# Research Statement

**Kevin Moore**

Throughout my education and my early career, I've been drawn to problems in both hardware and software. It was my desire to work on hardware that led me back to school after working for two years as a software developer. After joining a hardware-oriented research group as a graduate student, it was my experience with software that led me to my research focus—the interaction between hardware and software. Through my research, I strive to facilitate the development of highly scalable, reliable and robust software by making the interface between hardware and software as clear, efficient and powerful as possible.

The hardware/software interface is particularly relevant today because the computer industry's transition from uniprocessor chips to chip multiprocessors (CMPs) has profound implications for the development of software. In contrast to the circuit and architecture-level improvements of the 90's, the introduction of CMPs, which incorporate multiple processing units on the same chip, requires that software change in order to reap the benefits of the enhanced hardware. For a computer system to take advantage of even the dual-core processors common today, it must use parallel software. With plans for more parallel CMPs in the near future, software will need to not only be parallel, but to scale to larger and larger numbers of processors to keep pace with evolving hardware.

A pressing challenge in the movement to an explicitly parallel model of computing is to give programmers, compiler writers and language designers a more powerful means of synchronization. My doctoral research attacks this problem by investigating practical, high performance *transactional memory* systems. Transactional memory gives programmers the ability to declare the synchronization properties their programs need, without requiring that they develop a mechanism (e.g., a locking scheme) to enforce these properties. Furthermore, transactions may be nested to allow programmers to build thread-safe libraries without exposing implementation details such as locking conventions to higher levels of software.

Transactional memory systems may be implemented in software, in hardware or in a combination of the two. Typically, transactional memory systems do not limit the amount of memory a transaction may access nor the length of time a transaction may run. Though intuitive for programmers, that model is poorly suited for direct implementation in hardware, which is limited to structures of fixed size. Ideally, an implementation of transactional memory should: (1) make the common case—short transactions that commit—fast; (2) defer rare cases and difficult-to-implement cases to software; (3) allow transactions of any size and duration; and, (4) allow programmers to nest transactions to arbitrary depths.

## LogTM

In my doctoral research, I developed Log-Based Transactional Memory (LogTM), a transactional memory system that combines software-based *version management* (with limited hardware support) and conservative hardware *conflict detection* to support

arbitrary-sized transactions with limited hardware. Version management is the maintaining of multiple versions of data values: "new" values generated inside a transaction that remain if the transaction commits and "old" values that remain if the transaction aborts. LogTM performs version management by eagerly updating memory in place during transactions and saving old values in a per-thread transaction log. No further action is needed to commit a transaction since new values are kept in place. To abort a transaction, LogTM restores saved values from the log. Fortunately, (for most workloads) aborts are rare. LogTM leverages that rarity to reduce hardware complexity by performing aborts in software. Conflict detection identifies overlaps between the write set of each transaction with the read set or write set of other concurrent transactions. Like other hardware transactional memory systems, LogTM detects conflicts on cached data by augmenting the cache coherence mechanism, adding a read (R) and write (W) bit to each cache line. LogTM extends this mechanism with *sticky states*, additional coherence states in which the directory continues to send coherence messages to a processor for a block that it has already evicted from its cache. Receiving coherence messages for evicted blocks (in sticky states) allows a processor to conservatively detect conflicts for blocks not present in the cache.

Before I began work on LogTM, the only published hardware transactional memory systems relied heavily on the cache. The size and associativity of the cache limited the scope of transactions. LogTM's unique log-based version management combined with innovative sticky states allow it to break this dependence on the cache without adding complex hardware. Unlike previous schemes, which relied on keeping old values in memory and storing new values in the cache, LogTM stores new and old versions in separate memory locations, which may be cached and evicted independently. Like the stack, the transaction log is a part of a thread's virtual memory and is effectively unbounded. Sticky states, although part of the coherence mechanism, break the dependence between conflict detection and caching. By allowing conflict detection on blocks after eviction, sticky states enable transactions whose read and write sets exceed the capacity or associativity of the cache. Perhaps most importantly, LogTM guarantees that transactions appear atomic to applications, but it allows some lower-level software to observe transactions' intermediate values. This allows LogTM to involve software in maintaining the atomicity of transactions. LogTM uses hardware to perform the most performance critical tasks, tracking read and write sets and detecting conflicts, but leaves rare and complicated tasks, such as aborting transactions, to software.

## Extending LogTM

It has been my great fortune over the past two years to see my simple idea grow into a successful research project. What began as a lonely effort by my advisor and I is now a sizeable team, including several members who specialize in programming languages and operating systems. Our multi-discipline collaboration has allowed us to improve LogTM by extending it to support nested transactions and to allow thread switching and paging within transactions. This process led us to explore the precise semantics of transactions

(especially nested transactions) and to design mechanisms for invoking operating system services within transactions.

