Managing Resource Limitation of Best-Effort HTM

Mohamed Mohamedin, Roberto Palmieri, Ahmed Hassan, Binoy Ravindran

Abstract—The first release of hardware transactional memory (HTM) as commodity processor posed the question of how to efficiently handle its best-effort nature. In this paper we present Part-HTM, a hybrid transactional memory protocol that solves the problem of transactions aborted due to the resource limitations (space/time) of current best-effort HTM. The basic idea of Part-HTM is to partition those transactions into multiple sub-transactions, which can likely be committed in hardware. Due to the eager nature of HTM, we designed a low-overhead software framework to preserve transaction's correctness (with and without opacity) and isolation. Part-HTM is effective: our evaluation study confirms that its performance is the best in all tested cases, except for those where HTM cannot be outperformed. However, in such a workload, Part-HTM still performs better than all other software and hybrid competitors.

Index Terms—Transactional Memory, Hardware Transactions, Concurrency

1 Introduction

Transactional Memory (TM) [1], [2] is a support that programmers can exploit while developing parallel applications so that the hard problem of synchronizing different threads, which operate on shared memory locations (or object otherwise), is solved. TM implementations are classified as *software* (STM) [3], which can be executed without any transactional hardware support, *hardware* (HTM) [1], [4], which exploit specific hardware facilities, and *hybrid* [3], which mix HTM and STM. Two events confirmed TM as a practical alternative to the manual implementation of thread synchronization: first, *GCC* (the famous GNU compiler) embedded interfaces for executing atomic blocks; second, Intel released to the customer market commodity processors equipped with *Transactional Synchronization Extensions* (TSX) [5], which allow the execution of transactions directly on the hardware through an enriched hardware cache-coherence protocol. IBM also released Power 8 [6], a processor with best-effort HTM capabilities.

Hardware transactions (or HTM transactions) are much faster than their software version because the conflict resolution and the roll back when aborting is inherently provided by the hardware cache-coherence protocol; however, their downside is that they do not have commit guarantees, therefore they may fail repeatedly, and for this reason they are categorized as *best-effort*. The eventual commit of a transaction that consistently fail using HTM is guaranteed through a software execution defined by the programmer (called *fallback path*). The default fallback path consists of executing the transaction protected by a single global lock (called GL-software path). In addition, there are other proposals that take the choice of falling back to a pure STM path [7], as well as to a hybrid-HTM scheme [8], [9].

In the current HTM implementations, three reasons force a transaction to abort: *conflict*, *capacity*, and *other*. Conflict failure occurs when two transactions access the same memory location and at least one of them wants to write it; a transaction is aborted for capacity if its memory footprint does not fit with the cache constraints of the architecture; and any extra hardware intervention, including interrupts, is also a cause of abort (see Section 2 for more details).

Many recent papers propose solutions to: *i)* handle aborts due to conflict efficiently, such that transactions that run in hardware minimize their interference with concurrent transactions running in the software fallback path [7], [8], [9]; *ii)* tune the number of retries a transaction running in hardware has to undergo before falling back to the software path [10]; and *iii)* modify the underlying hardware support for allowing special instructions so that conflicts can be solved more effectively [4], [11].

Despite this body of work, one of the unsolved problems of best-effort HTM is that there are transactions impossible to be committed in HTM due to the characteristics of the underlying hardware itself. Examples include transactions that require non-trivial execution time even accessing few memory locations and thus they are aborted due to a timer interrupt (which triggers the actions of the OS scheduler); or those transactions accessing several memory locations, such that the problem of exceeding the cache size arises (*capacity* failure). We group these two types of failures into one superset, where, in general, a hardware transaction is aborted if the amount of resources, in terms of space and/or time required to commit, is not available. We name this superset as *resource* failures.

None of the past works target this class of aborted transactions and we turn this observation into our core motivation: solving the problem of resource failures in HTM. To pursue this goal, we propose PART-HTM (whose initial design has already appeared in [12]), an innovative transaction processing scheme that avoids falling back to the GL-software path for those transactions that cannot be executed as HTM due to space and/or time limitation, instead it executes them as a set of hardware transactions.

- *Authors are with the Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA, USA. Dr. Mohamedin and Dr. Hassan are currently affiliated with University of Alexandria in Egypt.*
- *E-mails: {mohamedin,robertop,hassan84,binoy}@vt.edu.*

PART-HTM limits the transactions executed as GL-software path to those that retry indefinitely in hardware (e.g., due to extreme conflicting workloads), or those that require the execution of irrevocable operations (e.g., system calls).

PART-HTM's core idea is to first run transactions as HTM and, for those that abort due to resource limitations, a partitioning scheme is adopted to divide the original transaction into multiple, thus smaller, HTM transactions (called sub-HTM), which can be easily committed. However, when a sub-HTM transaction commits, its updates to memory locations are immediately made visible to others and this inevitably jeopardizes the isolation guarantees of the original transaction. We solve this problem by means of a software framework that prevents other transactions from accessing (or from committing after having accessed) those committed (but still locked) memory locations.

This framework should be non-invasive: a heavy instrumentation would annul the advantages of HTM, falling back into the drawbacks of adopting a pure STM implementation. PART-HTM uses locks to isolate new memory locations written by sub-HTM transactions from others, and a slight instrumentation of read/write operations using cache-aligned signature-based structures to keep track of accessed memory locations. In addition, a software validation is performed to serialize all sub-HTM transactions at a single point in time. PART-HTM does not aim at improving performance of those transactions that are systematically committed as HTM; PART-HTM commits transactions that are hard to commit as HTM due to resource failures without falling back to the GL-software path and by still exploiting HTM leveraging sub-HTM transactions.

PART-HTM ensures serializability [13], the well-known consistency level for on-line transaction processing. With serializability, all committed transactions observe a consistent view of the shared state according to some equivalent serial execution. However, many concurrent applications relying on speculative execution, as TM, are subject to possible erroneous computation due to the access of an inconsistent value even if the execution is doomed to abort. For this reason opacity [14] has been introduced as a correctness level for those applications because it prevents all transactions, including aborted ones, from observing inconsistencies in accessing the shared state. Hardware transactions are partially exempt by those dangerous computation because of *hardware sandboxing* [15], [16], which is a policy that aborts an HTM transaction in case of long or unexpected execution without propagating the possible error to the application itself. However, it has been shown in [16] that the sandboxing currently implemented by Intel TSX cannot prevent all bad executions from happening.

To address this issue, we designed another version of PART-HTM, called PART-HTM-O, that ensures opacity. Doing that means introducing additional code instrumentation that may easily lead to poor performance. We limit this overhead by exploiting a lightweight technique, which we name *address-embedded* write locks, to prevent transactional accesses to locked memory locations at encounter time. This technique deploys a bit-stealing scheme, therefore it has the downside of requiring the application to wrap accesses to primitive data types in an additional level of indirection to enable the manipulation of the requested memory location's address (details in Section 5.5).

We implemented PART-HTM and assessed its effectiveness through an extensive evaluation study including a micro-benchmark, a data structure, the STAMP suite [17], and EigenBench [18]. As competitors, we selected a pure HTM with GL-software path as a fallback, two state-of-the-art STM protocols and a recent HybridTM. In this evaluation, sub-HTM transactions are manually defined after a static code profiler. Results confirmed the effectiveness of PART-HTM. It is the best in almost all the tested cases, except those where HTM outperforms all (and therefore no competitor can perform better than that). In these workloads, PART-HTM still represents the best among other STM and HybridTM alternatives. The combination of these two contributions gives PART-HTM the unique characteristic of being independent from the application workload.

PART-HTM has been designed and evaluated using the Intel TSX implementation of HTM. Other HTM processors, such as IBM Power 8 and Blue Gene/Q, although all inherit the fundamental nature of being best-effort, they have additional features (e.g., Power 8's execution of non-transactional code inside an HTM transaction) and strategies to handle asynchronous events (e.g., CPU interrupts). Given that each HTM implementation has its own set of features, the current design of PART-HTM cannot embrace all of them especially because some require exploring different trade-offs between performance and functionalities. For that reason, in this paper we decided to focus on Intel TSX and scope out other HTM implementations. We selected Intel TSX because, at current stage, it is the most diffuse and affordable processor in the market.

2 PROBLEM STATEMENT

The current Intel HTM implementation is best-effort, namely no transaction is guaranteed to eventually commit because it enforces space and time limitations. In this implementation, the L1 cache (32KB) is used as a transactional buffer for read and write operations. Accessed cache-lines are marked as "monitored" whenever accessed. This way, the cache-line size is indeed the granularity used for detecting conflicts. When two transactions need the same cache-line and at least one wants to write it, an abort occurs. When this happens, the application is notified and the transaction can restart as HTM or can fall back to a software path.

In addition to those aborts due to data conflicts, HTM transactions can be aborted for other reasons. Any cache-line eviction (e.g., due to cache-associativity) of written memory locations causes the transaction to abort (however there is a specialized buffer for handling the eviction of a memory location previously read, but not written). This means that write operations of hardware transactions are limited in space by the size of the L1 cache. However, read operations can go beyond the L1 cache capacity by exploiting the L2 cache. Also, any hardware interrupt, including the interrupts from timers, forces HTM transactions to abort. We name the union of these two causes as *resource* limitation and in this paper we propose a solution for that.

