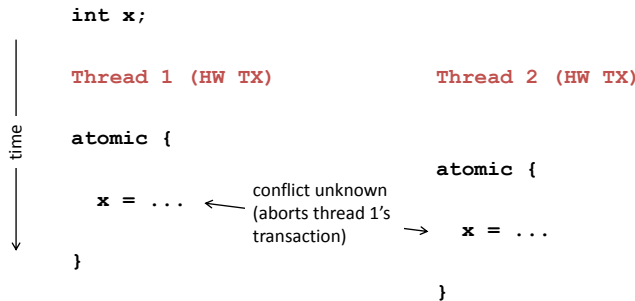


Figure 2. Scenario #1: Two hardware transactions executing concurrently.

Figure 2 illustrates an execution where two hardware transactions concurrently execute and conflict on a shared memory access to variable x . Although trivial, this example demonstrates a fundamental limitation of real HTMs; that is, the conflict that caused thread 1’s transaction to abort is generally unknown when the HTM’s abort handler is invoked. For example, Intel’s Haswell restricted transactional memory (RTM) abort handler contains an error code that identifies the reason why the transaction aborted, but it does not include the actual memory location that caused the conflict. The same is true for IBM’s Blue Gene/Q HTM. Yet, identifying precise conflict locations between transactions is critical to fixing performance and correctness bugs as explained in Section 2.

Consider the challenge of manually determining the source of a transactional conflict if two or more transactions ac-

²Note that while we do not discuss nested transactions, these techniques also apply to flat nested transactions. Complications arise in closed nested scenarios, but we do not discuss them here because all current HTMs in which we are familiar use flat nesting.

³This scenario has identical RnR behavior as executing one hardware transaction in thread 1 and one software transaction in thread 2.

cess many disjoint memory locations across many functions. What if some of these functions reside inside libraries that the programmer cannot see into? This approach seems generally intractable, yet, it is one way debugging is currently being performed on real HTMs. Using the RnR recording extensions we described in Section 3.2, we can move away from this model and instead record the necessary interleavings to track precise transactional conflicts which can later be used to provide meaningful debug information during replay.

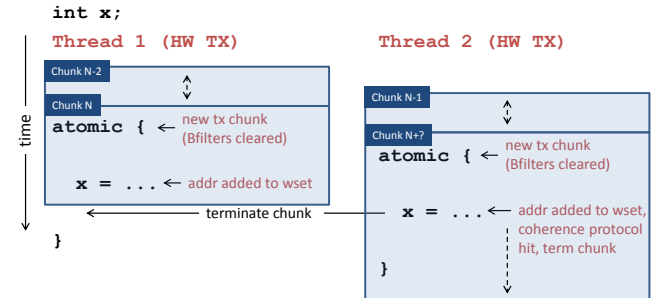


Figure 3. Scenario #1: Recording two hardware transactions executing concurrently.

Figure 3 illustrates the recording process for the transactions shown in Figure 2. Each rectangle represents a distinct chunk. Chunks N and $N+?$ are created by the recording machinery when thread 1 and 2’s respective transaction’s start. Thread 2’s chunk label “ $N+?$ ” means the chunk does not have a precise number associated with it because it has not been terminated yet, but it will be N plus some value. The Bfilters in the figure refer to the Bloom filters used in QuickRec to store read and write set accesses for a given chunk, while wset refers to the write set associated with those Bloom filters. When thread 2 writes to variable x , the chunk in thread 1 is terminated for two reasons. First, this access results in a conflict in thread 1’s write set Bloom filter, which causes a chunk termination using QuickRec’s general RnR system. Second, the same access causes the hardware transaction in thread 1 to abort, which also invokes a chunk termination as described in Section 3.2.

Once recorded, programs can be replay-debugged using the additional logic we proposed in Section 3.3, which monitors memory locations as they are accessed in the program binary. As shown in Figure 4, the replay process is similar to the recording process. However, the replayer uses the AddrMap and the Global Replay Data, described in Section 3.3, to enhance the debugging experience.

The replay process shown in Figure 4 works in the following way. The replayer sequentially replays each chunk of the program, eventually stalling thread 1 at chunk N after variable x is written to and added to thread 1’s AddrMap. At this point, thread 2 will become active and eventually

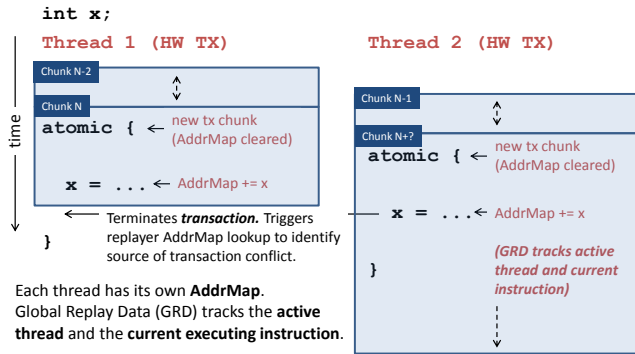


Figure 4. Scenario #1: Replaying two hardware transactions executing concurrently.

