Abstract—The availability of commercial hardware transactional memory (TM) systems has not yet been met with a rise in the number of large-scale programs that use memory transactions explicitly. A significant impediment to the use of TM is the lack of tool support, specifically profilers that can identify and explain performance anomalies. In this paper, we introduce an end-to-end system that enables low-overhead performance profiling of large-scale transactional programs. We present algorithms and an implementation for Intel’s Haswell processors. With our system, it is possible to record a transactional program’s execution with minimal overhead, and then replay it within a custom profiling tool to identify causes of contention and aborts, down to the granularity of individual memory accesses. Evaluation shows that our algorithms have low overhead, and our tools enable programmers to effectively explain performance anomalies.

Keywords—transactional memory; profiling; record and replay

I. INTRODUCTION

After two decades of research, Transactional Memory (TM) [1] has begun to deliver on its promise to simplify the creation of correct, efficient multi-threaded programs. The recent Draft C++ TM Specification [2], with a production implementation in GCC, makes it straightforward to use transactions in place of locks. Similarly, Intel Transactional Synchronization Extensions (TSX) [3] in the Haswell microarchitecture can be used to run language-level transactions with low latency.

Transactions that do not participate in memory conflicts should run in parallel. When a transactional program scales worse than expected, conflict-induced aborts are a likely culprit [4], [5]. In hardware TM (HTM) systems like TSX, aborts also may occur due to syscalls within the transaction, asynchronous interrupts, and accesses to more distinct locations than the hardware allows (typically determined by cache sizes). Upon abort, TSX provides a status code that helps determine if the transaction should be retried [6]. If not, the run-time system typically serializes all transactions by acquiring a global lock, re-executes the code of the aborted transaction, and then releases the lock. In TSX, the status code may indicate that there was a memory conflict, but does not provide information about the other thread(s) that participated in the conflict. System Z [7] can sometimes provide this information.

To understand a program’s contention, the programmer needs to know which source-level transactions conflicted, as well as their conflicting memory accesses (addresses and corresponding source lines of code). Gleaning this information at run time from existing HTM systems seems impossible: During execution in HTM, all memory accesses made by a transaction are invisible to outside observers. A hardware transaction cannot be single-stepped in a debugger, and when it commits or aborts, all information about the set of locations it accessed is immediately discarded.

In this paper, we present an approach to profiling transactions that addresses the aforementioned difficulties. Our work is independent of the underlying TM system and hence applies to a wide range of TM implementations, including today’s TSX systems. Our approach is driven by the following observations:

1) The real-time order of transaction attempts is sufficient to re-create transaction behavior in subsequent executions.
2) Real-time transaction commit and abort ordering can be recorded on existing machines without introducing scalability bottlenecks or noticeable perturbations to program behavior.
3) Accesses to program data by individual transaction attempts are not dependent on the underlying TM mechanism.

Given these properties, we employ a three-step process to identify and resolve abort-related performance pathologies in transactional programs. First, we first re-execute the program with a lightly-instrumented TM library, to measure all transaction start, commit, and abort times. Because transaction orders can be non-deterministic, we may need to re-execute until we record an execution with pathological slowdown. Second, we perform an offline analysis of the program recording to determine (i) the order of committed transactions, (ii) the aborted transactions that accounted for significant execution time, and (iii) the transactions that overlapped with those aborted transactions. Third, we re-execute the program with a specialized software TM (STM) library. This library allows single-stepping through all transaction attempts, even those that ultimately abort, in accordance with the previously recorded transaction order. The library records every memory access from the transactions identified in step 2, along with the corresponding lines of source code. A separate visualizer, such as [8], can use this information to show what interactions between transactions caused costly aborts.

The remainder of this paper is organized as follows. In Section II, we present a system model and the key assumptions made in this work. Section III discusses our technique for low-overhead, bottleneck-free recording of transaction order. Section IV discusses the offline analysis of the recorded information, and Section V introduces our techniques for stepping through transactions and identifying conflicting memory accesses. We evaluate performance in Section VI, contrast with related work in Section VII, and then draw conclusions and discuss future work in Section VIII.

II. SYSTEM MODEL

In this paper, we focus on programs that exclusively use transactions for inter-thread synchronization. Extending our algorithm to support programs with locks and C++ atomic variables would require recompilation (to transform lock operations and atomic variable accesses into transactions), but is otherwise straightforward, and adds no further algorithmic insight.

We assume that programs are race-free. In our opinion, correctness bugs should be addressed with different tools, and earlier in the development life-cycle, than performance bugs. Indeed, for programs that adhere to the Draft C++ TM Specification [2], races in transactional programs, whether due to missing synchronization
or over-decomposition of atomicity, are indistinguishable from the same errors in lock-based programs, and can be detected using the same tools and techniques as classic race detectors [9].

We also assume that transactions are implemented as a shared library, as is the case in GCC and Intel compilers [10], [11], and that the TM implementation is compatible with the Draft C++ TM Specification [2]. The compiler generates two code paths for each transaction in the source code: one path without per-access instrumentation, suitable for HTM, and another with a function call to the shared library on each memory access, suitable for STM. We require the two code paths to make the same memory accesses. Note that memory access reorderings on the HTM path, which can result from weak CPU memory models, do not present a problem. Our STM-based re-execution captures all possible memory accesses by the transactions it profiles, and presents a maximal set of potential sources of conflict.

