Optimizing Application Locking Performance on Large Multi-core Systems

Waiman Long
Principal Software Engineer, Red Hat Inc
Open Source Summit North America 2017
Agenda

- Why Optimizing for Multi-core Systems?
- Lock Types and Usage Considerations
- Locking Performance
- Recent Locking Development
Why Optimizing for Multi-core Systems?
CPU Core Counts are Increasing

- The table below shows the progression in the maximum number of cores per CPU for different generations of x86 CPUs.

<table>
<thead>
<tr>
<th>CPU Model</th>
<th>Max Core Count</th>
<th>Max threads in 4P Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>IvyBridge</td>
<td>15</td>
<td>120</td>
</tr>
<tr>
<td>Haswell</td>
<td>18</td>
<td>144</td>
</tr>
<tr>
<td>Broadwell</td>
<td>24</td>
<td>192</td>
</tr>
<tr>
<td>Skylake</td>
<td>28</td>
<td>224</td>
</tr>
<tr>
<td>AMD Ryzen</td>
<td>32</td>
<td>256</td>
</tr>
</tbody>
</table>

- Massive number of threads will be available for running applications.
- The question now is how to make full use of these computing resources. Of course, virtualization and containerization are all useful ways to use them up. Even then, the typical size of a VM guest or container is also getting bigger and bigger with more vCPUs in it.
Parallel Programming

- Parallel programming is needed to utilize the increasing number of cores available in the computer systems.

- Developers are usually good in writing serial code, but few are really good in writing parallel programs that have more than minimal amount of interactions among threads or processes.

- Locks and synchronization primitives in general are the preferred tools to help developers write correct parallel programs as they are simple to use.

- Amdahl's law – the speedup of a parallel program is limited by the amount of works that have to be serialized by a lock or synchronization primitive.

- Lockless algorithms like RCU (Read-Copy-Update) can provide a significant boost in performance but are usually hard to use as well as buggy in many cases.

- Using locks and synchronization primitives wisely is essentially to writing parallel programs with good performance.
Lock Types and Usage Considerations
Glibc Synchronization Primitives

- Mutex – `pthread_mutex_lock(3p)`, `pthread_mutex_unlock(3p)`
- Read/write lock – `pthread_rwlock_rdlock(3p)`, `pthread_rwlock_wrlock(3p)`, `pthread_rwlock_unlock(3p)`
- Condition variable – `pthread_cond_wait(3p)`, `pthread_cond_signal(3p)`, `pthread_cond_broadcast(3p)`
- Semaphore – `sem_wait(3p)`, `sem_post(3p)`
- Barrier – `pthread_barrier_wait(3p)`
- Spinlock – `pthread_spin_lock(3p)`, `pthread_spin_unlock(3p)`

Mutexes, Read/write locks and condition variables are the most commonly used synchronization primitives in glibc.
Spinning vs. Sleeping Locks

- In user space, spinning locks should only be used if the critical section is really short and the locks are unlikely to be highly contended. It is because spinning lock can suffer from lock holder preemption problem, where the lock holder can be preempted by the operating system and hence the lock waiters are just wasting CPU cycles spinning on the lock.

- Sleeping locks are generally recommended in user space under most circumstances except as noted above. Sleeping locks do have the problem that there is a fair amount of latency after the locks are released and before the lock waiters wake up and take the locks.

- On Linux systems, all these synchronization primitives except spin lock rely on the futex(2) system calls to enable threads to sleep and wait in the kernel until a lock is available or a condition is true.
Lock Granularity

- Lock granularity refers to how much data are being protected by a lock.

- Coarse-grained locks protect a large amount of data and are easy to code, but they limit the amount of parallelism that can be achievable in the application.

- Fine-grained locks protect only a limited amount of data and can be much harder to code because of the potential need for nested locking which can lead to deadlock. However, they do increase the amount of parallelism achievable and hence performance.

- Most parallel applications start with coarse-grained locks and gradually move over to finer grained locks over time to increase application performance.
Lock Types

- There are 2 main types – exclusive or shared.
- With exclusive lock, only one thread can hold own the lock at any one time. With shared lock, multiple threads can hold the lock simultaneously.
- A mutex is an exclusive lock, whereas a read/write lock is an exclusive lock for writing and shared lock for reading.
- Mutexes generally perform better than read/write locks on writer heavy workloads, though read/write lock performance has improved recently and thus reducing the gap in the latest glibc version (2.26). For reader heavy workloads, read/write locks will perform better.
- Read-mostly data should be protected by a read/write lock, whereas write-mostly data should be protected by a mutex. For those in the middle of the spectrum, some testings may have to be done to decide which type of locks should be used.
Transactional Memory

- Transactional memory attempts to simplify concurrent programming by allowing a group of load and store instructions to execute in an atomic way.

- Intel's TSX (*Transactional Synchronization Extensions*) is one attempt to implement the transactional memory semantics. It provides two software interfaces – Hardware Lock Elision (HLE) and Restricted Transactional Memory (RTM).

