Transactions have long been used as an organizing principle for concurrent operations. Originally formulated as a database concept, Knight, Katz, and Tinker used transactions to speculatively parallelize sequential code [Kni86, TK88, Kat89]. Herlihy and Moss later introduced transactions into the hardware memory system [HM93] in order to efficiently implement Lamport’s lock-free synchronization [Lam77]. Pure-software implementations of transactional memory followed [ST95, HLM03]. Rajwar and Goodman applied hardware transactional memory to speed up traditional lock-based programs [RG02].

As a strictly database concept, transactions have appeared in the E programming language [RCS93] and other specialty database languages. General-purpose languages acquired linguistic support for lock-based transactions and atomic actions in Argus [LS82, WL85, LCJS87, Wei90] and Avalon [Ler88, DHW88]. Transactions have been factored into persistence, undoability, locking, and threads as first-class objects in a Standard ML of New Jersey (SML/NJ) implementation [NW92, HKM+94].

In the database community, a variety of extensions beyond simple single-level transactions exist; for example, chained, concurrent, and overlapping transactions. Moss proposed nested transactions as an organizational principle for distributed systems [Mos81]. Nested transactions allow concurrency control within transactions by appropriately serializing subtransactions, and permit parts of a transaction to fail without necessarily aborting the entire transaction.

The goal of this project is to extend previous research by integrating non-blocking multiple-level transactions into a modern object-oriented language, Java. In addition to using these for synchronization (as per Herlihy, Rajwar), non-blocking multiple-level transactions ought to be able to be used to express optimistic and speculative concurrency (Knight). Finally, the “undoability” properties of transactions ought to allow their use for fault-tolerance: a crucial question here is how irreversible external I/O within a transaction ought to be handled.

I propose to design a non-blocking multiple-level transaction system for Java and use it to safely implement speculative parallelization of sequential codes. This is a reasonable goal as I have already written a Java compiler and a single-level transaction system. However, it might be necessary to hook my compiler backend up to cilk in order to get low-overhead threading; this may prove to be a lot of work.

As a fallback, I would concentrate on the design of an efficient multiple-level transaction system and its use to write non-blocking parallel codes. I would evaluate the efficiency of using transactions compared to traditional locks, especially as the size of the transaction increases. Interactions with the OS and with the memory subsystem could be systematically analyzed.

Sean Lie is interested in hardware implementations of transactional memories. As an alternative to this proposal, I might collaborate with him to explore the hardware/software interactions of transactions. Hardware tends to implement short transactions well and efficiently, but has trouble with large and long-lived transactions. Software transactions tend to be slower, but are potentially unbounded in size and length (subject to livelock considerations). One avenue for exploration might be integrating hardware and software implementations such that short/small transactions are fast and large/long transactions complete.

References


