Leveraging modern multi-core processors features to efficiently deal with silent errors

Diego Pérez Arroyo*, Esteban Meneses Rojas*, Cesar Garita Rodríguez*
*Costa Rica Institute of Technology

***Abstract*—Since current multi-core processors are more complex systems on a chip than previous generations, some transient errors may happen, go undetected by the hardware and can potentially mess up the result of an expensive calculation. Because of that, techniques such as replication or checkpointing are utilized to detect and correct these soft errors; however these mechanisms are highly expensive adding a lot of resource overhead. Hardware Transactional Memory exposes a very convenient and efficient way to revert the state of a core's cache, which can be utilized as a recovery technique. We created an experimental prototype that uses such feature to recover the previous state of the calculation when a soft error has been found. Through the combination of HTM, Hyper-Threading and or Memory Protection Extensions, the performance, applicability and confidence of our technique may be further improved.**

I. Introduction

Because of the power wall preventing single core processors manufactures from keeping the increase of frequency as expected, multi-core processors are becoming more complex system on a chip. These systems are more prone to transient errors compared to previous generations of processors, because manufactures continuously boost performance with higher circuit density using really small transistor sizes and at the same time achieving higher energy efficiency by operating at lower voltages [1]. The problem turns even more serious since a lot of these errors are not detected by the hardware, such as bit flips, and can potentially mess up the entire calculation. Such silent errors are considered a major problem for future very large data-centers and supercomputers [2]. There are a wide range of causes for such faults in CPUs, including dynamic voltage scaling, cosmic radiation, physical altitude of the data center, power supply faults, among others. The most common solution to deal with such silent errors is replication, however this technique is extremely costly adding typically 200% of resource overhead.

There are two main challenges when dealing with such kind of errors. The first one is being able to detect that an error happened. The second is correcting the error and ensuring that the final result of the calculation is correct. Recent contributions [1] [3] tend to show that checkpointing techniques provide the best compromise between transparency for the developer and efficient resource usage; still remaining expensive with about 100% of overhead.

To reduce this overhead, we think that some of the features provided by recent processors, such as hardware-managed transactions, hyper-threading, memory protection extensions, are great opportunities to implement efficient error detection and recovery. The pioneer work presented in [1] is a first step towards leveraging hardware transactional memory for recovering application state. The goal of this investigation is to explore opportunities to reduce the cost of rollback-recovery mechanisms by combining the use of mentioned features in current multicore processors.

For that, an experimental prototype has been built that detects and corrects artificially injected errors. Such program uses hardware transactional memory through Intel TSX technology as a recovery mechanism when an error has been found and scales properly when Intel Hyper Threading technology is in use.

The rest of the paper is structured as follows, in the next section we provide some concepts that are necessary to understand properly the rest of the document, in section 3 the background and previous work is presented, section 4 is where we list the details of our prototype that uses transactional memory as a recovery mechanism, in section 5 we show some preliminary results of said prototype and in section 6 we conclude and discuss future work.

II. Concepts

A. Hardware Transactional Memory

Hardware Transactional Memory (HTM) follows the same ideology of database transactions, where a set of instructions are managed as if they were just one instruction, if the transaction succeeds then every operation executed is committed to the database, if not every operation is reverted and the state of the database remain the same as when the transaction began. The difference with HTM is that the goal of a transaction is not to commit changes to a database but to main memory. When a transaction begins the values of variables modified are stored in the cache and if no collisions are detected then those changes are atomically committed to RAM, so every other core can view the new values [4].

Intel introduced the Transactional Synchronization Extensions (TSX) as part of the Haswell Instruction set architecture [5], it is the implementations of HTM of Intel but not the only one, in this document we center on multi-processors chips that have this feature, specifically the Intel Restricted Transactional Memory (RTM) interface, which exposes a new set of primitives:

- _xbegin, initializes a new transaction.
- _xend, marks the current transaction as successful, and therefore commits atomically the changes to RAM.
- _xtest, tells if one is inside a transaction.

