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 that have aged as well as the scheduler. Which is just another proof that scheduling is easy."

Linus, Torvals, 2001 [2]

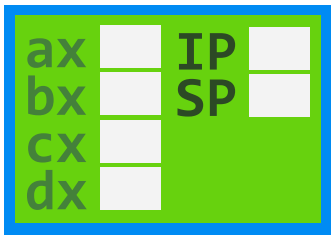
Introduction

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

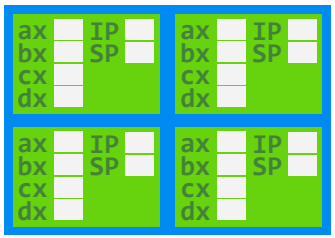
One of these changes was the development of multicore systems

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 processing unit contains *registers* which can be seen as a snapshot of the current state of a running program on that processing unit
- A processor with multiple cores allows it perform tasks concurrently on each core



Registers on a Single-core CPU



Quad-core CPU

Outline

Concepts

Thread Scheduling on Linux

Bug fixes and two new schedulers

Conclusions

Outline

Using Threads

- *Threads* allow a program to run multiple independent tasks at the same time
- Useful for programs:
  - with long, mostly-independent computations
  - with a graphical interface

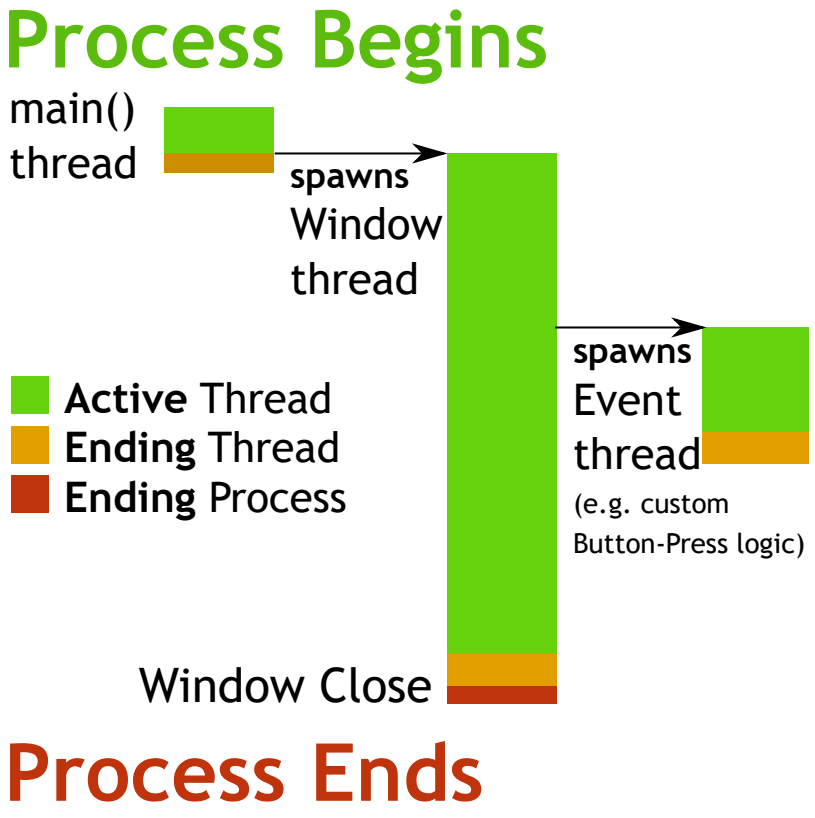


Figure: Example GUI Program. **Three** threads are created within **one** process

Using Threads

- A *multithreaded* program is a program that employs threads
- *Concurrent* computing techniques are techniques that allow many tasks to occur at the same time [W]
- *Parallel* computing techniques are techniques that allow many calculations to occur at the same time [W]
- Problems can be solved or improved using neither, either, or both of these techniques at once
- All you *need* to know about *parallel programs* for this talk is that they are used to perform extensive, often repetitive computations on large sets of data and do so efficiently by heavily utilizing threads

