

## Position paper, TRAMP Workshop, February 2007

Michael L. Scott, Sandhya Dwarkadas  
Virendra J. Marathe, Michael F. Spear, Arrvinth Shriraman,  
Vinod Sivasankaran, Hemayet Hossain, Luke Dalessandro, and Aaron Rolett

University of Rochester

Transactional memory (TM) has emerged as a promising addition to the parallel programmer's toolchest, with the potential to reduce the semantic complexity of multithreaded code. Our goal is to develop a functional and easy-to-use interface without sacrificing performance or requiring special-purpose (TM-only) hardware. Clearly transactions need to outperform coarse-grain locks in most cases, since those already exist and are comparably easy to use. Nonblocking semantics are also desirable, though probably not essential. Perhaps most important, there must be a clear migration path for legacy systems; transactions must run well on existing hardware for the coming decade with minimum single-thread overhead.

Based on experience with library-based STM, we believe that language and compiler support will be essential for programming simplicity. That said, a library-level API provides an excellent vehicle for implementation experiments. Based on many such experiments, we believe that acceptable performance is achievable in all-software systems, but that it will depend on policy flexibility: mechanisms that do not adapt to application characteristics will miss crucial opportunities.

Our current work has two principal research thrusts. First, we have developed a runtime environment with a wide range of implementation alternatives. Our RSTM system embodies multiple metadata layouts; both blocking and nonblocking progress; eager, lazy, and mixed conflict detection; pluggable contention managers; initial-access or every-access validation of individual objects or whole read sets (with common-case optimizations); several strategies for safe privatization of objects no longer visible to other threads; and pluggable memory managers (with delayed reclamation when required). A principal goal is to evaluate tradeoffs among these alternatives, and to develop automatic adaptation mechanisms. A public release of much of our code is available at [www.cs.rochester.edu/research/synchronization/rstm/](http://www.cs.rochester.edu/research/synchronization/rstm/).

Second, we are developing general-purpose architectural communication mechanisms that are flexible enough to accelerate a variety of applications. In particular, they support both bounded and unbounded transactions without constraining policy in either case (leaving that up to software). We have proposed an *alert on update* mechanism that allows a thread to receive fast, asynchronous notification when previously-identified lines are written by other threads, and a *programmable data isolation* mechanism that allows a thread to hide its speculative writes from other threads until software decides to make them visible. We are currently developing mechanisms for low-cost conflict tracking. We expect to integrate these mechanisms with additional techniques for explicit management of caches on clustered multicore machines. We are also exploring the interaction of transactions with OS-level scheduling and other resource management.

Over the coming year we expect to devote increasing attention to a pair of additional challenges: large-scale application experiments and programming model development. The latter, we believe, must be driven by the former. We are eager to collaborate with other groups on the creation of transactional benchmarks and on APIs that will allow these benchmarks to run on multiple systems. We see several major open issues: (1) Should transactions be used for thread-level speculation, synchronization of explicitly parallel threads, or both? (2) How can we cleanly interoperate with nontransactional operations like interactive I/O and calls to legacy libraries or lock-based code? (3) Should transactions be completely transparent, with full roll-back and retry of operations on arbitrary data, or should transactional data and actions be explicitly identified, to enable faster implementations or user-level optimizations? (4) What type of hardware support will best accelerate transactions without slowing the processor's critical path?