There are two ways to program HTM transactions: Hardware Lock Elision (HLE) and Restricted Transactional Memory (RTM). With HLE, each critical section protected

by a lock is attempted before as transaction and, in case of abort, the original lock is acquired and mutual exclusion is enforced. RTM allows for more flexibility because it provides programmers with the freedom of deciding how to activate a transaction even after its abort by not automatically falling back to the original lock, and also it does not require an already engineered lock-based application while it admits applications to be coded directly with transactions. However, while the commit of transactions in HLE is always guaranteed, in RTM the implementation is responsible to provide a fallback path where any transaction can commit eventually. In this paper we focus on RTM because of its flexibility; however applying PART-HTM to HLE's first speculative trial before the lock acquisition is a simple extension.

	% of Aborts				% of committed		
	Conflict	Capacity	Explicit	Other	GL	HTM	SW
A	10.11%	70.76%	0.04%	19.09%	49.6%	50.4%	N/A
B	93.95%	1.09%	1.14%	3.82%	0.1%	50.3%	49.6%

TABLE 1
Statistics' comparison between HTM-GL (A) and PART-HTM (B) using Labyrinth application and 4 threads.

In Table 1 we report a practical case of resource limitation. The table contains statistics related to the Labyrinth application of the STAMP benchmark. Here we can see how the sum between the percentage of HTM transactions aborted for *capacity* and *other* forms more than 91% of all aborts, forcing HTM to often execute its GL-software path. This is because more than 50% of Labyrinth's transactions exceed the size and time allowed for an HTM execution.

3 RELATED WORK

Before the release of Haswell, the first Intel processor with HTM support, AMD proposed Advanced Synchronization Facility (ASF) [4], which attracted researchers to design initial Hybrid TM systems [7], [19] and test them on AMD simulation stack. However, those systems used ASF support for non-transactional load/store inside HTM transactions, which is not allowed in the current Intel TSX extensions.

The release of Intel HTM processors attracted more research on how to boost HTM capabilities via software [8], [9], [10], [20], [21]. The best-effort nature of this processor motivated different research directions such as: tuning the number of trials in HTM before falling back to the software path [10]; using STM as a fallback path instead of global locking in order to reduce conflict between concurrent HTM and STM transactions [7], [9], [19]; and using reduced hardware transactions where only the STM commit procedure is executed as HTM transaction [8].

PART-HTM takes a different direction from the above proposals. Instead of falling back to either global locking or STM, PART-HTM does not give up HTM execution. In fact, it partitions transactions that fail in hardware due to resource limitations and executes each partition as sub-HTM transaction. Falling back to global locking is chosen only when a transaction cannot succeed in HTM (e.g., due to hardware interruption or irrevocable operations) or when the contention between transactions is very high.

The problem of partitioning memory operations to fit as a single HTM transaction is described also in [22]. In this approach authors used HTM transactions for concurrent memory reclamation. However, unlike PART-HTM, they do not provide a software framework for ensuring consistency and isolation of sub-HTM transactions. In [20], a similar partitioning approach is used to simulate IBM Power 8's rollback-only hardware transaction via Intel Haswell HTM. They ensure opacity by hiding writes that occur in the GL-software path until the end of the critical section without monitoring the reads.

SpHT [23] is a general and effective technique for splitting best-effort hardware transactions. Similar to PART-HTM, SpHT splits the transaction into multiple sub-HTM transactions. Each sub-HTM transaction keeps both an undo-log and a redo-log. The undo-log is used before committing a sub-HTM transaction to restore memory's old values (i.e., hiding the transaction's writes). Then, at the beginning of the next sub-HTM transaction, the redo-log is used to restore the values written in previous sub-transactions. This approach is effective if transactions fail due to computations (e.g., non-transactional code) that can be saved by breaking them down into multiple parts. However, if transactions abort because of resource limitations due to transactional work, which is PART-HTM's primary focus, its effectiveness is reduced because the last sub-HTM transaction still has a redo-log that is as big as the original transaction.

The way a transaction is divided into smaller sub-HTM transactions does not constitute the main contribution of the paper. We consider it an orthogonal problem because there exist several efficient policies that can be applied, often exploiting compiler supports, such as [22], [24], [25], [26], [27]. A close technique that fits PART-HTM's design is presented in [25], where *advisory locks* are used to serialize the portion of an HTM transaction that is more prone to conflict. Those locks are automatically placed by new passes added to the LLVM compiler. Their activation works as follows: transactions are statically analyzed and advisory locking points are identified. These points are possible positions where advisory locks can be placed. At run time, a locking policy decides which of these locking points to activate. Similarly, PART-HTM can statically instrument transactions by adding breaking points between segments, and activate them at run-time to detect boundaries of a sub-HTM transaction.

Transaction Chopping [27] is a methodology for splitting transactions to achieve higher concurrency. Although it is interesting and in principle applicable to cope with resource limitations as PART-HTM, Transaction Chopping assumes rollback-safe transactions: either rollback statements are not allowed or all the rollback statements in a transaction reside in its first chop. PART-HTM does not assume rollback-safe transactions, and thus has less limitations on the design of general purpose concurrent applications.

In [24], Xiang et al. presented ParT, a programming technique that represents another direction for partitioning transactions. ParT partitions transactions into two phases; a read-only (planning) phase, and a read/write (completion) phase. At the beginning of the completion phase, an application-specific validator is defined to validate the output of the planning phase. ParT works well for semantic objects, such as data structures, since their operations often expose a read-only phase that traverses the data structure or does computation. PART-HTM has two advantages over ParT. First, it fits applications where the two phases of ParT

cannot be easily identified. Second, it does not require a programmer-defined validator, and thus it is more generic.

In [28], authors address the problem of capacity abort due to best-effort HTM implementations of many-core architectures by proposing hardware redesigns. A partition mechanism is used to establish multiple logically independent transactional buffers, i.e., partitions, in shared transactional cache, furthermore, to make these transactional buffers dynamically expandable. In contrast with PART-HTM, the work in [28] does not assume a general purpose hardware and it partitions physical resources rather than software transactions.

Partitioned segments in PART-HTM are activated sequentially, one after the other. Activating those segments in parallel is an appealing enhancement. This can be applied in PART-HTM's design by leveraging techniques like those presented in [29], [30]. Such an extension is however left as a future development.

4 ALGORITHM DESIGN

The basic idea of PART-HTM is to partition a transaction that likely (or certainly) fails in HTM (due to resource limitations) into smaller sub-transactions, which could cope better with the amount of resources offered by HTM.

Although the idea of partitioning a transaction into smaller hardware sub-transactions has been already explored in [23], executing them efficiently in a way such that the global transaction's isolation and consistency is preserved still poses a challenging research problem. In this section we describe the design principles that compose the base of PART-HTM, as well as the high level transaction execution flow. The next section describes the algorithmic details. The presented PART-HTM design assumes a address-based TM (for simplicity in the description, we also used the term memory object to indicate its location).

Three-paths Execution. PART-HTM adopts a three-level fallback mechanism to work well in all possible scenarios:

- *Fast Path*. To cope with transactions that do not fail for resource limitations, PART-HTM first tries to execute each incoming transaction (we call it *global transaction* hereafter) as a single non-partitioned HTM transaction. This execution type is called *fast path*. One of our design principles is to minimize the instrumentation cost on that path in order to achieve comparable performance between PART-HTM and pure HTM execution in scenarios where most HTM transactions successfully commit without being split.
- *Partitioned Path*. In case the transaction experiences a resource failure, then our software framework "kicks in" by splitting that transaction and executing it as a sequence of *sub-HTM transactions*. This path is called *partitioned path*.
- *Slow Path*. Transactions that repeatedly fail due to reasons other than resource limitations fall back to an exit path, called *slow path*, where a global lock is acquired and the transaction is guaranteed to complete because it executes in mutual exclusion with respect to any other execution.

Eager Writing. In the partitioned path, we opt for using an eager approach: when a sub-HTM transaction T_{S1} of a global transaction T commits, the shared memory is directly modified with the new values of the memory locations written in T_{S1} without waiting for T to commit, and the old values are kept in a private undo-log.

Using eager writing is the main reason why in PART-HTM, unlike the earlier (lazy) approaches for partitioning transactions (e.g., [23]), sub-HTM transactions have smaller footprints than the global transaction that encloses them. Although using such an eager approach may have some side effects (e.g., increasing the lock holding time of a written location and therefore potential consequences, such as live-lock or higher abort rate), we consider it as a mandatory choice since reducing transactions footprints (and hence overcoming resource limitations) is our main objective. That said, using such a eager writing policy makes PART-HTM's design effective where there are large (in terms of number of accessed memory locations) transactions that are unlikely to conflict with each other. This claim is also confirmed by the evaluation study when applications that expose the above mix of transaction kinds have been tested.

Software Component for Conflict Management. Eager writing imposes three challenges when executing a sub-HTM transaction T_{S1} as a part of a global transaction T. First, it allows other transactions to potentially access the values written by T_{S1} before committing T, thus breaking the isolation of T. Second, once T_{S1} is committed, HTM does not keep any record of its read/written memory locations during the rest of T's execution, therefore it becomes challenging to enforce the consistency of T's (with all its sub-HTM transactions) reads. Third, the effect of committed sub-HTM transactions has to be undone if the global transaction aborts. PART-HTM adds a software component to address those challenges.

Let T^x be a transaction aborted for resource limitations, and let T^x_1, T^x_2, ..., T^x_n be the sub-HTM transactions obtained by partitioning T^x. Let T^x_y be a generic sub-HTM transaction. At the core of PART-HTM there is a software component that manages the execution of T^x's sub-HTM transactions. Specifically, it is in charge of: *1)* detecting accesses that are conflicting with any T^x_y already committed; *2)* preventing any other transaction T^k from committing if it reads or overwrites memory locations updated by T^x_y before T^x is committed (in PART-HTM-O, it prevents other transactions even from reading/overwriting those values); *3)* executing T^x in a way the transaction observes a consistent state of the memory; and *4)* undoing the writes done by sub-HTM transactions of an aborted global transaction.

