Commercial-Off-the-shelf Hardware Transactional Memory for Tolerating Transient Hardware Errors

Short Term Scientific Mission Report

Rasha Faqeh

December 25, 2014

The purpose of the STSM

Transactional Memory (TM) [1] programming model attempts to simplify concurrent programming for multi-core processors by transferring the concurrency management from the programmer to the system. In this model, transactions are atomic and isolated sequences of memory accesses that are executed speculatively. A transaction commits its updates to the memory if they do not conflict with any other concurrent transaction, thus, making them permanent. A transaction aborts if it is unable to commit its speculative updates to memory. It uses the TM checkpoint/roll-back mechanism to undo these changes. Many researchers used TM for concurrency control [2, 3, 4] while few attempts are directed into using TM for only failure control instead of concurrency control [5, 6, 7].

Transactional Encoding (TE) [6] is a fault-tolerance software based technique used to detect and mask transient hardware errors in order to build safe applications on top of unreliable hardware. Encoded processing is used to detect incorrect execution, while to recover from errors, a checkpoint/rollback of software transactional memory (STM) is used. The experiments show that TE tolerates execution errors with high fault coverage. On the other hand, although it uses an optimized library for reliability rather than for concurrent synchronization, the performance of such system is relatively low in comparison to the native execution even in the error-free execution (up to 5x slowdowm). Therefore, we want to reduce the performance degradation by implementing hardware transactional memory (HTM) version optimized for reliability. In addition to better performance, the reliability targeted hardware implementation will overcome the transaction-size restriction found in the concurrency targeted HTM's [8, 9]. It is feasible to have fine grained customizable granularity transactions as they are targeted for reliability not for parallelism [7]. Therefore, minimal resources are needed to implement such system, since resources are reused when transactions are committed or aborted.

The Technische Universität Dresden (TUD) and the Barcelona Supercomputing Center (BSC) intend to collaborate to build a cache extension that implements reliable targeted HTM. The idea is to extend the processor's L1 data cache to maintain the written addresses locally at the granularity of a cache line. Any modification to the memory will take place in the cache and not in the memory during the speculative execution of the transaction. So, the cache acts as a write-back log. However, due to time limitations and the design similarities with already exist commercial-off-the-shelf hardware in particular Intel Haswell TSX [10] Chapter 12), STSM plan was modified. So, instead of building the HTM hardware from scratch, we redirect our investigation to study to what extent transient fault recovery can be implemented leveraging the abort mechanism of commodity hardware transactional memory (HTM) (i.e. Intel Haswell TSX, IBM Power8 [11]).

Hardware fault tolerance can be implemented in software or in hardware. Software based solutions are flexible since they are easier to use, cheaper, faster to develop and allow to use the most up to date hardware. However, they have higher performance overhead than hardware based solutions. While hardware based solutions have better performance, they are hard-coded in silicon which reduces their flexibility, moreover, the hardware used soon will be outdated. This infers that using customized hardware to implement fault tolerance mech-

anisms is not practical for general purpose systems. Which implies that the only practical solution is to use commodity hardware to implement the fault tolerance. Additionally, different applications might require different levels of protection from transient faults, so they can trade reliability for performance, which induce the use of software based error detection mechanisms to provide the needed flexibility. On the other hand, general purpose systems major concerns are the performance and how much overhead induced by the usage of fault tolerance techniques. Therefore, we propose to decouple the error detection from the recovery and restrict the use of hardware to implement fault tolerance into the error recovery by reusing already exist facilities in the COTS hardware (error recovery in HTM). This allows to interchange the error detection techniques based on the targeted fault model in addition to provide a low overhead error recovery.

Description of the work carried out during the STSM

Recent commodity hardware has implementations of hardware based transactional memory for concurrency control [10, 11]. The idea is to execute concurrently critical sections inside transactions instead of serializing by lock acquisition. If transactions access the same data in a conflicting way (at least one is a writer), TM has an abort mechanism that undo the changes done by the conflicting transactions, then re-execute. In a different context, TM abort mechanism can be used to achieve fault tolerance by recovering from transient faults [5, 6, 7]. The re-usability of the same commodity hardware is the stimulus in this study for *using the commodity HTM to recover from transient faults.*

In this work, our focus is on investigating the limitations and the values of using commodity hardware transaction memory to recover from transient faults, therefore, transient error detection mechanisms are out of the scope of this report.

