Linux multi-core scalability

Andi Kleen
Intel Corporation
Germany
andi@firstfloor.org

1 Abstract

The future of computing is multi-core. Massively multi-core. But how does the Linux kernel cope with it? This paper takes a look at Linux kernel scalability on many-core systems under various workloads and discusses some some known bottlenecks. The primary focus will be on kernel scalability.

2 Why parallelization

Since CPUs hit the power-wall earlier this decade single threaded CPU performance has been increasing at a much lower pace than it historically used to in earlier decades[1]. The recent trend in hardware is to go multi-core and multi-threaded for more performance instead. Multi-core means that the CPU package has more than one CPU core inside and acts like multiple CPUs. Multi threaded CPUs are using multiple virtual CPUs inside each CPU core to use execution resources more efficiently. Also larger systems have always used multiple CPU packages for better performance.

Exploiting the performance potential of these multiple CPUs [2] requires software improvements to run parallel. Traditionally only larger super computers and servers needed major software scalability work, because they have been using many CPU sockets, while cheaper systems only had a low number of CPUs (one or perhaps two) so that major scalability work was not needed. But since individual CPUs are getting more and more cores and threads this is changing and even relatively low end systems require extensive scalability work now. The following table has some current example systems, showing these trends.

CPUs	Visible CPUs	Memory	Description
2 cores	2	2GB	Low end x86 desktop system 2008
4 cores x 2 threads x 2 sockets	8	4-8GB	Middle-end x86 desktop system 2009
4 cores x 2 threads x 2 sockets	16	8-32GB	Standard low end x86 server 2009
6 cores x 4 sockets	24	32-128GB	Standard 4 socket x86 server 2009
8 cores x 2 threads x 4 sockets	64	128-512GB	Standard 4 socket x86 server 2010
8 cores x 2 threads x 8 sockets	128	128GB-1TB	8 socket x86 server 2010
2 cores x 32 sockets	64	512GB-2TB	High end commercial server 2008
2 cores x 512 sockets	1024	>1TB	Super computer 2007

Table 1: Linux systems and their CPU numbers

[1] But contrary to common statements the single-thread performance increase has not plateaued, but is just increasing a much lower rate and does not directly follow Moore's law anymore

[2] A note on terminology: "*CPU*" in the rest of the paper refers to a CPU thread on a multi-threaded CPU or a core on a single-threaded CPU. Essentially any CPU visible in linux in /proc/cpuinfo.

This table shows that even desktop systems have upto 8 CPUs now, and even reasonably cheap server systems upto 24 CPUs These numbers are likely to increase further in the future. This makes it critical that software scalability keeps up.

Another factor in scalability is not only scaling to many CPUs, but also scaling to large memory sizes. Each CPU in a system needs enough memory for its working set, this implies that with more CPUs the systems also need more memory. This trend is also visible from the table above. Scaling to larger memory sizes is also a challenge for a kernel (or for a user program), but this is out of scope in this paper.

3 Parallelization basics

When talking about parallelization commonly Amdahl's law[1] is cited as a limitation. Amdahl's law essentially states that parallelization performance improvements are limited by the serial sections in the algorithm when an algorithm working on a given data set is parallelized. But in practice – and especially for kernel tuning – we tend to be more guided by Gustafson's law[2]: when doing more in parallel usually also the data set sizes increases and that more than offsets any speedup limitations from Amdahl's law. To apply this to the kernel: the kernel will not necessarily get faster for single operations by parallelization [3] , but it will be able to do a lot of single operations in parallel, allowing more processes or threads to run concurrently to each other.

I call this the *"library style"* of tuning. For classic parallelization of single workloads (let's say for a given computation like a weather model) the main challenge is to split up the existing algorithm, working on a single data set, into multiple independent tasks and execute them concurrently. But for code like the kernel which acts more like a library and offers relatively short lived individual operations to other programs it is typically not worth it to try to parallelize individual operations[4] Instead the goal is to make sure that multiple individual operations are able to execute in parallel as far as possible. This *"library style"* of tuning also applies to many network servers (all that handle relatively short lived operations) and are also to a lesser degree to OLTP databases.

Code that only accesses data local to the current operation can run in parallel, the main challenge is to synchronize access to any shared data structures that may be concurrently accessed by multiple threads. The main work in "library style" tuning is therefore to scale access to shared data structures.

Examples of such data structures would the directory and *inode* caches, the file tables, the page cache, essentially every data structure that is not local to a thread.

The simplest approach to scale access to a shared data structure is to use a lock for all the code in the subsystem that accesses the data structure. This is called *code-locking*. It limits parallel operation to one access to the data structure in parallel and tends to not scale well. On the other hand it's a relatively simple approach with little overhead.