Lastly, we assume in this paper that programs do not exhibit nondeterminism for reasons other than the order of transactions. For the benchmarks used in this paper, this entails ensuring that threads are created and initialized in a deterministic order. For more general programs, a large body of work already addresses the various sources of I/O and timing nondeterminism [12]–[14]. We assume that using these techniques does not affect the occurrence of performance pathologies.

### III. Lightweight Transaction Recording

While the algorithms in this paper can be easily adapted to most TM algorithms (hardware, software, and hybrid), the implementation presented in this section is specific to HTM, implemented via Intel’s Transactional Synchronization Extensions (TSX) [3], [15].

TSX consists of four instructions, _xbegin_, _xend_, _xabort_, and _xtest_. _xbegin_ serves as a full memory fence, and attempts to begin a transaction. Compiler intrinsics in GCC make it appear that _xbegin_ returns −1 upon successfully starting a transaction. _xend_ ends a transaction. It takes no parameters, has no return value, and also serves as a full memory fence. _xtest_ is used to determine if a hardware transaction is executing, which is useful for hybrid TMs that execute both hardware and software transactions.

When a transaction is aborted automatically, e.g., due to cache overflow, faults, system calls, or conflicts with concurrent memory accesses by other threads, or manually, by a call to _xabort_, the architectural state is restored to just prior to the thread’s _xbegin_ call, and the eax register is set with an approximation of the error that caused the transaction to abort. Compiler intrinsics in GCC make this value of eax appear to be the return value of _xbegin_.

Our system is compatible with every serializable TM implementation of which we are aware, so long as it is compatible with the Draft C++ TM Specification. Our algorithms require a feature of recent Intel processors: the invariant rdtscp instruction. rdtscp reads the processor cycle counter, and is a load memory fence. As implemented on the Haswell microarchitecture, the rate at which the clock advances is independent of voltage and frequency scaling, and reads of the clock from multiple cores are coherent (i.e., if an rdtscp from core A is known to occur before an rdtscp from core B, the return value to core A will be smaller than the return value to core B).\(^1\) For convenience, we refer to the values returned from rdtscp as times.

Given a TM implementation, we determine a suitable total ordering of transactions by placing a store, load, and rdtscp immediately prior to the linearization point [17] of the transaction (in Haswell, _xend_). In all HTM and STM implementations of which we are aware, the transaction must have acquired read or write ownership of all locations it accessed before this point. This sequence ensures that the read of the time occurs after all memory operations within the transaction body, because rdtscp is a load fence. Therefore, if the transaction commits, the system knows a time at which (a) the transaction was valid, and (b) it had already completed all of its memory accesses.

To understand the sufficiency of this algorithm for reproducing the commit order of a program, consider Figure 1. In this example, two transactions, running on two separate cores, reach their commit points. On the left, T1 reads its clock before T2, and commits before T2. In this case, R1 < R2, as expected. On the right hand side, there is an arbitrary delay between when T1 reads the clock and when it commits. In between, T2 may perform memory operations, read the clock, and commit. Thus T2 commits before T1, but R1 < R2. However, if T1 commits, then it must hold that T1 and T2 do not have conflicting memory accesses. Otherwise, the fact that their executions overlap means that one or the other would be aborted by the underlying TM in order to preserve atomicity. Thus, for this case, the program’s behavior does not depend on the order in which T1 and T2 execute, otherwise either T1 or T2 would have aborted. Therefore, the incorrect commit order of [T2, T1] will not affect the reproducibility of the program.

This information is sufficient to re-create an equivalent commit order of successful transactions. However, when profiling a program that exhibits performance anomalies, it is the failed transactions that are most interesting. While our recording algorithm does not concern itself with the actual memory accesses performed by transactions (they can be determined postmortem, by our region-of-interest replay library), it must record information about failed transactions. Specifically, it must identify (a) every transaction attempt; (b) the ultimate result of that attempt, be it a successful commit, or a failure accompanied by an abort status code; and (c) a tight bound on the time during which the transaction could have interacted with other transactions.

In the absence of escape actions [18], there is no benefit from

\(^1\)It appears that this behavior is not yet guaranteed across chips without an additional memory operation [16].
executing additional rdtscp within aborted transactions. Instead, we add two more bits of instrumentation. First, we perform an rdtscp immediately before the transaction begins (i.e., before the \_xbegin). Second, we place an rdtscp as the first instruction of the code that runs upon transaction abort.

This instrumentation is imprecise. It allows determination of a maximal set of transactions that might overlap with a specific aborted transaction, but can report unrelated transactions (i.e., false positives). Consider the execution trace in Figure 2, which introduces arbitrary delays before or after certain rdtscp calls (denoted by Rxxx). The order of clock accesses is indicated by solid arrows. When analyzing why T2 aborted, we conclude that its execution overlapped with T1, T3, and T4, based on the times. However, T1 is the only transaction whose memory accesses occurred during the period between T2’s begin and abort. Adding T3 and T4 to the conflict set represents two false positives. Similarly, T4’s conflict set includes T1 and T2, where T2 is a false positive. In the experiments in Section VI, we did not find false positives to be a significant factor, but false positives are determined by both the workload and number of threads on the machine. We leave a more detailed study of false positives for future work.

