

Transactional Memory: An Overview

Osman S. Unsal, Adrian Cristal and Mateo Valero
Barcelona Supercomputing Center,
Barcelona, Spain

Faced with diminishing returns (ILP) on increased investment (silicon real-estate and power) and hitting the brickwall of power density; CPU manufacturers are betting their futures on TLP and multi- and many-core computing. A new Moore's law is emerging which states that the number of cores on die will double in 18 months. However, a programmer productivity problem is looming on the horizon, and if one is not careful, we will hit this brickwall instead. The problem is that writing parallel code is very tedious now: currently locks are used to synchronize access to shared data which are tedious to use, difficult to understand and prone to deadlock. Transactional Memory has been recently proposed as a simple mechanism which will eliminate the problems associated with locks and make parallel programming easy for the masses.

Transactions replace locking with atomic execution units, so that the programmer can focus on determining where atomicity is necessary, rather than on the mechanisms that enforce it. For example, the following code segment shows an example atomic region in a simple kernel that computes the histogram of an array:

```
atomic {  
hist[array[i][j]]++;  
}
```

With this abstraction, the programmer identifies the operations that form a critical section, while the TM implementation determines how to run that critical section in isolation from other threads.

Typical TM implementations optimistically run transactions in parallel, assuming that the transactions won't perform conflicting memory accesses. Generally, "conflicting" means in violation of a temporal order. //Author: Correct?// Most often, a load (read) operation from an ongoing transaction has failed to use the result of a store (write) operation from a previous transaction. Here, "previous" usually has a serial temporal-ordering sense, although other logical orders are also possible.

If the transactions don't conflict, the optimism has paid off. The transactions did not have to contend for a common mutual exclusion lock for the data they updated. And, in many implementations, a transaction making read-only accesses to shared data allows all the data to remain in the core's data cache in shared mode, helping scalability.

However, if transactions do attempt conflicting accesses, the optimism has not paid off. The TM must abandon the work of one of the conflicting transactions, ensuring that any side effects of their attempted work are not visible to other threads, before reexecuting the abandoned transactions. The mechanisms by which the TM detects conflicts and contains the effects of abandoned transactions are the focus of the implementation techniques and case studies discussed in this article.

Compared with TM, locks are pessimistic. With mutual exclusion locks, only one thread can hold a lock at a time, whereas in most TM implementations, more than one thread can access a critical section simultaneously. Because actual conflicts are rare in many programs, the optimistic TM approach makes more sense as a future programming model.

The two main TM implementation styles are hardware based and software based. Historically, the earliest design proposals were hardware based (HTM). Researchers proposed software transactional memory (STM) to address the inherent limitations of HTM: resource limitations and adaptability. However, HTM has a considerable speed advantage when compared to STM. To have the best of both worlds, we argue that future hardware-supported TM-designs should address such concerns. We provide some examples below.

Accelerating STM: Hardware assistance can identify STM bottlenecks such as frequently aborting transactions through suitably appended hardware performance counters and increase the priority of the particular thread running this transaction so that the transaction can continue when the TM system aborts the transaction it is conflicting with instead.

Accelerating HTM: The cost of frequent aborts is prohibitive, there is simply too much wasted computation. To decrease the cost of the abort, checkpoints could be taken and which break the main transaction into smaller transactions, which we term an implicit transaction. The instructions between two consecutive checkpoints are considered part of one implicit transaction, which appears to the rest of the system as a single memory transaction. In particular, the system manages memory updates (store instructions) associated with a transaction as a group and performs them globally and atomically when the corresponding checkpoint commits.

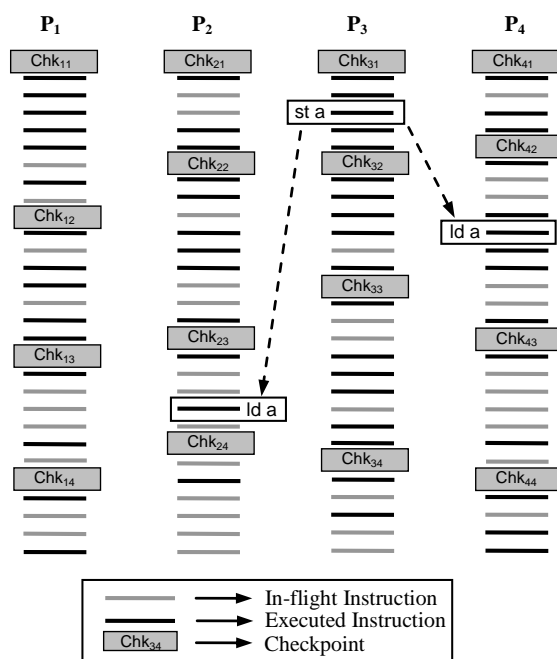


Figure 1 - Implicit Transactions checkpointing and conflict resolution mechanism.

Figure 1 shows an example of the execution flow for four processors and their respective checkpoints. A processor's oldest checkpoint can commit when all of its corresponding instructions—the ones that come after the checkpoint and before the next checkpoint—are finished. For example, processor P3 can commit checkpoint Chk31 when all instructions up to Chk32 have completed. Meanwhile, a general speculation substrate buffers the speculative read set and searches for conflicts with any committed update, causing a processor to roll back in case of violation. This mechanism guarantees correct memory consistency. For example, in Figure 4, the broadcast of a store (st) to memory location A conflicts with two other processors that have already speculatively loaded from location A, and the loads have not yet committed. In this example, P2 rolls back to Chk23, causing instructions from Chk24 to Chk23 to be discarded. Also, P4 rolls back to Chk42, forcing its newest instructions to be discarded.