Then the next step typically, when the data structure is frequently accessed, is to move on more fine-grained data-locking, by adding locks to individual objects in the data structure. For example for a hash table this could be a per bucket lock instead of a table lock, and also possibly adding an own lock to each object inside the hash table to protect its private state. This tends to scale better than code-locking. As long as different operations don't access the same objects they can execute independently in parallel and will scale well.

These different phases while tuning for more parallel performance can be described as the *"parallelization tuning cycle"*. The cycle consists of initial measurement of a workload, identifying using profilers like oprofile, simple tuning, then more measurements and if needed more tuning using more complex tuning methods.

[3] In fact parallelization on the kernel level tends to make single operations somewhat slower because of locking and atomic operations overhead.

[4] The individual operations still rely on classic non parallel code tuning and on CPU single thread performance improvements to become faster.

4 Communication latencies

For highly tuned parallel code often communication latency is the limit. When the code is already using fine-grained locking the hold times of the locks tend to be relatively short. But still the cache lines of the locks and any other shared data have to be transferred between different CPUs when the ownership of the critical section moves from one CPU to another. This is informally known as "cache line bouncing".

Communication latencies are limited by physics, as in how fast the signal can be transferred on wires between chips. They can be a significant problem, especially in larger systems with many CPU sockets, which have longer latencies when transfering signals over longer distances between boards. When a workload causes significant cache line bouncing the time lost waiting for accessing shared variables in memory can be significant.

Luckily on many-core systems which have multiple cores on a CPU die (like a 4 core or 6 core CPU today) the problem is less severe, because the communication latencies inside a die are significantly lower, than when talking to a different chip. But even there the latency is typically significantly higher than the base cycle time of the CPU. Also modern point-to-point interconnects on smaller systems (<= 4 sockets) are faster than interconnects on older large multi node systems. This makes it easier to scale software on a modern many core system with the same number of visible CPUs than on an older large SMP/NUMA system. Still there is typically significant effort required to optimize algorithms for local memory, and avoid cache line bouncing, to get good parallel performance.

One classical initial approach is to eliminate "false sharing": making sure that data frequently accessed by different CPUs does not share cache lines in memory. This can be addressed by adding padding around variables to ensure that they are in different cache lines[5] Areas with this problem can be identified using a profiler with a suitable performance counter measuring cache misses.

A more advanced approach to minimize communication latencies is to distribute data: giving each CPU has its own local data set that it can work on. This is called *"per-cpu data"* and is a common technique in the Linux kernel for critical subsystems. For example the kernel slab allocator uses per-cpu data for fast cpu local memory allocation with no cache-line bouncing[6]The drawback is relatively high memory overhead and also some complexity in dealing with CPU hotplug.

Using lock-less approaches like *Read-Copy-Update*[3] also eliminates cache line bouncing on locks, but they tend to be more difficult to develop and only work for some specialized cases.

5 Problems of SMP scaling

Scaling kernel software is not without problems. Typically a highly parallelized version of a subsystem is more complicated than the original less parallel version the tuning effort started from. This makes single operations slower. Often a highly parallel version also needs more memory and has larger code-size, which can be a problem on small embedded systems[7].

A special problem, when more locks are added, is that each lock adds some cost. Locks are implemented using atomic operations on the CPU level, and atomic operations are significantly slower than non atomic operations[8]. So more and more locks make the code slower. This is already visible in some simple statistics: for example the number of locks acquired and released for a simple system call like *read()* is already in the high double digits.

This has been colloquially described as "falling over the locking-cliff" (see [4] for more details, although the paper is somewhat outdated and a little polemic). In addition they also add complexity and subtle bugs (like lock ordering problems). This also has to be paid on smaller systems, like embedded systems on which Linux also runs. But nevertheless scalability is needed to run well on multi-core

[5] But of course too much padding can increase memory requirements significantly

[6] However there can be still significant cache-line bouncing in slab when data is freed on a different CPU than where it was allocated

[7] However embedded systems are also increasingly becoming multi-core

[8] Luckily the trend in modern CPUs is to make atomic operations faster, but they can be quite slow in some older systems.

systems.

How to get out of this dilemma? One approach is to carefully consider what subsystem should be tuned for scalability. Obviously subsystems that are not performance critical do not need much scalability, or only a lesser degree of scalability. So for example a infrequently used subsystem would likely not be modified to use per-cpu data or fine-grained locking. This has the risk of course that if the subsystem is actually frequently used in some special workload that it might become a bottleneck.

