Lock Holder Preemption Avoidance
via Transactional Lock Elision

Dave Dice
Oracle Labs
dave.dice@oracle.com

Tim Harris
Oracle Labs
timothy.l.harris@oracle.com

Abstract
In this short paper we show that hardware-based transactional lock elision can provide benefit by reducing the incidence of lock holder preemption, decreasing lock hold times and promoting improved scalability.

Categories and Subject Descriptors D.4.1 [Operating Systems]: Mutual Exclusion

General Terms Performance, experiments, algorithms

Keywords Concurrency, synchronization, threads, multicore, locks, mutexes, contention, involuntary preemption, hardware transactional memory, transactional lock elision

1. Introduction
Transactional Lock Elision (TLE) \[7, \ 8\] permits multiple threads to concurrently enter and execute critical sections guarded by a given lock \(L\). The critical section is executed in optimistic transactional mode. If the hardware transaction aborts because of conflicting accesses or other reasons, the lock system can retry with another transaction. If there are excessive aborts in a given lock acquisition episode, then, to ensure progress, the system reverts as necessary to classic pessimistic physical locking.

The benefits of TLE are commonly taken to be the ability to leverage disjoint access parallelism \(^1\) and, for promiscuous locks \(^2\), avoidance of so-called cache line sloshing – cache-to-cache coherence traffic related to lock metadata. We identify and demonstrate yet another mode of benefit for TLE: lock-hold preemption avoidance (LHPA). By running a critical section as a TLE transaction, if the operating system preempts the thread, then the transaction immediately aborts and rolls back execution, leaving the lock available. Absent such TLE-based LHPA, convoys can form and the critical section durations can be artificially increased.

\(^1\) A classic application of TLE might be a hash table protected by a single coarse-grained lock where accesses to different buckets would be expected to be disjoint. Concurrent transactional threads operating on different buckets would be expected to run and commit without conflict aborts.

\(^2\) A promiscuous lock is typically uncontented, but is accessed in turn by multiple threads.

2. Related Work
Blasgen et al. \[3\] identified the convoying phenomena for locks. Edler et al. \[9\] suggested the idea of temporary non-preemption to allow lock holders to defer preemption. Kosche et al. \[12\] implemented a related facility in the Solaris operating system via the schedctl interface, where threads can request advisory and bounded preemption deferral. Schedctl has been employed in surprising ways in lock implementations \[5\]. In some environments, lock holder preemption can be avoided for short periods by masking the timer interrupt through which preemption is driven. Uhlig et al. \[16\] investigated lock-hold preemption avoidance for virtual machine monitors. In real-time systems the priority inheritance protocol \[15\] may be able to forestall lock holder preemption. Similarly, using elevated thread priorities for the lock holder may avoid LHPA, although this does not suffice for the default schedulers on commodity operating systems such as Linux or Solaris. Bershad et al. \[2\] emulated atomic instructions on unprocessors with restartable atomic sequences. Dice et al. \[6, \ 10\] used restartable critical sections to roll back preempted critical sections that access CPU-specific data. Similar ideas \[1\] have been recently rediscovered by the Linux kernel developer community. Harris et al. \[11\] introduced the concept of revocable locks implemented as a specialized software transactional memory.

3. Evaluation
To illustrate the benefits of LHPA we use a simple microbenchmark where \(T\) concurrent threads loop as follows: acquire a central lock \(L\); increment a shared variable; advance a shared random number generator \(^3\) 200 steps; release \(L\); advance a thread-local random number generator 100000 steps. At the end of a 10 second measurement interval we report aggregate number of iterations completed. We increment the shared variable to intentionally preclude any benefit from TLE that might otherwise allow critical sections to run concurrently in transactional mode.

We used an Oracle x5-2 \[14\] for our benchmarks. The system has 2 sockets, each populated with an Intel Xeon x5-2699v3 processor running at 2.3 GHz. Each processor has 18 cores, and each core is 2-way hyperthreaded. The system exposes a total of 72 logical CPUs. The system ran Ubuntu 15.04 with a 3.19 Linux kernel. The default energy management policies were used, with turbo mode enabled. Hardware transactional memory was explicitly enabled. The processors provide best-effort hardware transactional memory with a requester-wins conflict management policy.

We used two locks in our experiments: ttstle and ttsttle. Tts is a simple polite test-and-test-and-set lock \[1\]. Upon arrival, threads use an atomic XCHG operation to try to acquire the lock \(^4\). Failing that, they enter a busy-wait loop populated with a single PAUSE instruction. There is no back-off in the busy-wait loop. When the

\(^3\) We used the the PCG random number generator from \url{http://www.pcg-random.org/}

\(^4\) Transitioning the lock word from 0 to 1 via XCHG confers ownership.

Copyright ©2015 Oracle and/or its affiliates.
lock is then observed free, control exits the busy-wait loop and again retries the XCHG instruction.

Ttstle is just ttt augmented with TLE in a simplistic fashion. Arriving threads use the Intel TTS RTM [12]XBEGIN instruction to start a hardware transaction. The thread then checks the lock state, and if the lock is held, the thread immediately commits via XEND and reverts to the classic ttt path. Otherwise control passes into the critical section, and, absent aborts, the thread will successfully commit in the unlock operator. If the transaction aborts for any reason, control diverts into the ttt slow path. No retries are used, and there is no demarcation avoidance [8]. If two or more more threads try to simultaneously execute the critical section in transactional mode, then at least one will abort because of data conflicts on the variable that is incremented. We intentionally structured the critical section and TLE policies so that the sole benefit of using TLE would be lock holder preemption avoidance – that is, the data conflicts ensure that there is no opportunity for speculation to allow multiple critical section executions to run concurrently.

If a thread in the critical section in transactional mode is aborted by a preemption interrupt, that transaction aborts and, when the thread is again dispatched, control reverts to the classic ttt slow path. This acts to reduce lock holder preemption. In our case, the critical section duration is far less than any reasonable time slice length, so when the thread is dispatched and subsequently enters the critical section via the ttt path, it is less vulnerable to being preempted. In a sense, the tttt path shifted or “readigned” the critical section to a time interval that is less likely to be exposed to preemption. A freshly dispatched thread is unlikely to suffer immediate re-preemption at the start of a new time slice.

In Figure 1 we show the performance of the microbenchmark for ttt and ttstle on the Y-axis, varying the thread count on the X-axis (log scale). In our experiments the critical section length (CSL) is far shorter than non-critical section length (NCSL) even when 72 threads run concurrently. While the lock is promiscuous, contention and waiting are rare. Up to 72 ready threads, tts exhibits the same performance as ttstle – ttstle provides no benefit in this region. Conflict aborts are rare, and most critical sections manage to execute transactionally. Beyond 72 ready threads we encounter the onset of preemption, and ttstle shows better performance by virtue of lock holder preemption avoidance. Under ttt the lock holder is most likely to suffer preemption. Queuing and contention ensue until the lock holder is again dispatched onto a CPU, after which contention will abate. Preemption of the lock holder transiently increased the critical section length.

4. Conclusion

We show the existence of a non-traditional mode of benefit for TLE – lock holder preemption avoidance.

References


<sup>ttstle uses a conservative early subscription policy</sup>

<sup>quanta on Linux and Solaris are usually greater than 1 millisecond</sup>
Figure 1: Aggregate throughput