Leveraging Modern Multi-core Processor Features to Efficiently Deal with Silent Errors

Diego Pérez*, Thomas Ropars‡, Esteban Meneses*†
*School of Computing, Costa Rica Institute of Technology
†Advanced Computing Laboratory, Costa Rica National High Technology Center
‡Univ. Grenoble Alpes, CNRS, Grenoble INP^{φ} , LIG, F-38000 Grenoble France

***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 corrupt the result of an expensive calculation. Because of that, techniques such as Instruction Level Redundancy 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 (HTM) exposes a very convenient and efficient way to revert the state of a core's cache, which can be utilized as a recovery technique. An experimental prototype has been created that uses such feature to recover the previous state of the calculation when a soft error has been found. The combination of HTM, Hyper-Threading and Memory Protection Extensions may further improve the performance, applicability and confidence of our technique.**

***Index Terms*—Hardware Transactional Memory, Hyper-Threading, Replication, Soft Errors.**

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 systems on a chip. These systems are more prone to transient errors compared to previous generations of processors, because manufacturers 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 damage 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.

There are two main challenges when dealing with soft 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. Instruction Level Redundancy (ILR) is usually used at some level in common solutions. Instructions are replicated and frequent checks are placed to detect faults. In the case of Triplication, two extra copies of the calculation are performed and a majority vote is used each time to decide the correct result. However, this technique is extremely costly adding typically about 200% of resource overhead [3]. Recent contributions [1] [4] tend to show that checkpointing techniques provide the best compromise between transparency for the developer and efficient resource usage. In theses cases only one extra copy is executed, which is enough to detect the error. But, still the technique remains expensive with about 100% of overhead.

To reduce this overhead, some features provided by recent processors, such as hardware-managed transactions, Hyper-Threading and memory protection extensions, could be 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 detection and rollback-recovery mechanisms by combining the use of mentioned features in current multi-core 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.

The rest of the paper is structured as follows. The next section describes some concepts that are necessary to understand properly the rest of the document; in Section III the background and previous work are presented; Section IV explains the details of the prototype that uses transactional memory as a recovery mechanism; in Section V shows some preliminary results of said prototype. Finally, the conclusions and future work are discussed in Section VI.

II. Concepts

A. Hardware Transactional Memory

Hardware Transactional Memory (HTM) follows the same principles as 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 remains 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 the main memory of a processor. When a transaction begins, the values of modified variables 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 these new results [5].

Transactional Synchronization Extensions (TSX) is the Intel version of HTM and was proposed as part of the Haswell

φ Institute of Engineering Univ. Grenoble Alpes

Instruction set architecture [6]. This document focus on multi-processors chips that have this feature, specifically on 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 non conflictive operations to occur without the delay that coarse grained locks would have injected [7].

Internally in Intel TSX, transactions have a read and write sets, to keep track of what has been consulted 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; having two transaction with 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's write set, at least one transaction aborts. If such thing happens (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 it is not its original purpose, Intel TSX also provides strong isolation guarantees and a way to rollback that can be utilized as a recovery mechanism. But, there are several design choices that limit the use of this feature for fault tolerance. First of all, Intel does not guarantee that a transaction will eventually commit, even when applied to sequential code [8]. 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]. Lastly there are "unsafe" 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 [9].

Hyper-Threading does not do much for single thread workloads, but when multiple threads can run in parallel there may be 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 [9].

C. Memory Protection Extensions (MPX)

Memory Protection Extensions from Intel include a set of primitives for pointers bound checking, as well as new registers for storing bounds data. Its original purpose is to check that memory references defined at compile time do not become a source of uncertainty at runtime due to buffer overflows or underflows (either way the bounds are violated). Such task is accomplished using new hardware features that can be used by software. MPX main goal is a way to protect memory deficiencies in unsafe languages [10].

Whenever a pointer is used, Intel MPX validates that the requested memory reference is inside the pointer's associated limits, hence preventing out-of-bound memory accesses. Buffer overruns account for a considerable amount of all bugs encountered in a typical C/C++ application [11].

III. Previous work and Background

This section presents and reviews how current solutions for soft error management take advantage of modern multi-core features. The vulnerabilities and/or limitations of each solution are identified in order to establish our path to follow.

Intel TSX's primarily target is synchronization and exhibits several design restrictions that make the use of this technique for recovery purposes not trivial [1]. Hardware Assisted Fault Tolerance (HAFT) [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 transactions 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 re-executed without transactions.

The best effort approach in HAFT of retrying a transaction a fixed 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 could be 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.

Oleksenko *et al* present another option to use Intel's version of hardware transactional memory as recovery mechanism against transient errors [10]. They focus on errors caused by bit-flips in data pointers, which can be catastrophic especially if the pointer is not to a basic type but to a more complex data structure, because it can potentially provoke a considerable amount of data loss. Once an error of this sort is identified, they explicitly roll back the current transaction using Intel's TSX feature.

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 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 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 [10].