Linux also makes heavy use of compile-time configuration options to eliminate unneeded scalability code (for example locks expand to nothing on kernels built for non-preemptive single processor systems) This only works to a certain extent: a fine scale locked subsystem is usually still more complex than a simpler non parallel subsystem even when the locks are dropped by the compiler. This doesn't address the problem of the different needs of a two-core systems and a 512-core system being served by the same distribution kernel binary.

The potential correctness problems with locks, like deadlocks or lock-ordering problems, can be addressed by better tools. For example modern Linux has the *lockdep* subsystem which can automatically detect many mistakes in lock use[9] Unfortunately there are no similar facilities for user space applications currently.

6 History of Linux kernel scalability

The original SMP work was done for Linux 2.0. Back then the complete kernel, including the interrupt handlers, were all running all under a single big kernel lock (*BKL*). This preserved the behavior of a traditional non preemptive uniprocessor Unix kernel with very little code changes, but allowed no parallelism in kernel code. User space code was able to execute in parallel on multiple CPUs though.

In Linux 2.2 the *BKL* was still used for most of the kernel, but the interrupt handlers had own spin locks, allowing parallelism in interrupt handlers. During the 2.2 lifetime various distributions also back ported scalability improvements from the then 2.3 development tree, so in practice a lot of 2.2 based distributions (e.g. SUSE SLES8) did a bit better. Some advanced optimizations, like the first NUMA optimizations, were started on 2.2. The 2.2 kernels were the first to run successfully on larger systems (like 16 CPU system) on limited – primarily user space – workloads.

In Linux 2.4 more and more subsystems were moved out of the big kernel lock. Some data locking was introduced, but there was also still a lot of code locking (spinlocks covering whole subsystems). This improved scalability significantly and Linux already ran successfully on larger systems, but there were quite a lot of problematic bottlenecks left. Again several of them were addressed by backports of distributions from the then 2.5 development tree (for example the *page cache* locking conversion from a single code lock to multiple data locks)

Finally in the 2.6 series very little code was left under the big kernel lock. Most the code locking in common paths was converted to data locking and there was a lot of scalability tuning to eliminate shared cache line bouncing. For critical subsystems there was also some use of advanced lock-less approaches (e.g. *Read-Copy-Update (RCU)*[3]) A new multi queue CPU scheduler was also introduced, avoiding contention on global run queues shared by multiple CPUs and lowering overhead of scheduling processes in parallel. This scheduler was also back ported to some 2.4 distribution kernels later. There was also significant effort to parallelize the block IO subsystem, tuning them to a point, where they scale very well (some communication latency problems on cache lines remain). There is also increasing work to move data structures to be per CPU to avoid communication latencies at the cost of more memory foot print.

There is an ongoing effort in the 2.6 series to improve CPU scalability further.

There has been some work recently to address scalability to large memory sizes, especially on systems with small page sizes (4K).

A good non Linux specific overview of the various basic techniques used for kernel scaling is in [5].

[9] This requires using special debugging kernels because lockdep has significant runtime overhead.

Linux scalability development used many of the basic techniques described in this book.

7 Linux common locks with scalability issues

This covers the situation in 2.6.31. In general Linux has a lot of locks now. There are 1957 different spinlock declarations in Linux 2.6.31. However this severely underestimates the total number of active locks because locks in data structures often have many instances (one per object). Also especially hash tables with per bucket locks can have thousands of locks. There are other locks types (mutexes, semaphores etc.) not even accounted. On the other hand most of these locks are in drivers, which are only used for specific hardware.

7.1 *dcache_lock* and *inode_lock*

The directory cache caches all file names in a hierarchy and is a central concept in the virtual file system interface (VFS): most accesses in the Linux VFS go through a dentry object. Every lookup of a path (like in the *open* syscall) has to look the dentry up in the global *dcache*, which consists of a hash table and a hierarchy of objects.

The *dcache_lock* controls access to the *dcache*. The *inode_lock* protects access to the inode cache, which manages the global cache of all *inodes*. It is less critical than the *dcache_lock* – because most accesses are handled by the *dcache* – but nevertheless important.

Some critical paths, like some read-only lockups in the *dcache*, have been converted to *RCU* for some time[6], making them lock-less, but for important workloads (e.g. creating lots of files from multiple processes) the *dcache_lock* still can be very contended. Also there is still a lot of cache-line traffic on written data of common objects like the root directory.

There is currently a effort underway to split these locks into smaller locks, improving scalability of the VFS.

