Brief Announcement: On Scheduling Best-Effort HTM Transactions

Mohamed Mohamedin
Virginia Tech
Blacksburg, VA
mohamedin@vt.edu

Roberto Palmieri
Virginia Tech
Blacksburg, VA
robertop@vt.edu

Binoy Ravindran
Virginia Tech
Blacksburg, VA
binoy@vt.edu

ABSTRACT

This paper shows the issues to face while designing contention management policies that involve best-effort hardware transactions. Also, in this paper we present Octonauts, a solution for scheduling HTM transactions without relying on on-the-fly information. Octonauts learns the objects accessed by a hardware transaction while running and it uses them in case of conflict. It also proposes an innovative scheme for optimizing the communication between transactions running in hardware and software.

Categories and Subject Descriptors

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

Keywords

Scheduling, Hardware Transactional Memory, Synchronization

1. INTRODUCTION

Transactional Memory (TM) [8] is increasingly becoming a promising technology for designing and implementing concurrent applications. Recently, TM gained more traction because hardware vendors released the first commodity processors with transactional support (i.e., hardware transactional memory or HTM): Intel *Haswell* [11], and the IBM *Power 8* [3]. The common issue of all those processors is their *best effort* nature, namely transactions are not guaranteed to progress in HTM, thus a software fallback path is a mandatory requirement, which leads to hybrid TM.

Most TM implementations achieve high concurrency when the actual contention level is low (i.e., few transactions conflict with each other). At higher contention levels, transactions abort each other more frequently and a contention manager (CM) is often required to manage concurrency. A CM is an encounter-time technique: when a transaction conflicts with another one, the CM is consulted to decide which of the two transactions can proceed. A CM collects information about each transaction (e.g., start time, number of reads/writes, number of retries) and, according to the implemented policy, it decides priorities among conflicting transactions. This management guarantees more fairness and progress. A CM can work either during the transaction execution by using live (on the fly) information, or work prior the transaction execution. Schedulers in the latter category use information about transaction's potential working-set (reads and writes) defined a priori in order to avoid the need of solving conflicts while transactions are executing. Examples include [7, 5, 2, 13, 10, 6, 1].

The problem of defining a CM for handling HTM transactions is still not investigated because current HTM implementations embed the conflict resolution strategy entirely into the cache-coherence protocol. Roughly, *i)* the L1 cache of each CPU-core is used as a buffer for the transactional write and read operations; *ii)* the granularity used for tracking accesses is the cache line; and *iii)* the eviction and invalidation of cache lines defines when a transaction is aborted (it reproduces the idea of read-set and write-set invalidation of STM). In addition, the Intel Haswell documentation says "*Data conflicts are detected through the cache-coherence protocol. Data conflicts cause transactional aborts. In the initial implementation, the thread that detects the data conflict will transactionally abort.*". From the specification, it is impossible to unequivocally define which thread will detect the conflict as the details of the hardware cache-coherence protocol are not publicly available.

Since there is no way to change the provided HTM conflict resolution policies without modifying the hardware itself, classical CMs policies cannot be trivially ported for scheduling HTM transaction. In fact, when the hardware detects a conflict between threads (executing transactions) accessing the same cache line, the progress made by one of them is immediately aborted without giving the programmer a chance to either manage the conflict differently, or extract any runtime information (i.e., any written object is automatically discarded after an abort). As a consequence of this process, when the programmer receives the notification that a transaction is aborted, it is already too late for avoiding it or deciding which transaction is more convenient to abort.

HTM treats all reads and writes executed within the boundaries of a transaction as transactional, even if the accessed object is not shared or there is no need to guarantee atomicity on it. Non-transactional accesses inside a transaction cannot be performed in current architectures providing

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

HTM support. Customizing the conflict resolution policy and controlling which transaction aborts necessitates detecting the conflict before it happens. However, this fact means redoing what HTM already efficiently provides (i.e., the conflict detection). In addition, every access (read or write) to shared data should be monitored for a potential conflict. Also, per-object runtime information should also be kept (i.e., meta-data) and this leads to one of the major problems of CM in HTM: accessing shared meta-data for each object introduces more conflicts and reduces available cache-lines.

As an example of this claim, let us consider the case of adding a read/write lock for each shared object, which indicates that a transaction is currently reading/writing the object. In order to preserve the semantics of the lock, each transaction should read the value of the lock before accessing the related object. Let us now consider the case of two transactions both reading the object. In this case each transaction checks the availability of the read-lock, then it registers itself as another read-lock holder and proceeds by accomplishing the read operation. However, at the memory level, the acquisition of the lock means writing to the lock variable. Due to the HTM implementation, the lock is just an object enclosed in a cache line. Reading the lock status will add it to the transaction read-set, and acquiring (updating) the lock will add it to the write-set. Since all memory accesses in an HTM transaction are considered as transactional, once a transaction acquires the lock, it will conflict with all other transactions that read/wrote to the same lock. The issue described in the previous example can be applied to any meta-data used by the CM.