The software framework does not handle those conflicts that happen on T^x_y's accessed objects when T^x_y is still running; the HTM solves them efficiently. This represents the main benefit of our approach over a pure STM fallback implementation.

Signature-based Metadata. It is clear that the efficiency of PART-HTM depends on the design of the aforementioned software component. Although this component can be trivially implemented by populating the same metadata commonly used by STM protocols for tracking accesses and handling conflicts, applying existing STM solutions can easily lead HTM to lose its effectiveness and, consequently, can lead to poor performance. In the following we point out some of these reasons:

- STM metadata are not designed for minimizing the impact on memory capacity. Adopting them for solving our

problem would stretch both the transaction execution time and the number of cache-lines needed, thus consuming precious HTM resources;
- HTM already provides an efficient conflict detection mechanism, which is faster than any software-based contention manager; and
- HTM monitors any memory access within the transaction, including those on metadata, which takes the flexibility for implementing smart contention policies away from the programmer because concurrent updates on metadata during the transactional execution cause abort.

We do not use the classical address/value-based read-set or write-set as commonly adopted by STM implementations [3]; rather we rely only on cache-aligned Bloom filter-based metadata (just Bloom filter hereafter) to keep track of read/write accesses. We recall that HTM monitors all memory accesses, thus if two HTM transactions write different parts of the Bloom filter (thus different objects), one transaction will be aborted anyway (behavior also known as *false conflict*). Note that, we refer to a Bloom filter [31] as an array of bits where the information (memory addresses in our case) is hashed to a single bit in the array.

Metadata handling is done as follows (the complete list of metadata is in Section 5).
- One Bloom filter shared across all global transactions is used to keep track of memory locations written by committed sub-HTM transactions. This Bloom filter acts as a shared lock table and is updated by sub-HTM transactions before committing to announce its written locations.
- Two Bloom filters per global transaction are used for recording the memory locations read and written by its sub-HTM transactions. The purpose of these Bloom filters is to let read/written memory locations survive even after the commit of a sub-HTM transaction, allowing the framework to check the validity of the global transaction at any time. For this reason, these two Bloom filters are not visible outside the global transaction.
- A value-based undo-log is kept for handling the abort of a transaction having sub-HTM transactions already committed. Unfortunately, this undo-log cannot be optimized using Bloom filters because it needs to store the old values of written and committed locations. Although we consider the undo-log as the biggest source of overhead in PART-HTM, our experimental results show that this overhead is dominated by the gain of partitioning transactions when they face resource limitations. Also, since there is no need to have such an undo-log in the fast path (because it is executed as a single unpartitioned transaction), this overhead is not paid when global transactions fit in HTM.

Non-transactional Code. The design of PART-HTM has a positive side-effect of allowing the execution of non-transactional computation, originally included (and not desired) inside the HTM transactions, as a part of the software framework. However, as a direct consequence of the eager writing policy of PART-HTM, non-transactional code is allowed to access only memory locations locally visible and not globally. This is because, given that non-transactional code is not instrumented, it can access a memory location written by a sub-HTM transaction whose global transaction is not yet committed, therefore ignoring the existence of the lock and overwriting the value. Also, any side effect of non-transactional code cannot be rolled back on abort. This limitation of non-transactional work represents a downside for PART-HTM's design.

Strong Atomicity. Intel HTM implementation provides strong atomicity, which is a property that aborts hardware transactions if a conflicting transactional and non-transactional code is executed. PART-HTM cannot guarantee strong atomicity due to the early exposure of written locations (although locked). In this regard, it is important to recall that HTM-based concurrency controls that admit a software fallback path are subject to the same limitation. Specifically, if a non-transactional operation interferes with locations accessed by the fallback path, then the fallback execution may not be consistent anymore. To protect computation in those scenarios there are orthogonal solutions that can be applied, such as [32], [33].

5 ALGORITHM DETAILS

Figure 1 shows the pseudo-code of PART-HTM's core operations. In Section 5.1 we list all metadata used by PART-HTM. Then, in the subsequent three sections, we show in detail how the transaction is executed in each of the three paths (i.e., fast, partitioned, and slow) mentioned in Section 4.

5.1 Protocol Metadata

As we mentioned before, in order to reduce the metadata size, most of them are Bloom filters (i.e., a compact representation). We refer to any Bloom filter-based metadata as *signature*. Conflict detection using Bloom filters can cause *false conflicts* because the hash function could map more than one address into the same entry. To reduce false conflict, in our implementation Bloom filters are bit-arrays of 2048 bits (4 cache-lines) with a single hash function. Bloom filters are updated using HTM transactions, thus two HTM executions that aim at updating different bits of the same Bloom filter might still conflict if both the bits are stored into the same cache-line due to HTM conflict resolution policy. Having Bloom filters of 4 cache-lines alleviates this problem.

PART-HTM uses two types of metadata: some of them are local, thus visible by only one global transaction (including all its sub-HTM transactions); and others are shared by all transactions.

Local Metadata. Each transaction has its own:
- *read-set-signature*, where the bit at position i is equal to 1 if the transaction read an object at an address whose hash value is i; 0 otherwise.
- *write-set-signature*, where the bit at position i is equal to 1 if the transaction wrote an object at an address whose hash value is i; 0 otherwise.
- *aggregate write-set-signature*. This signature is used only in the partitioned path. In this path we need one signature to save the writes performed by only the current sub-HTM transaction, and a separate signature where all the writes of the enclosing global transaction are saved. The above *write-set-signature* is used for the former and the *aggregate write-set-signature* is used for the latter.
- *undo-log*, it contains the old values of the written objects, so that they can be restored upon the transaction abort.

```
Fast Path

fast_tx_begin()
1. _xbegin();
2. if (GLock) _xabort();

fast_tx_read(addr)
3. read_sig.add(addr);
4. return *addr;

fast_tx_write(addr, val)
5. write_sig.add(addr);
6. *addr = val;

fast_tx_commit()
7.  if (write_locks ∩ write_sig
      || write_locks ∩ read_sig)
8.       _xabort();
9.  if (!is_read_only)
10.     ts = ++timestamp %
                      RING_SIZE;
11.     ring[ts] = write_sig;
12. _xend();
13. post_commit();

fast_post_commit()*
14. write_sig.clear();
15. read_sig.clear();

Partitioned Path

partitioned_tx_begin()*
16. while (Glock) PAUSE();
17. atomic_inc(active_tx);
18. if (Glock) tx_abort();
19. start_time = timestamp;
```

```
Sub-HTM

sub_tx_begin()
20. _xbegin();

sub_tx_read(addr)
21. read_sig.add(addr);
22. return *addr;

sub_tx_write(addr, val)
23. undo_log.add(addr, *addr);
24. write_sig.add(addr);
25. *addr = val;

sub_tx_commit()
26. others_locks = (write_locks - agg_write_sig);
27. if (others_locks ∩ write_sig
        || others_locks ∩ read_sig)
28.      _xabort();
29. write_locks ∪= write_sig;
30. _xend(); post_commit();

sub_post_commit()*
31. in_flight_validation();
32. agg_write_sig ∪= write_sig;
33. write_sig.clear();

in_flight_validation()*
34. ts = timestamp;
35. if (ts != start_time)
36.   for (i=ts; i >= start_time + 1; i--)
37.     if (ring[i % RING_SIZE] ∩ read_sig)
38.       tx_abort();
39.   if (timestamp > start_time + RING_SIZE)
40.     tx_abort(); //Abort at ring rollover
41.   start_time = ts;
```

```
partitioned_tx_commit()*
42. if (is_read_only)
43.    atomic_dec(active_tx);
44.    return;
45. atomic {
46.    ts = atomic_inc(timestamp) % RING_SIZE;
47.    ring[ts] = agg_write_sig;   }
48. atomic {
49.   write_locks = write_locks - agg_write_sig;}
50. agg_write_sig.clear();
51. read_sig.clear();
52. atomic_dec(active_tx);

partitioned_tx_abort()*
53. undo_log.undo();
54. atomic {
55.   write_locks = write_locks - agg_write_sig;}
56. agg_write_sig.clear();
57. read_sig.clear();
58. atomic_dec(active_tx);
59. exp_backoff();
60. restart_tx();

Slow Path

slow_tx_begin()*
61. while (!CAS(GLock, 0, 1));
62. while (active_tx);//Wait for active tx

slow_tx_read(addr)
63. return *addr;

slow_tx_write(addr, val)
64. *addr = val;

slow_tx_commit()*
65. Glock = 0;
```

Fig. 1. PART-HTM's pseudo-code. Procedures marked as * are executed in software.

- *starting-timestamp*, which is the value of the *global-timestamp* (see later) of the system at the time the transaction begins.

Global Metadata. All transactional threads share:

- *global lock*, which is the lock that implements mutual exclusion between the slow path and any other execution.
- *write-locks-signature*, a Bloom filter that represents the write-locks array, where each bit is a single lock. If the bit in position *i* is equal to 1, it means that some sub-HTM transaction committed a new object stored at the address whose hash value is *i*. The write-locks-signature has the same size and hash function as other signatures.
- *global-timestamp*, which is a shared counter incremented whenever a global writing transaction commits.
- *global-ring*, which is a circular buffer that stores committed transactions' write-set-signatures, ordered by their commit timestamp. The global-ring has a fixed size and is used to support the validation against committed transactions, in a similar way as proposed in RingSTM [34].
- *active_tx*, a counter that stores the number of transactions currently running in the partitioned path.

