

Unbounded Transactional Memory

C. Scott Ananian, Krste Asanović, Bradley C. Kuszmaul, Charles E. Leiserson, Sean Lie

Background: Programming in a shared-memory environment often requires the use of atomic regions for program correctness. Traditionally, atomicity is achieved through critical sections protected by locks. Unfortunately, locks are very difficult to program with since they introduce problems such as deadlock and priority inversion. Locks also introduce a significant performance overhead since locking instructions are expensive and performing deadlock avoidance can be slow. In addition, locks are simply memory locations so there is an added space overhead associated with locking as well.

Hardware Transactions: To overcome the problems with locks, Herlihy and Moss proposed a hardware transactional memory (HTM) [1] scheme that gives the programmer a more intuitive atomicity primitive, a transaction. A transaction is an atomic region that either completes atomically or fails and has no effect on the global memory state.

Two regions are atomic if, after they are run, they can be viewed as having run in some serial order with no interleaved instructions. HTM ensures atomicity by simply running the atomic region speculatively. If no other processor accesses any of the same memory locations as the atomic region, the speculative state can be committed since atomicity has been satisfied. On the other hand, HTM must provide the mechanism to detect conflicting memory accesses if they do occur. In such a case, HTM will abort all the speculative state and restore the processor to its state right before the transaction started. The transaction can then be retried until there is no conflict thus ensuring atomicity.

Unbounded Transactions: Previous HTM schemes imposed a restriction on transaction size since the speculative state is stored in a hardware structure such as a cache or buffer with a fixed size. Previous HTM schemes simply aborted if such a limit was reached.

We propose that HTM support transactions of arbitrarily large size, while still ensuring that small transactions run efficiently. Our unbounded transactional memory implementation, called UTM, holds tentative updates in a cache but allows them to spill out of the cache into main memory if necessary.

Our studies show that, in the Linux 2.4.19 kernel, most atomic regions are small but some are quite large. In the kernel, 99.9% of the transactions touch less than 54 64-byte cache lines while the largest transaction touches over 7000 cache lines. Thus, we can automatically and efficiently “transactify” the Linux kernel only if unbounded transactions are supported. This evidence suggests we can afford a higher overhead mechanism to handle the large transactions as long as the common case is handled efficiently.

Implementation: We add the `xBEGIN` and `xEND` instructions to the processor ISA to mark the start and end of transactions. If the transaction commits, all transactional memory operations are performed atomically with respect to all other memory operations. If the transaction is aborted, all the memory operations performed in the transaction are discarded and the processor jumps directly to the abort handler which is given at `xBEGIN`.

UTM uses the processor cache as the primary storage for speculative transactional state. Unlike previous HTM schemes, UTM allows the transaction to continue running even when the cache is full. This is accomplished by allowing the transactional state to overflow from the cache into main memory. UTM effectively extends the cache into main memory when more transactional storage is needed. UTM overflows transactional data into an unsorted linear array data structure in main memory.

Transactional data is primarily stored in the cache as shown in Figure 1. Each cache line has an additional bit (T) marking whether or not it contains transactional data. On each transactional memory operation, the corresponding cache line is marked transactional. In addition, an additional bit (O) per cache set is added to mark if the set has overflowed.

Using the cache to store transactional state is convenient since the speculative state can be committed by simply clearing all transactional bits. The speculative state can also be aborted and rolled-back by simply invalidating all transactional lines. In addition, transactional conflicts can be detected using the cache coherency protocol without any changes. A conflict is signaled when an external cache intervention is received for a line marked transactional.

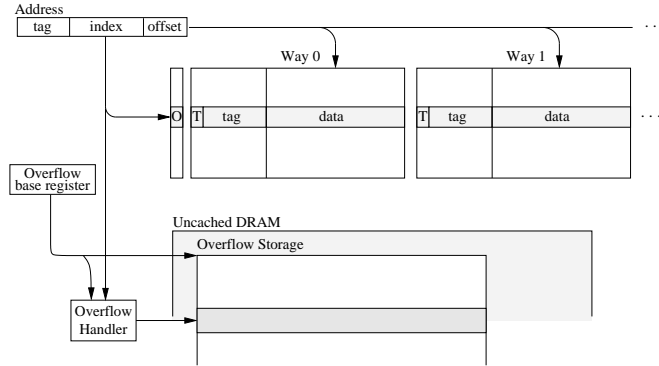


Figure 1: Transactional state storage. One additional bit (T) is added per cache line. One additional bit (O) is added per cache set. Transactional data is overflowed from the cache into uncached memory.

In addition to restoring the memory state after an abort, UTM also provides the ability to restore the processor register state. This is done by taking a register snapshot on each `xBEGIN` and then restoring that snapshot when jumping to the abort handler.

Evaluation: We implemented UTM in the UVSIM [2] multiprocessor simulator and measured 1-processor overheads for the SPECjvm98 benchmark suite. Our simulation results shown in Figure 2 show that the hardware changes impact the processor pipeline and speculative execution units minimally. In almost all cases, the overflow data structure was used but, in all but one case, UTM performance overhead is less than 10% over the serially executing non-transactional code. This is much lower than the overhead incurred by locks. However, in `213_javac` UTM overhead was almost 14x over the serial case. This unreasonable slowdown is caused by the fact that the entire benchmark code is wrapped in one huge transaction. Therefore, the overflow data structure is used much more than we expected. This result suggests that a linear search overflow handler is insufficient to cover all cases. We plan to investigate other data structures as part of our future work.

Benchmark	Base time (cycles)	Locks time (of Base)	Trans time	Time in trans (% of Trans time)	Overflow overhead
200_check	8,062,716	1.24x	1.02x	32.9%	0.00366%
202_jess	75,012,964	1.41x	1.07x	59.3%	0.00761%
209_db	11,784,220	1.42x	1.05x	53.9%	0
213_javac	30,688,574	1.70x	14.70x	99.0%	93.0%
222_mpegaudio	98,961,244	1.00x	1.00x	0.817%	0
228_jack	261,441,564	1.75x	1.04x	32.2%	0.00260%

Figure 2: SPECjvm98 1-processor performance and overheads. The *Base time* uses no synchronization. The *Locks time* uses locks. The *Trans time* uses UTM. The *Time in trans* is the time spent running a transaction. The *Overflow overhead* is time spent handling overflows.

Research Support: This research is supported in part by the National Science Foundation Grant ACI-032497, in part by the Singapore-MIT Alliance, and in part by a grant provided by Silicon Graphics, Incorporated.

References:

- [1] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Conference on Computer Architecture (ISCA)*, pages 289–300, San Diego, California, May 1993. ACM Press.
- [2] Lixin Zhang. URSIM reference manual. Technical Report UUCS-00-015, University of Utah, August 2000.