Red Hat
Enterprise Linux

Linux Kernel Performance & Scalability on NUMA System

Waiman Long
Principal Software Engineer

July 31, 2024

Overview

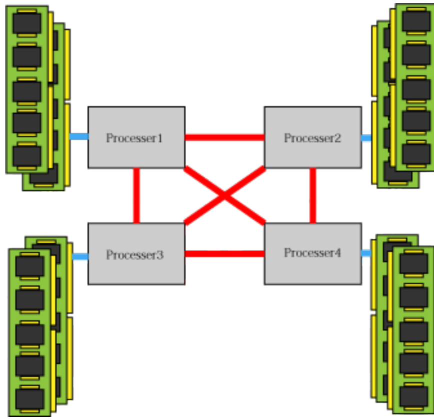
- As core count per processor is increasing and computer vendors are building high-end systems with more and more processors, we need to make sure that the kernel is able to efficiently use all the computing resources available in the system. For example, HPE/SGI can sell you a computer system with up to 32 sockets. That means up to thousands of cores and threads in a single box.
- Of course, such a large system may not be able to boot up the Linux kernel at all without the advancements in the synchronization subsystem made in the last 10 years. There are actually many challenges to properly utilize large NUMA systems efficiently.
- This presentation discusses the the following three major challenges involved and ways to diminish their impacts.
  1) Data locality
  2) Huge pages
  3) Cacheline contention/bouncing

Linux Kernel Performance

- Performance of the Linux kernel is one of the major focus areas of many kernel developers.
- Some of kernel code paths are considered performance critical as their performance can have major impact on the performance of user space applications. These performance critical paths are sometimes called fast paths. So most of the performance tuning done by kernel developers is focusing on all these performance critical paths.
- Other kernel code paths are less performance critical. They are sometimes referred to as slow paths. Performance gain in those slow paths is still nice to have.

What is NUMA?

- NUMA stands for Non-Uniformed Memory Access. It is a multi-socket computer system design where the memory access time varies depending on where the memory DIMM is located in the system relative to the processor accessing it.
- Earlier multi-socket systems used a SMP (Symmetric Multi-Processing) design where all the processors and memory chips are attached to a shared bus. The problem with this design is that it can’t scale up to too many sockets/processors.
- The access time of locally attached memory is the lowest. Remote memory has higher access time. Depending on the exact topology of the processor connection. Remote memory can be more than one hop away. In general, the more hops you need to access the remote memory, the slower it will be.
- The access time of a memory channel also depends on how many memory DIMMs are present. The more loading the bus has, the slower it will be.



Sockets, Nodes, Cores, Threads

- In this presentation, a socket refers to a physical computer chip installed in a physical CPU socket.
- A node refers to an entity that contains CPU with local memory attached and connected to other nodes via inter-processor interconnect like QuickPath (Intel) or HyperTransport (AMD).
- The two terms are not equivalent as a socket can contain more than one node, though one node per socket is the most common case. A good example is the AMD EPYC (Ryzen) processor that can be configured to contain up to 4/8 nodes within a single socket.
- A core refers to a physical CPU in silicon that includes (in most cases) the execution units and the L1 and L2 caches. Each node/socket can have multiple cores. L3 cache is usually shared by multiple cores.
- A thread or logical CPU refers to an instruction stream execution context that the operating system can treat as a standalone CPU. X86 processor supports up to 2 threads per core. CPUs from other architectures may support 4 threads/core or even 8 threads/core.

Data Locality

- For maximal performance, all the data access should be to or from local memory. In reality, remote memory access is unavoidable. It is just a matter of maximizing local memory access versus remote access.
- Many large enterprise applications like database provide many knobs for tuning. A big portion of those tuning knobs are for controlling data locality.
- The scheduler in the Linux kernel will move tasks around to different CPUs to balance the load. In case the CPU is on another memory node, that will mean the data that was local will become remote.
- Linux kernel has an autonuma feature that enables the kernel to make the data follow the task as it is being moved around. That may improve performance in some cases. However, the performance gain is usually not as good as careful manual tuning with CPU and memory pinning which requires expertise and experience.
- For example, on a 4-node system, running a single database instance on all the nodes will not perform as good as running 4 database instances each of which runs exclusively on one of the nodes.