The absence of false negatives is more subtle. Since the first clock read (Rxxx) occurs before the transaction begins, and aborted transactions read the clock after aborting, the system cannot miss a true overlap involving two aborted transactions. Overlaps between committed transactions are inconsequential to the determination of the conflict set of an aborted transaction. However, suppose that T3 aborts. The conflict set shown here would involve only T2, even though T1 had not yet committed at the time when T3 began. Intel’s TSX implementation employs a requester wins conflict resolution strategy [19]. Thus given that there are no memory accesses by T3 between Rxx and T1’s commit, we must conclude that T3 did not perform memory accesses between its begin and T1’s commit that conflicted with T3. If it had, T1 would have aborted. Note that if TSX used responder-wins conflict detection (as in LogTM [20]), then an access by T3 before T1’s commit, to a location T1 accessed prior to Rlb, could cause T3 to abort. In that case, there would be a false negative unless we added an additional rdtscp after commit. In that case, the rdtscp before committing would still be needed, but only to record a correct commit order. The post-commit rdtscp would be used for determining overlapping transactions.

For completeness, we present the final instrumentation for our recording library in Figure 3. The underlying TM is TSX with fallback to a single lock [15], [21]. The fallback is taken when a transaction aborts too many times, or when a transaction cannot finish in hardware, due to limits of the HTM implementation.

The log mechanism is implemented via a 24KB memory buffer which is flushed to disk whenever it fills. Each OS-level thread writes to its own file, so that there is no inter-thread serialization when writing to disk. Each log entry consists of 24 bytes (due to padding of fields to word size on a 64-bit system): the time at which the transaction began, the time at which it ended, and a 64-bit status word describing the result (commit-in-hardware, commit-via-lock, or the abort status code given by the HTM).

Note that the modification of the transaction nesting depth in each function places a store and dependent load before any call to rdtscp, thereby preventing the rdtscp from bypassing instructions in program code. Likewise, the commit rdtscp is followed by \_xend (a full fence), the begin rdtscp is followed by a full fence (\_xbegin or an atomic read-modify-write used for lock acquisition), and the abort rdtscp is followed by a check

### Figure 2: An execution trace with aborts

<table>
<thead>
<tr>
<th>Time</th>
<th>Txn T1</th>
<th>Txn T2</th>
<th>Txn T3</th>
<th>Txn T4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R1a</td>
<td>Begin</td>
<td>R2a</td>
<td>Begin</td>
</tr>
<tr>
<td></td>
<td>Begin</td>
<td></td>
<td>Abort</td>
<td>R3a</td>
</tr>
<tr>
<td></td>
<td>Commit</td>
<td></td>
<td>Begin</td>
<td>abort</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>R4a</td>
<td>Begin</td>
<td>R4b</td>
<td>abort</td>
</tr>
</tbody>
</table>

void beginTransaction()
// fast-path when nested
if (++nesting > 1) return
// try the transaction 5 times
attempts = 0
while (true)
  attempts++
  // get time just before beginning
  addToLog(rdtscp(), Start)
  status = \_xbegin()
  // abort the transaction if lock held
  if (!lock.held())
    if (isOk(status))
      lock.held()
      \_xend()
      \_xabort(LockHeld)
  else
    \_xabort(LockHeld)
  else // this runs only on abort
    // get time just after aborting
    addToLog(rdtscp(), status)
    // wait for lock to be released
    while (lock.held()) spin()
    fullMemFence()
    // fall back to lock?
    if ([attempts > 4] || cantCommitInHW(status)
      lock.acquire())
      break

void endTransaction()
// fast-path when nested
if (--nesting > 0) return;
// commit via transaction?
if (in_transaction())
  addToLog(rdtscp(), HtmCommit)
  \_xend()
else // commit via lock
  addToLog(rdtscp(), LockCommit)
  lock.release()
of the spin lock. The spin lock is a C++ atomic variable, and the access includes a full fence. Thus our implementation prevents reordering of timestamp reads, without excessive synchronization.

IV. USING REPLAY FOR VISUALIZATION AND ANALYSIS

The information recorded above is sufficient to guide deterministic re-execution of the program, as it captures the real-time commit order of transactions. However, while it can be processed to show that instances of transactions conflict, it does not correlate those transaction instances to specific source-level transactions. Furthermore, it is not sufficient for diagnosing memory contention, as it does not include individual memory accesses.

We can remedy the first problem by leveraging lightweight deterministic replay. Our philosophy is simple: when recording, it is best to minimize overheads so as to perturb the behavior of the program as little as possible; when replaying, it is possible to use a functionally equivalent replay that has additional instrumentation.

We process the threads’ logs offline to produce per-thread functional replay scripts (FRS). An FRS has as many entries as there are committed transactions for that thread. If we think of the global transaction commit order as a logical clock, then a thread’s FRS indicates the times its transactions committed. For example, given the commit order: \([T_1, T_2, T_3, T_2, T_2, T_3, T_2, T_1, T_2]\), \(FRS_1 = [0, 7]\), \(FRS_2 = [1, 3, 4, 6, 8]\), and \(FRS_3 = [2, 5]\). Our sequential deterministic replay library uses the threads’ FRSs, along with a single counter \((txOrder)\) incremented after each commit, to enforce an order on all transactions. Non-transactional code runs concurrently.