- _xabort, explicitly causes an abortion of the current transaction and restores the state of the core as it was at the latest successful execution of _xbegin.

Transactional Memory was originally proposed as a better way to achieve high performance lock-based synchronization, yet easy to implement, in applications with concurrent access to shared memory. Intel TSX ensures the same results as having a coarse-grained lock, but allows not conflictive operations to occur without the delay that coarse grained locks would have injected [6].

Internally in Intel TSX, transactions have a read and write sets, to keep track of what has been read and modified, such information is temporarily stored in L1 cache. The execution model is optimistic, meaning it does not block (as a mutex does) a concurrent execution of the code, instead to detect conflicts in parallel transactions an optimized cache coherency protocol is used; two transaction having the same memory location in their read sets does not cause an abortion, but if one reads a variable that is also present in another transaction write set, the first transaction is aborted. If a transaction was aborted (explicitly or implicitly) the execution jumps to an abort handler (that has to be provided), where usually the transaction is retried a number of times before going to a fallback path where progress should be guaranteed, in case the code cannot be executed transactionally [1].

Even though its original purpose, Intel TSX also provides strong isolation guarantees and a way to rollback that can be utilized as a recovery mechanism, yet there are several design choices that does not make the use of this feature applicable to fault tolerance obvious. Firs of all, it does not guarantees that a transaction will eventually commit, even when applied to sequential code [7]. Since it uses the core's cache to temporarily keep track of reads and writes, the amount of memory is limited to the physical capabilities of the processor. Also there is a time limit (based on the interval of timer interrupts) of how much a transaction can last before it is aborted [1]; and lastly there are unfriendly operations (such as system calls) that force a core to abort any active transactions [1]. Thus, the importance of a fallback path is vital.

B. Hyper-Threading

Hyper-Threading Technology from Intel makes a single physical processor appear as two logical processors; the physical execution resources are shared and the architecture state is duplicated for the two logical processors. From a software or architecture perspective, this means operating systems and user programs can schedule processes or threads to logical processors as they would on multiple physical processors. From a micro-architecture perspective, this means that instructions from both logical processors will persist and execute simultaneously on shared execution resources [8].

Hyper threading does not do much for single thread workloads, but when multiple threads can run in parallel there is a significant performance improvement, because it ensures that when one logical processor is stalled the other logical processing unit (on the same core) could continue to make forward progress (without context switching). A logical processor may be temporarily stalled for a variety of reasons, including servicing cache misses, handling branch mispredictions, or waiting for the results of a previous instruction. [8].

C. Memory Protection Extensions (MPX)

Memory Protection Extensions from Intel includes a set of primitives for pointers bound checking, as well as new registers for storing bounds data. Its original purpose is to check, using new hardware features that can be used by software, that memory references defined at compile time don't become a source of uncertainty at runtime due to buffer overflow or underflow (either way the bonds are violated). MPX main goal is a way to protect memory deficiencies in unsafe languages [9].

Whenever a pointer is used, with Intel MPX the requested memory reference is validated to be inside the pointed associated limits, hence preventing out-of-bound memory access. Buffer overruns account for a considerable amount of all bugs encountered in a typical C, C++ application. [10]

III. Previous work and Background

In this section we present and review how HTM, hyper threading and MPX have been used to deal with soft errors. We try to identify the vulnerabilities or limitations of each solution in order to establish our path to follow.

Intel TSX is primarily target for synchronization and exhibits several design restrictions that make the use of this technique for recovery purposes not trivial [1]. HAFT: Hardware Assisted Fault Tolerance in [1] relies on Instruction Level Redundancy (ILR) to detect faults and HTM (Intel TSX) to correct them. In order to accomplish fault tolerance, first it replicates the instructions of the application and integrity checks are recurrently added. Once this step is performed the application is wrapped in HTM-based transaction in order to be able to recover from a fault. When an error is detected by the ILR checks, the transaction is explicitly rolled back, the state of the application is restored before the transaction began and the execution is retried a fixed number of times until it is executed without transactions.

