

Data-centric Synchronization

Jan Vitek Frank Tip Mandana Vaziri

Software Transactional memory is a programming abstraction that aims to facilitate the task of programming concurrent systems by reducing opportunities for synchronization errors, and yielding scalability improvements. This position paper lists research directions that we would like to pursue.

Data-centric Synchronization: TM does not offer assistance to developers in selecting *where* transactional boundaries should be placed in the code. Just as with locks, it is easy to forget to protect access to a variable or select an overly coarse transactional boundary. *An alternative approach is to use declarative specifications of data consistency drive the process of selecting transactional boundaries.* In the paper on *atomic sets* Vaziri et al. showed how to generate most of the synchronization policy for simple Java data structures from declarative specifications. Open questions are whether the approach scales to large systems, how to integrate atomic sets with transactional memory, and what support in the language and type system is required for an efficient implementation.

Value Types: Immutable types are an important enabler for efficient implementations of transactional memory as they do not require special treatment by the compiler. We are specially interested in *deep-immutability with purely functional semantics.* Adding value types to Java entails ensuring that (1) `this` references do not escape during object initialization, (2) reflection native calls do not modify final fields, (3) ensuring that all methods invoked on value types do not have side effects.

Type-based Isolation: The use of monads in Haskell allows for a clean separation between transactional code and non-transactional. This, together with the purely functional nature of Haskell, has made it possible to implement software transactions efficiently. Type-based isolation is possible in object-oriented languages by relying on approaches based on ownership types. *An ownership type system can be used to delineate transactional boundaries where strong atomicity needs to be enforced. Outside of the ownership domain, we would provide either no transactional support or, at best, weak atomicity.*

Mixed-mode synchronization: Transactional memory can cover only a portion of concurrent programming idioms. Currently, there is little in the way of integrating transactions with monitors, message-passing, barriers, etc. *We need to model the interaction between transactions and other synchronization mechanisms.*

Transparent Performance models: With lock-based synchronization, the performance of a well-synchronized program is at worst that of the serialized computation running on a single processor with very little additional memory requirements (for the dynamically allocated monitor objects). With a STM, performance is a function of the work performed within a transaction – an unlucky choice of transactional boundaries can cause both space and time requirements of the computation to increase significantly. We are thus faced with a tradeoff between buffering and blocking. *A transparent performance model for STMs is needed to give developers a clear view of the impact of transactions on the non-functional characteristics of their code.*

Workloads and benchmarks: To date, empiric evaluations of STM implementations have suffered from the lack of realistic benchmarks. Measuring performance in an overly simplified setting can be at best uninformative and at worst misleading as it may steer researchers to try to optimize irrelevant aspects of their implementations. Preliminary work on *STMBench* at EPFL and Purdue has uncovered some performance issues in existing STMs and usability issue with some proposed APIs, but more research is needed to create a rigorous benchmark suite with realistic workloads.

Experimental platform: One of the key challenges that we face in this area comes from the lack of research infrastructure for experimenting with STM technology. The requirements for a Java STM platform are the following: (1) *Competitive performance* – performance and scalability are key factors for the adoption of TM, there is little point in studying transactional implementations in a setting that is not performance competitive; (2) *Customizability* – the infrastructure must allow for customization and extensions; (3) *Open source* – the infrastructure should be accessible to academic partners.