perform the write operation that causes thread 1’s transaction to abort, sending it an HTM abort message. Thus far, all of this behavior is nominal to HTMs and chunk-based RnR processing [10, 16]. When the abort message is received by thread 1, and because the replayer sequentially replays chunks and tracks the current thread’s execution and instruction in the GRD, the instruction and memory location that caused thread 1’s transaction to terminate is known. The replay system determines if this is a true or false conflict by performing a lookup into thread 1’s AddrMap when the chunk is terminated, sending it the GRD’s current instruction information as input to the lookup.

An RnR without specific TM hooks can replay this TM program exactly as it was observed, guaranteeing that no causal shared memory interleavings are lost. However, by extending an RnR system to record and replay transactions as described in Sections 3.2 and 3.3, the system can also identify the memory address and source location of the transactional conflict, which is not possible using an RnR without such support.

Scenario #2: Two Software Transactions. Because unbounded HTMs seem unlikely in the foreseeable future [20], any debugging system for TM should also be able to handle software transactions that are likely to execute when transactions cannot be executed in hardware or have reached some threshold of failure. Figure 5 illustrates an example similar to Figure 2, except that instead of executing two hardware transactions, it uses two software ones.

Like the prior example, when recording this program, as shown in Figure 6, the chunk in thread 1 is still terminated when thread 2 writes to variable x. However, unlike the prior example, new chunks are not automatically constructed when the software transactions begin because there is no hardware event to trigger such behavior.

Because of this, there is a greater chance that conflicts outside the scope of the software transactions will negatively

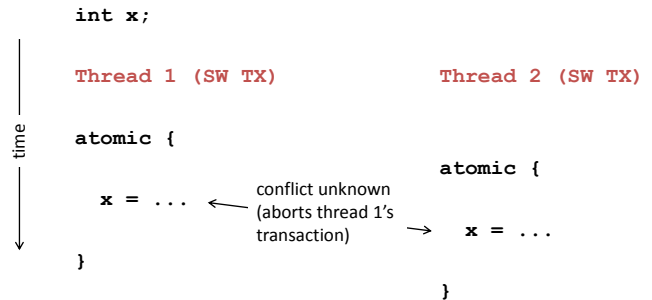


Figure 5. Scenario #2: Two software transactions executing concurrently.

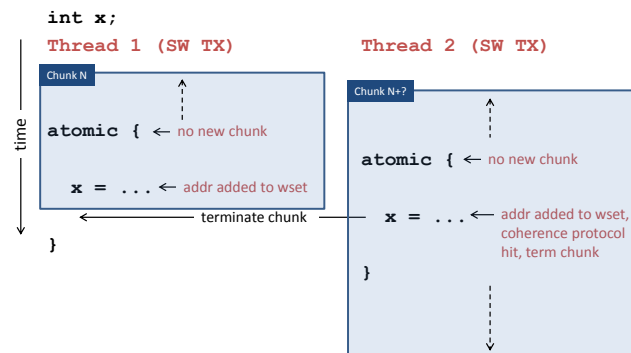


Figure 6. Scenario #2: Recording two software transactions executing concurrently.

influence the precision of the transactional conflict detection mechanism of the recording system. For example, a shared memory access might exist prior to starting a software transaction that then prematurely causes a chunk to terminate while a transaction is still in flight. If this happens, the RnR write and read sets will be cleared, possibly preventing it from capturing conflicts that exist in the actual transaction.

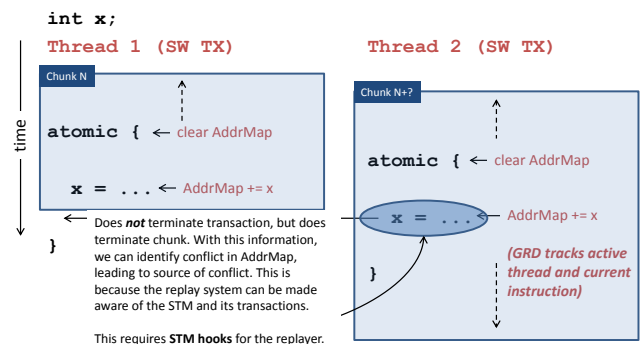


Figure 7. Scenario #2: Replaying two software transactions executing concurrently.