Huge Pages

- All modern operating systems use paging to map virtual addresses to physical addresses via multi-level page tables. Today, x86-64 system supports either 4-level or 5-level page table.
- Page table translation is an expensive operation. A translation look-aside buffer (TLB) is used to cache recently used translations to avoid the expensive table lookup process for most of the memory accesses.
- For x86-64, the TLB can cache a 4k page, 2M page or a 1G page. There is a finite limit on how big the TLB can be. Using 2M pages or even 1G pages can cover a larger address space without needing to perform expensive page table lookup.
- For applications that have large working set size, the use of huge pages will help performance.
- The Linux kernel provides 2 mechanisms for utilizing huge pages – transparent huge pages (THP) and hugetlbfs. THP is managed by the kernel and is transparent to the applications, but hugetlbfs has to be managed by the applications themselves.
- Dependent on the memory access pattern, THP may or may not improve application performance. In fact, some enterprise applications actually recommend turning off THP and let them handle huge pages via hugetlbfs directly.

Cache Coherence Protocol

- Modern CPUs have at least 3 levels of caches (L1, L2, L3). Data in caches can be accessed much faster than those in memory. So efficient use of caches is essential to application performance.
- A NUMA system must also make sure the content of the cache within different processors remain consistent. This is achieved via the cache coherence protocol.
- A cacheline (64 bytes for x86) in a cache can have different states in its life time. Most cache coherence protocols support 4 or 5 different states. Intel CPUs use the MESIF (Modify, Exclusive, Shared, Invalid and Forward) protocol, whereas AMD CPUs use the MOESI (Modify, Owned, Exclusive, Shared, Invalid) protocol.
- When multiple CPUs are trying to access the same cacheline across multiple nodes, the cache coherence protocol makes sure that only one CPU can have exclusive right to write to the cacheline. Other copies of the same cacheline in other CPUs will be invalidated.
- A contention in a cacheline by multiple CPUs across different nodes will cause significant cache coherence traffic in the interprocessor links to bounce around ownership right of the cacheline limiting bandwidth available for other uses and increasing latency.

Cacheline Contention (Bouncing)

- Cacheline contention (bouncing) is a major cause of applications slowdown on a NUMA system, especially a large one with many CPUs and nodes.
- Cacheline contention can be caused by:
  1) False cacheline sharing – the CPUs are accessing different and independent data, but they happen to reside in the same cacheline.
  2) Contention on the same datum.
- False cacheline sharing can be solved by moving frequently accessed, but unrelated data items into different cachelines. Tools like perf-c2c(1) can help in spotting this kind of problem.
- In the Linux kernel, the data items that are most likely to be contended by multiple CPUs are locks and counters.
- Many counters in the Linux kernel has been switched to use per-cpu counters. The remaining shared ones are usually used sparingly and so shouldn’t cause any performance problem.

Lock Contention

- Lock contention slows things down by forcing the CPU to wait until the lock become available. The CPU can be doing nothing (spinning lock) or the task can sleep and let other running task takes over the CPU (sleeping lock).
- The level of lock contention can be reduced by breaking down a coarse-grained lock into multiple finer-grained ones or trying to use as much shared locking as possible. This is usually easier said than done in many cases. Another alternative is to use as much lockless algorithm as possible, like RCU (Read, Copy, Update). However, the conversion to lockless algorithm can have many pitfalls and is easier said than done.
- Another side effect of spinning lock contention is lock cacheline contention as the CPUs have to constantly spin on the lock cacheline to see if it will become available.
- Lock cacheline contention used to be a serious problem for some workloads on large NUMA systems as their performance may actually decrease the larger the systems become.
- Recent advances in the locking infrastructures in the Linux kernel have largely solved this lock cacheline contention problem by making almost all lock waiters spinning on their own local cachelines, *NOT* on the lock cacheline while waiting for the lock to become available for acquisition.

