

# Thread Scheduler Efficiency Improvements for Multiprocessor Multicore Systems Draft

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

This is an incomplete Draft

## ABSTRACT

[Abstract contents]

## Keywords

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

## 1. INTRODUCTION

[Introduction contents]

## 2. BACKGROUND

In this Section we will broadly establish how threading, scheduling and caching works. We will establish causes of cache misses in highly parallel programs mediated by the current implementation of thread scheduler written for Linux.

### 2.1 Threads and Scheduling

Using any modern computer, there is an expectation that the operating system is running at all times (largely in the background) and that multiple different user programs and system programs should be able to run concurrently. Modern computer programs also often need to run more than one independent task at one time. This can be accomplished by employing *threads*, or equivalently, making the program *multithreaded*. Programs that involve long independent computations or programs with a graphical interface often benefit from employing threads.

For an example, in the case of image editing software, say you are working on a large picture and decide to perform an expensive filtering operation. If the program wasn't multithreaded, the user interface and the filter operation would be executed in the same thread (same *context*). When you instruct the program to execute the filter, the user interface wouldn't be able to respond to any events (like clicking the mouse) until the filter operation was finished. To make the program more responsive, separate the user interface into its own thread and employ more threads when the user initiates an expensive function.

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 storing and restoring the state of a process or thread so that execution can be paused or resumed. [4]. Context switching is typically faster for threads than processes because most of the state information between threads in a process is the same [5].

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 [2].

### 2.2 Completely Fair Scheduler (CFS)

Scheduler implementations vary per operating system. The scheduler used in Linux is called the Completely Fair Scheduler (CFS.) We will discuss the CFS as presented in Lozi et al. [2]. The CFS is an implementation of the weighted fair queuing (WFQ) scheduling algorithm. The goal of the WFQ is to divide available CPU cycles among threads, prioritizing more cycles for threads with larger weights.

Threads that are running accumulate *vruntime*, which is the runtime of a thread divided by its weight. Once a thread's *vruntime* is greater than the amount of runtime that the thread was scheduled for and there is another thread waiting to take this thread's place, the running thread is preempted from the CPU. A thread may also become preempted if another thread with a smaller *vruntime* awakens. Threads are organized in a red-black tree called a *runqueue* in which the leftmost node is always the thread with the smallest *vruntime*. [2]

This algorithm becomes much more complex for multi-core systems. Each core must have its own *runqueue*. If all cores shared one *runqueue*, each core would need to frequently make expensive calls for thread context information from other cores. [2]. For the scheduler to function properly and efficiently, it must keep each of the *runqueues* balanced to minimize expensive external calls and maximize the use of available cores. Most schedulers, including the CFS, run a load balancing algorithm that tries its best to keep *runqueues* balanced. Load balancing was simple for single-core systems but with multi-core systems, bugs have existed even until today. Before we can meaningfully discuss these bugs, we must first understand how caching works for most multiprocessor multi-core systems.

"Context switches are on a critical path, so they must be fast": Understand what the 'critical path' is.

## 2.3 Cache on NUMA Systems

*Non-uniform memory access (NUMA)*: There are many levels of cache and they exist in a hierarchy. L1 Cache exists on a processor or on a small group of processors. L2 (and higher) Cache exists on increasing sizes of groupings of processors. Cache misses happen and they reduce performance substantially.

It's faster to access data in L1 Cache than L2 and above. Allocating related threads to processors such that they share cache improves locality of cache lookups and are less likely to have cache misses.

Include cache locality into the cache section because it works.

## 2.4 Faults of the CFS

CFS modifications were made to work for multiprocessor systems and it is buggy and leaves threads waiting while a processor is idle for short, previously undetected amounts of time. Give stats for time idle [2]. Kumar et al. defines LLC miss rate as the last level cache misses per thousand instructions (MPKI) [1].

## 3. METHODS

So far we've seen how threads are being mismanaged by the CFS and how they impact performance. Next, we'll look at some recent developments that reduce lock contention.