5.2 Fast Path

In the fast path, PART-HTM tries to execute an incoming global transaction as a single HTM transaction. However, in order to synchronize that with transactions executing in any other path, the fast path cannot be a pure HTM transaction and it has to be slightly instrumented according to the following rules.

Begin [lines 1-2]: the global lock is checked right after starting the HTM transaction in order to abort the transaction if that lock is, or will be, acquired by a transaction falling back to the slow path.

Read/write [lines 3-6]: When a memory location is read/written in the fast path, HTM solves any conflict on that location with concurrent transactions executing in the fast path or as sub-HTM transactions. However, HTM will not detect the case when that location is written by a committed sub-HTM transaction whose global transaction is still executing (we refer to such a location as a *non-visible* location hereafter because that location would not have been visible if the global transaction was executed entirely without partitions). For that reason, every memory location is recorded into the read-set-signature (write-set-signature) before it is read from (written to) the shared memory. This information will be used by the HTM transaction before proceeding with the commit phase.

Commit [lines 7-13]: Before committing the transaction, validation is needed to solve two issues. First, the transaction should not overwrite any non-visible memory location because, in this case, the committed sub-HTM transaction that wrote that location has its global transaction not yet committed. Thus, overwriting that location means a potential serialization problem for the global transaction because subsequent sub-HTM transactions may access objects written by the transaction that overwrote the non-visible location. Second, the transaction should not read the value of non-visible location, in order to prevent the exposition of uncommitted (partial) state of a global transaction.

Both issues can be solved by comparing the transaction's read-set-signature and write-set-signature with the global write-locks-signature. This is because (as we detailed in the next section) non-visible locations are locked by sub-HTM transactions before committing using the write-locks-signature. The comparison is done through a bitwise *AND* (i.e., the intersection between the two Bloom filters [Line 7]). If the result is a non-zero Bloom filter, it means that the HTM transaction wrote some location that was locked, thus it should abort [Line 8]. It is worth to note that the abort in the fast path is handled by the HTM implementation itself and there is no need for an explicit abort handler.

The next step in the commit phase is to add the transaction's write-locks-signature to the global-ring if it is not

read-only [Line 9-11] (i.e., at least one write occurred during the execution). This is done by incrementing the global timestamp (no need to use atomic increment because we are still within an HTM context) and using the new value to identify the target ring location. If the transaction is read-only, there is no need to add it to the ring.

The last step after committing a fast path transaction is to clear the local signatures.

5.3 Partitioned Path

5.3.1 Partitioning Phase

When transactions fail in the fast path, partitions are created and the execution falls back to the partitioned path. In Section 3 we listed possible approaches to automate the partitioning process of transactions according to different heuristics. In this paper, partitions are manually made and determined based on static profiler analysis. This analysis splits transactions into multiple basic blocks, and measures the size of accessed shared objects and the duration of each basic block executed sequentially. A partition will be then composed of one or more basic blocks according to their capability of fitting HTM resource limitations. We also manually excluded basic blocks that access no shared objects from being executed in sub-HTM transactions.

When a transaction falls back from the fast path to the partitioned path, it first calls a *begin* subroutine for the global transaction. Then it executes the sub-HTM transactions one after another. Finally, it calls a global *commit* subroutine (or a global *abort* subroutine if it fails in the *in-flight-validation*).

5.3.2 Global Transaction: Begin

Similar to the fast path, the transaction checks the global lock and aborts if it is acquired. In addition, and before checking the global lock, it atomically increments active_tx [lines 17-18]. Since transactions running in the slow path do the opposite (acquire the global lock and then check active_tx [lines 61-62]), this guarantees mutual exclusion between transactions in the partitioned path and those in the slow path. Finally, the value of the global-timestamp is stored as the start_time of the transaction [Line 19].

5.3.3 Sub-HTM Transaction: Begin

No instrumentation is needed when a sub-HTM transaction starts.

5.3.4 Sub-HTM Transaction: Read/Write

Locations are read directly from the shared memory as in the fast path and recorded into read-set-signature even if that location may have been written before by the same global transaction. In fact, in case a previous sub-HTM transaction, belonging to the same global transaction, committed a new value of that location, this new value is already stored into the shared memory since HTM uses the write in-place technique. If the read object has been already written during the current HTM transaction, then the HTM implementation guarantees the access to the latest written value.

For writes, the only difference between fast path and partitioned path is that in the latter the global transaction could abort in the future, although the current sub-HTM transaction is committed. If this happens, the previous values of written locations should be replaced into the shared memory in the global transaction abort handler. For this reason, before to finalize the write operation, the old value of the location is logged into the local undo-log [Line 23].

5.3.5 Sub-HTM Transaction: Commit/Abort

The first step in the commit phase of a sub-HTM transaction is to validate the read-set-signature and the write-set-signature similar to the fast path [lines 26-27]. The only difference is that locks acquired in previous committed sub-HTM transactions have to be excluded from the global write-locks-signature before doing the validation. This is because, due to the nature of the Bloom filters, a lock is just a bit and has no ownership information. Thus, a transaction is not able to distinguish between its own locks (i.e., acquired by previous sub-HTM transactions of the same global transaction), and others' locks. We solve this issue by a simple bitwise operation between the transaction's aggregate write-set-signature and the global write-locks-signature [line 26], which allows each sub-HTM transaction to know whether the locked location is owned by its global transaction or not.

After validation and before committing the sub-HTM transaction, the new values of written locations should be protected against accesses from other transactions. This is done by updating the global write-locks-signature [Line 29]. It is worth to note that, every update to a shared metadata, such as the write-locks-signature, causes the abort of all HTM transactions that read the specific cache-line where the metadata is located, even if they updated or tested different bits (false conflict). For this reason, in order to minimize false conflicts, the write-locks-signature is updated in the commit phase of the sub-HTM transaction, rather than after each write operation.

In practice, the task of notifying that a new location has been just committed, but is non-visible, is very efficient: the write-locks-signature is updated to be the result of the bitwise *OR* between transaction's aggregate write-set-signature and the write-locks-signature itself.

The last step in the commit phase of sub-HTM transactions, after calling *_xend*, is to call the *in-flight-validation* routine (detailed below) and to update the aggregated write-set-signature [line 32].

Aborted sub-HTM transactions are handled based on the abort reason. If the sub-HTM transaction aborts due to a conflict on the global write-lock, the software framework propagates the abort to the enclosing global transaction. Otherwise, sub-HTM transaction retries for a limited number of times before aborting the enclosing transaction (for simplicity, this is not shown in the pseudo code).

5.3.6 Global Transaction: In-flight-validation

This validation is done by the software framework after the commit of every sub-HTM transaction in case some global transaction (including those executing in the fast path) committed in the meanwhile, whereas transactions in the fast path do not need to call it. The in-flight-validation is needed for ensuring that the memory snapshot observed by the global transaction is still consistent after the commit of a sub-HTM transaction. The following example shows the need of the in-flight-validation to preserve correctness.

Assuming the scenario with two global transactions T^x and T^y, both having two sub-HTM transactions each. Let us assume that T_1^x reads the value of object o and commits. Let us also assume that o is not locked at this time. After that, T_2^y is scheduled. It overwrites and locks o, invalidating T^x. T^x is able to detect this conflict during the validation done before T_2^x commits, but let us assume that the commit of T^y is scheduled before T_2^x's commit (in fact, T_2^y is the last sub-HTM transaction of T^y). As we will show later in the commit procedure (Section 5.3.7), all transaction's locks are cleared from the write-locks-signature when the global transaction commits. This means that, the intersection between T_2^x's read-set-signature and the write-locks-signature does not report any conflict on o, therefore T_2^x can commit even if T^x's execution is not consistent anymore.

The in-flight-validation solves this problem by comparing the transaction's read-set-signature against the aggregate write-set-signature (or write-set-signature in case of fast path) of all concurrent and committed global transactions [Line 34-38]. As we will show later, retrieving committed transactions is easy because, upon commit, each of them add itself into an entry in the global-ring, which is also associated with its commit timestamp. The selection of concurrent transactions through the global-ring is straightforward because they have a commit timestamp that is higher than the starting-timestamp of the transaction that is running the in-flight-validation [Line 36]. If a transaction detects an overflow in the ring while validating its entries, it aborts [Lines 39-40]. After a successful in-flight-validation, the transaction's starting-timestamp is advanced to the value of the global-timestamp at the time of the validation [Line 41]. This way, subsequent in-flight-validations do not pay again the cost of validating the global transaction against the same, already committed, transactions.

It is worth to notice that the in-flight-validation is done after each sub-HTM transaction mainly for performance reason (except for PART-HTM-O where it is mandatory). In fact, in order to ensure serializable executions, the in-flight-validation could be done just one time after the commit of the last sub-HTM transaction and before commit. We decided to perform it after each sub-HTM transaction because detecting invalidated objects early in the execution avoids unnecessary computation, saves HTM resources, and makes the software framework's execution always consistent.

5.3.7 Global Transaction: Commit/Abort

This section details the commit/abort procedure of a global transaction running in the partitioned path.

The commit phase of the partitioned path is different from the one of the fast path transactions in four points. First, incrementing the global-timestamp and adding the transaction's aggregate write-set-signature to the global-ring must be atomic [lines 45-47] since this commit phase is not part of any HTM execution. The way we guarantee in our code that those two lines together are atomic is somehow complicated (to be optimized), so we did not include it in Figure 1 and we only marked them as an *atomic* block. Second, there is no need to re-validate the transaction because it has been already validated by both the the last sub-HTM transaction and the in-flight-validation called after it. Third, active_tx has to be atomically decremented [line 52]. Finally, the transaction's write locks should be released [Line 48-49]. Recall that in each sub-HTM transaction, write operations are directly applied to the shared memory and the written locations are protected by modifying the global write-locks-signature. In order to release write locks, a transaction executes an atomic bitwise *XOR* between the transaction's aggregate write-set-signature and the global write-locks-signature.