The best effort approach in HAFT of retrying a transaction a static number of times before trying again without using HTM may be considered quite dangerous. If an error occurs in this non-transactional moment ILR has no choice but to permanently abort the execution of the program; this is definitely worse than adding extra overhead by trying a different recovery technique and ensuring a correct completion of the application. Mostly this design choice in HAFT is driven by the restrictions that Intel TSX currently exhibits, the technique was originally thought for small critical sections and therefore there are constraints that limits the use of the feature. Intel TSX transaction size is limited by the CPU cache size and by the timer interrupt interval; also there are a lot of "unfriendly" instructions (signals/interrupts) that cause the transaction to abort. HAFT therefore presents a "transactification" algorithm that heuristically wraps the application code in HTM-transactions, taking into account the limitations of Intel TSX.

In [9] Oleksenko et.al present another option to use Intel version of hardware transactional memory as recovery mechanism against transient errors. They focus more on errors caused by bit-flips in data pointers, which can be catastrophic especially if the pointer is not to a basic type but a more complex data structure, because it can potentially provoke a considerable amount of data loss.

In order to detect soft errors in pointers they use Intel Memory Protection Extensions. This technology basically adds a set of instructions for pointers bound checking. The main idea for identifying the errors in pointers lies in the fact that if a fault occurs in a pointer, the new value will probably violate the corresponding bounds; the authors also mention that if more than a single event upset (SEU) happens in the same pointer then it will be more probable to detect such incorrect state, because the bound checking will more likely fail. [9].

MPX original purpose is a way to protect memory limitations in unsafe languages, like buffer over runs and even though the idea of using it for fault detection in pointers is quite novel and ingenious, the authors in the paper [9] admit there are a lot subjects not yet explored that they have to keep working on. Another drawback of the solution is that it only detects pointer fault, making it a technique which has to be used in coordination with another one in order to be achieve complete fault tolerance.

Another reference where HTM is used as a method recovery mechanism is in [11] called POSTER, Fault Tolerant Execution on COTS Multi-core Processors with Hardware Transactional Memory Support. In it a software/hardware hybrid approach is proposed which leverages Intel TSX to support implicit checkpoint creation and fast rollback. The authors combine a software based redundant execution for detecting faults with hardware transactional memory to restore the state of the application if necessary.

The main idea of the paper is to redundantly execute each user process and to instrument signature-based comparison on function level. The error detection is allowed given the loosely coupled execution of both processes and with the encapsulation of blocks in transactions, error recovery is realized. For an efficient comparison of both instances, for each block a signature is created (which uniquely identifies it) and shared from the master project to its duplicate. Before committing the transaction, the locally calculated signature and the leading process signature are compared and if an error is detected a restart of the block is initiated [11].

This proposal has a disadvantage that HAFT also has, in which since the use of TSX alone does not guarantee that a non-conflictive transaction will eventually commit and therefore random aborts of the whole application may occur.

IV. Design of the Prototype

We have created a prototype that uses Intel's Hardware Transactional Memory (HTM) to recover from artificially injected errors and use replication to detect them. Basically the following happens, we keep a global array of work, in which for each entry a calculation (calculus function) must be performed, every thread created is assigned a range of this array to work on, by the end the length of the array is divided as equally as possible among the number of threads so each one has about the same amount of work. This scheme represents a very common way of designing parallel programs, multiple values can be calculated this way. In the calculus function there is a fixed probability that an error occurs (meaning the returned result may differ in different executions of the same function with the same parameters). The routine each threads executes is a loop that iterates over the global array of work only in its personal range, for each entry before executing the calculus function twice (replication) it begins a hardware memory transaction, if both executions of the calculation differ then it aborts the current transaction, causing all variables that were modified in such iteration to revert its state to the one they had before starting the transaction. Once in the error handler routine we identify possible causes of the transaction failure (it could have been implicitly or explicitly aborted, due to a memory conflict with another thread or a soft error detection) and if a retry is possible we execute the iteration a fixed number of times (5 in our case), before trying again without transactions; otherwise it moves to the next entry of the array. After the loop, each thread writes in another global array the partial result of its execution, so all results are merged at the end into one.

