RETROSPECTIVE: Transactional Memory Coherence and Consistency

Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, Kunle Olukotun

Stanford University

I. Motivation

In the early 2000s, the scaling limitations of single-core chips were clear. With IBM, Intel, and Sun Microsystems working on their first multi-core chips, thread-level parallelism was quickly emerging as a key method for scalable performance in future systems of all sizes. Within this context, the *Transactional Coherence and Consistency (TCC)* project aimed to address two key challenges.

The difficulty of parallel programming: The cache-coherent, shared-memory model supported by multi-core chips makes it easy to communicate between concurrent threads. Nevertheless, programmers have to manage synchronization when multiple threads access potentially shared data, using primitives such as locks, mutexes, and condition variables. *Coarse-grain synchronization* often leads to reduced parallelism due to lock contention. *Fine-grain synchronization* is error-prone as it can lead to races, deadlocks, or livelocks. Finally, performance tuning is difficult as bottlenecks are linked to low level events, such as coherence misses and false sharing, which are not intuitive for most programmers.

The complexity of cache coherence and memory consistency: The hardware implementation of cache coherence involves tracking numerous pending events at the granularity of cache lines and reaching consensus on ownership across a wide range of corner cases. The hardware implementation of memory consistency involves tracking individual loads and stores as they are buffered and reordered throughout the system. Correct and performant implementations in systems with tens of cores are challenging. Relaxed consistency models are often the source of software bugs as many programmers do not understand the subtleties of hardware ordering rules.

II. The TCC Approach

The TCC project embraced *transactional memory (TM)* [8] as the key abstraction for managing parallelism. Transactional memory allows parallel threads to execute small chunks of code (*transactions*) in an *atomic* and *isolated* manner. All instructions in a transaction either complete as an atomic unit and commit their updates to shared memory, or are rolled back and re-executed. Concurrent transactions are isolated from each others' memory accesses, as if all transactions executed in some sequential order without any overlapping.

Unlike earlier proposals that used transactions sporadically to replace short lock sections, TCC proposed to *make transactions pervasive*. Each parallel thread becomes a sequence of transactions and there is no code execution outside of an atomic transaction. Hence, the motto of the TCC project was *"all transactions, all the time"*.

For programmers, pervasive use of transactions means that they no longer need to struggle with the granularity of synchronization in their parallel code. They simply mark transaction boundaries where their code naturally completes tasks that should update shared memory, for example at the end of purchase tasks on an e-commerce site. Transactions across threads can be *unordered* if the programmer starts with a parallel algorithm, or explicitly *ordered* if the programmer starts with a sequential algorithm. Transactions can be as coarse-grain as needed and programmers do not have to guarantee that they are independent. The hardware will roll back and re-execute any concurrent transactions that have true dependencies (read-after-write) on shared data. Performance feedback is provided at the level of transactions (program tasks) and shared memory variables.

At the hardware level, what made TCC fundamentally different from previous and later work on this topic is that *it replaced the conventional coherence protocol and consistency model with transactions*, rather than layering transactions on top existing coherence and consistency techniques. TCC enforces cache coherence at the boundary of transactions, updating shared memory at once for all cache lines written within a transaction. TCC enforces memory consistency at the boundaries of transactions, ensuring all processor cores and the threads they execute perceive the same transaction commit order. Both coherence and consistency become simpler and coarser-grain (bulk) operations that benefit from the high communication bandwidth available in multi-core chips.

The hardware implementation we proposed used *lazy buffering* and *optimistic concurrency control*. Each core privately tracked the set of addresses read and written by the pending transaction. The write set is sent to other cores lazily only when the transaction is ready to commit. All cores concurrently execute transactions, optimistically assuming that no true dependencies exist. When a transaction commits, all other pending transactions compare their read set to the write set of the committing transaction to validate that the optimism was correct. If not, the pending transaction with a true dependency to the committing one is rolled back and re-executes. A core rolls back its pending transaction by purging the transaction read and write set from its private caches. The original TCC proposal allowed one transaction commit at the time, but implemented optimizations such as double buffering to maintain high throughput. The paper showed that the private caches in processor cores and the bandwidth available in multi-core chips were sufficient to support the pervasive use of transactions for a wide range of popular benchmarks.

III. The Evolution of the TCC Project

With transactions as the only abstraction for managing parallelism, the TCC project had to explore their use and impact across the system stack. Over seven years, we pushed the boundaries of TM technology across several dimensions.

Hardware: We defined robust ISA-level semantics so that TM supports the breadth of functionality in programming languages and operating systems [10]. We designed scalable and cost effective TCC hardware and techniques to validate its correctness [5], [11]. We also used FPGAs to build two full-system TCC prototypes [4], [13].