Closed nested transactions, in which child transactions commit atomically with their parent, facilitate software composition by allowing programmers to make library calls within transactions that may contain transactions themselves. Open nested transactions increase concurrency by exposing updates immediately upon commit, independent of the status of their parent transaction. My colleagues and I extended LogTM to support both open and closed nested transactions (Nested LogTM). Nested LogTM provides version management for nested transactions by segmenting the transaction log into a stack of log frames. Nested LogTM provides conflict detection for a few of levels of nested transactions by replicating the R and W bits in the cache. In the course of this work, we reasoned about the precise semantics for nested transactions, discovering conditions under which previously published TM systems produced unintuitive behavior. We further proposed non-transactional *escape actions*, which neither abort nor cause other transactions to abort, to facilitate the execution of system calls and irreversible actions.

A recurring problem in computer systems is the need to *virtualize* limited hardware resources by saving and restoring state in software. Hardware transactional memory systems maintain state in both conflict detection and version management that must be virtualized to allow transactions to span system events such as thread switching and virtual memory paging. LogTM, because it stores old values in virtual memory (the transaction log), virtualizes its version management. The R/W bits, however, pose a challenge to virtualization because those bits cannot be easily accessed by software. Additionally, they limit the depth to which transactions can be nested. To overcome these limitations, my colleagues and I developed LogTM Signature Edition (LogTM-SE), which decouples conflict detection from cache tag and data arrays.

LogTM-SE tracks read and write sets using compact Bloom filter-like *signatures*. Unlike R/W bits, signatures can be easily saved and restored by software. LogTM-SE supports unbounded nesting by saving the signature to the transaction log on each transaction begin (e.g., in the header of the parent transaction's log frame) and restoring the parent's signature on a transaction abort, or open transaction commit. Signatures may also be combined. LogTM-SE includes a *saved signature* per thread context, which represents the combined signatures of all suspended transactions in the current thread's process. All load and store instructions check their local saved signature to detect conflicts with suspended transactions. Software manipulation of signatures also allows LogTM-SE to support virtual memory paging in transactions—e.g., by adding a migrating page's new physical address to the appropriate signatures.

## Future Work

This is an exciting time to be a computer scientist. The transition to CMP is a fundamental shift in the computer industry. Now, more than ever, chip makers are interested in the interaction between hardware and software and are considering new interfaces, including transactional memory. As I enter the next phase of my research career, I am eager to continue promoting the development of efficient, scalable and

reliable software by both improving transactional memory and developing additional hardware mechanisms that will enhance the capabilities of software designers.

The biggest challenge I see to the adoption of transactional memory is to balance the generality of the interface against the complexity of the hardware. Based on my experience developing LogTM, I feel the key to meeting that challenge is for the hardware to provide simple, flexible mechanisms whose state is accessible to software. Developing LogTM, I held the view that an atomic transaction was not a primitive provided by the hardware, but a property guaranteed to applications by a transactional memory system comprised of both hardware and software components. Carrying this philosophy a step further, instead of providing *begin* and *commit* transaction instructions, the hardware could directly expose the internal mechanisms provided to support transactional memory. LogTM-SE, for example, provides hardware assistance for logging, tracking of read and write sets (in signatures), and conflict detection. These simple primitives will be easier to implement in hardware and, if exposed separately, may be applied by language designers and compiler writers in many ways, including, but not limited to, implementing transactional memory.

What we need most as we enter the era of many-core computing is highly concurrent software. In order to develop highly concurrent software using transactions, programmers are likely to need new tools to debug and optimize transactional software, hardware support for other synchronization techniques and a more relaxed model of atomicity such as open nested transactions. In my experience, converting lock-based programs to use transactional memory programs does not always immediately increase concurrency. Pitfalls, such as conflicts caused by false sharing and contention on shared resources (e.g., free lists), can erode the performance gains of transactional memory. Tools that report the frequency of transaction conflicts and aborts caused by contention on particular data structures will help detect bottlenecks in transactional memory programs. Furthermore, while transactional memory provides a simple replacement for mutual exclusion locking, it does not directly address other common synchronization paradigms, such as barriers, reductions and condition variables. Hardware support for these forms of synchronization will become increasingly important as transactional memory eliminates other synchronization bottlenecks. Finally, closed nesting in transactional memory can lead to serialization when long-running transactions access shared resources (e.g., by allocating memory). Open nesting can increase concurrency by raising the level of abstraction at which transactional semantics are guaranteed. Working with Nested LogTM, I saw that open nested transactions could dramatically increase throughput, but that its semantics were unintuitive in many cases. I believe that finding ways to help programmers reason about atomicity at higher levels of abstraction will bring us much closer to the concurrency we need to effectively utilize emerging many-core computer systems.