In the context of a shared-memory approach to exposing thread-level parallelism, Transactional Memory (TM) appears to offer some potentially signficant advantages over traditional mechanisms for ensuring mutual exclusion (*e.g.*, locks). Nevertheless, there are many issues related to a realistic implementation that remain open challenges.

Evaluating Transactional Memory: TM's advantages with respect to locks can be categorized roughly into to two categories: those relating to performance (optimistic execution and fine-grain synchronization) and those relating to programmability (no need to associate a synchronization variable to critical sections, natural composition, fault tolerance, and avoiding priority inversion). To date much of the technical evaluation of TM has focused on the former, with very little attempt (to our knowledge) to quantify the impact of the latter.

We believe that the importance of the latter needs to be understood because much of the performance benefits of TM can be achieved using a much more straight-forward, evolutionary technique: Speculative Lock Elision (SLE). SLE is a hardware technique for performing an optimistic execution of conventional (*e.g.*, locked) critical sections with fine-grain conflict detection. In the presence of modest sized critical sections ($\leq$10k dynamic instructions) and modest contention, SLE will match or exceed the performance of any Hardware TM (HTM) system and outperform Software TM (STM) systems. Most importantly, however, SLE is very straight-forward to implement—it is already shipping in the Azul Systems Java Appliance— as unbounded "transactions" need not be supported and no source code changes are required for its use.

With the challenges remaining in how TM should be cleanly and efficiently architected (some of which we discuss below) and the significant software investment that will be required to enable its widespread adoption, it is import to understand the true programmability benefits of TM. This will require the involvment of software engineering experts and studies of programmer productivity. In such efforts, we suggest that in addition to comparing TM against locks, such work also considers SLE as another benchmark against which TMs benefits must be demonstrated.

Workloads/Benchmarks: The most glaring short term problem for TM research is the conspicuous lack of credible workloads. Much of the existing research on TM has used either microbenchmarks or programs that have been automatically transalted from conventionally-synchronized (*i.e.*, those that use locks). We strongly believe that these classes of programs are representative of only one behavior for which TM transactions will be used.

We expect two classes of transactions to be present in TM programs: 1) short transactions updating highly-contended data structures, and 2) large transactions observing little or no real contention. The existing benchmarks may correspond well to the first class, where the pressure to create short critical sections is present both with traditional synchronization as with transactions. We view this class of transactions to be less compelling from a TM research standpoint, as techniques like SLE appear to be quite effective for this class of critical sections (from both performance *and* programmability standpoints). In the latter case, however, the lack of contention permits transactions to grow without negatively impacting concurrency. Especially given that the most compelling motivation for TM (in our minds) is the potential for composition of synchronized libraries, these larger transactions represent a more important case in which to evaluate TM systems. Furthermore, as TM is intended to facilitate parallelization, ideal workloads would be ones that have historically been sequential (because of the difficulty of traditional parallelization), but whose parallelization TM enables.

We tried to push the development of TM workloads through organizing a *Workshop on Transaction Memory Workloads* in conjunction with PLDI 2006. A surprising outcome from this workshop was that there was a lack of consensus on relatively fundamental features of a TM programming interface that would preclude the production of "portable" TM benchmarks[1]. Reaching such a consensus would be a significant step toward a portable TM benchmark suite.

TM and Legacy Code: Applications are rarely written from scratch. Even new programs written with transactions will likely incorporate a significant amount of conventionally-synchronized code, which is un-

[1]The second notable outcome was that, in the most compelling of the proposed TM workloads (a Delaunay mesh generator), achieving good performance required restructuring the application in ways that effectively obviated the need for TM.

likely to be re-written to use transactions. While the notion of automatically "transactifying" locked code, at least three impediments exist to automatically translating such code to use transactions: 1) doing so naively can potentially prevent forward progress (a subject I'll leave for Milo Martin), 2) transactions only provide isolation, locks can be used for synchronization, coordination, mutual exclusion, etc. and 3) critical sections may contain side-effects. Alternatively, can transactional code and traditionally-synchronized code be composed while ensuring the isolation properties specified by each? Naively, we can ensure safe composition by preventing transactions and locked critical sections from concurrently executing, but is there an approach that does not restrict concurrency? Alternatively, can existing static analysis techniques be extended to verify when such a composition will be safe without intervention?

TM and I/O: An obvious limitation for TM is its inability to handle side-effecting operations. Entirely precluding I/O from transactions is unlikely to be a realistic solution, as I/O regularly occurs within conventional critical sections. A number of approaches for handling I/O within transactions have been proposed, each with its own limitations. We believe that these approaches are complimentary in nature and, based on data we've collected about the use of I/O in critical sections, we believe that TM systems will likely require supporting multiple approaches so that the most appropriate can be selected for a given instance.

Ensuring TM virtualization: We believe that some form of hardware support for TM is inevitable, as the slowing growth of single-thread performance will make the overhead of software-only approaches uncompelling. What form this hardware will take remains an open research topic. An important constraint for how this hardware is implemented that has not received significant attention is ensuring that it can be virtualized, so that it can be used in conjunction with hypervisors/virtual machine monitors.

HTM Building Blocks: A number of mechanisms proposed for the implementation of HTM systems (*e.g.*, register checkpointing, isolated speculative execution) have potential applications outside TM. Two examples include the aforementioned SLE as well as providing atomic regions to improve compiler optimizations, permitting the compiler to generate speculatively-optimized versions of code that abort themselves when their optimizations do not apply, rolling back and transferring control to a less aggressively optimized version of the code. When architecting TM, it will be important to consider these additional applications to maximize the return on the hardware complexity invested.