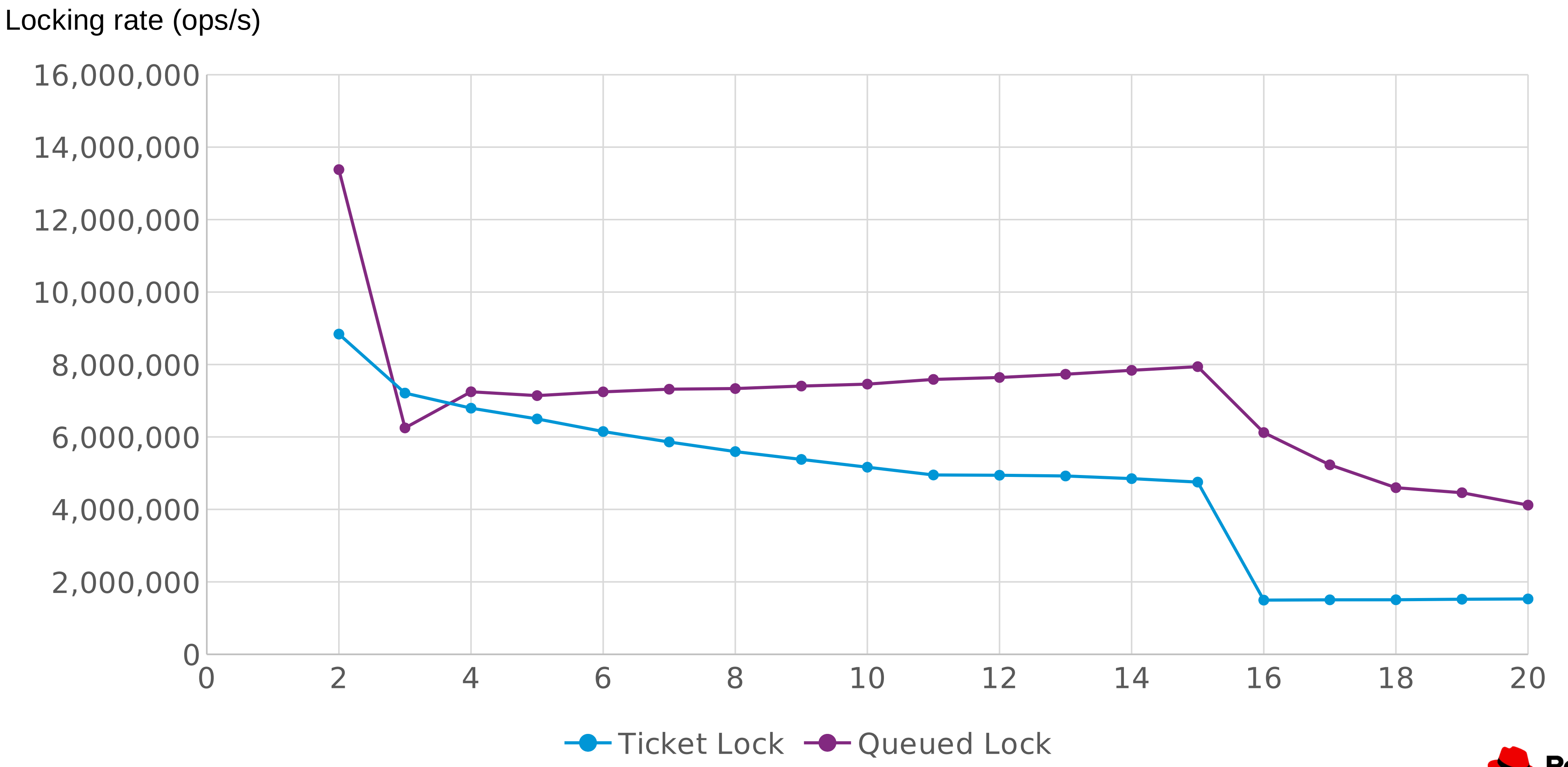
Recent Changes in Linux Kernel Locks

- A major building block for recent changes in Linux kernel locks is the MCS lock. It is a locking algorithm that allows scalable synchronization in SMP systems. Variants of MCS locks are used internally by the kernel locks for queuing up the lock waiters without spinning on the lock cacheline.
- Queued spinlock (qspinlock) was merged into the 4.2 kernel. Lock waiters are put into a MCS queue. Only the queue head (the MCS lock owner) will spin on the lock cacheline, the rests will spin on their own per-cpu cachelines.
- The original read-write lock implementation is unfair which can lead to live lock and hard panic. Queued read-write lock (qrwlock) which is a fair version of read-write lock was merged into the 3.16 kernel. An internal raw spinlock is used for queuing the lock waiters. Coupled with queued spinlock, it provides a fair read-write lock without lock cacheline contention problem.
- For mutex (a sleeping mutually exclusive lock), lock waiters are allowed to spin on the lock as long as the lock owner is running. This creates a lock cacheline contention problem for a contended mutex. In the 3.10 kernel, patch was merged to create a waiting queue for lock waiters based on the MCS lock. This solved the lock cacheline contention problem. The queuing code had since been enhanced and extracted into the optimistic spinning queue code in the kernel.
- RW semaphore (a sleeping read-write lock) had adopted the use of optimistic spinning queue since the 3.16 kernel to achieve performance parity with mutex for workloads that use mostly write lock.

