Thread Scheduler Efficiency Improvements for Multicore Systems

Daniel C. Frazier
Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota, USA 56267
frazi177@morris.umn.edu

ABSTRACT

Thread scheduling is a problem that has been around since the 1960s. By the early 2000s, thread scheduling was commonly believed to be solved in the Linux community. However, with the rising popularity of multiprocessor and multicore systems, and the rapidly developing requirements driven by new hardware, the thread scheduling problem space has become considerably more complex. This paper will describe some newly found issues in the Linux scheduler, their fixes, and two new schedulers designed for improved performance. Each of the developments meaningfully improve the average efficiency of popular relevant benchmarks.

Keywords

Scheduling; thread migration; multicore; multiprocessing; lock contention; last-level cache misses; efficiency

1. INTRODUCTION

Using any modern computer, there is an expectation that the operating system is running at all times (largely in the background) and that multiple programs should be able to run concurrently. Modern programs also often need to run more than one independent task at one time. A program can achieve this by employing *threads* because they allow several paths of execution to occur in parallel. Programs that involve long independent computations or programs with a graphical interface often benefit from employing threads.

For example, imagine a photo editing program that can apply an expensive filter operation. If the program was not multithreaded, the user interface and the filter operation would be executed in the same thread. When instructing the program to execute the filter on a large image, the user interface would not be able to respond to any events (like clicking the mouse) until filtering was finished. To make programs with user interfaces more responsive, we separate the interface onto its own thread and spawn new threads when the user initiates expensive operations.

Processors on most modern computing systems employ multiple cores. Some systems have multiple processors, each of which also have multiple cores. To take advantage of this hardware, threads should be distributed across the cores wisely. In *The Linux Scheduler: a Decade of Wasted Cores*, Lozi et al. describe bugs they found within the Linux scheduler that result in significant performance degradation.

The purpose of this paper is to cover some important decisions and trade-offs that are made in order to allocate tasks to cores to maximize performance. The next section will provide necessary background on threads, scheduling, and caching. The purpose of section 3 is to discuss the bugs (and their fixes) and approaches by Kumar et al. and Jo et al. to create new schedulers that overcome scalability problems with the Linux scheduler.

2. BACKGROUND

This section will broadly establish how threading, caching, locks, and scheduling works. These concepts will prepare us to establish causes of something called "cache misses" in the execution of programs using the current Linux thread scheduler. We will establish information necessary to understand some recent efforts to improve thread scheduling on multiprocessor and multicore systems on Linux.

2.1 Threads and Scheduling

Threads are tied to the *processes* that spawn them. A process always has at least one thread. Processes are typically independent of each other, while threads exist within a process. *Context switching* within a CPU is the process of saving and restoring the state of a process or thread so that execution can be paused or resumed. A process' state consists of resources that each of its threads need access to, such as compiled code and data.

The *scheduler* is the part of the operating system that is responsible for managing and distributing the CPU runtime that each of these processes and their respective threads receive. New threads and processes are added to the scheduler when they are made. [4] The current implementation of the scheduler on Linux is hierarchical and consists of modules. The primary module is the *scheduler core*, and it interfaces with modules called *scheduler classes*. The *Completely Fair Scheduler* (CFS) is the default scheduler class and is used for tasks that are not real-time critical. The core scheduler runs a tick function every $\sim$1000 times per second and delegates to the tick function of the appropriate scheduler class. Real time tasks must complete in a strict time frame (e.g. flying a helicopter). For real-time tasks, other scheduler classes are used, but out of brevity, we will not be covering these. [1]

Before we discuss how CFS schedules tasks, we should first understand how cache is used on multicore systems, how a system can maintain cache coherence, and how threads can work on the same data without conflicting with each other.

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/4.0/.
UMM CSci Senior Seminar Conference, December 2017 Morris, MN.

2.2 Cache on NUMA Systems

When a program is running, its working memory is stored in RAM. RAM exists far away from the CPU relative to the *cache*. Imagine a program that sums up an array of ten thousand integers and they all fit into RAM. It would be very slow for the CPU to request from RAM the integers it needs one at a time. Cache exists to speed this up by improving *locality*. If the system predicts a chunk of data will be used frequently, it migrates that data from RAM into cache. This improves locality because it makes the data more "local" to where it needs to be and improves performance because the system works less hard to load the data it needs.

