A Review of Hardware Transactional Memory

Silvio Ricardo Cordeiro, Nicolas Maillard
Universidade Federal do Rio Grande do Sul
Instituto de Informática
{srcordeiro,nicolas}@inf.ufrgs.br

José Nelson Amaral
University of Alberta
Department of Computing Science
amaral@cs.ualberta.ca

Abstract

In the last two decades, Transactional Memories have been researched as a way of simplifying the implementation of concurrent programs, while at the same time guaranteeing that the resulting code will execute efficiently. In this paper, we discuss the landmark Hardware Transactional Memory implementations that have been proposed by the industry, as well as their intrinsic limitations. We also review related software-based tools that have been developed to deal with some of these limitations. Finally, we present the Syncchar performance tuning infrastructure and comment on possible future investigation of auto-tunable characteristics in transactional code.

1. Introduction

Programmers have always relied on the fact that successive hardware implementations would execute their sequential code faster than before. However, the hardware industry has reached the physical limits of heat dissipation in a processor chip. In order to maintain the performance improvements, computer architects decided that the best approach would be the replacement of this single-core processor design with an architecture that will be able to sustain multiple energy-efficient cores in a single chip [1].

Faced with the task of writing efficient multi-threaded applications, programmers will usually adopt some lock-based synchronization approach. Due to the inherent difficulties of shared-memory programming, one often employs locking primitives in a conservative fashion, using coarse-grained critical sections for the sake of code simplicity. These efficiency problems, allied with the peril of deadlocks, have led researchers to work on other synchronization techniques, such as Transactional Memories.

Transactional Memories (TM) were initially proposed [9] as a hardware-implemented synchronization construct that aims at being more abstract than the lock-based competitors. Memory access synchronization is performed by means of a transaction, which is an explicitly delimited sequence of operations that is guaranteed to be seen as indivisible by any other thread of execution.

The proposed implementations of Hardware TM (HTM) closely resemble the semantics of the LOAD-LINK (LL) and STORE-CONDITIONAL (SC) hardware instructions. When used in conjunction, the former behaves as a normal LOAD operation; whereas the latter behaves as a STORE if the target address has not been modified since it was loaded, failing if it has. These operations can be used by software libraries in order to implement high-level locking primitives.

Transactions in a TM system behave similarly to LL/SC in that, once their execution has started, it may fail at any moment due to conflicts with concurrently executing memory-access operations. Just as in the case of locking primitive implementations, the underlying application is required to restart the execution of the transaction until no conflict is detected and the committing operation succeeds.

The remainder of this paper is organized as follows: Section 2 presents the commercial Hardware TM implementations that have been proposed and implemented over the last years. Section 3 describes the most common constraints faced by Hardware TM designers. Section 4 briefly presents some current efforts toward making TM more efficient. Finally, Section 5 presents some conclusions.

2. Current Implementations

Many Hardware TM proposals have been made by researchers since its first proposal, but commercial implementations have only started to appear more recently.

2.1. Azul Vega

The first commercial implementation of Hardware TM is the Azul Vega machine [5]. This system features up to 16 processors (each with 54 cores) running Java applications on a 64-bit RISC architecture. This is a scenario where customers wish to write standard lock-based multi-threaded
Java programs and be able to use all 864 cores efficiently, so they built TM support directly in the Java Virtual Machine to accelerate lock operations [6].

In the Vega machine, each core has a private 16KB+16KB L1 cache and a 2MB L2 cache that is shared in groups of 9 processors. Hardware TM support was implemented in this system through the addition of two extra bits in each cache line: `speculatively-read` and `speculatively-written`. Due to the fact that no changes were performed on L2, cache misses always abort an executing transaction.

The instruction set defines three HTM instructions: `SPECULATE`, which starts tracking loads and stores; `ABORT`, which stops speculation mode and rolls back to the beginning of the transaction; and `COMMIT`, which stops speculation mode, either writing the data to main memory or aborting execution and rolling back to try again.

The Java Virtual Machine is implemented in such a way that, whenever it sees a lock, it decides whether to actually acquire the lock or to try to speculate that section of code in a transaction. Initially, it tries to speculate, but if speculative execution fails too often, it switches back to acquiring the given lock in the first trial.

Early experiments showed that memory access conflicts would restrict speedup over sequential execution to less than $1 \times$. Capacity overflow, on the other hand, was reportedly very uncommon, and did not impact the runtime efficiency.

### 2.2. Sun Rock

The Rock processor implemented a checkpoint-based architecture, where a checkpoint of the current hardware state can be taken at an arbitrary instruction. This was implemented as part of a scheme of improving the efficiency of executable code through the execution of the same software thread in two different physical threads. Whenever the first thread became blocked due to a high-latency operation (like a cache miss or a slow floating-point operation), the other one could speculatively execute the next instructions, and, if no conflict was detected, those operations could be committed [4].

