Thread Scheduler Efficiency Improvements for Multicore Systems

Daniel Collin Frazier

Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota, USA

18 November 2017
UMM, Minnesota

Introduction

- *Thread scheduler* is an important system component that manages the processing programs receive in a given time
- Always running, so it must be efficient
- Most computers before 2001 were equipped with one processor containing one core
- At the end of the single-processor single-core era (early 2000s) thread scheduling was largely considered a solved problem by the Linux community

"...not very many things ... have aged as well as the scheduler. Which is just another proof that scheduling is easy."

Linus, Torvals, 2001 [2]

Introduction

Popular hardware changed rapidly throughout the 2000s and those developments made thread scheduler implementation much more complex

One of these driving changes was the increasing affordability and adoption of multicore systems

Unfortunately, this increasing complexity led to bugs

A Decade of Wasted Cores

Lozi et al. found four bugs in the Linux thread scheduler and fixed them [2]

Undetected by pre-existing diagnostic tools, required the development of new tools to notice them

https://goo.gl/3wsfVU

- The Scheduling Group Construction bug
- The Group Imbalance bug
- The Overload-on-Wakeup bug
- The Missing Scheduling Domains bug

Scheduling Group Construction bugfix results

To gauge improved efficiency, Lozi et al. used a group of 9 computational applications that heavily utilize threads called the *NAS Parallel Benchmark (NPB)*, developed by NASA

Application	Time w/ bug (sec)	Time w/o bug (sec)	Speedup fac
is	271	202	1.33
mg	49	24	2.03
lu	1040	38	27

Table: "NPB applications with least, median and most improvement between a bugged and fixed scheduler" [2]

Group Imbalance and Overload-on-Wakeup bugfixes

Bug Fixes	TPC-H request #18	Full TPC-H benchmark
None	55.9s	542.9s
Group Imbalance	48.6s (-13.1%)	512.8s (-5.4%)
Overload-on-Wakeup	43.5s (-22.2%)	471.1s (-13.2%)
Both	43.3s (-22.6%)	465.6s (-14.2%)

Table: "Impact of these bug fixes on a popular commercial database (values averaged after five runs.)" [2]

They disabled and re-enabled a core before each run of the benchmark

Missing Scheduling Domain bugfix results

Occurs rarely, but easily reproducible

Application	Time w/ bug (sec)	Time w/o bug (sec)	Speedup fac
ep	72	18	4.0
mg	81	9	9.03
lu	2196	16	137.59

Table: "NPB applications with least, median and most improvement between a bugged and fixed scheduler" [2]

Outline

Outline

Introduction

- The computer processor is responsible for executing compiled code.
- A single-core processor has one *processing unit*, while a multi-core processor has many *processing units*.
- A processor with multiple cores allows it to perform tasks concurrently on each core

Using Threads

Threads allow a program to run multiple independent tasks at the same time

- Imagine you're using photoshop but it has one thread
- You load a large image and perform an expensive filter operation

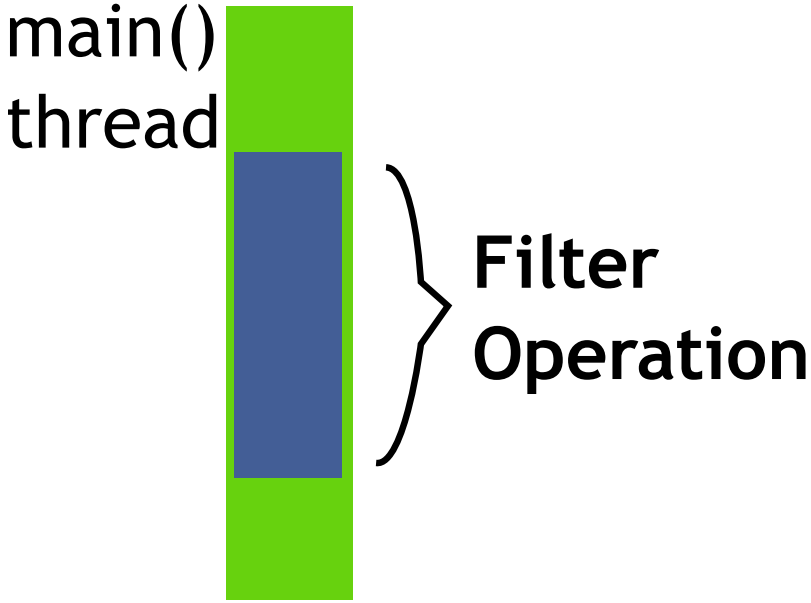


Figure: Running a filter operation in a hypothetical single-threaded version of photoshop