- HLE helps improve locking performance by eliminating lock taking if there is no memory access conflict. Glibc currently supports HLE for mutex only by setting the `PTHREAD_MUTEX_ELISION_NP` mutex type. On hardware that doesn’t support TSX, the new mutex type is a no-op.

- Intel C compiler provides intrinsics that can invoke the RTM feature.

- Intel TSX is not a panacea. It does not guarantee performance improvement. Performance can actually deteriorate if it is used in the wrong places.

- The best places to use HLE it is a coarse-grained lock that covers a fairly large number of independent data items where data access conflict is relatively infrequent.
Use Standard APIs or Build Your Own Locks

- It is easy to use standard glibc synchronization primitives for an application locking needs. Being conforming to standards like POSIX and supporting all the different attributes, however, contribute to additional performance overhead in their use.

- By building custom locking APIs, developers can tailor the APIs to the need of the application and squeeze out unnecessary overhead. However, not many developers are familiar enough with locking in general and the Linux futex(2) system call in particular to be able to build a reliable locking APIs with good performance. As the custom APIs need maintenance, only large projects with sizable team can afford that.

- Many large development projects start with standard APIs. When the development teams finds the standard APIs lacking in features or performance, they may build some custom locking code for their performance critical paths.

- Standard APIs provided by glibc, for example, also have the performance of their locking APIs improving over time.
Locking Performance
Linux Kernel Locking Performance

- Ever since the 3.10 kernel, the performance of the Linux kernel locking APIs (including the futex(2) system call used by user space synchronization primitives) have improved substantially especially on system with large number of CPU cores.

- The two major factors contributing to the performance improvement are:
  1. Reducing cacheline contention (cacheline bouncing)
  2. Optimistic spinning on sleeping locks

- Cacheline contention is reduced by limiting the number of running threads that are snooping on the lock cacheline including data items that are around the lock. This helps to improve lock cacheline access latency as well as power efficiency. Similar technique is also applicable in user space to help locking performance.

- Optimistic spinning refers to spinning on a sleeping lock as long as the lock holder is still running, not sleeping. This is done by lining up spinners in a queue with only the head of the queue spinning on the lock. This technique is not currently applicable in user space as there is no easy way to find out if the lock holder is actually running and queuing suffers from lock waiter preemption problem.
Data Cacheline Alignment

- On x86 as well as most other CPU types, a cacheline is 64 bytes in size. A cacheline is the basic unit in all the physical reads and writes from/to memory.

- False cacheline sharing refers to the fact that two or more independent and unrelated data items happen to be in the same cacheline. This increases the chance of cacheline contention when these data items need to be accessed frequently by independent threads.

- It is important to examine the data layout of critical data structures that are to be shared among multiple threads that false cacheline sharing won’t be an issue that will affect application performance.

- Padding and/or compiler directives, like gcc's __attribute__((__aligned__(64))), can be used to make sure that false cacheline sharing won’t happen for those critical data.
NUMA Awareness

- For applications that are to be run on systems with multiple sockets (nodes), NUMA awareness is important to avoid unexpected performance shortfalls.

- Cross-socket memory transfers can have a much higher latency as well as bandwidth limitation that local memory transfers. That includes lock cacheline contention. Therefore, locking performance deteriorates when the lock contenders are spread across different nodes instead of all in the local node.

- Partitioning resources (cpus and memory) according to node boundary and trying to do as much local node memory access as possible can be done with tools like numactl, libnuma library or Linux kernel control group. This will help the application scale better on systems with multiple nodes.
Recent Locking Development
Introducing TP Futex

- Even though the performance of Linux kernel futex(2) system call has been improved in recent years, it is still a sleeping lock that suffers from extended latency in lock ownership transfer.

- Most user space implementations of locks use a type of futexes called wait-wake (WW) futex. This name comes from the fact that the FUTEX_WAIT and FUTEX_WAKE command codes are used for this type of futex.

- A new futex type called throughput-optimized (TP) futex had been proposed to enable the Linux kernel to perform lock optimistic spinning on behalf of user application threads that wants to acquire a user space lock. The user space lock can be a mutex or a read/write lock.

- Further works will be needed to enable glibc to support a new mutex and read/write lock type using the the TP futex.
TP Futex Performance

- Using a locking micro-benchmark, the following table shows the per-thread locking performance in\nterm of the number of locking operations per second of locks implemented using TP and WW futexes\non a 2-socket 36-core x86-64 system with one locking thread per core.

<table>
<thead>
<tr>
<th>Lock/Futex Type</th>
<th>Glibc</th>
<th>WW Futex</th>
<th>TP Futex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutex</td>
<td>110,186</td>
<td>136,291</td>
<td>158,897</td>
</tr>
<tr>
<td>Read/Write lock</td>
<td>41,267</td>
<td>86,004</td>
<td>138,280</td>
</tr>
</tbody>
</table>

- The WW futex column refers to custom implementations that do not fully conform to standards like\nPOSIX and so can perform better than glibc.

- The read/write lock benchmark results are for a 1-to-1 ratio in the number of read and write lock calls.

- The locks implemented with TP futexes are also more fair and less susceptible to lock starvation.
Thank You!