The pseudocode for this library’s transaction instrumentation appears in Figure 4. As in the record library, each thread uses buffered I/O to read one page from the FRS at a time. Since transactions run sequentially, there is no need for per-access instrumentation or HTM involvement.\(^2\)

Transaction begin is a function, and thus at begin time we can use the GNU `backtrace` library to extract from the call stack the program counter corresponding to the source-level `__transaction` statement. This library causes \(3\times\) slowdown, making its use during the recording phase impractical. Each thread saves these program counters to a per-thread program counter log (PCL). There is a one-to-one correspondence between FRS entries and PCL entries. Then, offline, we use the GNU `addr2line` tool to correlate PCL entries with source lines of code.

Armed with this information, we are able to produce a variety of tools to visualize and analyze a program’s execution at a high level. In appearance, the functionalities are most similar to TMProf \([8]\). The main difference is that, at this stage, we cannot see the exact memory accesses performed by the program, though we can see: every transaction attempt; the thread issuing that attempt; the times when the attempt began and ended; the source code corresponding to that attempt; the transaction abort code, if it failed; if the transaction fell back to the lock, if it committed; and the set of transactions with which each transaction might have overlapped.

Among the analyses these data afford are:

- Frequency of necessary serialization for each source-level transaction: computed as a function of the number of aborts a transaction incurs before committing in serial mode, and the value of the status register for each of those aborts.
- Frequency of serialization due to contention for each source-level transaction: the total number of serialized transactions minus the previous quantity.
- Average transactional load: the maximum number of transactions in flight at any time, and their necessary serialization rate, can give insight into whether symmetric multi-threading is limiting transactional capacities.

Note that TSX treats abort status codes as hints, and thus these metrics are approximations.

The data also identifies transactions that abort repeatedly, either by instance, or by source location. After identifying a specific instance of a specific source-level transaction that appears to be suffering from contention (i.e., it aborts repeatedly, other transactions run concurrently with it, and status codes do not indicate that there is a capacity issue), the programmer can use our tools to generate a “region of interest” (ROI) script. The ROI script is used in conjunction with our interactive replay library, described in the next section.

V. REGION OF INTEREST REPLAY

In Section IV, we produced a functionally equivalent deterministic replay of a program by eliminating all aborted transactions from the recorded execution, and then forcing transactions to execute in the same order as in the recorded execution. A particularly powerful feature of this approach is that the same output can be produced using any TM algorithm, as long as commit order is preserved. Previously we used a sequential algorithm to replay the program to determine the source line corresponding to each transaction begin. We now introduce parallel and interactive algorithms, with which we can achieve our remaining profiling goals.

A. Parallel Partial Replay

Given a pre-determined commit order, we can execute the program as a set of speculative ordered transactions \([22]\). Figure 5 presents the foundation for our parallel partial replay algorithm (PPR). We use lazy subscription \([21]\) to enable multiple threads to execute transactions in parallel, and then to enforce the original order at commit time. The `order` object abstracts away details of the threads’ FRSs and is used to enforce ordering. The algorithm is designed for partial replay, in that the `order` object also can be connected to an external profiler, described later in this section.

---

2The current Draft C++ TM Specification is evolving with regard to explicit self-abort and cancellation of transactions. Supporting this feature once it finalizes is trivial.
void beginTransaction()
// fast-path when nested
if (++nesting > 1) return
while (true)
// start a transaction, ignore commit
// order for now
status = _xbegin()
if isOK(status)
    break
// on abort, retry immediately unless
// transaction can’t complete in hardware
else
    if cantCommitInHW(status)
        // wait until this transaction is
        // oldest, then run without TM
    while (order.getNext() != myThreadId)
        spin()
    break

void endTransaction()
// fast-path when nested
if (--nesting > 0) return;
// if in transaction, don’t commit unless
// allowed by commit order
if in_transaction()
    if (order.getNext() != myThreadId)
        _xabort(NotMyTurnYet)
        _xend()
// allow next transaction to commit
order.increment()

Figure 5: Instrumentation for parallel replay

The role of the PPR algorithm is to enable fast-forwarding of a program through a prefix of the execution history, in order to reach a set of transactions that the programmer has identified as being involved in a contention hot-spot. We note that PPR is an optional optimization: it is correct to use a sequential algorithm that executes transactions in order. This is a critical feature since PPR is not, in general, safe. There are a number of problems that can plague lazy subscription algorithms [23], which even the state of the art in static analysis and control flow integrity cannot solve efficiently. We allow the programmer to choose whether to use PPR or a sequential algorithm when profiling.

B. Interactive Abort-Based Transaction Inspection

In the Draft C++ TM Specification, aborted transactions cannot have side effects. When a transaction fails, whether due to contention or hardware capacity, it must roll back its changes completely, so that both shared and thread-local program state are indistinguishable from immediately prior to the failed attempt.

Our interactive replay library (IRL) relies upon this behavior. As in Section IV, we use the FRS to replay a program. However, when IRL loads, it connects via a Unix named pipe to an interactive profiler. The profiler allows a programmer to advance the execution by committing transactions from the FRS, using PPR or a sequential ordered transaction library. Thus the programmer can replay transactions, one-at-a-time or in batches, until the program reaches a point where conflicts manifested in the original execution (the programmer would have determined this point via the techniques presented earlier). The programmer can also use an ROI script to automatically advance to this point.

To profile contention and identify its cause, IRL also allows the profiler to execute transactions in a special “inspect” mode. Inspect-mode transactions run serially, using a custom software TM runtime, and can be selected in any order, even an order incompatible with the FRS. However, these transactions never commit. When they reach the commit point, they send a memory trace to the profiler, and then abort and undo their executions.