The cache that a system has is hardware dependent. In a *non-uniform memory access* (NUMA) system, there are many levels of cache and they exist in a hierarchy. The defining property of NUMA systems is that levels of cache that are physically closer to cores or processors are faster than levels that are farther. Lower levels of cache also hold less data because of space restrictions. Most modern systems are multiprocessor multicore NUMA systems. L1 is the lowest level of cache. L1 cache is also called last *level cache (LLC)*. If data is not found in L1 cache, it is called an *LLC miss* and the data is searched for in further levels of cache. If the data can not be found in any level of cache, it is called a *cache miss* and external memory is consulted.

An important property of a multicore system that employs cache is that it should be *cache coherent*. A system that is cache coherent satisfies the invariant that any cached data that is read must be consistent with the most recent write to that portion of cached data. When data is written into an L1 cache, it must propagate up the cache hierarchy. If any *other* L1 cache holds the same data as that L1 cache, it must also refresh. The most common approach to maintain strict consistency in cache is to somehow *invalidate* cache entries. Cache invalidation is beyond the scope of this paper. [1] [6]

Context switching between two cores involves ensuring that the replacing thread has the resources it needs available on its new core's cache. Recall that a processor's state is much larger than thread's state. Because of this, context switching is typically fastest between threads who share a process because the processor's state would not need to migrate to another cache.

2.3 Synchronicity and Locks

When two threads have read and write access to the same portion of the memory, any read or write to that memory must be made synchronous to avoid race conditions. A race condition is a timing dependent error where two threads updating the same memory at the same time overwrite each other in a way that might cause the threads to function incorrectly. A system can be made synchronous by employing *locks*. Saltzer and Kaashoek defined a lock as "A flag associated with a data object, set by a thread to warn concurrent threads that the object is in use and that it may be a mistake for other threads to read or write it." [6] When a thread needs to use an object that is associated with a lock, the thread should check the lock first before operating on the object. If an object's lock is already acquired, the thread should wait until the lock is released, then acquire the lock for itself and release the lock when finished.

An important problem faced by massively parallel programs is lock contention. *Lock contention* can be found in multithreaded programs where many threads frequently compete for access to the same lock. Threads of a program that are contending and which reside on different processors experience a higher than average LLC miss rate. This is a result of cache coherency. Because both of these threads are continually modifying the same data in separate cache, each change to one cache must propagate to the other before the other core reads the data.

2.4 Completely Fair Scheduler (CFS)

The *Completely Fair Scheduler (CFS)* was introduced in version 2.6.23 (2007) of the Linux kernel. [2] We will discuss the CFS as presented in Lozi et al. The CFS is an implementation of the weighted fair queuing (WFQ) scheduling algorithm. The goal of the WFQ is to divide an arbitrary number of CPU cycles among threads, prioritizing more cycles for threads with larger weights. [4]

Threads that are running accumulate *vruntime*, which is the runtime of a thread divided by its weight. [1] Threads are organized in priority queues called *runqueues* that sort ascending on vruntime. Under normal conditions, the thread that replaces the current thread is the thread which needs most to run, which is the thread with the least vruntime, the first thread in the runqueue. If a thread with a smaller vruntime awakens, it may preempt the executing thread. [4]

On a multicore system, each core should have its own runqueue. If all cores shared a runqueue, cores would be need to make frequent synchronous requests for threads. For the scheduler to function properly and efficiently, it must keep each of the runqueues balanced. If runqueues are not balanced, then a core is left idle and needs to request work from another core to keep busy. External calls between cores are expensive and should be minimized. The CFS, runs a load-balancing algorithm that tries to keep runqueues balanced. Load balancing was simple for single-core systems, but on multi-core systems, bugs have found their way into the system and persist until at least Linux kernel version 4.3. [4]

3. METHODS

So far we have established what threads are, how cache works and where it resides on NUMA systems, how the CFS scheduler works, how locks work, and why they are important. Now, we will discuss four bugs Lozi et al. found in the CFS load-balancer and their fixes, which substantially improved scheduler efficiency. Later, we will show two new schedulers that improve efficiency for certain kinds of programs running on certain kinds of multicore NUMA systems.