The abort of a global transaction in the partitioned path due to failing in the in-flight-validation requires to restore the values of memory locations written by its committed sub-HTM transactions. This operation is done traversing the transaction's undo-log [Line 53]. After that, the transaction's write-locks are released from the global write-locks-signature [Line 54-55]. Finally, active_tx is atomically decremented and a retry in the partitioned path is invoked after an exponential back-off time [Line 58-60]. The transaction is retried 5 times before falling back to the slow path.

5.4 Slow Path

A transaction that falls back to the slow path acquires the global lock and waits until all active transactions (i.e., those running in the partitioned path) complete [Lines 61-62]. This guarantees that no transaction, running in any path, is concurrent with a transaction executing in the slow path, thus the slow path can execute without instrumentation, and release the global lock at the end.

5.5 Ensuring Opacity

PART-HTM cannot guarantee opacity. This is because the consistency of the execution history is not verified encounter time but only before committing a sub-HTM transaction, as well as during the in-flight-validation. Roughly, the former validation checks if objects accessed during the current sub-HTM transaction were non-visible; the latter verifies that the memory snapshot observed by the global transaction is still consistent against all committed transactions. These validations do not prevent the transaction from performing a memory read if the object is non-visible or the global transaction's history is not valid anymore, whereas they "only" prevent the transaction from finally committing. Under specific programming patterns, such as invoking `jump` instructions to addresses computed at run-time as a result of previous read operations, such behavior may produce erroneous executions as those described in [16].

Two extensions are needed for making PART-HTM opaque: *1)* once a locked object is accessed, the global transaction should be immediately aborted; *2)* no memory access should be performed if the snapshot observed by the sub-HTM transaction, as well as the global transaction, is not valid. Figure 2 shows the pseudo-code of PART-HTM-O's core operations. For simplicity in the representation, we do not report the additional indirection level required by PART-HTM-O. In this sub-section, line numbers refer to Figure 2.

Encounter time lock detection. In principle, checking if an object is locked is straightforward because we could analyze the write-locks-signature just before performing the actual read. Unfortunately, the write-locks-signature is a global metadata, which is updated anytime a sub-HTM transaction commits any object. As a result, reading it

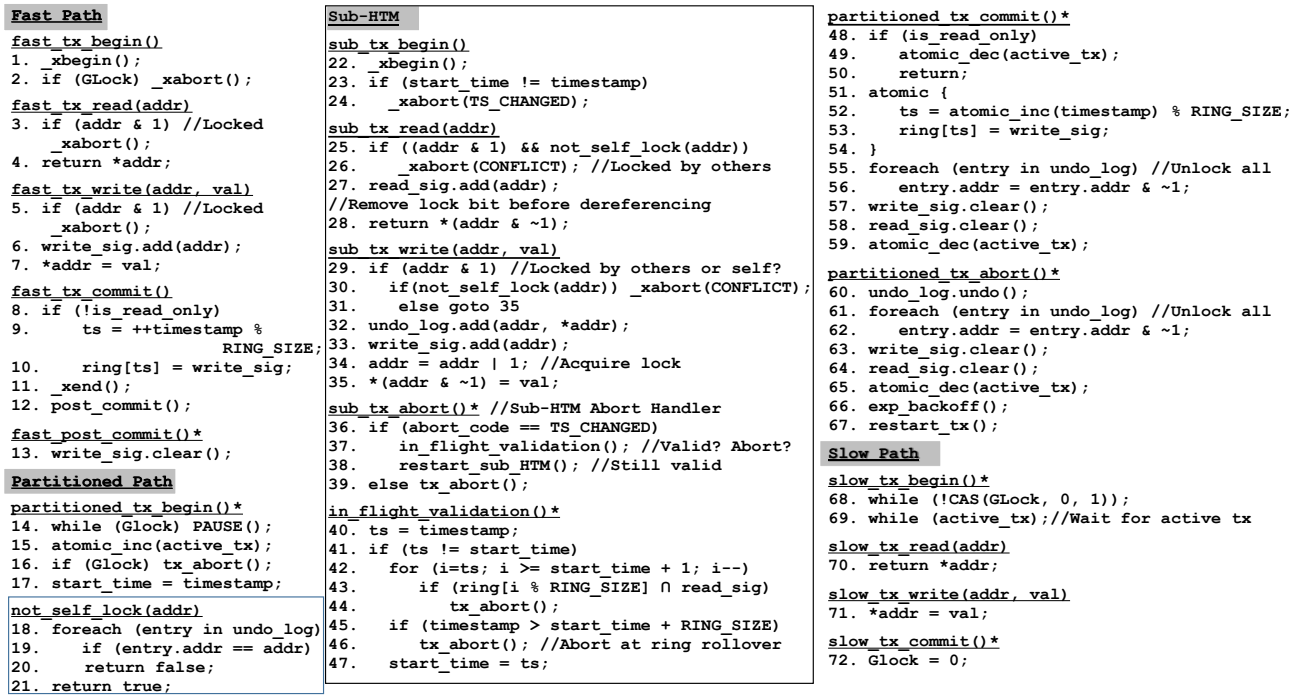


Fig. 2. PART-HTM-O's pseudo-code. Procedures marked as * are executed in software.

during an HTM transaction means being aborted anytime another sub-HTM transaction updates it, even if the accessed object is not the same and their entries in the write-locks-signature are different. Another solution is creating an external lock-table for storing locks, as done in [35]. However, this solution has the same drawback as the write-locks-signature because they both rely on a hash-function.

PART-HTM-O solves this problem by introducing the *address-embedded* locks, which is a technique never used in the HTM context, that embeds the information about the lock acquisition into the memory address of the shared object itself. With it, we assign the address of shared objects in a way such that they are always memory-aligned. If so, we set the least significant bit (meaningless because we know the object is always memory-aligned) to 1 (locked) or 0 (unlocked). With address-embedded locks we eliminate any false conflicts due to shared metadata. In practice, when an object is accessed inside a sub-HTM transaction, the least significant bit of its address is checked and if a lock is found, the transaction is explicitly aborted [Line 3, 5, 25, 29].

The deployment of address-embedded locks introduces two inherent downsides. First, it requires a memory location for storing the actual memory-aligned address that points to the shared object. Although it does not generate a significant performance overhead, the implementation of this indirect addressing layer needs a modification of the memory allocation of the application. As an example, if the shared object is a primitive type (e.g., integer), we need to manage the value of its pointer. This indirect addressing layer must be added. Note that, memory accesses occur only inside HTM transactions, therefore any request to a possible unmapped location due to the new additional level of indirection would cause the transaction to abort. The second downside is that concurrency control metadata become accessible to the application programmer given the exposed lock bit, thus allowing their manipulation as a consequence of a malicious behavior. Besides that, it is important to note that stealing bits is a widely used technique in coding operating systems.

We exploit the memory alignment of addresses, which allows the last bit to be manipulated arbitrarily without corrupting the address itself. If the application accesses a scalar X with address $addr(X)$, in order to change the last bit of $addr(X)$ we need an indirect reference to X ($wrap(X)$). This way, the value of $wrap(X)$ is $addr(X)$ but with the last bit ready to be used for locking. Therefore, the deployment of address-embedded-locks requires modifying the application (although simple) to use $wrap(X)$ rather than X. If X is a pointer, then no wrapper is needed and the lock is embedded in X itself. For instance, in a linked-list, nodes store pointers to other nodes (`Node* next`), thus we already have a container for modifying the addresses directly to embed the lock. Modifications are rather needed if there is a scalar (e.g., `int size`). If so, we wrap it with a pointer (`int* sizep = &size`) so it is only accessed indirectly via the wrapper (`*sizep`).

Consistent reads. Opacity requires that any memory access is performed only if it does not violate the consistency of the snapshot observed so far by the transaction. PART-HTM does no provide this because, with its scheme, it is not possible to detect if an object read in a previous sub-HTM transaction becomes not valid while executing a subsequent sub-HTM transaction. As a consequence, a read operation can access an object committed by a transaction whose history is not consistent with the global transaction. PART-HTM allows this anomaly and aborts the global transaction once the sub-HTM transaction is already committed exploiting the in-flight-validation. A trivial solution for ensuring consistent reads is to validate all objects accessed before

reading a new shared object, but this solution is unfeasible because it would generate several false conflicts and require maintaining all read objects, thus consuming resources.

PART-HTM-O adopts a strategy that overcomes the above limitations. At the beginning of each sub-HTM transaction, the global-timestamp is compared against the transaction's start_time [Line 23-24]. The goal is to abort the sub-HTM transaction anytime a new global transaction commits. Once this happen, the in-flight-validation is called and, in case it succeeds, just the sub-HTM transaction is restarted [Line 36-38], otherwise the whole global transaction is aborted [Line 39]. Reading the global-timestamp allows the sub-HTM transaction to avoid any validation while executing because, once a new global transaction commits and it is added to the global-ring, the global-timestamp is changed and this forces the sub-HTM transaction to abort due to the hardware conflict detection. The combination of both the above extensions make the sub-HTM's validation before commit useless in PART-HTM-O because its goal is already provided earlier in the execution.

6 CORRECTNESS

6.1 PART-HTM: serializability

Serializability requires that in any concurrent execution, committed transactions appear to execute sequentially. Unlike opacity, serializability does not provide any guarantees on aborted and live (i.e., not yet committed) transactions. We show, case by case, that the aforementioned property holds for every possible concurrent execution.

