# Lifting the Barriers – Integration of Monitors into a Distributed Transactional Memory System

Annette Bieniusa
University of Freiburg, Germany
bieniusa@informatik.uni-freiburg.de

Thomas Fuhrmann
Technical University Munich, Germany
fuhrmann@in.tum.de

Our work aims at exploring the potential of virtual machines (VM) that can provide a single system image across a potentially large number of heterogeneous processor cores. DecentVM [1] is a fully decentralized, proof-of-concept interpreting VM, which runs on clusters of many-core processors as well as in networks of embedded processors. Its memory structure reflects the requirements for consistency and coherence needed in multi-threaded applications. The primary consistency model in the DecentVM is transactional memory [2], but it offers backwards compatibility and integration with Java's synchronization techniques: monitors and volatiles.

In this talk, we discuss how transactional memory can carry over to code that uses Java's synchronization means such as monitors and volatiles. The Java memory model (JMM) specifies when write operations have to become visible to other threads. Cache coherency protocols – typically implemented in the processor hardware – further help to keep up the illusion of a single, system-wide control flow. It further defines the relation between thread-local operations and the system-wide behavior. In particular, it defines that the VM and its just-in-time (JIT) compiler must not reorder instructions that access volatile fields or monitors. On some processor architectures, the JIT compiler must even insert specific memory barrier instructions to keep the processor from reordering access to volatile fields.

Transactional memory (TM), on the other hand, defines explicit barriers for publishing modifications on shared data data. Usually, the updates becomes only visible after the transaction has successfully committed. Thus, ideally, a thread operates solely on its private cache, which is atomically written back to shared memory when the transaction commits.

## Mapping Java synchronization to transactions

Monitor enter and volatile load instructions both constitute the same kind of memory barrier which we call LOAD instructions. Monitor exit and volatile store instructions constitute another kind of memory barrier, which we summarize as STORE instructions. In the instruction sequence as executed by one thread, LOAD instructions can be moved up, i.e. to an earlier position (*prefetching*) whereas STORE instructions can be moved down, i.e. to a later position (*delayed write*), provided they maintain the program order.

In particular, the instructions can be reordered so that they form blocks that begin with one or more LOAD instructions and end with one or more STORE instructions. All other instructions such as non-volatile load and store instructions are executed inside these LOAD-STORE blocks (Fig. 1).

Even though the Java specification requires the VM to synchronize each access to a volatile field, it does not require the scheduler to schedule other threads so that they actually see each write access, or that they can modify a field between two read accesses. Thus, for each LOAD-STORE block the processor can first load all

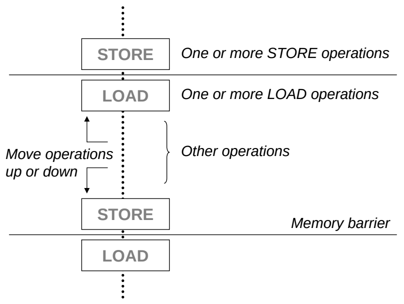
STORE
One or more STORE operations
LOAD
One or more LOAD operations
Move operations up or down
Other operations
STORE
Memory barrier
LOAD

**Figure 1.** Load-store-blocks and memory barriers.

required data, then process that data, and finally publish all modifications. This execution scheme is similar to an atomic transaction, but the JMM does not require the load and the publish action to be atomic. Thus, in the system as described so far, there are no read or write conflicts. In general, this ability to bundle instructions is not limited to one LOAD-STORE block. However, when extending this scheme to multiple blocks, conflict detection needs to be performed and rollbacks must be triggered if the mutual exclusion or visibility guarantees of the JMM are violated.

To show the correctness of our approach, we develop a hierarchy of formal models with the standard JMM implementation, single LOAD-STORE blocks, and multiple LOAD-STORE blocks with transactional execution, and prove their semantical equivalence.

## Integrating I/O

To treat the occasional rollback of critical sections, we assume buffering structures for (almost) all I/O devices. These buffers are treated similarly to all other memory objects – in fact, they are memory objects, which are only modified in form of their thread local copies. When a thread rolls back, no output has to be undone because nothing has been actually written to any device, and, conversely, all input can be processed again, as if it had not been touched before. Since I/O devices are shared resources, they are typically already protected with monitors so that their access induces a synchronization barrier.

***Acknowledgements*** This work was supported by the German Ministry of Education and Research under grant number 01IH08011.

## References

[1] A. Bieniusa, J. Eickhold, and T. Fuhrmann. The architecture of the DecentVM: Towards a decentralized virtual machine for many-core computing. In *VMIL '10*, pages 5:1–5:10, 2010.

[2] A. Bieniusa and T. Fuhrmann. Consistency in hindsight: A fully decentralized STM algorithm. In *IPDPS*, pages 1–12. IEEE, 2010.