3.1 Load-Balancing the CFS

Modifications were made to the CFS load balancer that introduced bugs that caused processors to remain idle even while there were threads available. Work by Lozi et al. has identified four bugs that were responsible for this behavior. These bugs have remained hidden because, while they corrode performance, they are not obvious. They do not make programs freeze or crash and their effects only last a few hundred milliseconds at a time, which is too short for common performance tools to detect. Lozi et al. designed new tools that observe the Linux scheduler more closely in order to track the source of the problems. [4]

[1] See Section 10.2.4 in Part II of Principles of Computer System Design by Saltzer, Jerome and Kaashoek, M. Frans for more information on cache invalidation.

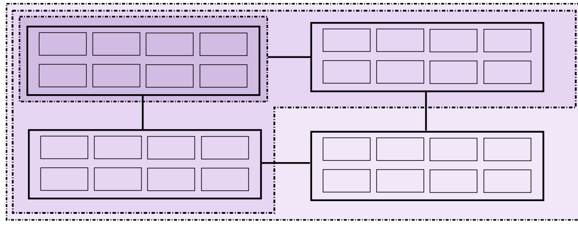

Figure 1: 32-core machine with four NUMA nodes. It takes at most two hops to get from any core to another node. The shades represent scheduling domains relative to the first core. From Lozi et al. [4]

In order to understand these bug fixes, as described in Lozi et al., we must explain a simplified version of the CFS load balancing algorithm. [4]

3.1.1 Load Metric

The load-balancing algorithm tracks a metric called *load* to best distribute threads to cores. Defining what load should be is tricky. A threads' load should be representative of the amount of CPU time it should receive relative to all other active threads. Balancing load such that each core has the same number of threads is not ideal because threads have priorities. If all of the high-priority threads happened to be placed on one core while all low-priority threads were placed on another, the low-priority threads would be receiving much more runtime than they should be in relation to the high-priority threads. Balancing load such that each core has roughly the same amount of weight is not ideal either, because, if there was one thread that was nine times more important than nine low-priority threads, that important thread would be left on a core all alone. That seems acceptable, but consider the case that this high-priority thread sleeps frequently. Its core would be left idle for an unacceptable amount of time. The idle core would need to ask other cores for more work to keep busy in the downtime, which is an expensive operation for both cores involved. [4]

The current implementation of CFS defines the load metric as a combination of a thread's weight and average CPU use divided by the number of all threads in the parent process. The division is in order to remain fairness so that two processes that have different numbers of threads of the same priority still get equal runtime. [4]

3.1.2 Load-Balancing Algorithm

A NUMA system contains *NUMA nodes*, but the definition of what a NUMA node is changes depending on how NUMA is implemented on a machine. Let us consider how NUMA nodes were defined for the system used in Lozi et al. On their system, groups of eight cores share last level cache and form a NUMA node.

For load balancing, cores exist in a hierarchy where each level is called a *scheduling domain*. The groups within each level are based on how cores share resources within the machine. The lowest level of scheduling domain is a single core. A *scheduling group* is one of the units that comprise a scheduling domain. In the machine described in Section 2.2 there were 32 cores. These cores individually represented the first level of scheduling domains. The second scheduling domains were determined by groups of eight adjacent processors which made four NUMA nodes. The third level was formed by groupings of nodes that are within one hop of each other. The final level was all of the nodes as one unit. The scheduling groups of the final scheduling domain were the four NUMA nodes. See Figure 1. [4]

Function running on each cpu *cur_cpu:*

```
1  forall sd in sched_domains of cur_cpu do
2      if sd has idle cores then
3          first_cpu = 1st idle CPU of sd
4      else
5          first_cpu = 1st CPU of sd
6      end
7      if cur_cpu ≠ first_cpu then
8          continue
9      end
10     forall sched_group sg in sd do
11         sg.load = average loads of CPUs in sg
12     end
13     busiest = overloaded sg with the highest load
             (or, if inexistent) imbalanced sg with highest
            load
             (or, if inexistent) sg with the highest load
14     local = sg containing cur_cpu
15     if busiest.load ≤ local.load then
16         continue
17     end
18     busiest_cpu = pick busiest cpu of sg
19     try to balance load between busiest_cpu and
        cur_cpu
20     if load cannot be balanced due to tasksets then
21         exclude busiest_cpu, goto line 18
22     end
23 end
```

Algorithm 1: Simplified CFS Load Balance algorithm from Lozi et al. [4] CPUs are cores.