The fact that GCC supports HTM makes it easy to test and use Intel's TSX primitives to mark the beginning, abortion or completion of a hardware memory transaction [13]. The use of Hyper Threading happens without explicit instructions from the programmer, however it has been taking into account for building the application. Since Hyper Threading duplicates the architectural state for the two logical cores on each real core and HTM uses cache memory to save local values before pushing them to RAM, the size of the transactions must be small enough so it does not fail for storage issues and Hyper Threading can successfully create a copy of the architectural state.

V. Results and analysis

In this section we want to measure two aspects regarding our prototype:

1) the expected total performance overhead.
2) the success rate of (how close was to identify and correct all soft errors).

For that we created an instance (a large sum of integers) of our prototype where we know the total result being calculated beforehand; in our scheme simply calling a new calculus function is enough to obtain different values: simple mathematical operations like sums, multiplications, factorials are ideal for such tests. Having known the correct result beforehand lets us know if the execution was able to reach the expected result, or if an artificially soft error was injected and not corrected. We use a fixed probability of $1/6000000 = 1.66666667 * 10^{-7}$ of introducing an error per iteration, which yields 4 errors on average per execution.

A normal execution, without soft error detection and correction, of the calculation is performed before our prototype; in both we measure the execution time and also the number of

times it computed the correct value; that gives us the ability to compare and evaluate our two main aspects: how much performance overhead was induced and how much the success rate is improved.

In a machine with a Intel Core i7 6600U processor and 16GB of RAM (more than enough for our test functions), we repeated the process using 1,2 and 4 threads 500 times each. The average of all configurations is shown in the next images.

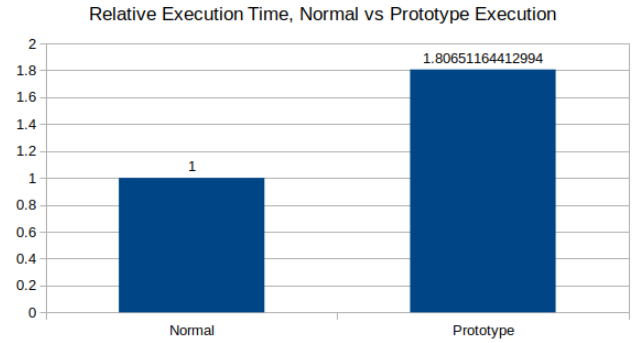

Fig. 1. Relative execution time of the normal executions vs our prototype.

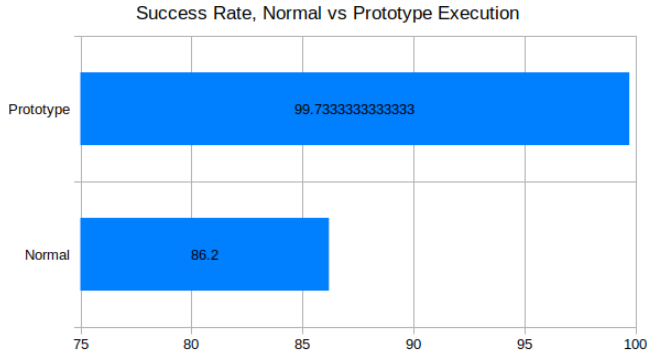

Fig. 2. Success Rate or the normal executions vs our prototype

In figure 1 we can see that the executions in general of our prototype induce an average overhead of 80.65% and in figure 2 we see that the success rate of our prototype nearly gets to 100%; this happens because, as stated before there are multiple reasons a transaction may abort and we only retry an iteration 5 times before trying again without transactions (hence if an error happens here it is not detected).