Using Threads

- Useful for programs:
  - with long, mostly-independent computations
  - with a graphical interface

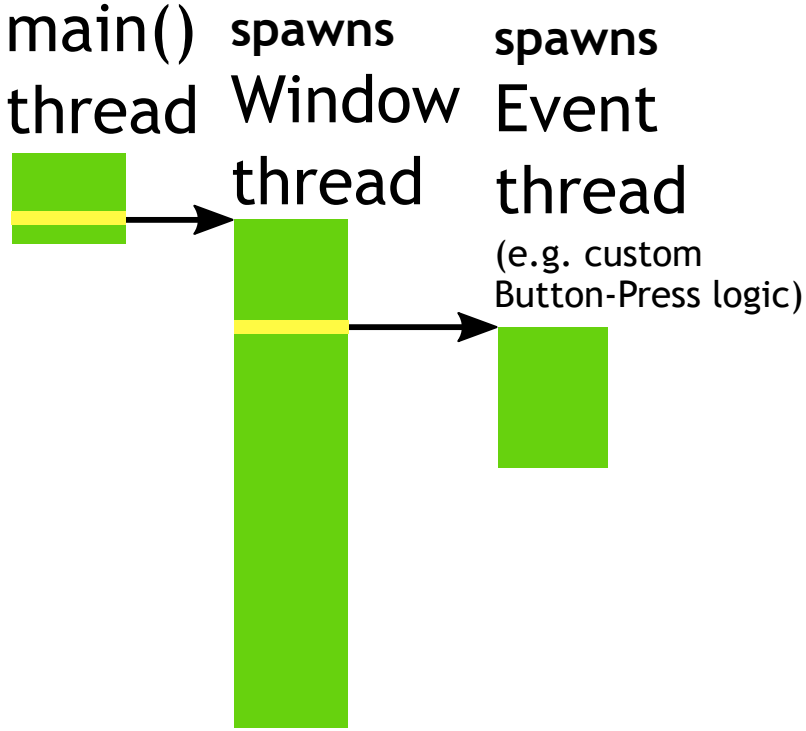


Figure: Example GUI Program. **Three** threads are created within **one** process