To compensate for this, the STM can be augmented for replay so that transactional reads and writes are added to the AddrMap as previously described in Section 3.3. Then, as long as the AddrMap is cleared upon each new transactional scope, as shown in Figure 7 and the transaction’s writes and reads are added to it as they are accessed, the shared memory interleavings that are captured by the RnR’s recording machinery will recreate the proper execution interleavings, resulting in a deterministic replay that includes precise transactional conflict information for debugging. It is important to note that unlike Scenario #1, in this scenario the AddrMap requires special knowledge of the STM memory accesses and its begin and end events in order for the replayer to correctly identify transactional conflicts.

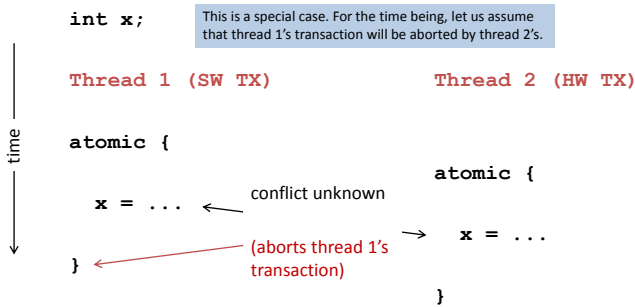


Figure 8. Scenario #3: A software and hardware transaction executing concurrently.

Scenario #3: Software and Hardware Transactions. Identifying precise conflicts between transactions in an RnR system is more challenging when such conflicts are caused by a hardware transaction accessing a memory location that was previously accessed by a software transaction. This is caused, at least, by the following two complications. First, the automatic HTM abort process, which was used to correctly replay and debug Scenario #1, does not apply to weakly isolated STMs. Second, instrumenting an STM’s read and write operations, as was done for Scenario #2, does not capture these types of conflicts because the offending operation occurs outside of the scope of a software transaction. This means that, unlike both prior Scenarios #1 and #2, this transactional conflict is not guaranteed to be precisely identified by either the record or the replay portion of the RnR system.

An RnR system may terminate a chunk when such a conflict occurs, as is shown in Figure 9, however, such chunk termination cannot be guaranteed to be a transactional conflict for the same reasons described in Scenario #2. That is, the chunk termination may be caused by a prior shared memory access before the software transaction started. Furthermore, even when using an instrumented STM at replay-time, the conflict that caused the chunk termination cannot be guaranteed to be a transactional one, because such a conflict origi-

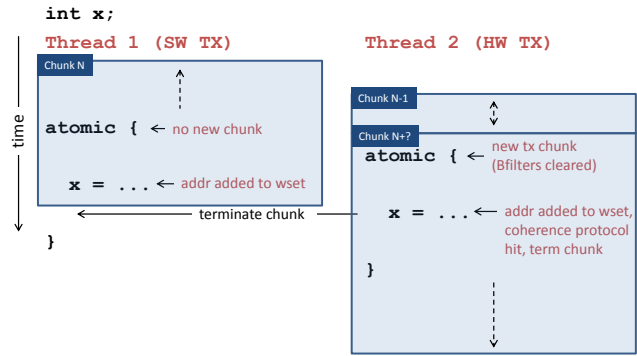


Figure 9. Scenario #3: Recording a software and hardware transaction executing concurrently.

nates from an access from a hardware transaction, which is outside the scope of instrumenting an STM.

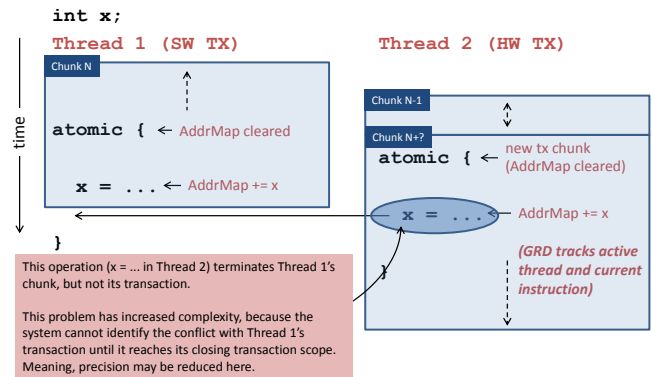


Figure 10. Scenario #3: Replaying a software and hardware transaction executing concurrently.

With this in mind, our proposed design can still pinpoint the conflict memory location in the software transaction by using the AddrMap in the same manner it was used for Scenario #2. Furthermore, while it may not be possible to locate the precise source of the conflict from the hardware transaction, a list of potential candidate locations can be generated. For trivial cases like the one shown in Figure 10, the conflict location is known because the AddrMap contains the conflicting address and we can simply match the memory access to the source code line in thread 2. For more complex cases, where there may be multiple conflicts from multiple disjoint accesses, therefore the RnR system will only be able to provide a list of potential source locations of the conflict.

5. Conclusion

Compared to locks, TM offers a simplified programming model and may lead to improved performance. Yet, debug-

ging TM programs is notably more complex. Traditional debugging techniques, such as using breakpoints and single-stepping through code, are a poor match for programs that use transactions. Ad hoc debugging techniques, such as using `printf` to capture shared memory interleavings, are likely to have limited suitability, or none at all, for TM programs.