2. HTM-AWARE SCHEDULER

To circumvent the aforementioned problem, we propose a CM that operates on "static" information about incoming transactions (i.e., not collected during the transactional execution, but rather before or after it). According to these information, only those transactions that are non conflicting with each other can be concurrently scheduled. One would say that, given such a CM, transactions are not needed anymore because no conflicting executions can happen at-a-time. However the previous case is ideal because, due to the "static" nature of the runtime information, the CM could erroneously activate two conflicting transactions, concurrently. In such a case, correctness is still preserved through the conflict resolution of HTM.

We propose OCTONAUTS, an HTM-aware scheduler that deploys the idea illustrated above by using queues (called *scheduling-queues*) that guard shared objects. A transaction is associated with the objects that will be potentially accessed during the execution (called *working-set*). Then, the working-set is used for deciding which scheduling-queue(s) the transaction should be subscribed to. The subscription process is atomic. If a transaction needs to subscribe to more than one scheduling-queue, it locks the required queues in a deterministic order to avoid deadlocks, and then it proceeds with the actual enqueuing process. When the transaction reaches the top of all subscribed queues, it can execute.

The working-set could be either statically declared by the programmer or dynamically built as follow. Once submitted, HTM transactions are firstly activated as hardware transactions without waiting any decision from OCTONAUTS. While executing, the accessed objects are collected and, in case the transaction is committed, the just-composed working-set is used for classifying the next incoming transactions of the same profile (e.g., the New Order transaction profile of the TPC-C benchmark [4]). This way, after a period of application execution time, accesses can be predicted and the respective transaction will be enqueued in the appropriate scheduling-queues. This approach to populate the working-set is not deterministic and it depends on the actual runtime execution, therefore it should be refined as a result of subsequent commits.

In practice, in order to efficiently implement OCTONAUTS's scheduling scheme, we use a system inspired by the synchronization mechanism where tickets are leveraged. We associate two integers (i.e., `enq_counter`, `deq_counter`) and a lock with each scheduling-queue. The subscription process of a thread T to a scheduling-queue Q_s consists of the following steps. T increments the `enq_counter` of Q_s (in other words T obtains a ticket on Q_s). After that, T waits until the `deq_counter` reaches the value of the acquired ticket. To prevent deadlocks, T must subscribe to all required scheduling-queues atomically. To accomplish this task, T acquires all locks associated with the required scheduling-queues before incrementing all `enq_counter`. After executing a transaction, T increments the `deq_counter` of all subscribed scheduling-queues, thus allowing next (conflicting) transactions to proceed.

Using the described technique, two read-only transactions accessing the same object are not allowed to execute concurrently, although such transactions cannot conflict with each other because they just read. Serializing those read-only transactions could significantly limit the overall concurrency, especially in read dominated workloads. To overcome this drawback, we modified the aforementioned ticketing technique to accommodate reader and writer tickets. Threads owning reader tickets can proceed concurrently if there is no active writer. Rather, conflicting writers are serialized.

2.1 Handling HTM's Software Fallback

Transactions that cannot be committed in HTM (even if they run alone), need a software fallback path to ensure the application's progress. However, the communication between the STM and HTM paths should have minimal overhead, otherwise the OCTONAUTS's goal of increasing concurrency is nullified.

To minimize the aforementioned communication overhead, we use a phasing approach inspired by [9]. We execute HTM transactions in two ways: *plain* HTM and *instrumented* HTM. When the whole transactional workload runs in HTM, then the plain mode is adopted. Once an STM transaction is activated, it makes a notification so that all new HTM transactions after this point start in the instrumented HTM mode. In such a case, the STM transaction waits until all the plain HTM transactions finish, and then starts its execution. When all STM transactions are committed, the execution returns to the plain HTM mode.

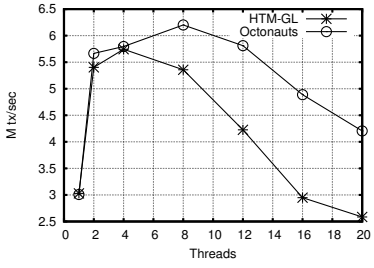
When the instrumented HTM mode is active, we propose to use a circular buffer (called the *ring* [12]), which contains the write-set signatures (i.e., Bloom filters) of each committed HTM transaction. An HTM transaction gets an empty entry from the ring before starting the transaction (i.e., non-transactionally using a CAS operation). During the HTM transaction, every write operation to a shared object is logged into a local write-set signature (e.g., a Bloom

filter). Before committing the HTM transaction, the local write-set signature is stored into a preemptively reserved entry of the ring. This design eliminates false conflicts due to shared HTM-STM meta-data (in our case, the ring). The ring entry is reserved before starting the HTM transaction and each HTM transaction writes to its own private ring entry. For STM transactions, they read only the ring entries so that they cannot conflict (at memory level) with any HTM transaction.

