# Brief Announcement: Managing Resource Limitation of Best-Effort HTM

Mohamed Mohamedin, Roberto Palmieri, Ahmed Hassan, Binoy Ravindran
Virginia Tech
Blacksburg, VA
{mohamedin,robertop,hassan84,binoy}@vt.edu

## 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, the first 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).

## Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features

## Keywords

Transactional Memory, Hardware Transactions, Concurrency

## 1. INTRODUCTION

Transactional Memory (TM) [4] is one of the most attractive recent innovations in the area of concurrent and transactional applications. TM is a support that programmers can exploit while developing parallel applications so that the hard problem of synchronizing different threads, which operate on shared objects, is solved.

Very recently 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 since its version 4.7; second, Intel released to the customer market the *Haswell* processor equipped with *Transactional Synchronization Extensions* (TSX) [7], which allow the execution of transactions directly on the hardware through an enriched hardware cache-coherence protocol.

*SPAA'15,* June 13–15, 2015, Portland, OR, USA.
.

Hardware transactions (or HTM transactions) are much faster than their software version because the conflict resolution 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 an HTM transaction is guaranteed through a software execution defined by the programmer (called *fall-back 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 fall back to a hybrid-HTM scheme [5, 1].

Leveraging the experience learnt from recent papers on HTM [2, 1], three reasons that force a transaction to abort have been identified: *conflict*, *capacity*, and *other*. Conflict failure occurs when two transactions access the same object and at least one of them wants to write it; a transaction is aborted for capacity if the number of cache-lines accessed is higher than the maximum allowed; and any extra hardware intervention, including interrupts, is also a cause of abort.

Many recent papers propose solutions to handle aborts due to conflict efficiently (e.g., [1, 5]), and to tune the number of retries a transaction running in hardware has to accomplish before falling back to the software path [2]. Despite this body of work, one of the main unsolved problems of best-effort HTM is that there are transactions that, by nature and due to the characteristics of the underlying architecture, are impossible to be committed as hardware transactions. Examples include transactions that require non-trivial execution time even accessing few objects and thus they are aborted due to a timer interrupt (which triggers the actions of the OS scheduler); or those transactions accessing several objects, 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, are 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, an innovative transaction processing scheme, which prevents those transactions that cannot be executed as HTM due to space and/or time limitation to fall back to the GL-software path, and commit them still exploiting the advantages of HTM.

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 objects 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) objects.

This framework is designed to be low overhead: 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 objects 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 any accessed object. In addition, a software validation is performed to serialize all sub-HTM transactions at a single point in time.

With this limited overhead, Part-htm gives performance close to pure HTM transactions, in scenarios where HTM transactions are likely to commit without falling back to the software path, and better than pure STM transactions, where HTM transactions repeatedly fail. This latter goal is reached exploiting sub-HTM transactions, which are indeed faster than any instrumented software transactions. In other words, Part-htm's performance gains from HTM's advantages even for those transactions that are not originally suited for HTM due to resource failures.

Opacity [3] is the reference correctness criterion for TM implementations because it avoids any inconsistency during the execution, independently from the final transaction outcome (either commit or abort). However, ensuring opacity in Part-htm is challenging because its overhead could nullify the achieved benefits. While acknowledging the importance of an opaque hybrid-TM protocol, in this paper we briefly introduce two versions of Part-htm. One aims at obtaining the best performance by relaxing opacity in favor of serializability, the well-known consistency criterion for online transaction processing, and by relying on the HTM protection mechanism (i.e., sandboxing), which protects from faulty computations (e.g., division by zero). In the second version, we enriched Part-htm for ensuring opacity but, at the same time, we present a set of innovations (e.g., *address-embedded* write locks) for reducing the transaction's memory footprint so that the overhead is kept limited (less than the achievable gain).

## 2. PROBLEM STATEMENT

In this section we briefly overview the principles of Intel's HTM transactions in order to highlight their limitations and motivate our proposal. The current Intel HTM implementation of the Haswell processor, also called Intel Haswell Restricted Transactional Memory (RTM) [7], is a best-effort HTM, namely no transaction is guaranteed to eventually commit. In particular, it enforces space and time limitations. Haswell's RTM uses L1 cache (32KB) 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.

## 3. ALGORITHM DESIGN

Despite the simple main idea of partitioning a transaction into smaller hardware sub-transactions, executing them efficiently in a way such that the global transaction's isolation and consistency is preserved 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. Hereafter, we refer to the original (single block) transaction as a *global* transaction and the smaller sub-transactions as *sub-HTM transactions*.

A memory transaction is a sequence of read and write operations on shared data that should appear as atomically executed at a point in time between its beginning and its completion, and in isolation from other transactions. This also entails that changes on the shared objects performed by a transaction should not be accessible (visible) to other transactions until that transaction is allowed to commit. The latter point clashes with the above idea: when a sub-HTM transaction $T_{S1}$ of a global transaction $T$ commits, its written objects are applied directly to the shared memory, by nature. This allows other transactions to potentially access these values, thus breaking the isolation of $T$. Moreover, once $T_{S1}$ is committed, there is no record of its read/written objects during the rest of $T$'s execution, therefore also $T$'s correctness becomes hard to enforce.