The first trivial case is when all transactions are running in the fast path. In that case, since every transaction in the fast path runs as a single non-partitioned HTM transaction, serializability is inherited from the HTM guarantees: if two transactions are conflicting (i.e., they access the same location and at least one of them is writing, HTM aborts one of them). Note that, due to the conflict detection implemented by HTM, the provided safety guarantee is much stronger than Serializability.

The second case, which is when some transactions run in the partitioned path, is the core of our proof. Adding transactions running in the partitioned path to a concurrent execution moves part of the conflict management that preserves serializability to the software framework. In the following, we show that the software framework handles the three possible types of conflicts that can invalidate the transaction execution: *write-read*, *read-write*, and *write-write*.

Let T_r^x and T_w^y be two sub-HTM transactions with conflicting accesses. Specifically, T_r^x performs a read operation and T_w^y a write operation on the same memory location. T_r^x belongs to the global transaction T^x, whereas T_w^y belongs to the global transaction T^y. Clearly, T_r^x and T_w^y are both in the partitioned path. The case when one transaction is in the fast path and the other is in the partitioned path can be handled (with slight adaptation) as a special case where the former is considered to be partitioned into only one sub-HTM transaction.

A *write-read* conflict happens when T_w^y writes an object o read by T_r^x. If the conflicting operations of T_r^x and T_w^y happen while both the transactions are running, the HTM conflict detection will abort one of them. Otherwise, it means that the write operation of T_w^y on o is executed after the commit of T_r^x. If so, T^x will detect this invalidation through the validation performed at the end of the sub-HTM transaction that follows T_r^x. If there is no sub-HTM transaction after T_r^x, it means that T^x commits before T^y thus it serializes itself before T^y. Any future serialization issue that may occur between T^x and T^y will be detected by T^y during its validation phases. If T^y commits after T_r^x but while T^x is still executing, T^y's aggregate write-set-signature will be attached to the global-ring. In this case, the in-flight-validation performed by T^x before committing will detect the conflict and abort T^x.

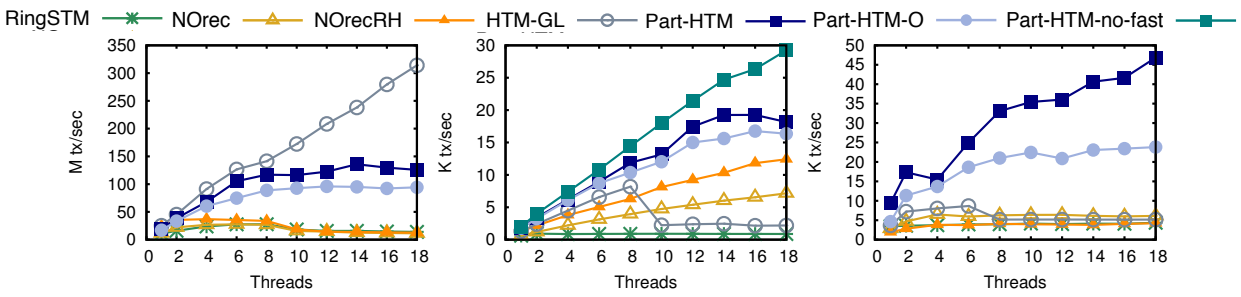
A *read-write* conflict happens when T_r^x reads an object o that T_w^y already wrote. As before, if the conflict is materialized while both are running, the HTM conflict detection handles it. If T_r^x reads after the commit of T_w^y, but T^y is still executing, then T_r^x will be aborted before it attempts to commit thanks to the HTM-pre-commit-validation, which detects a lock taken on o by T^y. If T^y commits just after T_w^y, this is not a problem because it means that T_r^x accessed to the last committed version of o.

The *write-write* conflicts are detected either by exploiting the HTM conflict detection if both the write operations happen during HTM executions, or before commit all HTM transactions perform the HTM-pre-commit-validation, which detects a taken lock by intersecting the transaction's aggregate write-set-signature (or write-set-signature if the transaction is committed in the fast path) with the global write-locks-signature. Handling write-write conflicts is important because, if ignored (as many serializable concurrency controls do), a read operation on an object already written inside the same transaction could return a different value.

The last case is when some transactions, in addition to those running in fast and partitioned paths, are running in the slow path. Those transactions cannot break serializability because they run in isolation (thanks to the global lock and the *active_tx* counter).

6.2 PART-HTM-O: opacity

Considering that PART-HTM reads and writes only using HTM transactions, there is the possibility that doomed transactions (those that will be aborted eventually) could observe inconsistent states while they are running as HTM transactions. In fact, locks are checked only before committing the HTM transaction, thus a hardware read operation always returns the value written in the shared memory, even if locked. The return value of those inconsistent reads could be used by the next operations of the transaction, generating non-predictable execution (e.g., infinite loops or memory exception). This behavior does not break serializability because aborted transactions are not taken into account by the correctness criterion. However, for in-memory processing, like TM, avoiding such scenarios is desirable, as defined in [14]. As a partial fallback plan, the HTM provides a sandboxing feature, which eventually aborts misbehaving HTM transactions that generate infinite loops or erroneous computations. However, without guaranteeing Opacity, the protocol cannot prevent corner case situations where a sub-HTM transaction is committed skipping the pre-HTM validation [16].



(a) N=M=10. (b) N=100K, M=100. (c) N=M=100.

Fig. 3. Throughput using N-Reads M-Writes benchmark.

PART-HTM-O addresses this problem by avoiding any memory operation in case *A)* the snapshot observed by the transaction is not consistent anymore, and *B)* if the memory access itself would break the consistency of the transaction.

(A) We ensure the point *A* by monitoring the global-timestamp as the first operation of a sub-HTM transaction. This way, if some object is committed just after the in-flight-validation performed before the activation of a sub-HTM transaction or if some global transaction commits while a sub-HTM transaction is executing, then the global-timestamp is changed and any HTM transaction is aborted and forced to perform a validation of all accessed objects.

(B) If a sub-HTM transaction accesses an object already locked (if the object becomes locked after the access, then the HTM will detect the conflict), then before to finalized the access the HTM transaction is explicitly aborted by leveraging the address-embedded write locks.

7 EVALUATION

PART-HTM has been implemented in C++. We used four benchmarks: *N-reads M-write*, a configurable application provided by RSTM [36]; the linked-list data structure; STAMP [17] (v0.9.10), the popular suite of applications used for evaluating STM- and HTM-related concurrency controls; and EigenBench [18], a customizable TM benchmark.

As competitors, we included two state-of-the-art STM protocols, RingSTM [34] and NOrec [3]; one Hybrid TM, Reduced Hardware NOrec (NOrecRH) [37]; and one HTM with the GL-software path as fallback (HTM-GL). Also, the ring used by RingSTM and PART-HTM have the same size and signature. NOrecRH and HTM-GL retry a transaction 5 times as HTM before falling back to the software path. All are implemented such that they do not suffer from the lemming effect [38]. As suggested in [38], a transaction does not retry until the global lock is not released. In this evaluation study we mainly used the Intel Haswell Core i7-4770 processor (4-core) hosted in a single socket machine, but for some micro-benchmark we used the Intel Xeon E7-8880v3 (18-core HTM enabled) processor on a single socket machine. Hyper-threading is enabled in the former CPU and disabled in the latter. Both the CPUs have the following cache sizes per core: L1=32 kB, L2=256 kB.

We used GCC 4.8.2. Transactional barriers (read and write) are inserted manually in the used applications; no GCC transactional extensions have been used. All data points are the average of 5 repeated execution. To show the viability of using the address-embedded write locks, we also included the performance of PART-HTM-O in most of the used applications.

As a general comment of our evaluation, PART-HTM represents the best solution in almost all the tested workloads, except for those where pure HTM transactions always commit. In these cases, outperforming HTM is impossible without additional hardware support, but our approach does not pay a significant performance penalty thanks to the fast path HTM transactions.

7.1 Micro benchmarks

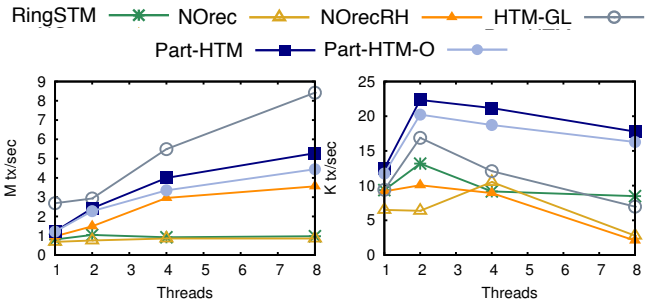
N-Reads M-Writes. In this benchmark each transaction reads N elements from one array and writes M to another. Both the arrays have a fixed size of 100k elements. This benchmark is configured to access disjoint elements (i.e., no contention) in all the experiments because we aim at evaluating our approach in scenarios where the aborts due to non-false conflicts of HTM transactions are minimized.

Figure 3(a) shows the results of reading and writing 10 elements. Here, few transactions are aborted for resource failure, thus almost all commit as HTM. As expected, HTM-GL has the best throughput, followed by PART-HTM. This scenario is not the best case for PART-HTM but still, thanks to the lightweight instrumentation of fast path HTM transactions, it shows a limited slowdown up to 8 threads over HTM-GL, whereas the best competitor (NOrecRH) is much slower than PART-HTM. When number of concurrent threads increases, PART-HTM suffers from false conflicts on metadata, which reduces its scalability. However, it remains the closest competitor to HTM-GL. Recall that HTM-GL does not suffer from this meta-data overhead, and thus it keeps scaling even after 8 threads. PART-HTM-O is slightly slower than its non-opaque version due to the additional indirection level to implement the address-embedded write locks, which in this case represents just an overhead because application workload is already disjoint.