Error Recovery

In this work, we study how to use two commodity hardware transactional memory (Intel Haswell TSX and IBM Power8 HTM) to implement the recovery from transient faults:

Intel TSX: Intel TSX has Restricted Transactional Memory (RTM) mode that provides an interface to start, end and abort transactions. It allows to define an atomic region of the program code by surrounding it by xbegin() and xend() calls. All write operations inside the atomic region are allowed to be externalized to the memory only when the transaction commit successfully. Concurrent transactions conflict if there is mutual access to some memory address between concurrent transactions and at least one of them is a writer. In case of conflict, the transaction is aborted, by invalidating all changed done by that transaction and roll-back to the state the was before the transaction started. After rolling back a transaction, software implemented failure handler will execute, by either retrying the transactional execution again, or fall-back to lock based execution

to insure the forward progress. *The existence of abort handler and the explicit abort call seduce us to use Intel TSX for fault tolerance.*

Challenges when using Intel TSX for fault tolerance: Intel TSX is build for concurrency control, therefore, it has some implicit assumptions on the work-flow of transactions. In the following, we list the challenges found when we try to use Intel TSX for fault tolerance:

1. **Conflict Detection is fixed:**

   Conflict are detected in TSX using cache coherency protocol at L1 cache line granularity (physical addr), so it is done implicitly by the hardware. Therefore, no enhancement can be applied to the conflict detection. This prevent the direct usage of any error detection mechanism that execute in parallel any code in replicated fashion (similar to FaulTM [7]), since the hardware will detect that the two transactions are conflicting and they will be aborted immediately.

2. **TSX is best effort:**

   Best effort implies that there is no guarantee that any transaction can finally commit successfully. This is why in TSX, a non transactional execution path should always be provided to guarantee forward progress. Transactions in TSX might abort due to many reasons: conflict with concurrent transaction, L1 cache capacity limit reached, timer interrupts and system calls. Most of the reasons can be avoided in the reliability context. Aborts due to conflicting transactions can be avoided by not having concurrent transactions. To prevent aborts due to system calls, they can be excluded from the transactional execution, so we can end transaction before and start new transaction after system calls. Capacity aborts can be avoided. The size of reliability transactions is flexible, which allows us to reduce the size of transactions to not reach the capacity limit. On the other hand, aborts due to timer interrupts might not be voided, specially with larger sized transactions. To handle these aborts, we can simply restart the transactional execution. However, since TSX is best effort, then we limit the number of retries to a maximum threshold, afterwhich, that particular code will be executed without transaction protection. Executing code without transaction protection will reduce the reliability, but will allow the program to proceed.

IBM power8: IBM Power architecture is another commercial architecture that added hardware support for transactional memory. It's most recent processor implements HTM is Power8 [11]. Power8 has similar transactional memory semantics to Intel Haswell TSX. TM_begin and TM_end marks the start and end of transactional execution, which provide the transactional mode of execution. Power8 HTM includes a new exclusive mode known as *suspended mode* in which the transactional execution is suspended to allow software to execute non transactionally for a period of time, afterwhich the transaction will resume the execution in the transactional mode. The new mode is entered using TM_suspend and TM_resume. Instructions executed after TM_suspend will have the same effect as if they are executed outside transaction. TM_resume

resumes the transactional execution and allows the speculative execution of instructions. One usage of such suspended mode is for debugging, by printing transactional values, in addition to executing some of the instructions that are not allowed to be executed inside transactions including system calls or interrupt handlers. Another important usage of suspend mode is to implement the HTM pre-commit validation that allows to implement a customized error detection mechanisms. Additional exclusive implementation to the Power8 HTM is a special kind of hardware transactions called *Rollback-Only Transactions (ROT)*s, which track only the memory writes and do not undergo conflict detection. They are intended to be used for single threaded software speculation. They are not used to manipulate shared data (atomicity is not provided). They support rollback by tracking only memory writes of transaction (write set), therefore, fixed sizes buffer allows for larger transactions than in case of regular atomic transactions. No barriers or serializability of atomic transactions are required. *By encapsulating the instructions in a ROT, one can hide the memory writes issued until successful commit and provide a lazy subscription without exposing them to data inconsistencies, which make them perfect for the usage of error recovery for reliability.*

Challenges when using Power8 for fault tolerance Atomic Power8 transactions are similar in principle to Intel Haswell TSX, therefore, they incur the same challenges:

1. **Conflict Detection is fixed:**

   Conflict are detected in Power8 using cache coherency protocol at L2 cache line granularity, so it is done implicitly by the hardware, and can not be changed when atomic transactions are used. However, using Power8 flexible interface allows to overcome this problem. To implement a reliable error detection, we need to prevent the usage of concurrency conflict detection, in addition to provide our own validation algorithm to check for the correctness of the execution. The former can be achieved by using Power8 ROT transactions where no conflict detection is used. The later can be implemented either in the suspended mode to validate with an external state, or in the transactional mode if that would be enough.

