DOI:10.1145/1506409.1506430

Technical Perspective
Highly Concurrent Data Structures

By Maurice Herlihy

THE ADVENT OF multicore architectures has produced a Renaissance in the study of highly concurrent data structures. Think of these shared data structures as the ball bearings of concurrent architectures: they are the potential "hot spots" where concurrent threads synchronize. Under-engineered data structures, like under-engineered ball bearings, can prevent individually well-engineered parts from performing well together. Simplifying somewhat, Amdahl's Law states that synchronization granularity matters: even short sequential sections can hamstring the scalability of otherwise well-designed concurrent systems.

The design and implementation of libraries of highly concurrent data structures will become increasingly important as applications adapt to multicore platforms. Well-designed concurrent data structures illustrate the power of *abstraction*: On the outside, they provide clients with simple sequential specifications that can be understood and exploited by nonspecialists. For example, a data structure might simply describe itself as a map from keys to values. An operation such as inserting a key-value binding in the map appears to happen instantaneously in the interval between when the operation is called and when it returns, a property known as *linearizability*. On the inside, however, they may be highly engineered by specialists to match the characteristics of the underlying platform.

Scherer, Lea, and Scott's "Scalable Synchronous Queues" is a welcome addition to a growing repertoire of scalable concurrent data structures. *Communications'* Research Highlights editorial board chose this paper for several reasons. First, it is a useful algorithm in its own right. Moreover, it is the very model of a modern concurrent data structures paper. The interface is simple, the internal structure, while clever, is easily understood, the correctness arguments are concise and clear. It provides a small number of useful choices, such as the ability to time out or to trade performance for fairness, and the experimental validation is well described and reproducible.

This synchronous queue is *lock-free*: the delay or failure of one thread cannot delay others from completing that operation. There are three principal nonblocking progress properties in the literature. An operation is *wait-free* if all threads calling that operation will eventually succeed. It is *lock-free* if some thread will succeed, and it is *obstruction-free* if some thread will succeed provided no conflicting thread runs at the same time. Note that a data structure may provide different guarantees for different operations: a map might provide lock-free insertion but wait-free lookups. In practice, most non-blocking algorithms are lock-free.

Writing lock-free algorithms, like writing device drivers and cosine routines, requires some care and expertise.

Lock-free operations are attractive for several reasons. They are robust against unexpected delays. In modern multicore architectures, threads are subject to long and unpredictable delays, ranging from cache misses (short), signals (long), page faults (very long), to being descheduled (very, very long). For example, if a thread is holding a lock when it is descheduled, then other, running threads that need that lock will also be blocked. With locks, systems with real-time constraints may be subject to *priority inversion*, where a high-priority thread is blocked waiting for a low-priority thread to release a lock. Care must be taken to avoid deadlocks, where threads wait forever for one another to release locks.

Amdahl's Law says that the shorter the critical sections, the better. One can think of lock-free synchronization as a limiting case of this trend, reducing critical sections to individual machine instructions. As a result, however, lock-free algorithms are often tricky to implement. The need to avoid overhead can lead to complicated designs, which may in turn make it difficult to reason (even informally) about correctness. Nevertheless, lock-free algorithms are not necessarily more difficult than other kinds of highly concurrent algorithms. Writing lock-free algorithms, like writing device drivers or cosine routines, requires some care and expertise.

Given such difficulty, can lock-free synchronization live up to its promise? In fact, lock-free synchronization has had a number of success stories. Widely used packages such as Java's java.util.concurrent, and C#'s System.Threading.Collections include a variety of finely tuned lock-free data structures. Applications that have benefited from lock-free data structures fall into categories as diverse as work-stealing schedulers, memory allocation programs, operating systems, music, and games.

For the foreseeable future, concurrent data structures will lie at the heart of multicore applications, and the larger our library of scalable concurrent data structures, the better we can exploit the promise of multicore architectures. C

Maurice Herlihy is a professor of computer science at Brown University, Providence, R.I. He is the recipient of the 2004 Gödel Prize and the 2003 Dijkstra Prize and is a member of the editorial board for *Communications*' Research Highlights section.