All these problems can be trivially solved by instrumenting HTM operations for populating the same meta-data commonly used by STM protocols for tracking accesses and handling conflicts. However, applying existing STM solutions can easily lead to HTM losing its effectiveness and, consequently, lead to poor performance. In the following we point out some of these reasons:

- STM meta-data 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 a conflict detection mechanism faster than any software-based contention manager.
- HTM monitors any memory access within the transaction, including those on the meta-data or local variables, which takes the flexibility for implementing smart contention policies away from the programmer.

Part-htm faces the challenge of how to exploit the efficiency of sub-HTM transactions, which write in-place to the shared memory, by minimizing the overhead of the instrumentation needed for maintaining the isolation and correctness of global transactions, and taking into account the above points. Given that HTM transactions commit directly to the shared memory and Part-htm always executes transactions using HTM (except when the GL-software path is invoked), we opt for using an *eager* approach. Part-htm first executes incoming transactions as HTM with few instrumentations (called *first-trial* HTM transactions). In case they experience a resource failure, then our software framework "kicks in" by splitting them.

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 reading and committing or overwriting values created by $T^x_y$ before $T^x$ is committed; and *3)* executing $T^x$ in a way the transaction observes a consistent state of the memory.

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

For the achievement of the above goals, the software framework needs a hint about objects accessed by sub-HTM transactions. In order to do that, we do not use the classical address/value-based read-set or write-set as commonly adopted by STM implementations; rather we rely only on cache-aligned Bloom-filter-based meta-data (just Bloom-filter hereafter) to keep track of read/write accesses. Just before committing, a sub-HTM transaction updates a shared Bloom-filter for notifying its written objects, so that no other transaction can access them.

Two Bloom-filters per global transaction are used for recording the objects read and written by its sub-HTM transactions. In fact, these Bloom-filters are passed by the framework from one sub-HTM transaction to another. Therefore, they are not globally visible outside the transaction. The purpose of these Bloom-filters is to let read/written objects survive even after the commit of a sub-HTM transaction, allowing the framework to check the validity of the global transaction at any time. A value-based undo-log is kept for handling the abort of a transaction having sub-HTM transactions already committed.

As mentioned before, in this paper we provide also a version of Part-htm (called Part-htm-o) that guarantees opacity by introducing some (but limited) overheads. Specifically, any sub-HTM transaction of Part-htm-o performs the following two additional checks. First, once an object is accessed by a sub-HTM transaction, the existence of a write lock is immediately detected. In order to minimize the impact on the memory footprint, we introduce the *address-embedded* write locks, which are locks that do not use additional memory location, whereas they are implemented by "stealing" the last bits from the accessed address. This prevents any false conflicts on the shared write locks set. Second, a sub-HTM transaction is immediately aborted once a global transaction commits. This is achieved leveraging the HTM conflict resolution itself by monitoring a shared timestamp, incremented anytime a global transaction commits, inside sub-HTM transactions.

## 4. PRELIMINARY EVALUATION

Figure 1 plots the results of Part-htm using the Labyrinth application of the STAMP benchmark [6]. We selected this application because more than half of the generated transactions in Labyrinth are large and long, thus their original version cannot complete using HTM.

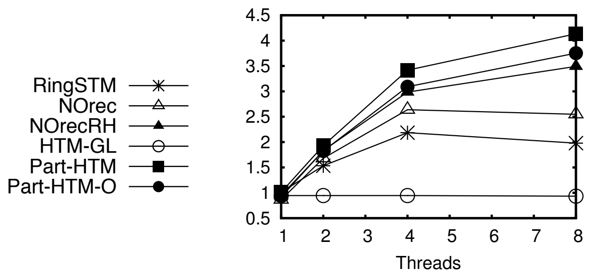

**Figure 1: Speed-up over sequential (non-transactional) execution using Labyrinth of STAMP.**

Table 1 reports the breakdown of the reason transactions are aborted in Labyrinth using HTM and Part-htm. Here we can see how the percentage of HTM transactions aborted for both *capacity* and *other* forms more than 91% of all aborts, forcing HTM to often execute its GL-software path. Part-htm solves this issue.

| | Conflict | Capacity | Explicit | Other |
|---|---|---|---|---|
| HTM-GL | 10.11% | 70.76% | 0.04% | 19.09% |
| Part-htm | 93.95% | 1.09% | 1.14% | 3.82% |

**Table 1: Decomposition of aborted transactions using Labyrinth and 4 threads.**

## 5. ACKNOWLEDGMENTS

Authors would like to thank Alexander Matveev for the important feedback. This work is supported in part by US National Science Foundation under grant CNS 1217385.

## 6. REFERENCES

[1] I. Calciu, J. Gottschlich, T. Shpeisman, G. Pokam, and M. Herlihy. Invyswell: A hybrid transactional memory for haswell's restricted transactional memory. In *PACT*, pages 187–200, 2014.
[2] N. Diegues and P. Romano. Self-tuning intel transactional synchronization extensions. In *ICAC*, pages 209–219, 2014.
[3] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPoPP*, pages 175–184, 2008.
[4] T. Harris, J. Larus, and R. Rajwar. Transactional memory, 2nd edition. *Synthesis Lectures on Computer Architecture*, 5(1), 2010.
[5] A. Matveev and N. Shavit. Reduced hardware transactions: A new approach to hybrid transactional memory. In *SPAA*, pages 11–22, 2013.
[6] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC*, pages 35–46, 2008.
[7] J. Reinders. Transactional synchronization in haswell. *Intel Software Network.*, 2012.