MPX original purpose is a way to protect memory limitations in unsafe languages, like buffer overruns. Even though the idea of using it for fault detection in pointers is quite novel and ingenious, the authors in the paper [10] admit there are a lot of subjects not yet explored that they have to keep working on. Another drawback of the solution is that it only detects pointer faults, making it a technique which has to be used in coordination with another one in order to achieve a more complete fault tolerance.

Another reference where HTM is used as a recovery mechanism is [12], Fault Tolerant Execution on COTS Multi-core Processors with Hardware Transactional Memory Support. This is a software/hardware hybrid approach 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, since processes (not threads) are practically independent from each other. 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 process 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 [12].

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. Methodology

A prototype has been created that leverages Intel's Hardware Transactional Memory to recover from artificially injected errors and uses replication to detect them. Basically the following happens, there is 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. 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) a hardware memory transaction is initiated. If both executions of the calculation differ then the current transaction is explicitly aborted, 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, a transaction status is returned, which helps identify possible causes of the transaction failure. Some of these causes permit a transaction retry, for example if it was explicitly aborted. If a retry is possible the iteration is executed a fixed number of times (5 in the current implementation), before trying again without transactions. Otherwise, if no error was encountered, the next entry of the array is processed. 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 [1].

V. Results and analysis

This section measures two aspects regarding our prototype:

1) the total performance overhead.
2) the success rate (the number of successful runs despite soft errors).

For that, an instance (a large sum of integers) of our prototype was created where the total result being calculated is known 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. Knowing the correct result beforehand lets us detect if the execution was able to reach the expected result, or if an artificially soft error was injected and not corrected. On average 4 soft errors per run are introduced.

A normal execution, without soft error detection and correction, of the calculation is performed before running the

[1]GCC supports all transactional memory primitives of RTM: GCC-x86-transactional-memory-intrinsics

prototype. In both of them, it is measured the elapsed time and 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 introduced and how much was the success rate improved.

In a machine with a Intel Core i7 6600U processor and 16GB of RAM, the process is repeated using 1,2 and 4 threads 500 times each. The average of all configurations is shown in the next figures.

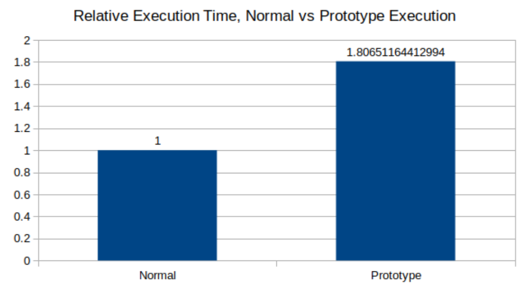

Fig. 1. Relative execution time of the normal executions vs our prototype.

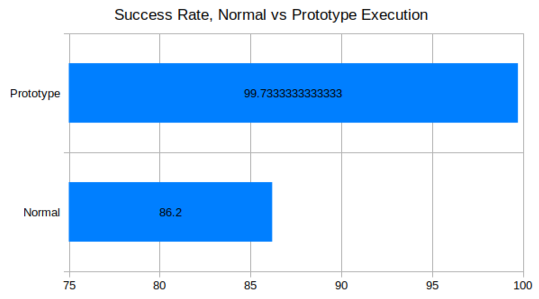

Fig. 2. Success Rate of the normal executions vs our prototype

Figure 1 shows that the executions in average of our prototype introduce a mean overhead of 80.65%; which is very likely to increase as the prototype is completed, since right now is not a generic technique for soft error management. In Figure 2 is clear that the success rate of our prototype does not get to 100%. This happens because, as stated before there are multiple reasons why a transaction may abort. Further, an iteration is retried 5 times before running it again without transactions (hence if an error happens here it is not detected). In general, these are preliminary results 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 [13].

VI. Conclusions and Future Work

Even though Hardware Transactional Memory was originally 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, especially when dealing with soft errors. The prototype built begins to supports this idea through the preliminary results.

In terms of future work, the average amount of soft errors injected per run may not be the most representative of real life scenarios. Although it provides a good idea of how the detection and correction phases behave, a more elaborate and complete way of soft error injection is necessary. FlipIt in [14] is an LLVM-based fault injection tool that can be used. It provides the ability to insert bit flips into instructions with a user defined distribution.

MPX can help detect soft errors that would otherwise be harder to discover using replication, such as those that happen in data pointers. The use of Hyper-Threading might be exploited in the error detection phase, performing the replication of the calculus. Thus, the combination of these two features would allow us to detect different kinds of errors and improve the performance of our technique, respectively.

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] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "Swift: Software implemented fault tolerance," in *Proceedings of the international symposium on Code generation and optimization*. IEEE Computer Society, 2005, pp. 243–254.

[4] 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.

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

[6] 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.

[7] 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

[8] 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.

[9] D. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty *et al.*, "Hyper-threading technology architecture and microarchitecture," *Intel Technology Journal*, vol. 6, no. 1, pp. 4–15, 2002.

[10] 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.

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

[12] 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.

[13] 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.

[14] J. Calhoun, L. Olson, and M. Snir, "Flipit: An llvm based fault injector for hpc," in *European Conference on Parallel Processing*. Springer, 2014, pp. 547–558.