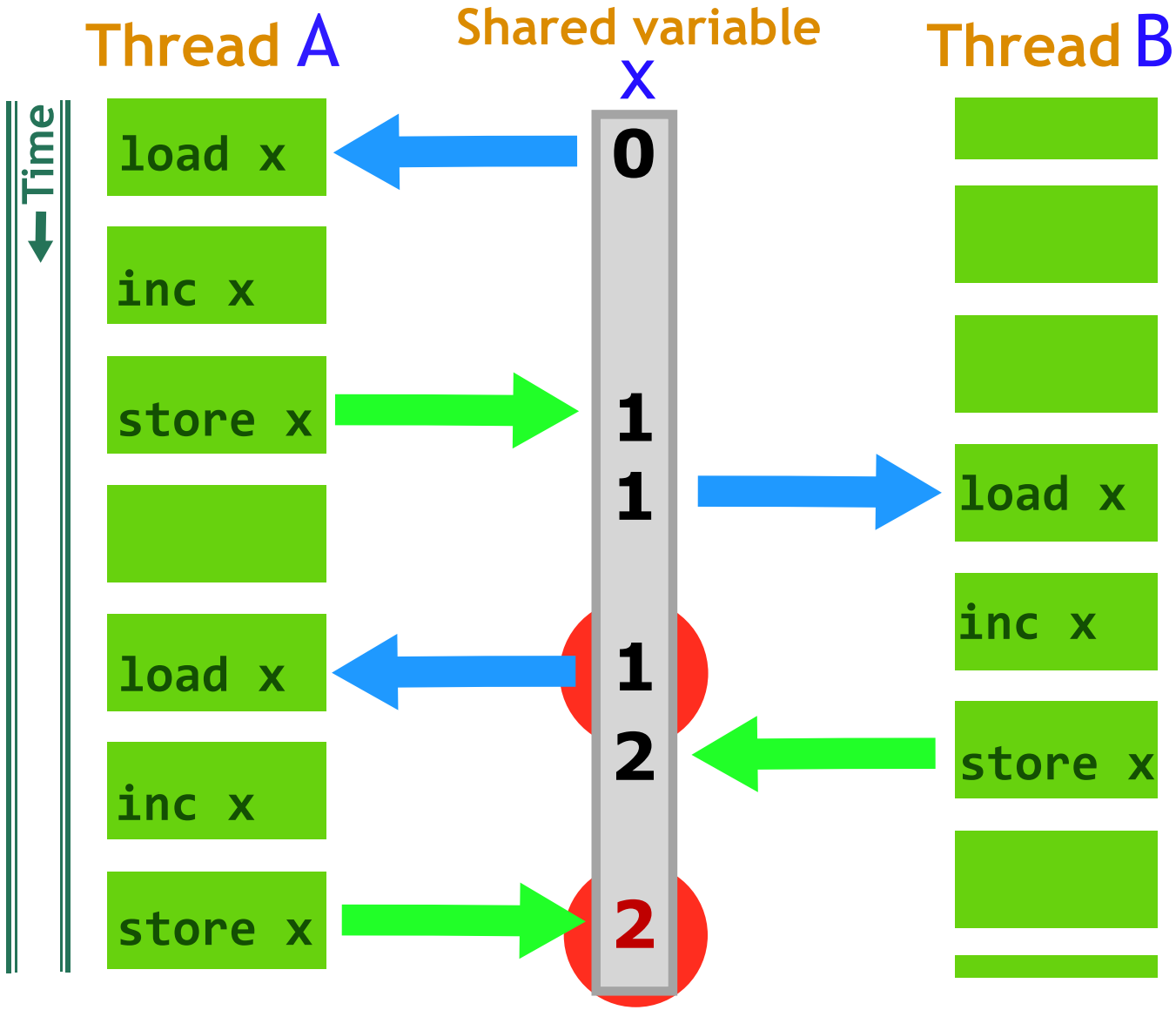
Race Conditions

One problem multithreaded programs face are called *Race Conditions*

Defined in Saltzer and Kaashoek as “A timing-dependent error in thread coordination that may result in threads computing incorrect results”

