Hardware Transactional Memory on Haswell

Viktor Leis

Technische Universität München
transactional memory is a very elegant programming model

\[
\begin{align*}
\text{transaction} \{ & \\
& a = a - 10; \\
& b = b + 10; \\
\} \\
\text{Transaction 1} \\
\end{align*}
\]
Transactional Memory Implementations

- idea: keep track of read/write sets, abort on read/write or write/write conflict
- software transactional memory (hash table)
- Sun Rock: store queue (32 entries)
- IBM Blue Gene/Q: multi-versioned L2 cache (20MB)
- Intel Haswell: L1 cache (32KB)
Intel Transactional Synchronization Extensions

- Restricted Transactional Memory (RTM):
  - XBEGIN: begin
  - XEND: commit
  - XABORT: rollback

- Hardware Lock Elision (HLE):
  - XACQUIRE prefix: acquire lock speculatively
  - XRELEASE prefix: release lock speculatively
Cache Coherency

- cache coherency protocol is used to detect conflicts
- L1 cache serves as a buffer
- tracking is done at cache line granularity (64 bytes)
Limitations of Haswell’s HTM

- size (32KB) and associativity (8-way) of L1 cache limit transaction size
- interrupts, context switches limit transaction duration
- certain (fairly uncommon) instructions always cause abort
- no forward-progress guarantees (fallback to lock necessary)
The Problem with Locks

- 4-socket, 32-core Nehalem EX
- Lookups in an Adaptive Radix Tree (16M dense integer keys)
- Fine-grained read-write spinlocks
- No logical contention
- Lock acquisition causes write
- Cache coherency misses destroy performance ("cache line ping pong")
The Problem with Locks

- 4-socket, 32-core Nehalem EX
- lookups in an Adaptive Radix Tree (16M dense integer keys)
- fine-grained read-write spinlocks
- no logical contention
- lock acquisition causes write
- cache coherency misses destroy performance ("cache line ping pong")

![Graphs showing M lookups per second (throughput) and cache misses per lookup with and without latching and synchronization.](image)
HTM Performance

- 4-core Haswell
- lookups in an Adaptive Radix Tree (16M dense integer keys)
- lookups depend on each other
- no cache coherency misses with HTM
Transaction Granularity

- out-of-order execution complicates the picture
- independent lookups
- small transactions: memory fencing overhead
- large transactions: capacity limitations
Lock-Free Data Structures

- avoid the problem of cache coherency misses
- without HTM the lock-free synchronization (e.g., Hekaton) is necessary for good performance
- lock-free data structures are complex
- and often require additional indirections (e.g., delta records and page table for BW-tree)
Database Transactions

- HTM can replace fine-grained latching (without complex lock-free data structures)
- but we want ACID and transactions of arbitrary size
- idea: use HTM as a building block
HyPer

▶ www.hyper-db.com
HTM-based Concurrency Control

- split database transaction into multiple HTM transactions
- “glue” together HTM transactions with timestamp ordering
- TPC-C with 32 warehouses

![Graph showing transactions per second vs. multiprogramming level (threads) for different concurrency control methods: partitioned, HTM, serial, 2PL.](image-url)
Conclusions

- HTM is a new synchronization primitive that can improve performance and simplify the implementation at the same time.
- Very good fit with high-performance database systems.
References

- “Exploiting Hardware Transactional Memory in Main-Memory Databases”, Viktor Leis, Alfons Kemper, Thomas Neumann (forthcoming)
- David Kanter’s analysis: http://www.realworldtech.com/haswell-tm/