2. **Power8 HTM is best effort:** TM in Power8 is best effort HTM. There would be no guarantee of success for any transaction. Therefore, we need to specify a failure handler allowing for non-transactional alternatives. Transactions in Power8 might abort due to many reasons: conflict with concurrent transaction, cache capacity limit reached, timer interrupts and system calls. In the reliability context, we can use ROT transactions to prevent aborts due to conflict detection in case we have multiple transactions executed in parallel, or we can limit the execution to single transaction at any time in parallel. To prevent aborts due to system calls, they can be excluded from the transactional execution, either by ending transaction before and starting new transaction after system calls, or executing system calls inside the suspended mode. However, to handle output commit problem, it is more desirable to commit previous code before executing system call, so to execute system calls in a non-transactional mode. The suspended mode can be used to handle timer interrupts and to allow

transactions to commit successfully despite having interrupts. This model is called suspend-on-interrupts model, whereas Intel Haswell TSX supports abort-on-interrupts model. To handle aborts caused by interrupts, we can simply restart the transactional execution. However, since Power8 is best effort, then we limit the number of retries to a maximum threshold, afterwhich, that particular code will be executed without transaction protection. Executing code without transaction protection will reduce the reliability, but will allow the program to proceed. Power8 flexible interface reduces the probability that transactions aborts in comparison to Intel TSX.

Description of the main results obtained

Intel TSX:

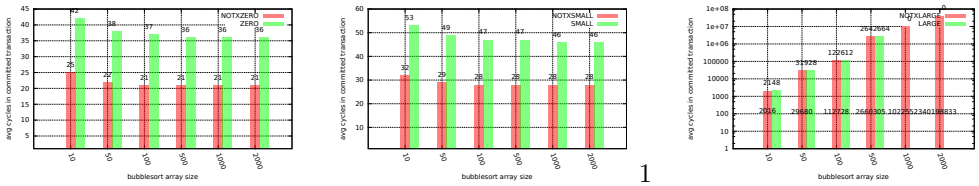
Start and Commit Transaction Overhead

The goal is to measure the number of cycles needed for starting and ending Intel TSX transactions. No error detection mechanism has been applied yet. We executed part of the application code inside transactions and compared the cycles needed to the native application without transactions. For simplicity we use bubble-sort application. Different sizes of transactions are used: LARGE, SMALL and ZERO. Listing 1 presents the boundaries of transactions used. ZERO size transactions indicate the overhead of begin (store the snapshot of registers) and end transactions without any memory access overhead. SMALL and LARGE size transactions indicate the overhead of committing small and large number of changes to memory respectively, in addition to the overhead of begin and end transactions. To measure cycles spend in each transaction size, we used *real time stamp counter* (RDTSC) instruction provided by modern Intel processors [12]. RDTSC is useful to measure the cycles in a section of the code. We record cycles spent inside transactions between xbegin() and xend() calls. Therefore, if transaction aborts, the abort handler will restart the transactional execution by calling xbegin() again, which will reset the timer again. Therefore, the time wasted by the aborted transaction is not included in the cycles recorded. To amortize the cost of reading the time stamp counter, the same measurements is taken for the non transactional code. The difference between cycles counted in case of NOTXZERO and ZERO bars indicates the cycles needed to start and commit an empty transaction in Intel TSX.

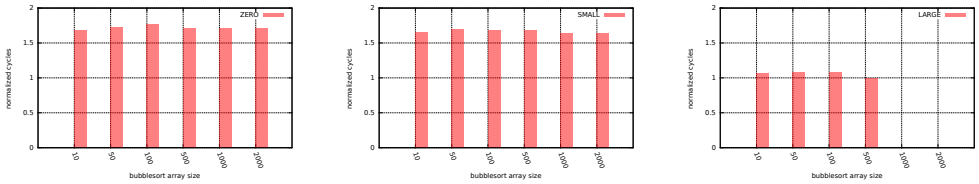
```
1  void bubble_sort(int iarr[],int arraysize) {
2    int i, j, temp;
3
4    // begin_tx: LARGE
5    for (i = 1; i < arraysize; i++) {
6      for (j = 0; j < arraysize - 1; j++) {
7        if (iarr[j] > iarr[j + 1]) {
8          // begin_tx: SMALL
9          temp = iarr[j];
10         iarr[j] = iarr[j + 1];
11         iarr[j + 1] = temp;
12         // end_tx: SMALL
13         // begin_tx: ZERO
14         // end_tx: ZERO
```

```
15          }
16        }
17      }
18      // end_tx: LARGE
19  }
```