Implementations of the Draft C++ TM Specification generate two versions of any code that might run in a transactional context: one where individual memory accesses are not instrumented, and one in which they are. Both versions are produced from the same initial source code, and hence access the same shared memory locations. The latter calls the TM library on any load or store of shared memory, to enforce atomicity and isolation. It also calls the library to checkpoint any write to a thread-local variable whose scope extends outside of the transaction.

Of these two versions of a function, the former is safe to run outside of transactions or from within a hardware or serial-ordered transaction, and the latter is only run within software transactions. When a transaction runs using a STM library, the transaction begin function returns a status code that instructs program execution to use the instrumented control flow path.

PPR and our serial replay libraries may use the un-instrumented code path. IRL uses the instrumented path, even though transactions run in isolation. As Figure 6 shows, IRL instrumentation captures transactions’ shared memory accesses.

In IRL, the FRS and the counter that determines which transaction to run next are no longer visible to the TM runtime. Instead, the profiler monitors this information. When it wishes for a transaction to commit (e.g., when the programmer is advancing forward to the point at which contention occurs), it uses Commit as the mode, indicating that the next transaction should commit. The transaction will execute using un-instrumented code, reach its commit point, and then notify the profiler that the transaction is complete. The profiler can then instruct another transaction to run.

In Inspect mode, a transaction determines at begin time that it must use instrumentation. On every memory access, we log the access type (read or write), the virtual address, the number of bytes that were accessed, and the program counter at the time of the access (achieved via the GNU backtrace tool). We then perform the requested memory access. At commit time, the transaction sends its entire list of memory accesses back to the profiler, and then aborts. The abort results in the transaction undoing shared and local effects and then performing a nonlocal jump to the first line of beginTransaction.

While the transaction is rolling back, the profiler (via programmer input or an ROI script) decides whether it should run another transaction in Inspect mode, or commit the next transaction from the FRL. Note that the next transaction in the FRL can also be run in Inspect mode. The profiler also passes the PCs it received to addr2line, to match source lines of code to virtual addresses.

IRL offers additional modes. Force is used to force a specific transaction to commit, in disagreement with the FRL. Nondet abandons the FRL for the remainder of the replay, instead allowing the program to proceed in whatever order occurs naturally. The motivation for these is that manually changing the commit order and then running the remainder of the program reveals whether
void beginTransaction()
// wait for permission to run
while (profiler getNext() != myThreadId)
spin()
// should it run in instrumented mode?
if (myMode == profiler getNextMode())
setTxMode(useUninstrumented)
else // myMode == Inspect
setTxMode(useInstrumented)

void endTransaction()
// if we ran in instrumented mode,
// send accesses to profiler and undo
// the transaction attempt
if (myMode == Inspect)
profiler.send(reads, writes)
undoThisTransaction()
// abort() // jumps to beginTransaction
// otherwise the effects are permanent.
// inform profiler that transaction is done
else // mode == Commit
profiler.send(Done)

T transactionRead(T* addr)
reads.add(addr, sizeof(T), backtrace())
return *addr

void transactionWrite(T* addr, T val)
writes.add(addr, sizeof(T), backtrace())
undolog.add(addr, sizeof(T))
*addr = val

Figure 6: The interactive replay STM library

the performance of the suffix of the execution improves once the contention is resolved. CommitHW runs a transaction in hardware, and to attempt to commit it. This aids in diagnosing TSX status codes: if a transaction aborts due to capacity, then re-running it in CommitHW reveals whether the application’s transactions are simply too big, or if there was some transient cause of abort. In the latter case, the transaction would succeed when running in CommitHW mode, even though it failed due to capacity during the recorded execution.

C. Usage

Suppose that the program visualization indicates that transaction \( T_c \) by thread 0, experiences contention and ultimately requires serialization to commit. Further, suppose that there are 3 threads total. TSXProf will show which transaction attempts by threads 1 and 2 overlapped with the first attempt of thread 0’s transaction. Because thread 0’s transaction aborted due to conflict, some such transaction must exist. Let \( T_e \) be the earliest such transaction, based on start times. TSXProf would use Commit mode to commit all transactions up to \( T_e - 1 \). It would then be able to profile both \( T_e \) and \( T_c \) using Inspect mode. The profiler would then report a list of locations on which \( T_e \) and \( T_c \) might conflict, and the corresponding lines of code, so that the programmer could begin to design a workaround.

There are two caveats. First, in both STM and HTM, there is the risk of false negatives due to false sharing. Suppose that \( T_e \) writes a byte at address \( A_c \), and that \( T_e \) writes a byte at address \( A_e \). If \( A_e \) and \( A_c \) are on the same cache line, then TSX will abort one of the transactions. Surprisingly, so too will most TM algorithms (the exception is algorithms that detect conflicts using values instead of metadata [24]). Because any such addresses will have all but the lower \( \log_2(\text{sizeof}(\text{ cacheline})) \) bits in common, it is easy for our profiler to diagnose false sharing at the same time as it performs an intersection of the accesses of \( T_e \) and \( T_c \).