A naive approach to load balancing would be to compare load on each core and transfer tasks from cores with the highest load to cores with the lowest load. This approach would not put into consideration improving thread or cache locality. Instead, the CFS load-balancing algorithm is executed on each CPU for each scheduling domain that the CPU is a part of, starting from the first level to last levels of scheduling domain. A simplification of the CFS load balancing algorithm can be found in Algorithm 1. The goal of each iteration of the outer loop is to find another CPU that is busier than *cur_cpu* and "try to balance the load" between them. This simplification does not explain the intricacies of load balancing, however is sufficient for our purposes of understanding the bugs that emerged.

In the CFS load balancer, one core per scheduling domain performs the load-balancing for that domain. That core is either the first core that is idle or the first core of the scheduling domain (Lines 2-9). Then, for each scheduling group within the scheduling domain, the average load is computed (Lines 10-12) and the group with the highest load is determined to be the busiest group (Line 13). If the busiest group is less busy than the group containing this CPU (local), then the load is considered balanced for this level and continues on to consider the next scheduling domain (Lines 14-17). At lines 18 to 23, the current core balances load between the busiest core and itself. [4]

Several optimizations made the scheduler such that it runs the load balancing operation less often. Lines 2 to 8 is an example. If any of the cores are idle, the first idle core in the scheduling domain is chosen to consider load balancing. If all cores are busy, the first core of the scheduling domain is chosen. There is also a power optimization where idle cores do not run the load balancing algorithm until awoken by an overloaded core. To improve cache locality, when an idle thread is awoken by an active thread, the scheduler prefers to assign the awoken thread to the same core so that they share a last-level cache. [4]

3.2 Bugs and Fixes for the CFS

Now that we know how the load balancer makes some decisions, we will now dive into the bugs that occur as a result of the complex, strict requirements that have built up over time on the CFS. Performance results from Lozi et al. were gathered using the *NAS Parallel Benchmark (NPB)*. The NPB is a set of parallel programs that are executed and monitored to evaluate the efficiency of massively parallel computing systems (supercomputers).

3.2.1 The Group Imbalance bug

Lozi et al. found that the scheduler was periodically not balancing load due to two reasons, the hierarchical design of scheduling groups and complexity of the load metric. The researchers were running a 64 threaded Process A on node A simultaneously with a single-threaded Process B on node B. Because of the group schedule feature in the definition of the load metric where the load of a thread is divided by the number of threads in the parent process, the amount of load for each of A's threads are 1/64 of the load of the one B thread. On their system, they observed that two nodes were underloaded and should have been stealing work from more loaded cores on other nodes, but this did not happen. This is due to the hierarchical design of the load balancer. When the load balancer considers stealing threads, it does not consider the load of a single core but rather the load of the whole scheduling group. In this case, the node with the B process had the same load as the node with the A process, so node B did not try to steal work from node A even though there were cores idle and available for work. [4]

They fixed this problem by, instead of defining the load of a scheduling group as the average load of cores in the group, defining it by the load of the core with the minimum load in the scheduling group. By fixing this issue, the efficiency of the single-threaded process B remained the same, however the completion time of process A decreased by 13 percent. On a certain parallel program within the NPB called *lu*, this bug fix improved performance by 13 times because the bug compounded lock contention by colocated threads. [4]

3.2.2 The Scheduling Group Construction bug

There is a function in Linux called *taskset* which allows an application to pin its threads to certain available cores. On certain machines, sequentially numbered NUMA nodes may be located more than one hop from each other. When a new thread is spawned on Linux, it is placed on the same node as the parent thread. The first scheduling group is constructed by all adjacent nodes to one node (say, Node 0), and following scheduling groups are constructed from nodes that were not in any of the previous groups and their adjacent nodes. On a system it is possible for two nodes to