### 3.1 Shuffler and Jumbler

The CFS doesn't 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 schedulers improve upon this.

#### 3.1.1 Shuffler

Performance of multithreaded applications on multicore multiprocessor systems with high lock contention is dependant on the distribution of those threads across processors. The Shuffling approach is for the scheduling algorithm to take into account what threads are contending for locks on what processors and migrate threads to share processors. The Shuffling Framework does this by the following. First we find the expected arrival time of locks on threads. Then we sort these threads by their expected arrival times and group them in as many groups as there are processors. Then we distribute these groups of threads to their own respective processors. See Algorithm 1. The migration of threads between processors is costly, but the thread that is waiting during that time is not doing any useful work anyway. In addition, contending threads that are co-located in one processor can share data much faster and avoid LLC misses. For these reasons it is preferable to migrate the whole thread rather than its data and the lock. This will be shown in the Performance section. [1]

For the first step of the algorithm you must find the expected arrival time of threads. The *lock time* of a thread is measured by the percent of time it spends waiting for locks. This is only done for threads in user space. There exists a daemon thread that contains a data structure that maps

threads to their lock times and processor ids. For the monitor we must choose a rate to sample lock times and a rate to perform thread migration (shuffling). We use `prstat(1)` to monitor lock times. Kumar et al found that finer sampling rates allow for detailed monitoring but also more overhead. They found that for sampling rates less than 200 ms, the overhead was significantly higher. For a lock sampling rate of 200 ms the process of sampling took less than 1% system time. The Shuffling interval was chosen by experimentation. We tested various shuffling intervals on 20 programs and chose 500 ms.

On an iteration of the grouping-forming procedure (every 200 ms), the daemon checks the total amount of time that was spent resolving locks on each thread and if that time exceeds a preset limit, then groups are formed again.

On an iteration of the shuffling procedure (every 500 ms), shuffling checks to make sure that if any threads aren't on processors that they were grouped to, they are migrated. Threads that are already on the processor that they are assigned do not migrate. If nothing substantially different changed in how threads interacted, the shuffling step is effectively skipped. [1]

**input** : N: Number of threads;  
C: Number of Processors.

**repeat**

**i. Monitor Threads** – sample lock times of N threads.

**if** *lock times exceed threshold* **then**

**ii. Form Thread Groups** – sort threads according to lock times and divide them into C groups.

**iii. Perform Shuffling** – shuffle threads to establish newly computed thread groups.

**end**

**until** *application terminates*;

**Algorithm 1:** The Shuffling Framework as presented in Kumar et al.

#### 3.1.2 Jumbler

#### 3.1.3 Shuffler and Jumbler Performance

In Kumar et al we studied the lock times of 33 programs on a 64-core, 4-processor machine running Oracle Solaris 11. We identified 20 of those programs that experienced overall high lock times and used those programs to compare shuffling versus the standard scheduler.

## 3.2 FLSCHED for Xeon Phi Manycore Processor

[Body text]

### 3.2.1 Lockless Thread Scheduler

[2, 3] [1]

### 3.2.2 FLSCHED Performance

[Body text]

## 4. CONCLUSIONS

[Conclusion text]

## Acknowledgments

## 5. REFERENCES

- [1] 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.
- [2] 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.
- [3] U. B. Nisar, M. Aleem, M. A. Iqbal, and N. S. Vo. Jumbler: A lock-contention aware thread scheduler for multi-core parallel machines. In *2017 International Conference on Recent Advances in Signal Processing, Telecommunications Computing (SigTelCom)*, pages 77–81, Jan 2017.
- [4] Wikipedia. Context switch — Wikipedia, The Free Encyclopedia, 2017. [Online; accessed 11-October-2017].
- [5] Wikipedia. Thread (computing) — Wikipedia, The Free Encyclopedia, 2017. [Online; accessed 10-October-2017].