The second caveat is that \( T_e \) and \( T_c \) report a maximal set of memory accesses. Suppose that both transactions ought to write every location in a doubly-linked linked list, in the same order, and \( T_e \) attempted to write the first element before \( T_c \). In TSX, when \( T_e \) performed its write to the first element, \( T_c \) would abort: the conflict would have occurred on precisely one location. Since we Inspect \( T_e \) and \( T_c \) separately, and record their full sets of accesses, we generate many potential conflicts (one for each element in the list). This property is useful: Suppose \( T_e \) and \( T_c \) start at opposite ends of the list; on any two runs, the transactions would conflict on one arbitrary location in the list, depending on timing. Attempting to resolve such single-location conflicts is fruitless: the program needs to be redesigned so the transactions do not conflict on the list. By reporting the full set of potential conflicts, TSXProf discourages the programmer from resolving one conflict, re-running the program, and starting over at another position in the list. Instead, the programmer will see that the conflict is inherent in the program’s design, and can use that knowledge to rewrite the code once.

VI. Evaluation

Our evaluation of TSXProf consists of two parts. First, we investigate overheads, using the STAMP benchmark suite [25]. We use the most recent version, which has been ported to use the Draft C++ TM Specification [26]. We use recommended “non-simulator” parameters. For KMeans and Vacation, we run both recommended contention settings. All times are the average of 5 trials. With the exception of Bayes, variance was low for all experiments. In the case of Bayes, the benchmark behavior is dependent on the order in which transactions commit, so high variance is expected.

Second, we explore the benefit that TSXProf offers to programmers, by profiling a transactional implementation of Krieger locks [27]. By using our tool, we are able to both (a) confirm specific patterns of cyclic abort, which are fundamental to the algorithm, and (b) evaluate claims made by Dice et. al [28] about the need for specific algorithmic optimizations.

All experiments ran on an Intel Core i7-4770 CPU running at 3.40GHz. The i7-4770 supports TSX, and contains 4 cores, each 2-way multi-threaded, for a total of 8 hardware threads. The machine ran Ubuntu 13.04 and GCC 4.9.0. The machine had 8 GB of RAM, and a consumer-grade 5400 RPM SATA hard disk. The sizes of the log files produced by our tools depend on the abort rate of the workload. For example, a three-thread Krieger test with 15M commits and 7M aborts produced roughly 166MB per thread.

A. Overhead

To evaluate overhead, we compared four TSX-based TM libraries. HTM-Base uses TSX to execute transactions. Following the advice of Yoo et al. [15], on five consecutive aborts, or when an abort status code indicates that the transaction will never succeed in hardware (e.g., due to making a system call), the transaction
acquires a global lock and then executes without HTM. When an HTM-Base transaction aborts, it waits until the lock is not held, and then retries immediately. In contrast, HTM-Backoff uses simple exponential backoff on abort, ranging from roughly 128 to 2048 cycles. This is a more general approach than HTM-Base.

We also compare two versions of our recording system. HTM-Record extends HTM-Base using the algorithm in Section III. This entails the use of one rdtscp instruction before beginning, another before committing, and another in the abort handler. It also involves writing 24 bytes to disk on any transaction commit or abort, to capture the start and end times of the transaction, along with the abort code. We use per-thread 1024-entry buffers to store transaction events; this size gave the best balance between frequency of I/O and number of TLB entries consumed by the implementation. When the buffer fills, we flush the buffer to disk using synchronous I/O.

Lastly, HTM-Record-NoWrite is identical to HTM-Record, except that the system call to flush the buffer to disk is elided. In this manner, we can isolate disk overheads, which ought to be common to any realistic record/replay system, from overheads specific to the use of rdtscp in our algorithm.

Table I reports single-thread slowdown for HTM-Record and HTM-Record-NoWrite, relative to HTM-Base. We observe no more than 6.6% slowdown from the use of rdtscp, and no more than 16.4% slowdown from the combination of rdtscp and disk I/O. Slowdown is proportional to the frequency of transactions, and inverse proportional to size. Thus SSCA2, with a high frequency of small transactions, incurs the most overhead. In some workloads with the inverse of this pattern, the execution time was marginally faster with rdtscp or disk enabled, though never by more than 1%. From these tests, we conclude that the overhead of recording should be acceptable, as long as it does not affect program behavior at higher thread counts.

To ensure program behavior is unchanged, we first investigated the throughput of HTM-Record. Our use of rdtscp, along with our use of per-thread log files, ought not to introduce bottlenecks. While overheads may be higher, the recording library should scale as well as HTM-Base.

Table II lists the average commit/abort ratio for each library and each of the recommended benchmark configurations, at 4 threads. With the exception of SSCA2, the rates are comparable. In SSCA2, contention is extremely low and there are more than 4M commits per thread per second. In this workload, the introduction of frequent disk writes has the effect of removing one thread from the execution at any time; the commit/abort ratio expresses this effect, with the record library at 4 threads having a ratio only 50% higher than backoff at 3 threads.

Lastly, we verified that the recorded execution provided sufficient information to deterministically replay the programs. To achieve this, we used Bayes, which produces a final value that differs depending on the order in which transactions commit, and a modified red-black tree microbenchmark, where the shape of the tree is printed after 1M random tree mutations are performed by multiple threads. We recorded an execution and then replayed it, testing for differences in output. In all cases, the final output of the replayed execution was identical to the output of the recorded execution, while output differed from one recorded output to the next. Naturally, these tests do not prove the sufficiency and
correctness of our recording mechanism. However, they support our claim that the system is correct, since perturbations in the commit order of transactions would alter the output.