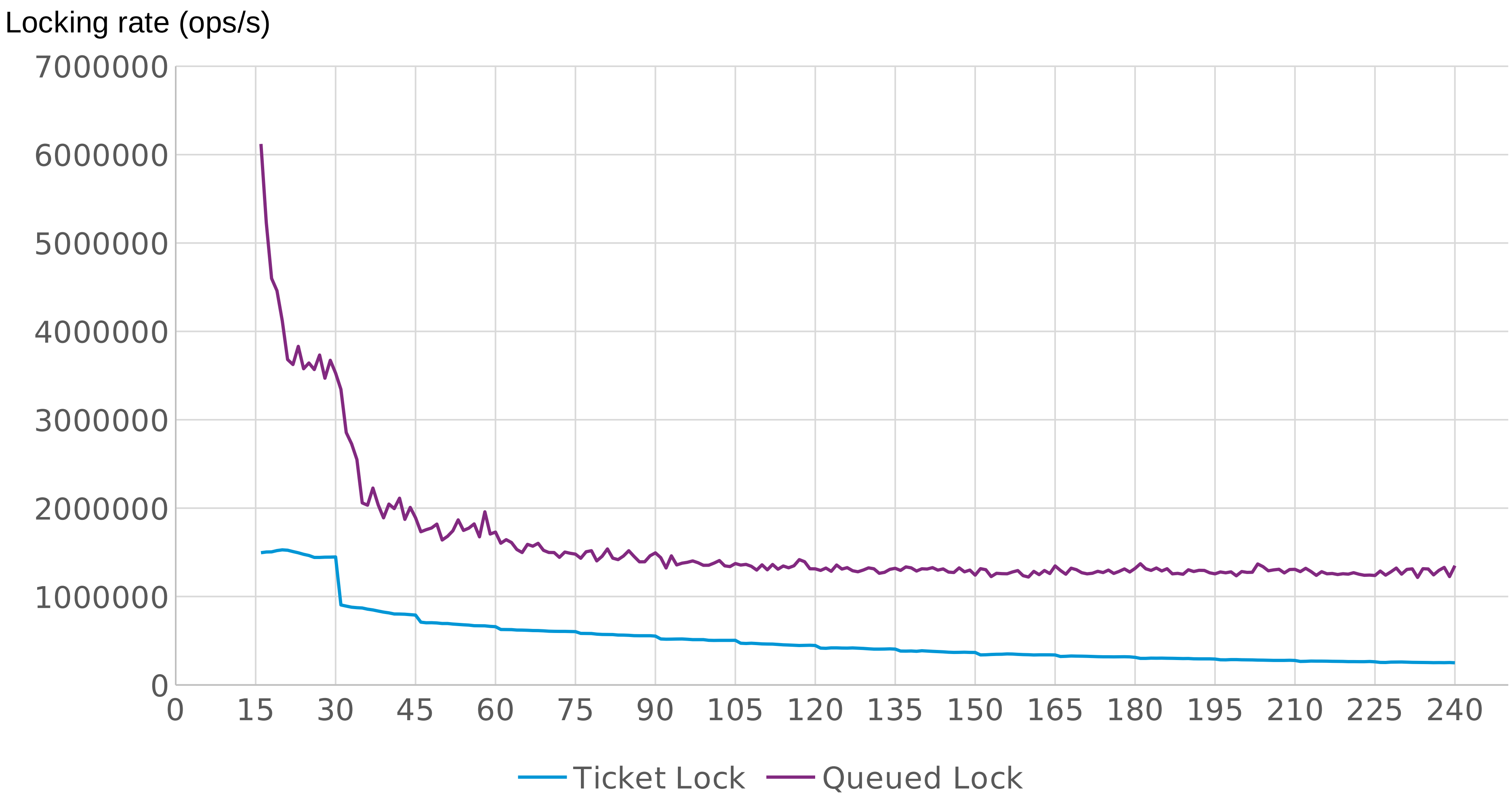
Impact of Lock Cacheline Contention

- To illustrate the performance impact of cacheline contention, a locking microbenchmark was run on the same test system on 2 different kernels – one with ticket spinlocks (prior to the 4.2 kernel) and one with queued spinlocks.
- The major difference in cacheline behavior between the ticket and queued spinlocks is that all the ticket lock waiters will spin on the lock cacheline (mostly read), whereas only the queue head of the queued lock waiters will spin on the lock cacheline.
- The charts in the next 2 pages show the locking rates (the total number of lock/unlock operations that can be performed per second) as reported by the microbenchmark with various number of locking threads running with an empty critical section. Note that if the lock owner reads from or write to data in the same lock cacheline, the lock contention performance degradation will get worse.
- The test system was a 16-socket 240-core IvyBridge-EX (Superdome X) x86-64 system with 15 cores/socket and hyperthreading off.

Ticket vs. Queued Locks (2-20 threads, 1 load)



Ticket vs. Queued Locks (16-240 Threads, 1 Load)



Conclusion

- The Linux kernel is doing a pretty good job in managing data locality for its internal data structures. Some NUMA aware applications are able to manage data locality by themselves with some tuning by users. For other non-NUMA aware applications that don’t need a large footprint, tools like numactl(8) or control group (cpuset) can be used to limit their execution to a single node or sub-node.

- Work is still ongoing on THP front to continuously enhance its functionality and usefulness to the average users. Application developers who have enough resources can also utilize hugetlbfs if they choose to.

- Cacheline contention is still an ongoing issue and it needs to be addressed when it pops up. Lock cacheline contention, however, is mostly solved. Though additional tuning and enhancements in the locking code is still ongoing.

Red Hat
Enterprise Linux

Q & A

Thank you!

Red Hat is the world's leading provider of enterprise open source software solutions. Award-winning support, training, and consulting services make Red Hat a trusted adviser to the Fortune 500.

in linkedin.com/company/red-hat

youtube.com/user/RedHatVideos

facebook.com/redhatinc

𝕏 twitter.com/RedHat