

Case Study: Rigel Task Model Livelock

John H. Kelm

Wed 6th May, 2009

The following case study discusses a livelock that was found while implementing the runtime system for the Rigel 1000+ core processor. A *livelock* occurs when multiple threads of execution are waiting on a shared resource, those threads continue to change state with respect to one another, but none of the threads ever acquire the resource and make forward progress. Livelocks can be transient, impeding forward progress for a finite period of time, or they can persist indefinitely. Persistent livelock conditions can appear very similar to *deadlocks* in which multiple threads try to access a shared resource, there are no state changes, and the threads make no progress. However, as we show in this case study, debugging livelocks can require a bit more ingenuity and may be hard to detect. We also show how we were able to remove the livelock that we discovered using fair locking mechanisms.

1 Implementation Overview

In the Rigel architecture, eight cores are grouped to form a cluster. These cores share a first-level cache, the cluster cache. Multiple clusters share a second-level cache, the global cache, which has an order of magnitude higher access latency compared to the cluster cache. The more memory requests that can be serviced by the cluster cache instead of the global cache, the higher the performance of the system.

Rigel has a runtime system that distributes work dynamically using a task queuing model named the Rigel Task Model (RTM). Logically, task descriptors defining a unit of work are inserted into a single queue, which is resident in memory, and are later dequeued by idle cores that then begin processing the task. The Rigel architecture supports 1000+ cores and is designed to execute fine-grained tasks. A real implementation using a single queue creates a bottleneck and limits performance.

In lieu of a single queue, the underlying implementation of RTM supports hierarchical queues. For simplicity in this case study, we will assume a two-level hierarchy and a Rigel design with 16 clusters (128 cores). Tasks can be enqueued at a global task queue while dequeue operations occur at distributed local task queues, one per cluster. There are two major benefits of the hierarchical design. The first benefit of having two queue levels is that it allows fast dequeue operations in the common case by accessing local tasks queues resident in the cluster cache. The second benefit of this approach is that enqueued work migrates to idle cores. Work is inserted into the global task queue, resident in the global caches, and filters down to local task queues in chunks on-demand, thus achieving better load balance, a desirable property in parallel systems.

Dequeue operations proceed by acquiring a local task queue lock, removing a task descriptor from the linked list of tasks, and releasing the lock. As local task queues empty, the first core at a cluster to find the empty queue will grab the local task queue lock, see that the head of the linked list is NULL, and attempt to dequeue a group of task descriptors at the global task queue and enqueue them locally.

A design choice was made to decouple global task dequeues from local dequeues. The choice was made to allow for local dequeues to continue to occur while another core prefetches task descriptors from the global task queue. The goal is to avoid seven cores waiting for work while an eighth is trying to obtain new work, thus exposing the latency of the global dequeue access to all of the cores in a cluster unnecessarily. To achieve the decoupling, two locks are used: a local task queue lock ($lock_{local}$) and a per-cluster global task queue lock ($lock_{global}$). When a core finds the local task queue empty, it drops $lock_{local}$ and tries to acquire $lock_{global}$. If it succeeds, it attempts to dequeue tasks from the global task queue. If it fails, there must be another core already accessing the global task queue so it tries to dequeue at the local task queue again. When the holder of $lock_{global}$ for the cluster succeeds in dequeuing tasks from the global queue, it must grab $lock_{local}$, insert the tasks, and if it is done prefetching tasks from the global task queue, drop $lock_{global}$ and $lock_{local}$, and rejoin the other cores in removing tasks from the local task queue.

2 Manifestation of Livelock in RTM

The locks used in the initial implementation were simple spinlocks implemented with load-link/store-conditional operations performing test-and-set. For many weeks of daily runs, this implementation was sufficient and free of deadlock/livelock conditions. Eventually, one particular benchmark began failing to complete. We assumed a deadlock situation and added debug statements prior to regions of the code that were suspect. The bug disappeared and the benchmarks began to complete again.

Adding print messages to benchmark code changes the state of the program and can cause a race or, as in this case, a livelock to be masked. We removed the printouts and instrumented the simulator to print out the program counter (PC) of each core at regular intervals. The PC values can be used to identify at which point in time and where in the code the benchmark stopped making forward progress. After a few hours of simulation, the PCs showed that most of the cores had completed all of the available tasks they could access (they were at a barrier) and were waiting for one cluster to finish. On the remaining cluster, seven of the cores were stuck in a tight loop containing a call to `spinlock_lock()` trying to acquire `locklocal`. The last core was trying to continually add tasks to the local task queue that it had removed from the global task queue.

