

The Pros and Cons of Transactional Memory

**Rajesh Bordawekar, Trey Cain, Calin Cascaval, Siddhartha Chatterjee,
Maged Michael, Xiaowei Shen, Christoph von Praun
IBM Research**

The benefits of transactional memory are well-known: transactional memory primitives will allow CS 101 students to program irregular algorithms on massively parallel machines, with perfect scaling. Or at least come close... right?

Although our outlook on transactional memory is nearly this rosy, there remain many challenges that must be addressed before its widespread adoption. For example, TM must co-exist with conventional locking primitives, and must support modular software engineering practices, where a function call within a transaction may end up in a library performing I/O, surrounded by several intermediate levels of nested transactions and conventional locks. In terms of composability, the straightforward conversion of locks to transactional concurrency control (as frequently used for the evaluation of TM systems) may be insufficient to achieve high scalability and may be unrepresentative of the common use of transactions. TM is not a panacea to solve all parallel programming issues; if an algorithm contains little concurrency, transactional memory will not increase its performance. The challenge of supporting large transactions without negatively affecting programmer productivity also remains. Despite these issues, the performance and productivity benefits of TM remain compelling (compelling enough to warrant the organization of this workshop!).

Primitives not Solutions

A majority of papers on Transactional Memory seem to have ignored the basic set of principles for defining the interface between hardware and software (ref. William Wulf's seminal paper "Compilers and Computer Architecture") Wulf argued that it should be possible to divide a machine's definition into a set of separate concerns and define each in isolation from the others (i.e. orthogonality). He also expressed that it was always better to provide good primitives from which solutions could be synthesized, rather than the solutions themselves.

Because those who don't know history are destined to repeat it, similar arguments are now occurring on the subject of transactional memory. Luckily, transactional memory *can* be divided into a set of independent concerns, e.g., atomicity, register checkpointing, conflict detection, nesting, transaction logging, and recovery processing. These concerns can be further decomposed into sub-problems. While there has been some recent work on decomposing TM concerns into sub-problems (for example Saha et al. from Micro-06) most conference papers have attempted to build complete solutions that intermix all of these orthogonal issues.

Our approach to the TM problem involves identifying key orthogonal concerns, and tailoring solutions to each using a combination of software and hardware. In terms of acceptance, the TM interface must be simple while also being amenable to evolution over time, as unforeseen requirements and uses arise. This flexibility implies that software must be used when possible, and hardware added only when performance dictates. Consequently, the hardware primitives should be agnostic to high-level language choices (e.g. open nesting vs. closed nesting). This evolution will be similar to the evolution of instruction set architectures (recall that there were once advocates of implementing FORTRAN keywords in hardware, rather than synthesizing their semantics from primitives).

Decomposing the problem into a set of primitives may also have other advantages. Once decomposed, it may also become clear just how many other problems exist which could be greatly aided by one of these primitives. Reverse-debugging, application checkpointing, and speculative multi-threading may all benefit from the set of primitives that are designed specifically for transactional memory. Who knows what else?

Alternatives to Transactional Memory

As applications and algorithms are tuned for parallelism, other models of scalable shared memory programming and concurrency control will become common. Examples of such mechanisms today include non-blocking algorithms and data structures, read-copy-update, alert-on update, and copy-on-write. Although not as general as transactional memory, in certain situations these models provide a scalable alternative to locks. Usage scenarios of such mechanisms are largely orthogonal to transactions; in some cases efficient implementations are possible on current systems or could be supported with generic architectural primitives that are equally useful to support TM. With the advent of optimizing runtime systems, lock implementations can be dynamically tailored and customized using such primitives. In such dynamic compilation environments, with structured workloads that exhibit little or no contention, the additional benefits provided by transactional memory may be negligible.