Software transactional memory (STM) can lead to scalable implementations of concurrent programs, as the relative performance of an application increases with the number of threads that support it. However, the absolute performance is typically impaired by the overheads of transaction management and instrumented accesses to shared memory. This often leads a STM-based program with a low thread count to perform worse than a sequential, non-instrumented version of the same application.

We propose FastLane, a new STM system that bridges the performance gap between sequential execution and classical STM algorithms when running on few cores (see Figure 1). FastLane seeks to reduce instrumentation costs and thus performance degradation in its target operation range. We introduce a family of algorithms that differentiate between two types of threads: One thread (the master) is allowed to commit transactions without aborting, thus with minimal instrumentation and management costs and at nearly sequential speed, while other threads (the helpers) execute speculatively. Helpers typically run slower than STM threads, as they should contribute to the application progress without impairing on the performance of the master (in particular, helpers never cause aborts for the master’s transactions) in addition to performing the extra bookkeeping associated with memory accesses.

FastLane is implemented within a state-of-the-art STM runtime and compiler. Multiple code paths are generated for execution: sequential on a single core, FastLane (master and helper) for few cores, and STM for many cores. Applications can dynamically select a variant at runtime, depending on the number of cores available for execution. Preliminary evaluation results indicate that our approach provides promising performance at low thread counts: FastLane almost systematically wins over a classical STM in the 2-4 threads range, and often performs better than sequential execution of the non-instrumented version of the same application (see Figure 2).

The synchronization of the master and helper threads is based on a shared counter that keeps track of the progress of the master and ensures mutual exclusion with helpers trying to commit their changes. The counter is odd when it is owned by a transaction, and even otherwise. We use an array of monotonically increasing integers to protect a set of memory addresses. Each memory address is mapped to one entry in the array (by hashing the address modulo the size of the array). The entry contains the value of the counter at the last time the address was written. The master acquires the counter at transaction begin and releases it at commit. Additionally it requires only instrumentation for store operations that reflect the update by writing the current counter value into the corresponding array entry.

Helper threads execute speculatively and therefore postpone the counter acquisition until commit time. They require instrumentation of loads to detect inconsistent values that were updated after its transaction started. Stores also check for inconsistencies and buffer their updates until commit. We evaluate several commit variants for helpers (described in Figure 3) that differ in...
the validation strategy: (1) validate in isolation after acquiring the counter, (2) wait until the counter becomes unowned to obtain an even value and pre-validate, acquire the counter and re-validate if the counter changed in the meantime because another transaction committed, (3) obtain an even counter and validate, then try to acquire the counter using the obtained value or start over if the acquisition failed.

Our current focus is on evaluating further optimizations of the algorithms such that we can improve performance in certain conditions.

One interesting optimization and tuning aspect is the interference between master and helper threads. While the master thread should not degrade in throughput compared to the sequential baseline when contention is increased, the helper threads should have a chance to commit their share of the workload for better scalability. Our approach is to reduce the accesses to transactional metadata, i.e., the counter and the shared array. An optimization for the master is to increment the counter and release it only if it is requested by an helper thread that wants to perform an update transaction. Helper threads can be optimized by shifting the consistency check for writes to the commit time with the drawback of detecting conflicts later. Also the validation can be skipped when no other thread committed concurrently. First results show that a good tradeoff can be achieved that enables FastLane to compete in scalability with low thread counts even in high contention workloads.

We are also interested in exposing the master thread as a pessimistic feature to solve irrevocable operations that cannot be executed within speculative transactions, e.g., external actions or I/O. The roles of master and helper can change dynamically during execution of the concurrent application and require great care to minimize the performance penalty on other threads. Compared to the quiescence modes of STM algorithms that stop all other threads, our approach allows helpers to continue their execution in parallel to irrevocable operations. Helper threads can also request to become the master when they recognize a lack of progress due to high contention, e.g., after aborting and restarting several times. One of our directions is to automatize the selection of the master and helpers threads at runtime, and allow fairness of allocation of the master role between threads during the execution.

An earlier version of this work with the basic algorithm appeared in [3].

References