The checkpoint-based design of Rock allowed it to easily support HTM, with a `CHECKPOINT <FAIL_PC>` instruction that explicitly began a speculative execution and jumped to `FAIL_PC` on abort. TM support was added for a restricted number of operations in 32 of the L2 cache lines; unsupported operations would automatically abort the transaction.

The Hardware TM implementation involved adding $k$ bits to each of the L2 cache lines, corresponding to the $k$ possible physical threads. Every speculative memory access sets the appropriate bit. Whenever a cache line is invalidated or evicted, the transactions that have accessed it are aborted. During the execution of the `COMMIT` instruction, cache lines that have been accessed by the current transaction are locked to prevent access from other threads.

Early research showed good results [8], but the project was canceled before release.

### 2.3. IBM BG/Q

In 2011, IBM announced Hardware TM support for their Blue Gene/Q processor [2], which is part of the Sequoia supercomputer. In this architecture, speculative state is stored in a multi-versioned 16-way L2 cache, such that each version of a cache line is stored in a different way.

The hardware organization was originally designed without TM support, in such a way that the L1 cache is unable to store speculative information. Since the HTM implementation buffers speculative state in the L2 cache, this processor implements two possible solutions to provide the correct interaction between L1 and L2, in the form of two modes of transaction execution. In the short-running mode, any store to an address will evict its line from L1, and subsequent access to this address will miss L1 until the transaction commits. In the long-running mode, TLB aliasing is used to version the address space in L1, allowing it to be used during the transaction, but requiring that the whole L1 cache be invalidated at the beginning of the transaction [15].

### 2.4. Intel Haswell

In early 2012, Intel released the specification for Transactional Synchronization Extensions (TSX), which are scheduled to be included in their Haswell architecture, due to ship in 2013 [13]. The proposed instruction set defines `XBEGIN`, `XEND` and `XABORT` operations, with no restriction on what instructions can be executed in the transaction [10]. Architectural details on the HTM implementation are yet to be made public.

### 3. HTM Limitations

Transactional Memory systems are frequently constrained by the choices that must be made in the architectural project. For example, Hardware TM poses an implementation dilemma regarding irrevocable actions, which cannot be rolled back (such as system calls). They must either prohibit some operations from ever being used in a transactional context, or roll back transactions whenever an offending action occurs. In the latter case, there must be a way to acquire some sort of global lock that guarantees the transaction will execute without any interference from other transactions.

A tradeoff also appears due to the fact that a Hardware TM system is required to store information about transac-
tional execution in a temporary memory (such as a cache or a special buffer), writing the data to memory when the transaction commits. Storing this speculative state in the L1 cache is a very efficient solution, but it is one that restricts transactions to fit this cache’s typically small size. Implementing conflict detection in L2, on the other hand, would increase the available speculative memory, at the expense of being slower and possibly more complicated. Similar problems are observed with the use of special buffer designs, with the added expense of introducing these buffers solely for the sake of the TM subsystem.

Some even more limitations can be seen on particular Hardware TM designs. For example, if each transaction is given a hardware identification, the amount of available IDs may be reduced to zero, which could force some TM systems to abort a transaction. The same would be true if the level of transaction nesting is too high in a given scope. Even the interaction with outside applications could affect the execution of a transaction, for example if the scheduling quantum is exceeded and a context switch happens before the transaction commits.

Aside from Hardware TM, there is also much ongoing research on Software TM, which lacks some of HTM’s deficiencies, such as memory usage restrictions. While it is still questionable whether Software TM could ever be efficient on its own [3], recent research suggests that Hybrid TM implementations systems could benefit from running applications on Hardware TM and falling back to Software TM on failure [7]. However, some effort is still necessary to efficiently adapt a TM application to a given Hardware TM system.

4. HTM Tuning

Over the last years, some attempts have been made at modeling the behavior of TM applications in order to help programmers improve their performance. The Syncchar tool [11] instruments TM applications and executes them in a simulator, in order to assist programmers in the identification of what portions of code should be modified to improve the performance of the application.

Syncchar executes its programs in the MetaTM [12] hardware simulator, which stores speculative state in L1 caches. The L1 caches are kept coherent with the XMESI [14] protocol, a modified MESI protocol with additional states for cache lines accessed in a speculative context. Whenever an irrevocable operation is executed, the transaction is aborted and re-executed after acquiring a global lock.

Due to all of the inherent shortcomings of Hardware TM, there is a strong margin of improvement through the use of auto-tuning that has not yet been widely explored. The Syncchar tool represents one of the first steps in this direction, modeling and predicting the behavior of TM applications, but further work is still required if one wishes to use this information to automatically tune user code.

5. Conclusions

In this paper, we have reviewed the Hardware TM proposals that have been undertaken by industry during the last years. These implementations are frequently accompanied by software-level tools that aim at supplementing the inherent limitations in their approach. Some of these works, such as Syncchar, can already model and predict the runtime behavior of TM applications. Future work is to be expected in order to apply this information in automated tuning of user code.

References