Along these lines, we presented three TM scenarios that exploited weaknesses of existing traditional and ad hoc debugging techniques in real HTMs. We then demonstrated how using an existing chunk-based RnR system, with only a few minor modifications for TM, can provide the programmer with the memory conflict address and source code location that caused a transaction to abort, thereby significantly reducing the challenge of fixing performance and correctness bugs in TM programs. We assert that, without something like RnR that supports *always-on* recording and deterministic replay, TM might only be used by expert programmers because the debugging challenge will be too great for the average programmer.

References

- [1] The first step in the multi-core revolution. Technical report, Intel Corporation, Apr. 2005.
- [2] Draft specification of transactional language constructs for C++, 2012. Version 1.1.
- [3] H. Chafi, C. C. Minh, A. McDonald, B. D. Carlstrom, J. Chung, L. Hammond, C. Kozyrakis, and K. Olukotun. TAPE: a transactional application profiling environment. In Arvind and L. Rudolph, editors, *ICS*, pages 199–208. ACM, 2005.
- [4] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *ASPLOS, ASPLOS XIV*, pages 157–168, New York, NY, USA, 2009. ACM.
- [5] J. E. Gottschlich and H.-J. Boehm. Generic programming needs transactional memory. In *TRANSACT*. March 2013.
- [6] J. E. Gottschlich and J. Chung. Optimizing the concurrent execution of locks and transactions. In *LCPC*, September 2011.
- [7] J. E. Gottschlich, M. P. Herlihy, G. A. Pokam, and J. G. Siek. Visualizing transactional memory. In *PACT, PACT '12*, pages 159–170, New York, NY, USA, 2012. ACM.
- [8] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proceedings of the ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 175–184, New York, NY, USA, 2008. ACM.
- [9] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Elsevier, Inc., 2008.
- [10] Intel Corporation. Intel architecture instruction set extensions programming reference (Chapter 8: Transactional synchronization extensions). 2012.
- [11] C. Jacobi, T. Slegel, and D. Greiner. Transactional memory architecture and implementation for ibm system z. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO '12*, pages 25–36, Washington, DC, USA, 2012. IEEE Computer Society.
- [12] Y. Lev. *Debugging and Profiling of Transactional Programs*. PhD thesis, Brown University, 2010.
- [13] P. Montesinos, L. Ceze, and J. Torrellas. DeLorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *Proceedings of the 35th Annual International Symposium on Computer Architecture, ISCA '08*, pages 289–300, Washington, DC, USA, 2008. IEEE Computer Society.
- [14] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting nested transactional memory in logtm. In *ASPLOS, ASPLOS XII*, pages 359–370, New York, NY, USA, 2006. ACM.
- [15] G. Pokam, K. Danne, C. Pereira, R. Kassa, T. Kranich, S. Hu, J. Gottschlich, N. Honarmand, N. Dautenhahn, S. King, and J. Torrellas. QuickRec: Prototyping an Intel architecture extension for record and replay of multithreaded programs. In *ISCA*, June 2013.
- [16] G. Pokam, C. Pereira, S. Hu, A.-R. Adl-Tabatabai, J. Gottschlich, J. Ha, and Y. Wu. CoreRacer: A practical memory race recorder for multicore x86 processors. In *MICRO*, December 2011.
- [17] H. E. Ramadan, I. Roy, M. Herlihy, and E. Witchel. Committing conflicting transactions in an STM. *SIGPLAN Not.*, 44(4):163–172, 2009.
- [18] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and B. Saha. Enforcing isolation and ordering in stm. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, PLDI '07*, pages 78–88, New York, NY, USA, 2007. ACM.
- [19] M. F. Spear, V. J. Marathe, W. N. Scherer III, and M. L. Scott. Conflict detection and validation strategies for software transactional memory. In *Proceedings of the 20th International Symposium on Distributed Computing*, Sep 2006.
- [20] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of blue gene/q hardware support for transactional memories. In *PACT, PACT '12*, pages 127–136, New York, NY, USA, 2012. ACM.
- [21] A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable transactions and their applications. In *SPAA*, 2008.
- [22] F. Zylkharov, T. Harris, O. S. Unsal, A. Cristal, and M. Valero. Debugging programs that use atomic blocks and transactional memory. In *PPoPP, PPoPP '10*, pages 57–66, New York, NY, USA, 2010. ACM.
- [23] F. Zylkharov, S. Stipic, T. Harris, O. S. Unsal, A. Cristal, I. Hur, and M. Valero. Discovering and understanding performance bottlenecks in transactional applications. In *PACT, PACT '10*, pages 285–294, New York, NY, USA, 2010. ACM.