B. Profiling Experience

In reviewing the performance in Figure 7, we see that most benchmarks scale well under TSX. The outliers are (a) Bayes, which exhibits high variance between executions, (b) 8-threaded executions, where hyperthreading effectively halves the capacity of hardware transactions, and (c) Yada, which has an abort rate over 40% at one thread, due to transactions exceeding hardware capacity, and hence falls back to locking too often to scale. These characteristics make STAMP a poor suite to profile.

Instead, we built a microbenchmark that stress tests Krieger locks. The microbenchmark allows the user to specify the number of random memory operations to perform (upon an array of 4096 integers) within each read or read/write critical section, and the ratio of read to read/write critical sections. Unlike the work of Dice et al. [28], we did not rely on strong atomicity to ensure correctness when transactional and non-transactional accesses overlap; instead, all shared memory accesses within the Krieger lock implementation were performed via transactions.

We conducted two experiments. First, we ran a workload with 100% read-only critical sections, with 50 accesses per critical section. We compared throughput for three implementations: HTM-Base, where transactions serialize after 5 failed attempts; HTM-1024, a version of HTM-Base where transactions do not serialize until 1024 failed attempts; and HTM-Backoff. Table III presents the performance of each configuration. Given the relatively short critical sections, and the relatively small transactions (maximum of 6 locations in 3 cache lines), we sought to understand whether the decrease in performance was due to real contention within the queue, or implementation artifacts such as false sharing.

To answer this question, we recorded the benchmark at 3 threads. The logs identified situations in which each of the threads was executing a different source transaction, and the transactions all repeatedly aborted due to contention. We then used region-of-interest replay to identify the exact accesses made by each
transaction in the conflict set.

Figure 8 presents one finding. By correlating the maximum set of locations that each of the transactions could access, we discovered that some of the conflicts we observed were unavoidable: there are symmetric read/write conflicts between $T_2$ and $T_3$, such that repeated attempts can cause them to livelock. This pathology would be detected in STM profilers, as well as HTM profilers. In contrast, conflicts between $T_1$ and $T_2$, which can also cause livelock in HTM, would not affect most STM algorithms, because the read in $T_1$ could simply be ordered before the write in $T_2$. In addition, we discovered cases where false sharing (represented by dashes) was causing additional conflict cycles among the three transactions.

We then ran a configuration of the benchmark in which 5% of operations were read/write critical sections, and write critical sections accessed 100 random locations. Read-only critical sections were unchanged from the previous experiment. In this test, we observed the conflicts from above. However, we now also discovered situations in which simple transactions (consisting of either a single read or a single write to a flag) aborted. When we performed region-of-interest replay to inspect the accesses of the transactions that ran concurrently with the aborted transaction, we discovered that they did not access the same location, though they did access adjacent locations. By replaying, we thus could prove that the aborts were due to false sharing of cache lines. During this replay, we did not encounter false positives. We note, however, that false positives are expected to depend on both the workload and number of cores. Whether false positives will ever be significant cannot be determined without broad usage studies.

Based on this discovery, we added padding to ensure the flag was on its own cache line. We also discovered that the flag was only accessed in transactions that performed no other shared memory accesses. That being the case, we applied an optimization from Dice et al. [28], and replaced the transaction with an access to a C++ atomic variable. Note that this does not require strong atomicity from the underlying TM implementation. The flag is never accessed in transactions that performed no other shared memory accesses.

Figure 9 shows the impact of this rewrite. Though this is a small change, and one that was already shown to be profitable [28], TSXProf was able to exactly pinpoint the reason why the modification improved performance, because its record and replay enabled us to prove that a transaction was experiencing a conflict due to false sharing. The effect is significantly reduced contention (two orders of magnitude fewer aborts at 4–8 threads), and much more stable throughput as contention increases. It is worth noting that the workload is homogeneous, and thus we were able to profile at the lowest thread count where pathological behavior began (3 threads), and then observe benefits out to the maximum number of threads supported by the machine.

### Table III: Krieger microbenchmark throughput, 100% read-only critical sections

<table>
<thead>
<tr>
<th>Threads</th>
<th>HTM-Base</th>
<th>HTM-1024</th>
<th>Backoff</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.77M</td>
<td>2.22M</td>
<td>2.20M</td>
</tr>
<tr>
<td>2</td>
<td>3.85M</td>
<td>3.36M</td>
<td>3.75M</td>
</tr>
<tr>
<td>3</td>
<td>4.65M</td>
<td>2.39M</td>
<td>4.73M</td>
</tr>
<tr>
<td>4</td>
<td>1.45M</td>
<td>1.62M</td>
<td>3.63M</td>
</tr>
<tr>
<td>6</td>
<td>0.35M</td>
<td>0.45M</td>
<td>3.42M</td>
</tr>
<tr>
<td>8</td>
<td>0.23M</td>
<td>0.24M</td>
<td>3.59M</td>
</tr>
</tbody>
</table>

In summary, by replaying an execution and considering the maximal set of potential conflicts, we were able to leverage specific characteristics of the Krieger lock implementation to draw exact conclusions. We could show both that the scalability of the read-only workload was hindered by an intrinsic three-way conflict cycle, and that the muted scalability of the read/write workload was due to false sharing. We were able to draw these conclusions by replaying the program, collecting exact information about the locations read and written by each transaction, and correlating that information with all possible overlaps between transactions within the region of interest. To the best of our knowledge, this sort of reasoning about program behavior has never been possible for HTM before, except in simulation, and for software TM it has only been possible through the introduction of shared memory bottlenecks.