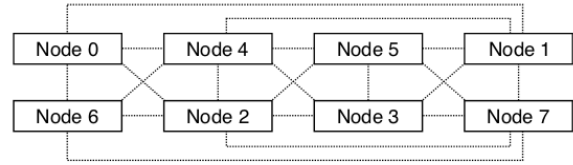

Figure 2: 8-node AMD Bulldozer machine from Lozi et al. [4]

be within one hop of the starting nodes of each group, but be two hops from each other. See Figure 2. [4]

Within that system, the two scheduling groups constructed are 0,1,2,4,6 and 1,2,3,4,5,7. Nodes 1 and 2 are two hops from each other but occur in both scheduling groups. When load balancing runs on Node 2 and it should steal work from an overloaded Node 1, it will look between the scheduling groups to compare which is lower. But both scheduling groups contain Node 1 and 2, so both scheduling groups' average load will be the same! Node 2 will never steal work. [4]

They fixed this bug by constructing scheduling groups relative to each core's own perspective. This allows nodes that are two hops away that should otherwise be able to share work to be a part of different scheduling groups and successfully contend for load balancing. Fixing this bug improved efficiency of 9 parallel programs in the NPB problems mostly by around 2 times, but for one problem, *lu*, up to 27 times. *lu* is an extreme example where all threads were being allocated only on one core. [4]

3.2.3 The Overload-on-Wakeup bug

There exists an optimization in thread wake-up code where threads that are awakened by other threads are placed on the same core for improved cache locality. This becomes a problem when the core with the requesting thread is already overloaded. The thread will join the runqueue on this busy core rather than consider being migrated to an idle core elsewhere. For some workloads this is acceptable, but simply making the most of cache is not always the best decision. This bug occurs primarily on database systems. The following sequence of events will illustrate how the Overload-on-Wakeup bug occurs. [4]

Consider two nodes running on a database system. Node A and Node B both have a database thread. Assume a temporary thread is created by the kernel on Node A for some background function such as system logging. During this time, the load balancer detects that the node is overloaded due to the new thread, and decides to migrate some thread to Node B. If it decides to move the temporary thread, that will pose no issue for performance. If it migrates the database thread, two database threads, which sleep and awaken frequently, will be running on the same node. The scheduler does not differentiate between cores that are idle for a short time versus a long time, so when the scheduler considers thread migration destinations, it might see a node, which happens to have a database thread that is currently idle, and decide that the node needs more work. This overloads the core with two intermittently busy threads. [4]

Lozi et al. fixed this bug by modifying the thread wake up code. The scheduler first checks to see if the core that the thread was already assigned to is idle. If it is, then wake up the thread on that core. If the power manager policy

of the system is set up so that cores never enter low-power mode and there are idle cores, the scheduler chooses the core that has been idle for the most time and assigns the thread to that core. This fix is only relevant to workloads where threads frequently sleep. Their fix improves performance by 13.2% on a full TPC-H database workload but up to 22.2% for certain queries. [4]

3.2.4 *The Missing Scheduling Domains bug*

This bug was already fixed but regressed on Linux kernel version 3.19 (up to at least 4.3) when an important line of code was removed in a refactor. Removing this line caused the system to misrepresent the number of scheduling domains that are available for threads to be distributed. This ended up causing load-balancing to never happen on all nodes, meaning processes, subprocesses, and their threads to stop looking for other NUMA nodes to join. Nodes that do not have any threads will never receive threads and nodes that do have threads will accumulate all of the spawned threads. This bug required that one of the cores become disabled and re-enabled. So while rare, reintroducing the removed line of code increased efficiency in a certain tested program by a maximum of 138 times. [4]

3.3 Shuffler

Researchers Kumar et al. suspected lock contention to be a significant actor in the non-scalability of parallel programs on Linux and Solaris. The operating system used was Oracle Solaris 11^{TM}. They assembled 33 parallel programs from various benchmarks and monitored time spent acquiring locks and the number of LLC misses experienced while running these programs. Twenty of the 33 programs had high lock times (>5%).

As mentioned in the concepts section, lock contention can be found in multithreaded programs where many threads repeatedly compete for access to the same lock. The costs in this involve the transfer of the lock and the propagation of cache associated with the lock. The problem is further compounded if the threads are not located within the same processor. If the threads of that multithreaded program were prioritized to be placed on one processor, then the program would experience a lower lock times and LLC miss rate.