In general, these are preliminary which tells us we are down a right path but should and will be further detailed using more complex benchmarks such as PARSEC 2.0 [12].

VI. Conclusions and Future Work

Even though Hardware Transactional Memory was originally as an alternative for traditional synchronization in concurrent shared memory applications, it exposes a very efficient way to rollback a core's cache which can be exploited as a recovery mechanism, specially when dealing with soft errors. We have presented a prototype that begins to supports this idea through the preliminary results.

It seems most of the overhead incurred in our prototype lies on the detection part of dealing with soft errors; but a more complex analysis is required to fully measure how much time is spent detecting and correcting the faults.

The fixed probability to inject an error we use, may not be the most accurate in real life scenarios, specially because it varies depending on a number of factors already described; it gave us a good idea of how well the detection and correction phases behaved, but more research is required in order to establish the best way to simulate soft errors.

Another approach related to the discovery of error phase, is use checkpointing techniques instead of replication; which in [1] [3] tend to show that provide the best compromise between transparency for the developer and efficient resource usage. Through the combination of other features such as MPX and more improvements for Hyper-Threading may allow us to detect different kinds of errors and improve the performance or our technique

References

[1] D. Kuvaiskii, R. Faqeh, P. Bhatotia, P. Felber, and C. Fetzer, "Haft: Hardware-assisted fault tolerance," in *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 2016, p. 25.

[2] V. Sridharan, N. DeBardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi, "Memory errors in modern systems: The good, the bad, and the ugly," in *ACM SIGPLAN Notices*, vol. 50, no. 4. ACM, 2015, pp. 297–310.

[3] Q. Liu, C. Jung, D. Lee, and D. Tiwari, "Clover: Compiler directed lightweight soft error resilience," in *ACM Sigplan Notices*, vol. 50, no. 5. ACM, 2015, p. 2.

[4] M. Herlihy and J. E. B. Moss, *Transactional memory: Architectural support for lock-free data structures*. ACM, 1993, vol. 21, no. 2.

[5] P. Hammarlund, A. J. Martinez, A. A. Bajwa, D. L. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar *et al.*, "Haswell: The fourth-generation intel core processor," *IEEE Micro*, vol. 34, no. 2, pp. 6–20, 2014.

[6] J. Reinders. Coarse-grained locks and transactional synchronization explained. [Online]. Available: https://software.intel.com/en-us/blogs/2012/02/07/coarse-grained-locks-and-transactional-synchronization-explained

[7] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar, "Performance evaluation of intel® transactional synchronization extensions for high-performance computing," in *High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for*. IEEE, 2013, pp. 1–11.

[8] D. L. H. G. H. D. A. K. J. A. M. M. U. Deborah T. Marr, Frank Binns, "Hyper-threading technology architecture and microarchitecture," *Intel Technology Journal*, vol. 6, no. 1, pp. 4–15, 2002.

[9] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, C. Fetzer, and P. Felber, "Efficient fault tolerance using intel mpx and tsx," in *Fast Abstract in the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2016.

[10] R. (Intel). Introduction to intel memory protection extensions. [Online]. Available: https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions

[11] F. Haas, S. Weis, T. Ungerer, G. Pokam, and Y. Wu, "Poster: Fault-tolerant execution on cots multi-core processors with hardware transactional memory support," in *Parallel Architecture and Compilation Techniques (PACT), 2016 International Conference on*. IEEE, 2016, pp. 421–422.

[12] C. Bienia and K. Li, "Parsec 2.0: A new benchmark suite for chip-multiprocessors," in *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, vol. 2011, 2009.

[13] R. M. Stallman and G. DeveloperCommunity, *Using The Gnu Compiler Collection: For Gcc Version 7.0.1*. GNU Press, 2016.