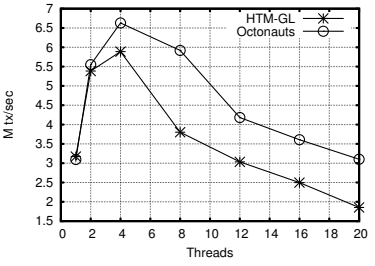
An STM transaction proceeds speculatively until its commit phase. Before committing, it validates its read objects against the concurrent HTM transactions (via the write-set signatures in the ring). If it is valid, the commit can take place. The proposed technique seems to favor HTM transactions, but since both HTM and STM transactions subscribe to the same object queues, HTM and STM transactions can only conflict due to an inaccurate definition of the working-set or due to Bloom filters' false conflicts. Thus, STM transactions cannot suffer from starvation.

3. PRELIMINARY EVALUATION

In order to show the practicality of our proposal, we built a preliminary version of OCTONAUTS using the TPC-C benchmark [4] configured with medium and high contention. The former has been enforced by selecting a total number of warehouses (the most contended object of TPC-C) as 20; whereas for the high contention case, we used 10 warehouses. We compared the throughput of OCTONAUTS against the pure HTM with the global locking as fallback. As testbed we used the Intel Haswell processor (i7-4770), which is equipped with 4 physical cores and 8 threads (given the hyper threading). Each hardware transaction retries 5 times before falling back to the software path. In this preliminary implementation, the working-set is defined by the programmer.



(a) Medium-contention.



(b) High-contention.

Figure 1: Throughput using TPC-C benchmark.

Figure 1 shows the results. When the contention level in TPC-C is high, OCTONAUTS is particularly effective, being able to reduce the number of conflicts significantly. In addition, when the number of threads is larger than cores, OCTONAUTS is still able to scale because scheduling transactions properly leads to more concurrency than just leaving transactions to contend (and abort) each other. As an evidence of that, in our experiments when the number of threads is larger than 8 (which is the maximum number of hardware threads supported in the Haswell processor), both HTM-GL and OCTONAUTS can run only 8 transactions at a time and schedule the others. However, HTM-GL selects those 8 transactions according to the policy of the operating system's scheduler, whereas OCTONAUTS gives more guarantees that the selected ones are not conflicting.

4. ACKNOWLEDGMENTS

Authors would like to thank Ahmed Hassan and the anonymous reviewers for their important comments. This work is supported in part by US National Science Foundation under grant CNS 1217385 and AFOSR Grant FA9550-14-1-0187.

5. REFERENCES

[1] M. Ansari, M. Luján, C. Kotselidis, K. Jarvis, C. C. Kirkham, and I. Watson. Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering. In *HiPEAC*, pages 4–18, 2009.

[2] H. Attiya and A. Milani. Transactional scheduling for read-dominated workloads. *Journal of Parallel and Distributed Computing*, 72(10):1386 – 1396, 2012.

[3] 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*, pages 225–236, 2013.

[4] T. Council. TPC-C benchmark. 2010.

[5] S. Dolev, D. Hendler, and A. Suissa. CAR-STM: Scheduling-based Collision Avoidance and Resolution for Software Transactional Memory. PODC, pages 125–134, 2008.

[6] A. Dragojević, R. Guerraoui, A. V. Singh, and V. Singh. Preventing versus curing: Avoiding conflicts in transactional memories. PODC, pages 7–16, 2009.

[7] R. Guerraoui, M. Herlihy, and B. Pochon. Polymorphic contention management. In *Distributed Computing*, volume 3724, pages 303–323. Springer, 2005.

[8] T. Harris, J. Larus, and R. Rajwar. Transactional memory, 2nd edition. *Synthesis Lectures on Computer Architecture*, 5(1), 2010.

[9] Y. Lev, M. Moir, and D. Nussbaum. PhTM: Phased transactional memory. In *TRANSACT*, 2007.

[10] W. Maldonado, P. Marlier, P. Felber, A. Suissa, D. Hendler, A. Fedorova, J. L. Lawall, and G. Muller. Scheduling support for transactional memory contention management. PPoPP, pages 79–90, 2010.

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

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

[13] R. M. Yoo and H.-H. S. Lee. Adaptive transaction scheduling for transactional memory systems. SPAA '08, pages 169–178, 2008.