### VII. Related Work

TSXProf performs dynamic profiling, but in theory it is possible to use static analysis to profile contention. Given whole-program analysis [29], a tool might produce a dependence graph [30] to express all possible conflicts between transactions. Recently, this approach was used by Cherem et al. to infer fine-grained locks [31], and by Kestor et al. to detect races in transactional programs [32]. This approach is necessarily conservative, and when used for profiling it risks inundating the programmer with too much information about uninteresting conflicts, e.g., conflicts that do not manifest frequently or affect overall program performance. In contrast, our technique is dynamic, and might miss input-dependent bottlenecks that do not occur during an observed run of the program. Combining static analysis with dynamic profiling is an interesting direction for future work.

Since TSXProf employs a form of record and replay (RnR), it is worth noting that an alternative approach could be to add support for TM to a hardware (RnR) mechanism [33], [34]. While originally envisioned as a tool for identifying hard-to-reproduce or nondeterministic bugs, a number of new uses, to include profiling, have recently been proposed for RnR systems [35], [36]. Profiling via hardware RnR could reduce overhead and employ compression to limit storage costs [13], [37], e.g., through chunking techniques. Similarly, there exist pure-software RnR systems [38], [39] that could potentially be used for performance profiling. The main drawback of hardware RnR is that it is unavailable on today’s
CPUs. The main drawback of software RnR systems is that they tend to focus on correctness of arbitrary applications, not just transactional programs. Thus they must incur substantial slowdown when recording. If the slowdown perturbs the interaction between transactions, it can alter the behavior of the program, leaving the programmer with an execution trace that does not exhibit the performance pathologies that necessitated profiling in the first place. While our work employs ideas from RnR, it benefits from the simplifications that TM provides. Specifically, we can capture high-level transactional behavior while recording very little information, and then employ re-execution to complete the profile and correlate memory operations to transactions.

In theory, profiling should be easier in deterministic programs. There exists a large body of work on software mechanisms for deterministic replay, with exemplar systems including CoreDet [40] and DThreads [41]. While these systems also focus on debugging, rather than profiling, they seek to run in always-on configurations, and thus must have extremely low overhead. For example, Kendo [42] enforces a deterministic order on all lock acquisition and release operations. By doing this, it can ensure determinism only for correctly synchronized lock-based programs. However, this constrained scope allows for an efficient implementation. Determinism has recently been extended to software TM (STM) [43], but the techniques carry high overhead and do not readily translate to HTM. Leveraging determinism to profile transactional programs is an open area of research.

We also note that several STM profiling tools have been proposed [5], [8], [44], [45]. These tools enable visualization of an execution, and correlation of conflicting memory accesses to individual lines of code. Their implementations also share several common attributes. First, they use per-access instrumentation to capture the precise read and write sets of transactions, so that upon an abort, they can identify both parties involved in the conflict. The implementation details vary among proposals, with Chakrabarti’s run-time abort graph being perhaps the most novel, and Gottschlich’s use of Bloom filters the most space-efficient. Secondly, these systems require some form of shared memory communication between transactions before the transactions commit. Third, they tend to rely on some sort of shared memory bottleneck to achieve communication. These characteristics are incompatible with our goal of profiling hardware transactions: in TSX the read and write sets are not visible for tools to observe, and transactions cannot communicate during execution. There are also performance concerns. First, the introduction of any bottleneck during profiling risks altering the interaction between transactions, and hence changing the commit/abort behavior of the program. Second, STM conflict detection mechanisms often produce fewer aborts than the “requester-wins” policy of HTM [19]. Third, profiling overheads in these systems are added to the overhead of STM execution. While these costs may be only 10% higher than simple STM, when added to STM overheads, performance may be over an order of magnitude slower than HTM.

**VIII. Conclusions and Future Work**

In this paper, we presented an end-to-end system for profiling transactional programs. While our techniques can work with most TM systems, we focused specifically on HTM systems using Intel TSX. Our work extends and enhances key ideas from the RnR, deterministic execution, and TM communities to achieve minimal overhead during execution. We also employ a novel use of timestamp counters, which does not introduce bottlenecks when recording programs, thereby ensuring that the execution we record is representative of the behavior of the original program. Our technique allows software replay of hardware-based transactions, and thus it enables previously proposed tools and approaches for profiling software transactions to be applied to hardware-based TM systems.

We showed the benefit of our off-line analysis and interactive replay by analyzing a microbenchmark based on Krieger locks. We discovered specific patterns of inter-thread contention that manifested repeatedly and caused serialization, as well as programming errors (false sharing, unnecessary use of transactions). These sorts of analyses were never before possible for HTM systems.

We plan to extend this work to other TM systems, and to programs that use locks and C++ atomics. We believe that our recording mechanism is compatible with any processor architecture with clocks equivalent to the Intel rdtscp feature, and with any serializable HTM or STM with clear linearization points for transactions. We are also interested in exploring the interaction between hardware RnR and transactional debugging. We expect there is much opportunity for our work to influence the design of performance-focused RnR systems, and for hardware RnR to simplify aspects of this work.

**Acknowledgments**

We thank our reviewers, and especially Jeff Hollingsworth and Tatiana Shpeisman, for their excellent advice. This material is based upon work supported by the National Science Foundation under Grants CAREER-1253362 and CCF-1218530. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

**References**