Let’s see an example where two threads increment a shared variable

Race Condition Example

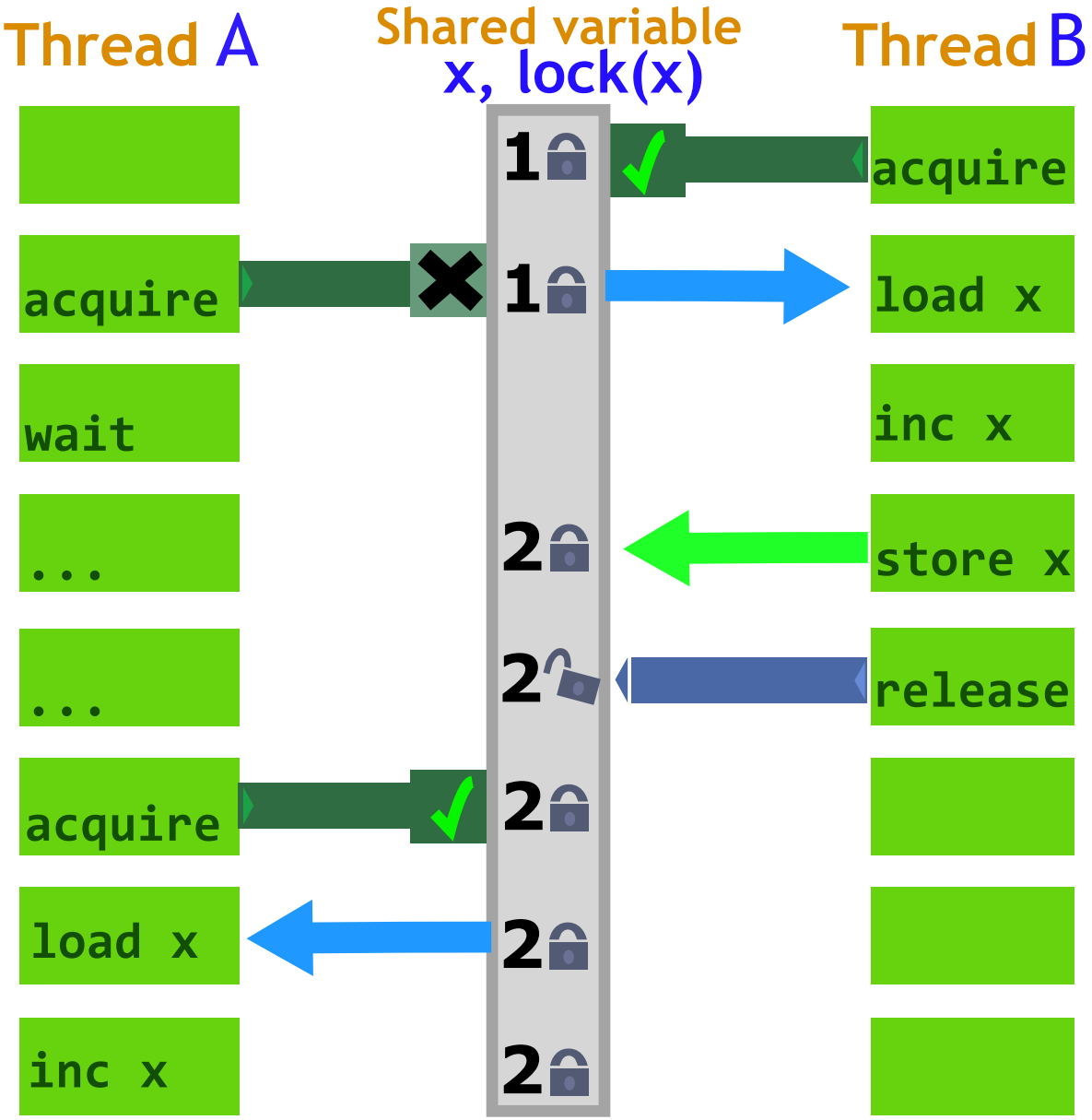


Synchronicity and Locks

- Race conditions can be fixed by controlling access to shared data.
- This control is achieved by employing locks
- *Locks* secure objects or data shared between threads such that only one thread can read and write to it at one time
- When a thread *locks* a lock, that thread **acquires** the lock
- When a thread *unlocks* a lock, that thread **releases** the lock

Now, let's fix the race condition in the previous example using locks

Lock Example



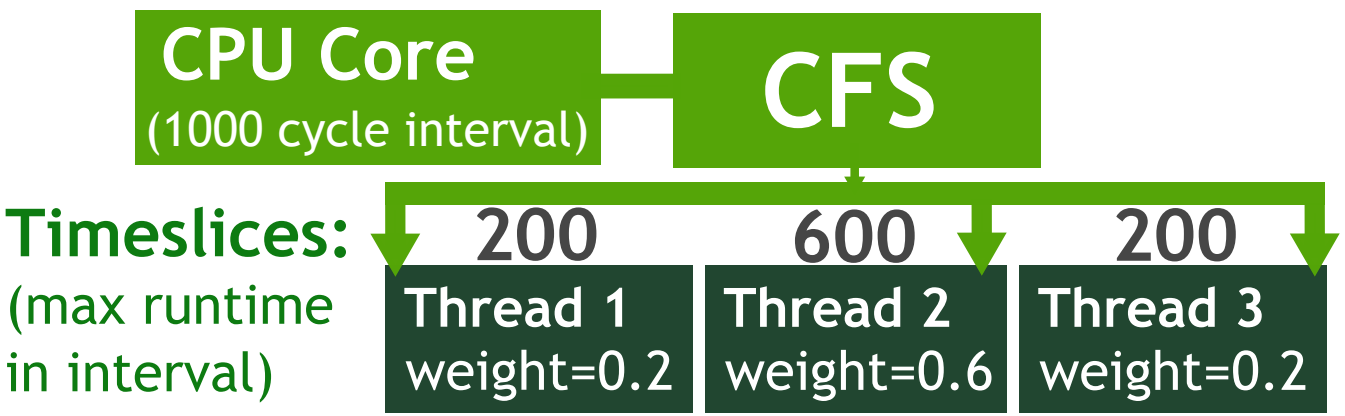
Outline

Completely Fair Scheduler (CFS)