Listing 1: bubblesort with transactions of sizes: ZERO, SMALL and LARGE



(a) Real cycles



(b) Normalized cycles

Figure 1: Cycles per committed single transaction

Figure 1a shows the cycles spend inside transactions with sizes ZERO, SMALL and LARGE. While figure 1b shows the normalized version to exclude the cycles spend in doing the measurements of RDTSC and to indicate to what extend the start and commit overhead can be amortized when using different sizes of transactions. The cycles needed to start and end transaction can be calculated by subtracting the number cycles from ZERO and the NOTXZERO transactions, which would be between 15-18 cycles. The difference between ZERO and SMALL cycles shows the overhead of committing a small number of transactional updates to memory. The difference between the number of cycles of the native NOTXLARGE with LARGE shows that with larger transactions, the transactional overhead can be negligible, since the overhead of start and end transaction would be very small in comparison to the time needed to execute instructions inside transactions. The overhead of executing part of the code inside a transaction in case of LARGE transactions is between 10% to 20% in comparison to the native execution, while with SMALL and ZERO it is between 50% to 70%. Therefore, to reduce the overhead of transaction start and commit, we should make larger transactions. Remember that TSX is best effort, therefore, their is no guarantee that any transaction can successfully commit. Which is the reason why the last two bars of LARGE with 1000 and 2000 bubblesort array sized are missing, since no transaction with the given sizes has committed successfully.

Aborts

In this part we wanted to study the effect of best effort transactions execution. Since it is best effort, their is no guarantee that the transaction can commit successfully. In the context of concurrency, it is not a real problem, since the transactional execution can always fallback to lock based execution after a number of retries. Which will not affect the correctness of the concurrent execution but will oppose an extra overhead since the execution will not concurrent, but serialized. However, in the context of reliability, if we are unable to execute application inside transaction even after a number of retries, falling back to execution without transaction would mean that we lost the ability to rollback the execution in case any error has been detected. However, if most transactions would commit, then we could scarify the fault tolerance for the forward progress.

To study how often aborts happen in transactions executed using Intel TSX, we used the same transactional sizes presented in Listing 1.

Figure 2 shows the normalized cycles needed to execute bubblesort application inside transactions with different sizes in comparison to the native NOTEX execution. Aborted transaction are allowed to retry for 0, 1 and 100 times before the application fallback to the native execution without transactional protection. Figure 2a represents the overhead of execution applications inside transactions. Transactions are not allowed to retry in case they abort. They will fallback to the native execution directly after the abort. Investigating the abort status of the previous run shows that most aborts are due to status 0, which indicates aborts due to others (usually timer interrupts). Which indicates that a retry will not solve the problem. if we just retry aborts with status 0, we will most likely retry a transaction that is doomed to abort, which results in an increase in the performance degradation as seen in Figure 2c with LARGE transactions.

The graph also shows that the overhead of starting and ending transaction manifest itself in the overall execution time (ZERO and SMALL bars). While with larger transactions, the overhead is almost negligible (LARGE). Additionally, the start and end overhead is also clearly manifested when comparing the overhead of using SMALL in case of small array sizes (10, 50, 100) with the larger array sizes (500, 1000, 2000) since the number of transactions started in the former case is (45, 1225, 4950) while in the later case is (124750, 499512, 1999030). Similar case when using ZERO sized transaction.

IBM Power8:

We are very thankful to Prof. Pascal Felber from Université de Neuchâtel for providing us with access to their new IBM Power8 machine to run experiments with it. However, due to time limitation, experiments on the Power8 HTM did not finished yet. Therefore, no results will be included in this part.