Programming: We defined transactional programming for Java and OpenMP [1], [2], as well as optimized libraries and collectives [3]. We developed two popular TM benchmark suites and contacted several application studies [12].

Systems: We developed virtualization schemes for TCC hardware [6] and showcased how to use TM to build novel profiling, debugging, and security tools [7]

IV. Lessons from the TCC Project

The three key lessons from the TCC project were:

- *Focus on the programmers:* The primary focus for TCC and other TM projects was not performance gains over conventional approaches. Their goal was to help non-expert programmers develop, debug, tune, and scale parallel programs that make the most of emerging multi-core chips. Ease-of-use can be transformational for any technology.
- *The benefits of a clean slate:* In a community that focuses heavily on backwards compatibility, the TCC project took a clean slate approach towards harwdare support for parallelism. This allowed us to simplify *both* parallel programming and parallel hardware. An evolutionary approach would add significant hardware or software complexity.
- *Explore the full stack:* To get practical value from a clean slate hardware approach, we must think of the whole system stack – languages, libraries, compilers, debuggers, and operating systems. We need to address the challenges and exploit the opportunities at each layer.

These lessons are applicable to any project developing parallel architectures. In 2023, a main focus of our community is accelerators for machine learning. These accelerators take a clean slate approach towards parallelism and efficiency. As the key hardware approaches for fast matrix operations are now known, we see the focus gradually shifting towards ease-of-use and system issues such as compilation, partitioning, scheduling, memory management, communication, and tuning. At this point, the most popular ML hardware (GPUs) is not necessarily the fastest but is definitely the easiest to program and build large-scale hardware and software systems with.

V. The Future of Transactional Memory

The encouraging results from TM research in 2000s motivated IBM, Intel, and Arm to introduce transactional extensions in their commercial ISAs. IBM and Intel produced multi-core chips that implemented TM extensions such as IBM Power8 and Power9 and Intel Skylake. IBM later discontinued TM support. Intel did the same for client chips, which now include Atom cores that never supported TM. Intel still supports TM in server chips such as chips in the Sapphire Rapids line. While TM has some important uses, its adoption across hardware and software projects is not as widespread as we expected. Some reasons frequently quoted are hardware implementation complexity, programming and tuning difficulties, and unexpected interactions with side-channel leaks in multi-threaded cores. The commercial TM implementations are layered on top of the conventional and already complex coherence and consistency mechanisms. They also require programmers to put significant effort in defining small transactions that fit within the limited hardware support. In a sense, these implementations take a different approach from the TCC project that had hardware simplicity and ease of programming as its main goals.

Given the success of atomic transactions with concurrency management in databases, it is possible that transactional memory will make a comeback in the future. Several hardware features that we now consider essential, such as wide vector instructions, took a few tries to find their market fit. For the moment, ideas similar to those of TCC are popular in distributed systems. Researchers argue that all tasks that interact with shared state in scale-out services should execute as atomic transactions supported by fast distributed databases [9].

References

[1] W. Baek *et al.*, “The OpenTM Transactional Application Programming Interface,” in *PACT’16*, 2007.
[2] B. D. Carlstrom *et al.*, “The Atomos Transactional Programming Language,” in *PLDI’27*, 2006.
[3] B. D. Carlstrom *et al.*, “Transactional Collection Classes,” in *PPoPP’12*, 2007.
[4] J. Casper *et al.*, “Hardware Acceleration of Transactional Memory on Commodity Systems,” in *ASPLOS’16*, 2011.
[5] H. Chafi *et al.*, “A Scalable, Non-Blocking Approach to Transactional Memory,” in *HPCA’13*, 2007.
[6] J. Chung *et al.*, “Tradeoffs in Transactional Memory Virtualization,” in *ASPLOS’12*, 2006.
[7] J. Chung *et al.*, “Thread-safe dynamic binary translation using transactional memory,” in *HPCA’14*, 2008.
[8] M. Herlihy and J. E. B. Moss, “Transactional Memory: Architectural Support for Lock-Free Data Structures,” in *ISCA’20*, 1993.
[9] Q. Li *et al.*, “A Progress Report on DBOS: A Database-oriented Operating System,” in *CIDR’12*, 2022.
[10] A. McDonald *et al.*, “Architectural Semantics for Practical Transactional Memory,” in *ISCA’33*, 2006.
[11] C. C. Minh *et al.*, “An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees,” in *ISCA’34*, 2007.
[12] C. C. Minh *et al.*, “STAMP: Stanford Transactional Applications for Multi-Processing,” in *IISWC’08)*, 2008.
[13] N. Njoroge *et al.*, “ATLAS: A Chip-Multiprocessor with Transactional Memory Support,” in *DATE’07*, 2007.