- Default Linux thread scheduler (there are others)
- Handles which threads are executed at what times on this core
- Spend a *fair* amount of runtime on all threads

Completely Fair Scheduler (CFS)

- Like any program, runs on one core
- Makes sure all threads run *at least once* within an arbitrary interval of CPU cycles
- Distribute *timeslices* (max CPU cycles) among threads
- Threads with higher priority (weights) get larger timeslices
- Monitors the number of cycles that the running thread receives and switches it out when it exceeds its timeslice



Runqueues

- The data structure within the CFS that contains threads is called a runqueue
- A *runqueue* is a priority queue that sorts for threads that have received the least cycles in the current interval
- When a thread reaches its maximum time, the first thread in the runqueue is chosen to replace

Runqueues on Multiple Cores

If each core needs work to do, how are threads distributed?

Before we can answer this question, we need to know some about switching threads, cache and processor state

Context Switching

- The scheduler *switches* active threads on cores by saving and restoring thread and processor state information.
- These switches are called *context switches*
- Scheduler performance

Process and Thread State

Process State

Consists of resources that each of its threads should have access to including compiled code, and data.

Thread State

Scheduler uses this information to pause and resume a thread's execution

Cache

- Memory and cache exists in a hierarchy
- How cache is arranged depends on the machine:
  - A cache can exist *on*, *built-in* to, or *outside* a processor
  - Cache can be one per *n* processors or one per *n* cores

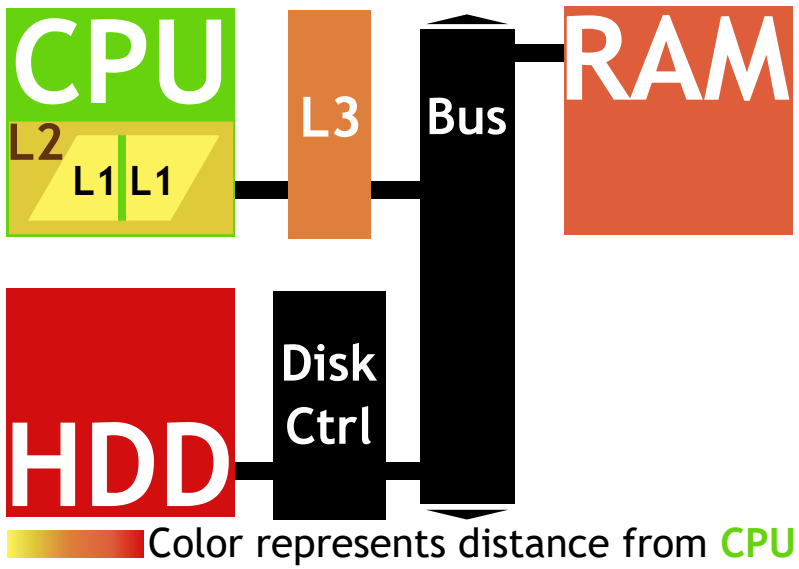


Figure: Distance of various forms of memory from CPU

Cache

- *Locality*: Speed of memory read and writes decrease as distance from CPU increases
- Cache is the fastest form of memory
- *Cache coherence:* Any changes to memory shared by two caches must propogate to the other to maintain correctness