Neither the CFS nor the Solaris scheduler differentiate between threads of a single-threaded program versus the threads of a multithreaded program. This prevents the scheduler from using that metadata in its thread distribution mechanism. The following thread scheduler named *Shuffler* by Kumar et al. takes this into account. The Shuffling framework was designed for multiprocessor multicore NUMA systems, and was implemented on a 64-core 4-processor machine running Oracle Solaris 11^{TM}. [3]

3.3.1 *The Shuffling Framework*

The Shuffling approach is to take into account which threads are contending for locks on what processors and migrate *whole threads* (rather than just lock and cache) such that they share processors. Threads that are reported to have higher lock acquisition times are threads that are assumed to be requesting locks from outside of their processor. These are the threads that should be migrated to share processors. Details on the shuffling framework are as follows.

Monitor Threads — First, Shuffler monitors and records the amount of time user threads spend acquiring locks.

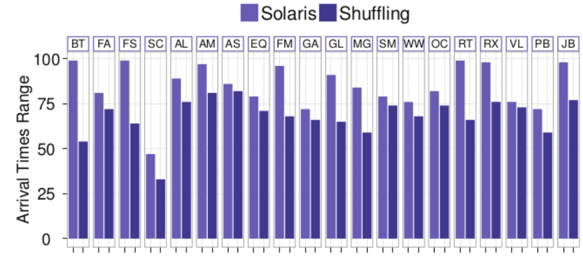

Figure 3: Lock arrival times ranges with Solaris vs. Shuffling. From Kumar et al. [3]

Form Thread Groups — If the sum of sampled lock times exceed a certain threshold, then all threads are sorted by their lock acquisition times and grouped by their order in the sorted data structure. There are as many groups as processors. This procedure is run every 200 ms to continue to form groups of threads that should share processors.

Perform Shuffling — On an iteration of the Shuffling procedure, it ensures that if any threads are not on processors that they were grouped to, they are migrated. Threads that are already on the processor that they were assigned to do not migrate. If threads continue to contend for locks with the same threads, lock times should remain below the threshold, and so Shuffling is not always run. [3]

3.3.2 *Shuffler Performance*

Now that we know how the Shuffling Framework works, let us review the results that implementing it gives us. Kumar et al. chose certain programs from several benchmarks that contained high lock contention to assess Shuffler performance. Each of the programs are multithreaded. The one they highlight the most was the Body Tracking (BT) algorithm whose performance improved by 54 percent. Program SC improved by 29 percent. The rest of the 18 programs improved between 4 and 19 percent. Figure 3 compares the efficiency of Shuffler versus Solaris for the same 20 programs. With these results we can say that multithreaded programs that had high lock contention perform faster under Shuffler than Solaris. They also tested the remaining problems that did not have high lock times. These programs received negligible improvements (less than 0.5%) in execution time. For more detailed results, see Kumar et al. [3]

We just saw that lock contention is a problem for multithreaded parallel programs. Next we will cover a scheduler that takes a different approach to alleviating the lock contention problem. FLSCHED is a scheduler whose implementation has no locks, reduces context switches, and makes more efficient scheduling decisions than CFS. [2]

3.4 FLSCHED for Xeon Phi

FLSCHED was built for maximal efficiency on manycore processors. Manycore processors contain upwards of about 20 cores. They continue to become more powerful and, as such, more popular. The Xeon Phi is a family of manycore coprocessors that allow a primary processor to offload expensive work. The latest version of Xeon Phi as of September 2, 2017 had up to 76 cores. The number of cores a processor has is expected to increase given the popularity and importance of highly-parallel algorithms such as machine learning.

Scheduler	bt	**cg**	ep	ft
CFS(%)	7.29	**10.73**	0.97	5.34
FLSCHED(%)	3.05	**4.11**	1.10	4.04

Scheduler	is	**mg**	**sp**	**ua**
CFS(%)	0.21	**6.84**	**8.23**	**14.63**
FLSCHED(%)	0.12	**2.85**	**3.58**	**5.96**

Table 1: Percent of time spent executing a certain type of lock called a spin lock. Lock times were measured throughout the runtime of the NPB with 1,600 threads. From Jo et al. [2]

The CFS was not built for this degree of parallelism. The cost of context switching continues to increase as hardware becomes capable of solving larger problems. Unfortunately, the density of cores on manycore processors causes them to have very small L1 cache. As the number of cores increase, the negative impact of lock contention on scheduler performance increases exponentially.