7.2 Big Kernel Lock (*BKL*)

The most important subsystems currently (as of 2.6.31) still under the big kernel lock are NFS, file locks (flock et.al.), fork, ptrace, reiserfs, various other file systems, some VFS callbacks and parts of many drivers (especially ioctl handler). Most of these are not critical, but for example the *BKL* use in NFS or reiserfs create problems under specific workloads. There is an effort underway to remove the *BKL* use for reiserfs and also some work to minimize *BKL* in general. On the other hand it's likely that the *BKL* will always stay at least for obscure drivers.

7.3 *mm_sem* and *pagetable_lock*

In addition to global locks like the ones discussed earlier, data structure private locks can also be a problem if the data structure is shared by multiple threads. A standard example here are the *mm_sem* read-write semaphore that protects the list of mappings in a process and the *pagetable_lock* that protects the pagetable state of a process. These locks are local to a process' address space. However when the process is using multiple threads then these threads will be able to access the address space in parallel, which can cause contention on these locks. A classical example here is the initialization of a large multi-threaded computing job that causes a lot of parallel page-faults. These page-faults will all run into contention on the *mm_sem* semaphores. Semaphores are sleeping locks and may run into convoying problems where waiting threads may get stuck at the end of the wait queue for a long time (see [7] for details on this phenomena).

Another case when these locks are contended is when multiple threads call system calls that modify the memory map in parallel (for example a user-space memory allocator calling the *mmap* syscall in parallel from multiple threads to get more memory)

8 Scalability benchmarks

Scalability benchmarks will be presented in the presentation.

9 How to deal with software that does not scale?

So far we only looked at Linux kernel scalability. But what about applications running in user space? Scaling applications is out of scope of this paper, although a lot of the techniques described earlier can be applied there too. Depending on where the applications are in the scalability tuning cycle they might not scale to the full number of CPUs in the system.

The simplest (and quite common) case is a single threaded program that isn't parallel at all. Luckily they are agnostic to the number of CPUs in a system, so at least they will not run slower on a large system due to scalability issues.

A classical example would be a compiler like gcc which can only use a single CPU for compiling a program. But often a program consists of multiple files, so gcc can be relatively easily parallelized on a coarse-grained base by using tools like "make -j". This is practical application of Gustafson's law[2].

For example a single threaded media encoder can encode multiple media files in parallel by using xargs:

```
find -type f | xargs -n1 -P$(grep -c processor /proc/cpuinfo) encoder
```

This technique can be applied to many other computing-intensive command line tools.

Then there are programs which have some degree of parallelism, but do not scale to large systems (where large can be as small as a 16CPUs systems). That might be because they still have too coarse-grained locking, run into communication bottlenecks or suffer from other scalability problems. Unfortunately these programs are often written to start as many threads as the system has CPUs, but when the system is larger than their scalability limit adding more threads might actually scale negatively (as in becoming slower when more threads are added)

The first measure is to limit them to the maximum number of threads that they can successfully scale to (or of course fix their scalability problems if possible). This can be done through benchmarking to find the scalability limits and then configure the program suitably.

This of course leaves some of the CPUs idle. This can be addressed by either run other instances of the process ("cluster in a box") or running different programs on the same systems too ("server consolidation")

This raises the question how to isolate different services on such a larger box. A popular approach is to use virtualization (like Xen or KVM) to run multiple smaller guests with less CPUs on a larger system. This has relatively high overhead in terms of memory consumption and administation overhead. An alternative is also to use the new cgroups subsystem in recent kernels to run applications in kernel-level containers, limiting them to the number of CPUs they can successfully handle.

10 Conclusion

Kernel scalability is critical for Linux to run well on today's and tomorrow's systems. While a lot of work has been done in this area, and it works reasonable well for a variety of workloads today, improving scalability further is a ongoing process that needs constant effort.

References

[1] Amdahl (1967) "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities" http://www-inst.eecs.berkeley.edu/~n252/paper/Amdahl.pdf

[2] Gustafson (1988) "Reevaluating Amdahl's Law" Communications of the ACM 31(5) http://www.scl.ameslab.gov/Publications/Gus/AmdahlsLaw/Amdahls.html

[3] McKenney, Appavoo, Kleen, Krieger, Russel, Sarma, Soni (2001) "Read-Copy Update", Ottawa Linux Symposium http://www.rdrop.com/users/paulmck/rclock/rclock_OLS.2001.05.01c.pdf

[4] McVoy (1999) "SMP scaling considered harmful" http://www.bitmover.com/llnl/smp.pdf

[5] Schimmel (1994) "Unix Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers" Addison-Wesley Publishing Company

[6] McKenney, Sarma, Soni (2004) "Scaling dcache with RCU" http://www.linuxjournal.com/article/7124

[7] Vahalia (1995) "UNIX Internals: The New Frontiers", Prentice-Hall.