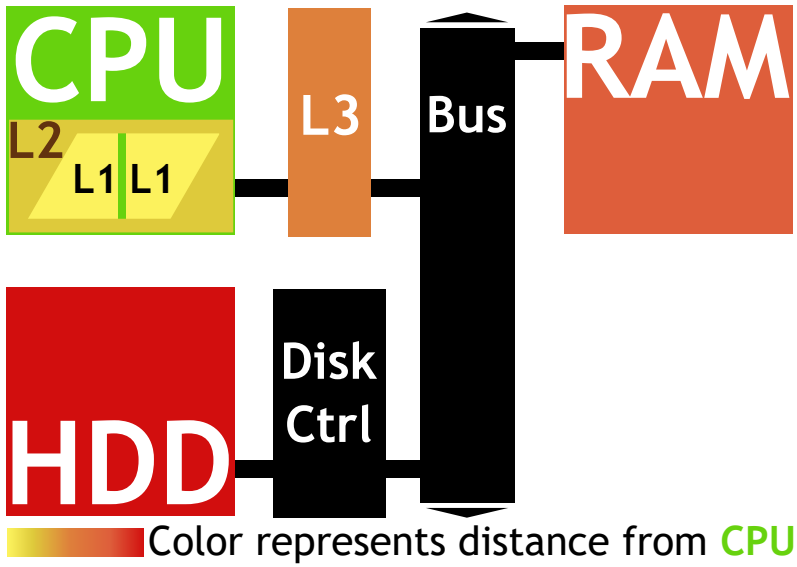


Figure: Distance of various forms of memory from CPU

Runqueues on Multiple Cores (Revisited)

Since process states are heaver than thread states, context switches between threads of different processes are more expensive

- If all cores shared one runqueue, access and changes to it would need to be synchronous and cache-coherent
- This would slow the system to a crawl
- So each core has its own runqueue and threads

- In order to best take advantage of available cores, the load on each of the core's runqueues must stay balanced.
- Most schedulers, including the CFS periodically run a load-balancing algorithm

Outline

Shuffler and FLSCHED

Both schedulers aim to solve the same problem, but for different processor types.

Shuffler	$\rightarrow$	multiprocessor multicore
FLSCHED	$\rightarrow$	single-chip manycore processor

Problem: As you add more threads to parallel computing problems, they start to operate slower rather than faster!

The term for what these systems face is called *lock contention* and it is to blame for the high acquisition times.

Shuffler

By researchers Kumar et al.

(in short, unfinished)
Shuffler

- Monitors the lock acquisition times of various executing threads
- Isolates groups of threads that have similar lock acquisition times, and migrate those threads to the same processor, assuming that those groups of threads may be contending with eachother for the same lock and would both have faster acquisition times if they were colocated.

FLSCHED

(in short, unfinished)
Designed by Jo et al. with manycore processors (specifically the Xeon Phi) in mind.

The Xeon and Xeon Phi manycore processors come with varying degrees of cores, from 24 to 76 cores. With such potential for parallelism and as the number of cores rise, any small hitch in a critical system component can lead to significant performance degradation.
The hithces that they isolated were due to locks emplaced by features and requirements of the CFS scheduler.

One requirement to rule them all: EFFICIENCY!!!

FLSCHED Aims to improve the efficiency by removing all locks from the scheduler component. In order to do this, they gutted the requirements and features of the CFS scheduler and simplified.

The requirements that they removed are:

- Fairness,
- Responsiveness,

The features that they removed were:

- Group scheduling

FLSCHED is a scheduler implementation without any locks[1].

Lock contention may exist in FLSCHED, but it isn't a problem in the same way as in Shuffler because none of the cores exist on other processors, there is only one processor, so that latency is not a concern.

[1] In Jo et al. they clarify it is still built on top of the *scheduler core* which still locks in it.

Outline

Conclusions

todo

Thanks!

Thank you for your time and attention!

Questions?

References


H. Jo, W. Kang, C. Min, and T. Kim.
FLsched: A lockless and lightweight approach to OS scheduler for Xeon Phi.
In Günther Raidl, *et al*, editors, *GECCO '09*, pages 1019–1026, Montréal, Québec, Canada, 2009.


R. Poli and N. McPhee.
A linear estimation-of-distribution GP system.
In M. O'Neill, *et al*, editors, *EuroGP 2008*, volume 4971 of *LNCS*, pages 206–217, Naples, 26-28 Mar. 2008. Springer.

See the GECCO '09 paper for additional references.