The system was not deadlocked: The PCs were observed to oscillated between a fixed set of values, even leaving and reentering `spinlock_lock()`. A deadlock would have left all of the cores stuck spinning on a lock that would never be unlocked. What was actually happening was a livelock involving three cores we will call **A**, **B**, and **C** executing on a single cluster together. The livelock occurs as follows. **A** finds the local task queue empty while **B** and **C** (and the other cores) continue to execute. **A** grabs `lockglobal` after releasing `locklocal` and spends a number of cycles accessing the global task queue. In the meantime, **B** and **C** complete their current task only to find the local task queue is empty. Each one in turn grabs the lock on the local task queue, `locklocal`, sees no new tasks, tries to grab the global task queue lock, `lockglobal`, fails, releases `locklocal` and tries again.

When **A** returns with new work, it tries to acquire `locklocal` and insert the new work into the local task queue. In most cases, **A** eventually acquires `locklocal`, inserts the tasks, and goes back to work. However, in this case, the system was in the unfortunate state that **A** could never acquire the lock. What continued to happen was **B** would get `locklocal`, find it empty, **A** would try to

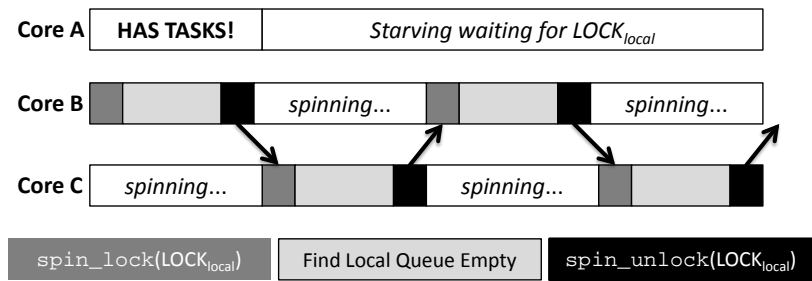


Figure 1: Timeline of cores participating in the livelock. Note that core **A** can never enqueue tasks because it is continually starved for $lock_{local}$ by **B** and **C**.

acquire the lock and fail, **B** would drop $lock_{local}$, **C** would acquire the lock, **A** would try again to acquire the lock and fail. This process would then repeat forever with cores **B** and **C** continually acquiring and releasing $lock_{local}$ without finding any work to do while core **A** never had the opportunity to insert the new work it had dequeued from the global task queue. What was occurring was clearly a livelock: while the state of the system continued to change, no forward progress was ever made. A timeline is depicted in Figure 1.

The causes of the livelock were three-fold: deterministic hardware, contention for a shared resource, and an unfair lock. Digital hardware systems are designed to be deterministic in the absence of external effects. The Rigel system modeled by our simulator is no different. When the state of the system moved into a bad state, it stayed there and kept cycling through the set of bad states ensuring that no forward progress would ever be made. The second issue was that there were multiple cores trying to access a shared resource repeatedly, with little delay, thus making the potential for starvation much higher. The last was the use of an unfair lock and ultimately what was changed in the final design to eliminate the livelock and ensure forward progress.

3 Correcting the Livelock

Instead of using a simple spinlock to protect the local task queue, a ticket lock was implemented. A ticket lock consists of two counters, similar to the line at a deli counter in the supermarket. The first counter is used by cores attempting to gain access to the lock. A core atomically increments the counter (with load-link/store-conditional in Rigel) and what is returned is the core's *unique* ticket

or reservation. This is akin to pulling a ticket at the deli counter with a number on it. Now the core continually reads the second counter, akin to compulsively checking the ‘Now Serving...’ sign in the deli example, until it matches the core’s number. The lock is released, and thus handed to the next core, by the lock holder incrementing the second counter.

The ticket lock avoids starvation by putting requesters in a line and servicing them in FIFO ordering. While a core may have to wait, as long as the cores ahead of it in line eventually releases the lock, i.e., there is no deadlock, the waiting core will eventually acquire the lock. In the example given above, instead of **B** and **C** continually starving **A**, **A** would now get a ticket and wait for **B** and **C** to drop the lock. **A** would acquire the lock, insert the new tasks, and the system would continue operating properly.

4 Lessons

Lessons from this case study are that livelocks are difficult to debug and may remain masked for long periods of time. Furthermore, even if the overall system never observably stops making forward progress, livelocks may be transient. The transience causes groups of cores to stop doing productive work, thus slowing down the system, possibly undetected.

Cores participating in livelock may transit through a large number of states making livelock difficult to identify. In this case, the livelock was constrained to a small number of cores and, since we were running in simulation, we had a high degree of visibility without intrusion. Livelocks in real systems with large numbers of threads may be much harder to isolate and remove.

The corrective measure for this case was a ticket lock that ensured fair access to *lock_{local}*. Note that not all locks are implemented using ticket locks in RTM. Ticket locks have added costs in terms of overhead even in the common case of the lock being free. A fair lock is not always necessary to eliminate livelock either. As an example, the global task queue lock remains implemented as a conventional spinlock since fairness in accessing it is not a constraint required by the system to ensure forward progress.