Figure 3(b) shows an experiment where 100k elements are read from one array and 100 of them are written to the destination. This scenario reproduces large transactions in a read-dominated workload. Here, HTM-GL still performs good up to 8 threads, because Intels HTM implementation can go beyond the L1 cache capacity for read operations. However, when number of concurrent transactions is more than 8, most of HTM transactions reach their capacity limits and fall back to the GL-software path, and the performance drops. For this reason, the benefit of partitioning and committing into sub-HTM transactions is evident and PART-HTM performs significantly better than HTM-GL. Both STM

protocols and NOrecRH suffer from excessive instrumentation cost due to having several operations per transaction. PART-HTM gains around 20% over PART-HTM-O. In Figure 3(b), we also added a new version of PART-HTM, which avoids the fast path execution and starts immediately by falling back to the partitioned path (we call it *Part-HTM-no-fast* in the figure). This version shows how, interestingly, the absence of fast path trials in such scenarios improves PART-HTMs scalability.

In Figure 3(c), each transaction reads one element from the source array, then it does some floating point operations, and then it writes its new value to the destination array in the same entry as the source array. This sequence is repeated 100 times on different objects by a single transaction. This way we emulate transactions that could be committed as HTM in terms of size but, for time limitation, are likely aborted (e.g., by a timer interrupt). Once partitioned, each sub-HTM transaction executes 25 of those iterations. When running in the partitioned path, non-transactional code is still executed transactionally by sub-HTM transactions. In this scenario, PART-HTM shows a significant speed-up compared to other competitors. HTM-GL executes all transactions using global locking. NOrecRH and NOrec perform similar but NOrecRH is slightly worst as it executes the transaction in hardware first. PART-HTM-O follows the same trend line as PART-HTM but with a performance gap due to the higher number of aborts as a consequence of the ring timestamp subscription.



(a) 1K, 50% writes. (b) 10K, 50% writes.

Fig. 4. Throughput using Linked-List.

Linked-List. In this benchmark, we do operations on a linked list. We change its size, and the percentage of write operations (insert and remove) against read operations (contains). Linked list transactions traverse the list from the beginning until the requested element. This increases the contention between transactions. Write operations are balanced so that the size of the list is stable.

Figure 4(a) shows the results of a 1K elements linked list using 50% of write operations. Due to the small size of the linked list, thus capacity aborts are rare, and given that concurrent transactions are only eight, thus conflict aborts are small, almost all transactions commit in hardware and HTM-GL has the best throughput. However, following the same trend as Figure 3(a), PART-HTM places its performance closer to HTM-GL.

Figure 4(b) shows a larger linked list with 10K elements. Here, most of the transactions fail in hardware for resource failures. As for the case in Figure 3(c), PART-HTM's throughput is the best as sub-HTM transactions pay a limited instrumentation cost and fast execution in hardware. PART-HTM gains up to 74% over HTM-GL. It is worth noting that all algorithms do not scale like Figure 4(a) because transactions become longer and the cost of aborting them increases.

7.2 STAMP benchmark

Figure 5 shows the results of STAMP applications. STAMP applications' transactions likely do not fail in HTM for capacity except for Labyrinth and Yada. However, most of the effort in the design of PART-HTM is focused on reducing overheads. In fact, STAMP applications's performance confirms the effectiveness of PART-HTM's design because it is the best in almost all cases, and the closest to the best competitors when HTM is the best. All data points report the achieved speed-up with respect to the sequential execution of the application.

Kmeans (Figure 5(b) and 5(a)), Vacation low-contention (Figure 5(f)), SCAA2 (Figure 5(c)), Intruder (Figure 5(e)), and Genome (Figure 5(i)) are application where HTM transactions do not fail for resource limitations, but they are mostly short and conflict due to real conflicts. In all those application, HTM-GL is the best but PART-HTM is always the closest competitor. Note that, SCAA2 shows the instrumentation overhead of PART-HTM while executing with only one thread.

On the other hand, applications like Labyrinth (Figure 5(d) and Table 1) and Yada (Figure 5(h)) are suited more for STM protocols than HTM. That is because more than half of the generated transactions in Labyrinth are large and long (thus HTM cannot be efficiently exploited), but they also rarely conflict with each other. As a result, NOrecRH and NOrec perform worse than, but closer to, PART-HTM. HTM-GL is the worst. We also observe a 10% of gap between PART-HTM and PART-HTM-O. This gap is basically the cost of performing the in-flight-validation once a global transaction commits and a sub-HTM transaction is executing. Labyrinth is not characterized by short transactions, thus updates of the global-timestamp are not very frequent, and this helps to reduce the gap between PART-HTM-O and PART-HTM.

In Figure 5(f) we observe the impact of hyper-threading (thus reduce number of cache-lines available per executing thread). Moving from 4 to 8 threads, the performance of HTM-GL drops due to the increased capacity aborts. Figure 5(h) shows the results of Yada. This application has transactions that are long and large, generating a reasonable high contention level. Thus it represents a favorable workload for PART-HTM and the plot confirms it. The big drop in performance of all competitors is due to the very high contention (in fact they are all slower than sequential).

We do not report the results using the Bayes application given its non-deterministic execution.

7.3 EigenBench

EigenBench is a comprehensive benchmark, which can generate transactions with different properties. We used it to build a workload with 50% long and 50% small transactions, thus the latter will likely fit in HTM. A small transaction does 50 read and 5 write operations to an array of 1024

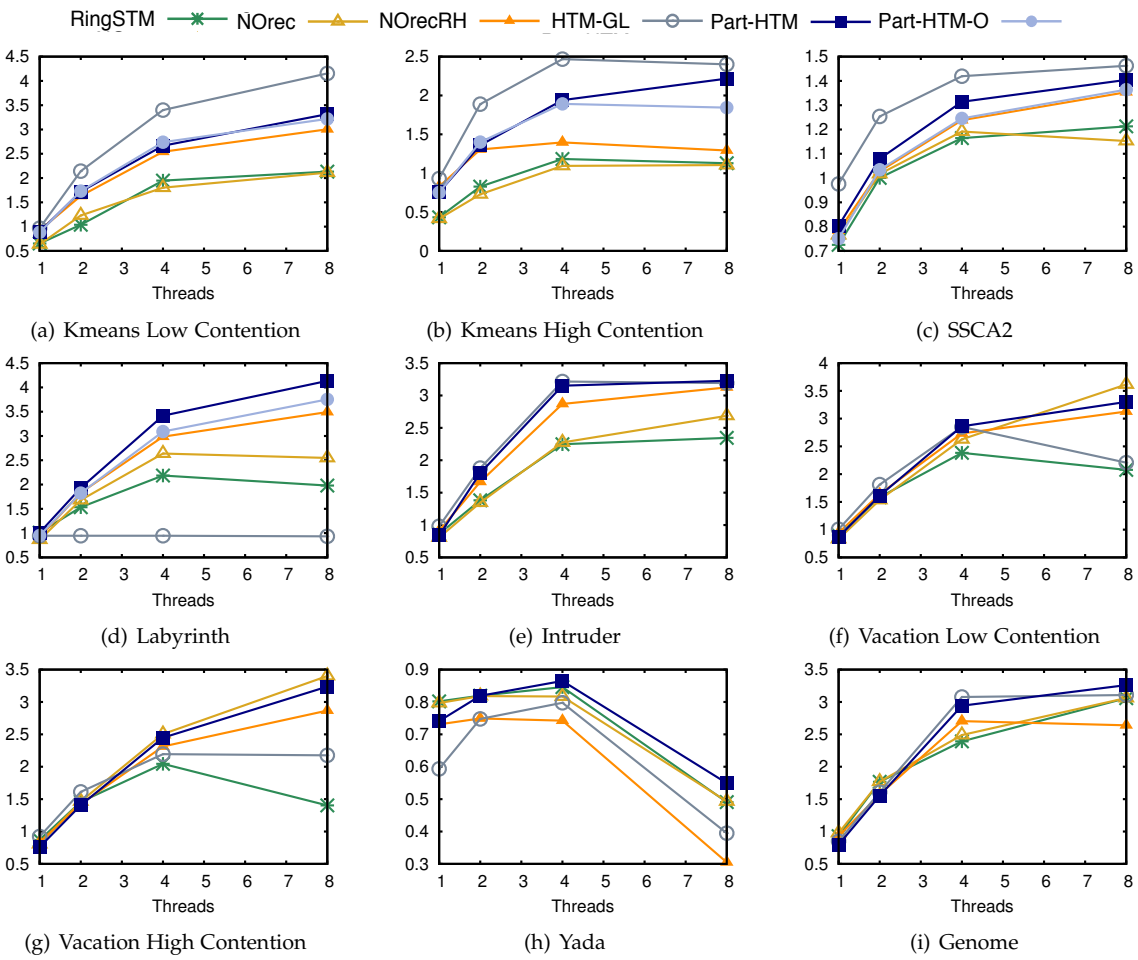
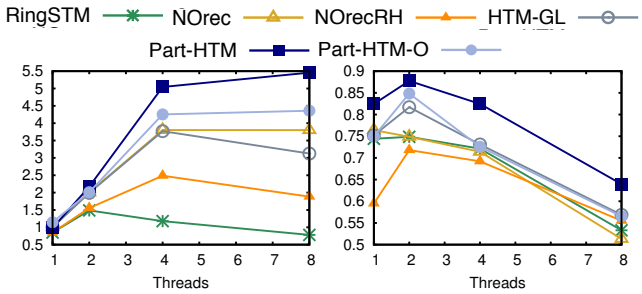


Fig. 5. Speed-up over sequential (non-transactional) execution using applications of STAMP Benchmark.

words while long transactions add non-transactional computation in between operations. Read and written elements are selected in a random way while providing disjoint accesses among threads. Figure 6(a) plots the results. PART-HTM has the best performance as it executes the long transactions efficiently and it is able to execute non-transactional computation outside sub-HTM transactions. PART-HTM-O follows with average overhead of 15%. Other competitors suffered with the long transactions.