Future collaboration with the host institution

The two participant TUD and BSC intent to continue the collaboration on using COTS hardware to achieve fault recovery. In addition to evaluate Intel TSX and IBM Power8 HTM implemenations of reliability targeted transactions, we

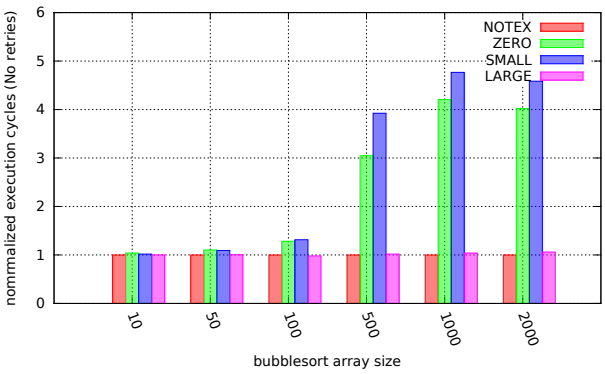
intend to integrate COTS error detection approaches (i.e. SWIFT, Encoded processing and replicated execution) and measure the effectiveness and the cost on both performance and reliability. Additionally, we intend to publish a joint paper to show the results of our study.

Foreseen publications/articles resulting from the STSM

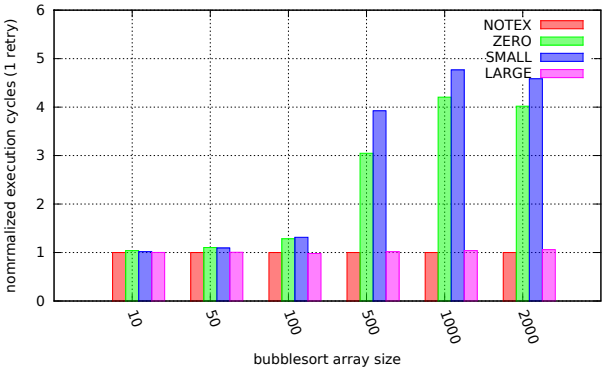
We started writing a paper on the study and we target PACT 2015 or ICCD 2015 conferences.

Confirmation by the host institution of the successful execution of the STSM

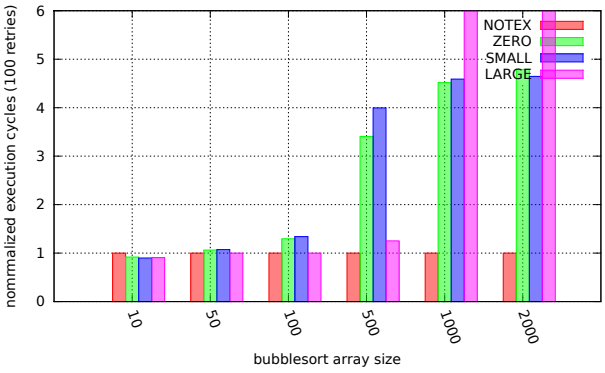
Dr. Osman Ünsal has sent the confirmation letter directly to the STSM coordinators.



(a) No retry



(b) One retry



(c) 100 retry

Figure 2: Normalized execution cycles to the native execution

Bibliography

[1] M. Herlihy and J. Moss, *Transactional memory: Architectural support for lock-free data structures*. IEEE Comput. Soc. Press, 1993.

[2] L. Yen, J. Bobba, and M. Marty, "LogTM-SE: Decoupling hardware transactional memory from caches," *. . . , 2007. HPCA 2007. . . .*, 2007.

[3] P. Felber, C. Fetzer, and T. Riegel, "Dynamic performance tuning of word-based software transactional memory," *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming - PPoPP '08*, p. 237, 2008.

[4] R. Yates and M. Scott, "A Hybrid TM for Haskell," *cs.rochester.edu*, pp. 1–7, 2014.

[5] C. Fetzer and P. Felber, "Transactional memory for dependable embedded systems," *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pp. 223–227, June 2011.

[6] J. Wamhoff, M. Schwalbe, and R. Faqeh, "Transactional Encoding for Tolerating Transient Hardware Errors," *Stabilization, Safety, and . . .*, 2013.

[7] G. Yalcin, O. Unsal, and A. Cristal, "FaulTM: error detection and recovery using hardware transactional memory," *Proceedings of the Conference on Design, . . .*, 2013.

[8] R. Rajwar, M. Herlihy, and K. Lai, "Virtualizing transactional memory," *Computer Architecture, 2005. ISCA . . .*, vol. 00, no. C, 2005.

[9] P. Damron, A. Fedorova, and Y. Lev, "Hybrid transactional memory," *ACM Sigplan . . .*, 2006.

[10] I. Coorporation, "(CH12-TSX_Intel ISA)Intel 64 and IA-32 architectures optimization reference manual," no. March, 2009.

[11] P. Isa, "Power ISA ^{TM}," 2013.

[12] I. Corporation, "How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures White Paper," no. September, pp. 1–37, 2010.