

# *The Challenges of Extended Transaction Semantics and of Generating and Managing Concurrency and of Developing Hardware*

**Tony Hosking and Eliot Moss**

In the current surge of interest in applying transactional approaches to programming applications running on increasingly parallel hardware, we have tried to make the case for the necessity of open nesting, or something like it. In brief, the case boils down to two distinct arguments:

- 1) Need for greater concurrency than can be achieved by flat or closed nesting, i.e., than can be achieved by operating at the level of semantics of memory units (bytes, words, cache lines, pages).
- 2) Need to go beyond strict serializability, for a number of reasons, one of which is to accomplish certain updates or actions permanently even if a containing transaction aborts, and another of which is applications that need greater control of ordering of actions, as opposed only to non-interference.

Open nesting, for which we have proposed memory level semantics and also language syntax and semantics for Java, is one approach to addressing these needs, but raises some important concerns:

- 1) How difficult is it to write suitable inverses (compensating transactions)? (We actually feel that for many, perhaps most, actions this is not that difficult, especially if extended transaction semantics are applied in a consistent way at the level of abstract data types, i.e., it is about classes and the abstractions they present, not actions.)
- 2) How difficult is it to express and manage concurrency control for the extended model? This is an area where we would agree that significant additional work is likely to improve programmability and likelihood of correctness. We wonder to what extent more powerful analyses, approaching the level of theorem proving (e.g., rely-guarantee approaches, invariants plus pre- and post-conditions, directly expressing a class's abstraction map to enable arguing about abstract effects and concurrency, etc.)
- 3) Can we guarantee properties of implementations, e.g., freedom from deadlock and from livelock, etc. (This is closely related to (2).)
- 4) Is our argument sound: that the extended model needs to be used exclusively or primarily by skilled developers of packaged libraries, and does that reduce the level of concern?

A second area that has received only limited attention to date is how best to generate (express) enough concurrency when writing at the language level (or perhaps in hardware). Explicit thread models as in Java are unwieldy and difficult to use to express concurrent algorithms in ways independent of the currently available degree of parallelism, and that work well in the face of varying resources (e.g., varying number of available cores, or adjusting degree of parallelism to degree of contention/conflict). It is not clear what models (do-all across data structures, fork-join, futures) will help here, or whether we might even need some new ones. We may be able to take some inspiration from the implicit parallelism in bulk data operations of databases, but our different context suggests that such parallelism will not address all the needs.

Further than the issue of models is the issue of actually managing concurrency -- of deciding,

dynamically, on the appropriate degree of parallelism, and of managing performance-related issues such as locality (sometimes it is better to wait and use a currently busy local core to work on data that is already in its cache than to use an idle remote core that will suffer cache misses to get started, etc.).

The third challenge area to the field is developing a suitable hardware model. At the current stage of evolution of designs, we would guess that a hardware-assisted or hardware-accelerated model would be the best outcome. One challenge is not to commit too much to hardware in the sense of "burning in" particular semantics where we may need more variety in the long run. Another challenge is to develop designs whose performance regimes do not exhibit drastic changes, but are more moderate. For example, having orders of magnitude difference between a small transaction and a slightly larger one is undesirable. Another significant challenge, which may fall more in the area of legal than technical, is having enough of a standard (whether or not it is formal) that applications can be portable, while each manufacturer can take its own approach to address issues of capacity, speed, etc. The "holy grail" here is perhaps more like the hardware assist we see for virtual memory (TLBs, a range of hardware versus software support for page table walking, etc.) than like the IEEE floating point standard (which is very detailed at the bit level).