Review #4

Decoupling Contention Management from Scheduling

F. R. Johnson, R. Stoica, A. Ailmaki, T. C. Mowry

ASPLOS, 2010

Jean-Pierre Lozi

April 30, 2010

1 Problem

Parallel applications tend to behave poorly under high load due to contention for shared objects. Two opposing contention management strategies are generally used: to wait for a lock to be released, a thread can either spin, actively polling the resource; or block, trusting the scheduler to wake it up when needed.

Spinning provides high reactivity (no context switch needed, instant detection of lock releases) but wastes significant CPU resources. Moreover, on an overloaded system using spinlocks, the scheduler often preempts lock holders to wake waiting threads up; these threads then waste their time slices actively polling a resource that cannot be released. This phenomenon is known as *priority inversion* and can drastically impede performance.

Block-based approaches usually fare better under high load but their lack of reactivity introduces high overheads on the critical path of computation, thereby increasing the likelihood that other threads will encounter contention and block. This causes a vicious cycle of extremely slow lock handoffs known as a *convoy*. This is especially true with programs that use fine-grained locking since their critical sections usually take less time to execute than a single context switch.

Hybrid solutions that somewhat improve performance have been developed, but they still perform rather poorly, simply offering other tradeoffs between reactivity and CPU usage.

2 Solution

According to the authors, the problem lies in the fact that contention management and load management (i.e. scheduling) are two orthogonal problems that should be treated separately. They propose a library that relies on a novel algorithm, known as *load control*, that regulates the number of active threads depending on the load. Basically, to acquire a lock, a thread either a) blocks, if there are too many threads running or b) spins otherwise. That way, threads are very reactive since they use spinlocks as much as they can and performance never collapses because the load is regulated.

To limit the load, threads are only allowed to run if they are unable to register themselves into a bounded array known as the *sleep slot buffer* because it is full. More precisely, the proposed algorithm —implemented by the load control library—works as follows: to acquire a lock, a thread spins until a) the lock is released or b) it is able to register itself into the sleep slot buffer. In the latter case, the thread blocks until it is removed from the sleep slot buffer or until 100ms pass, whichever comes first. Once the thread wakes up it restarts the lock acquire process as if it just arrived.

To control the number of running threads, a *controller* increases or decreases the size of the sleep slot buffer and wakes up threads when they get kicked out of the buffer. Thus, the controller acts as a local

scheduler that only manages the load whereas the clients only manage contention: contention and load management are effectively decoupled.

3 Performance

Experiments show that on various benchmarks, the proposed algorithm performs just as well as regular spinlocks under low load. Performance gracefully degrades under high load instead of collapsing as is the case with spinlock-based algorithms. Block-based algorithms' performance also degrades gracefully under high load, but they are initially much slower. Load control therefore offers the best of both worlds.

Another experiment shows that, on Solaris, the proposed algorithm is still efficient when it runs concurrently with applications that do not use it, even though these applications do not play nice under high load by limiting their number of active threads.

4 Benefits

The load control algorithm offers much better performance than traditional contention management algorithms under all loads. Moreover, the authors explain that performance could be improved even further if OSes provided additional facilities.

Also, the provided implementation is a regular library that can be transparently used by user applications.

5 Shortcomings

Load control behaves poorly when faced with large spikes or dips in load: the controller could respond erroneously to these variations, not realizing that they are only transient. The authors however explain that this is a well-known problem in control theory and that results from this field could help improving the algorithm (using a low-pass filter could work, for instance).

Moreover, the current version of load control responds poorly to nested critical sections: in this case, the algorithm can trigger priority inversions. However, the authors are confident that the load control algorithm could be extended to avoid this.