(a) 50% Long transactions & 50% Short transactions. (b) High Contention

Fig. 6. Speed-up over sequential (non-transactional) execution using EigenBench.

Figure 6(b) shows the results of EigenBench under high contention scenario. EigenBench is configured to access the shared `hot-array` of size 32K. Each transaction performs 10K reads and 100 writes with 50% repeated accesses. With this workload, transactions execute many operations with high chance of encountering contention. Here, transactions under HTM-GL abort each other often and then they fall back to the the global lock path. On the other hand, PART-HTM has the best performance because small sub-HTM transactions have a higher chance to commit and after that, they acquire locks on written objects, which avoid other HTM transactions to progress if the lock is acquired.

8 Conclusion

In this paper we presented PART-HTM, a hybrid TM, which aims at committing those HTM transactions that cannot be fully executed as HTM due to space and/or time limitation. The core idea of PART-HTM is splitting hardware transactions into multiple sub-transactions and run them in hardware with a minimal instrumentation.

PART-HTM's performance is appealing. In our evaluation it is the best in almost all the tested workloads, and it is close to HTM's performance where HTM performs best.

Acknowledgments

Authors thank anonymous IEEE TPDS reviewers for the very insightful comments. This work is partially supported

by Air Force Office of Scientific Research (AFOSR) under grant FA9550-14-1-0187.

REFERENCES

[1] T. Harris, J. Larus, and R. Rajwar, "Transactional memory, 2nd edition," *Synthesis Lectures on Computer Architecture*, vol. 5, no. 1, 2010.

[2] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *ISCA*, 1993, pp. 289–300.

[3] L. Dalessandro, M. F. Spear, and M. L. Scott, "Norec: Streamlining stm by abolishing ownership records," in *PPoPP*, 2010, pp. 67–78.

[4] D. Christie, J.-W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, and E. Rivière, "Evaluation of amd's advanced synchronization facility within a complete transactional memory stack," in *EuroSys*, 2010, pp. 27–40.

[5] J. Reinders, "Transactional synchronization in haswell," *Intel Software Network.*, 2012.

[6] H. W. Cain, M. M. Michael, B. Frey, C. May, D. Williams, and H. Le, "Robust architectural support for transactional memory in the power architecture," in *ISCA*, 2013, pp. 225–236.

[7] L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear, "Hybrid NOrec: A case study in the effectiveness of best effort hardware transactional memory," in *ASPLOS*, 2011.

[8] A. Matveev and N. Shavit, "Reduced hardware transactions: A new approach to hybrid transactional memory," in *SPAA*, 2013.

[9] I. Calciu, J. Gottschlich, T. Shpeisman, G. Pokam, and M. Herlihy, "Invyswell: A hybrid transactional memory for haswell's restricted transactional memory," in *PACT*, 2014, pp. 187–200.

[10] N. Diegues and P. Romano, "Self-tuning intel transactional synchronization extensions," in *ICAC*, 2014, pp. 209–219.

[11] C. Ananian, K. Asanovic, B. Kuszmaul, C. Leiserson, and S. Lie, "Unbounded transactional memory," in *HPCA*, 2005, pp. 316–327.

[12] M. Mohamedin, A. Hassan, R. Palmieri, and B. Ravindran, "Brief announcement: Managing Resource Limitation of Best-Effort HTM," in *SPAA '15*.

[13] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[14] R. Guerraoui and M. Kapalka, "On the correctness of transactional memory," in *PPoPP '08*, pp. 175–184.

[15] I. Calciu, T. Shpeisman, G. Pokam, and M. Herlihy, "Improved single global lock fallback for best-effort hardware transactional memory," in *TRANSACT '14*.

[16] D. Dice, T. L. Harris, A. Kogan, Y. Lev, and M. Moir, "Pitfalls of lazy subscription," in *WTTM*, 2014.

[17] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multi-processing," in *IISWC'08*.

[18] S. Hong, T. Oguntebi, J. Casper, N. Bronson, C. Kozyrakis, and K. Olukotun, "Eigenbench: A simple exploration tool for orthogonal tm characteristics," in *IISWC*, 2010, pp. 1–11.

[19] T. Riegel, P. Marlier, M. Nowack, P. Felber, and C. Fetzer, "Optimizing hybrid transactional memory: The importance of nonspeculative operations," in *SPAA*, 2011, pp. 53–64.

[20] Y. Afek, A. Matveev, and N. Shavit, "Reduced hardware lock elision," in *WTTM*, 2014.

[21] Y. Afek, A. Levy, and A. Morrison, "Software-improved hardware lock elision," in *PODC*, 2014, pp. 212–221.

[22] D. Alistarh, P. Eugster, M. Herlihy, A. Matveev, and N. Shavit, "Stacktrack: An automated transactional approach to concurrent memory reclamation," in *EuroSys*, 2014, pp. 25:1–25:14.

[23] Y. Lev and J.-W. Maessen, "Split hardware transactions: True nesting of transactions using best-effort hardware transactional memory," in *PPoPP*, 2008, pp. 197–206.

[24] L. Xiang and M. L. Scott, "Software partitioning of hardware transactions," in *PPoPP*, 2015, pp. 76–86.

[25] ——, "Conflict reduction in hardware transactions using advisory locks," in *SPAA*, 2015, pp. 234–243.

[26] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen, "Fast in-memory transaction processing using RDMA and HTM," in *SOSP*, 2015, pp. 87–104.

[27] D. Shasha, F. Llirbat, E. Simon, and P. Valduriez, "Transaction chopping: Algorithms and performance studies," *ACM Trans. Database Syst.*, vol. 20, no. 3, pp. 325–363, Sep. 1995.

[28] Y. Liu, X. Zhang, Y. Wang, D. Qian, Y. Chen, and J. Wu, *Partition-Based Hardware Transactional Memory for Many-Core Processors*, 2013.

[29] D. Niles, R. Palmieri, and B. Ravindran, "Exploiting parallelism of distributed nested transactions," in *SYSTOR*, 2016, pp. 10:1–10:11.

[30] J. Barreto, A. Dragojevic, P. Ferreira, R. Filipe, and R. Guerraoui, "Unifying thread-level speculation and transactional memory," in *ACM/IFIP/USENIX Middleware*, 2012, pp. 187–207.

[31] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, 1970.

[32] M. Abadi, T. Harris, and M. Mehrara, "Transactional memory with strong atomicity using off-the-shelf memory protection hardware," in *ACM SIGPLAN PPoPP*, D. A. Reed and V. Sarkar, Eds., 2009, pp. 185–196.

[33] T. Shpeisman, V. Menon, A. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and B. Saha, "Enforcing isolation and ordering in STM," in *PLDI*, 2007, pp. 78–88.

[34] M. F. Spear, M. M. Michael, and C. von Praun, "RingSTM: Scalable transactions with a single atomic instruction," in *SPAA*, 2008.

[35] D. Dice, A. Kogan, and Y. Lev, "Refined transactional lock elision," in *PPoPP*, 2016, pp. 19:1–19:12.

[36] V. Marathe, M. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. Scherer III, and M. Scott, "Lowering the overhead of nonblocking software transactional memory," in *TRANSACT*, 2006.

[37] A. Matveev and N. Shavit, "Reduced hardware norec: An opaque obstruction-free and privatizing hytm," *TRANSACT*, 2014.

[38] D. Dice, M. Herlihy, D. Lea, Y. Lev, V. Luchangco, W. Mesard, M. Moir, K. Moore, and D. Nussbaum, "Applications of the adaptive transactional memory test platform," in *TRANSACT*, 2008.

Mohamed Mohamedin is an Assistant Professor at Faculty of Engineering, Alexandria University. He received his BSc degree in Computer Engineering and his MS and PhD degree in Computer Engineering at Virginia Tech. His research interests include transactional memory, parallel programming, fault tolerance of transactional systems, and distributed computing.

Roberto Palmieri received the BSc in computer engineering, MSc and PhD degree in computer science at Sapienza, University of Rome, Italy. He is a Research Assistant Professor in the ECE Department at Virginia Tech. His research interests include exploring concurrency control protocols for multicore architectures, cluster and geographically distributed systems, with high programmability, scalability, and dependability.

Ahmed Hassan worked as a Postdoctoral Research Associate in the ECE Department at Virginia Tech. He received his BSc degree in computer science and his MSc degree in computer engineering at Alexandria University, Egypt. He received his PhD degree in Computer Engineering at Virginia Tech. His research interests include transactional memory, concurrent data structures, and distributed computing. Recently, he joined Alexandria University in Egypt as Assistant Professor.

Binoy Ravindran is a Professor of Electrical and Computer Engineering at Virginia Tech, where he leads the Systems Software Research Group, which conducts research on operating systems, virtualization, compilers, run-times, distributed systems, and real-time systems. Ravindran and his students have published more than 250 papers in these spaces, some of which have won best paper award nominations and awards. Some of his group's results have been transitioned to the US DOD, in particular, the Navy. Ravindran has graduated 18 PhD students, mentored 8 postdoctoral scholars, and is an ACM Distinguished Scientist.