3.4.1 Lockless Thread Scheduler

The design goals and requirements for CFS differ from FLSCHED. CFS was designed with responsiveness, fairness, and load-balancing in mind because it was intended for desktop and server use. FLSCHED was designed strictly with efficiency and computational throughput in mind. In the CFS, a great deal of state information is considered to decide what threads run and when. Since FLSCHED was intended for manycore parallel processors, it is more impactful to simply make decisions faster rather than more purposefully. [2]

The CFS implementation employs 11 locks which are used for load balancing mechanisms, runqueue management, runtime statistics updates, and additional scheduler features. FLSCHED itself has no locks. It manages this by removing runtime statistics entirely from the scheduler, as it does not depend on them to make decisions. FLSCHED does not use periodic load balancing, so it does not provide the feature to limit maximum CPU bandwidth nor any of the CFS group features. While the FLSCHED substitutes for the CFS, it still runs through the scheduler core which employs locks. Contexts switches are requested, rather than performed outright. Commitments to context switch requests are delayed on purpose to minimize the number of context switches. [2]

Timeslices in FLSCHED are assigned Round-Robin. The only managed scheduling information is the timeslices that threads receive. Thread preemption in FLSCHED occurs due to various reasons but mostly due to priority. Preemption is not performed immediately. Instead, FLSCHED reorders runqueues such that the important thread will come next after a normal task switch. [2]

3.4.2 FLSCHED Performance

To evaluate FLSCHED performance, Jo et al. used the NPB. NPB version 3.3.1 consists of 10 programs, but two of which can not run on the Xeon Phi due to memory constraints. These programs were run on FLSCHED and CFS. They found that FLSCHED scales better as the number of threads increases for six of those eight programs. Efficiency of the *ep* and *is* programs degraded under FLSCHED. [2]

The researchers traced the cause of the efficiency improvements to minimizing a certain kind of lock called a spin lock from their scheduler. A spinlock is a type of lock that loops indefinitely checking whether it has unlocked yet. These locks are used if the lock is expected to take a short amount of time to avoid giving up the core to another context. Table 1 shows the percent of time that each of these schedulers spend processing spin locks. Time spent on spin locks was more than halved for programs *cg*, *mg*, *sp*, and *ua*. For more specifics on FLSCHED, see Jo et al. [2]

4. CONCLUSIONS

Most of the modifications to the CFS in Section 3.2 improves user experience and system efficiency for Linux systems that use the CFS. The CFS functions well for the user and server systems it was designed to support but falls short when it comes to highly parallel programs. The Shuffler makes highly parallel programs running on multicore multiprocessor systems function more efficiently than on the CFS by migrating threads to make better use of hardware. The FLSCHED makes manycore parallel machines intended for problem-solving more efficient by removing features that a desktop or server system would normally have and implementing a simple, greedy approach to program execution.

Acknowledgments

Thanks to advisor Nic McPhee, and friends and classmates Skye Antinozzi, Dan Stelljes, and Miranda M. for providing invaluable feedback and corrections.

5. REFERENCES

[1] N. Ishkov. A complete guide to linux process scheduling, 2015. [Online; accessed 21-November-2017].

[2] H. Jo, W. Kang, C. Min, and T. Kim. FLsched: A lockless and lightweight approach to OS scheduler for Xeon Phi. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, APSys '17, pages 8:1–8:8, New York, NY, USA, 2017. ACM.

[3] K. Kumar, P. Rajiv, G. Laxmi, and N. Bhuyan. Shuffling: A framework for lock contention aware thread scheduling for multicore multiprocessor systems. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 289–300, Aug 2014.

[4] J.-P. Lozi, B. Lepers, J. Funston, F. Gaud, V. Quéma, and A. Fedorova. The Linux scheduler: A decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 1:1–1:16, New York, NY, USA, 2016. ACM.

[5] O. Mutlu. Computer architecture: Simd/vector/gpu, 2017. [Online; accessed 28-October-2017].

[6] J. Saltzer and M. F. Kaashoek. *Principles of Computer System Design*. Morgan Kaufmann, 2009.