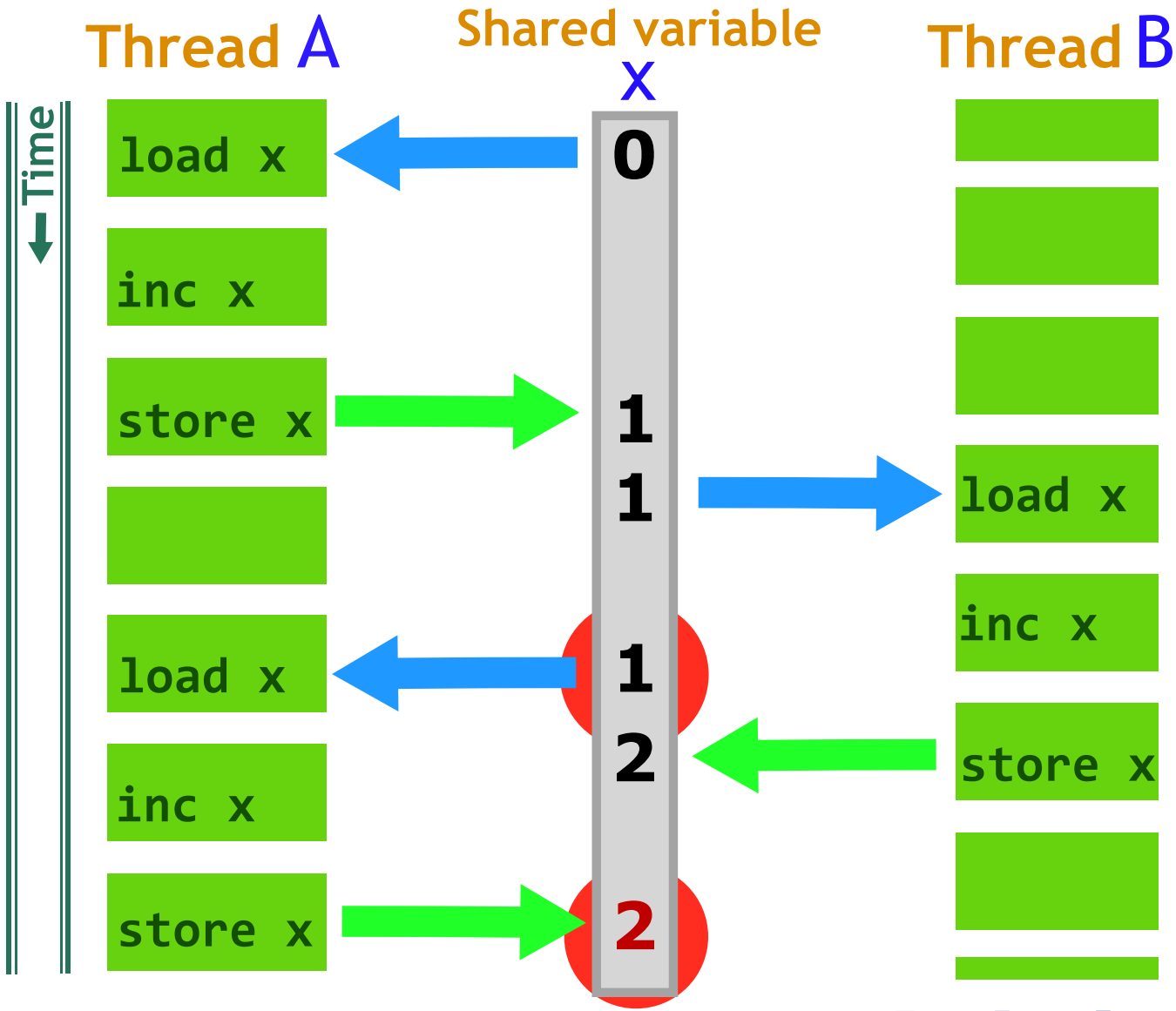
Race Conditions

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

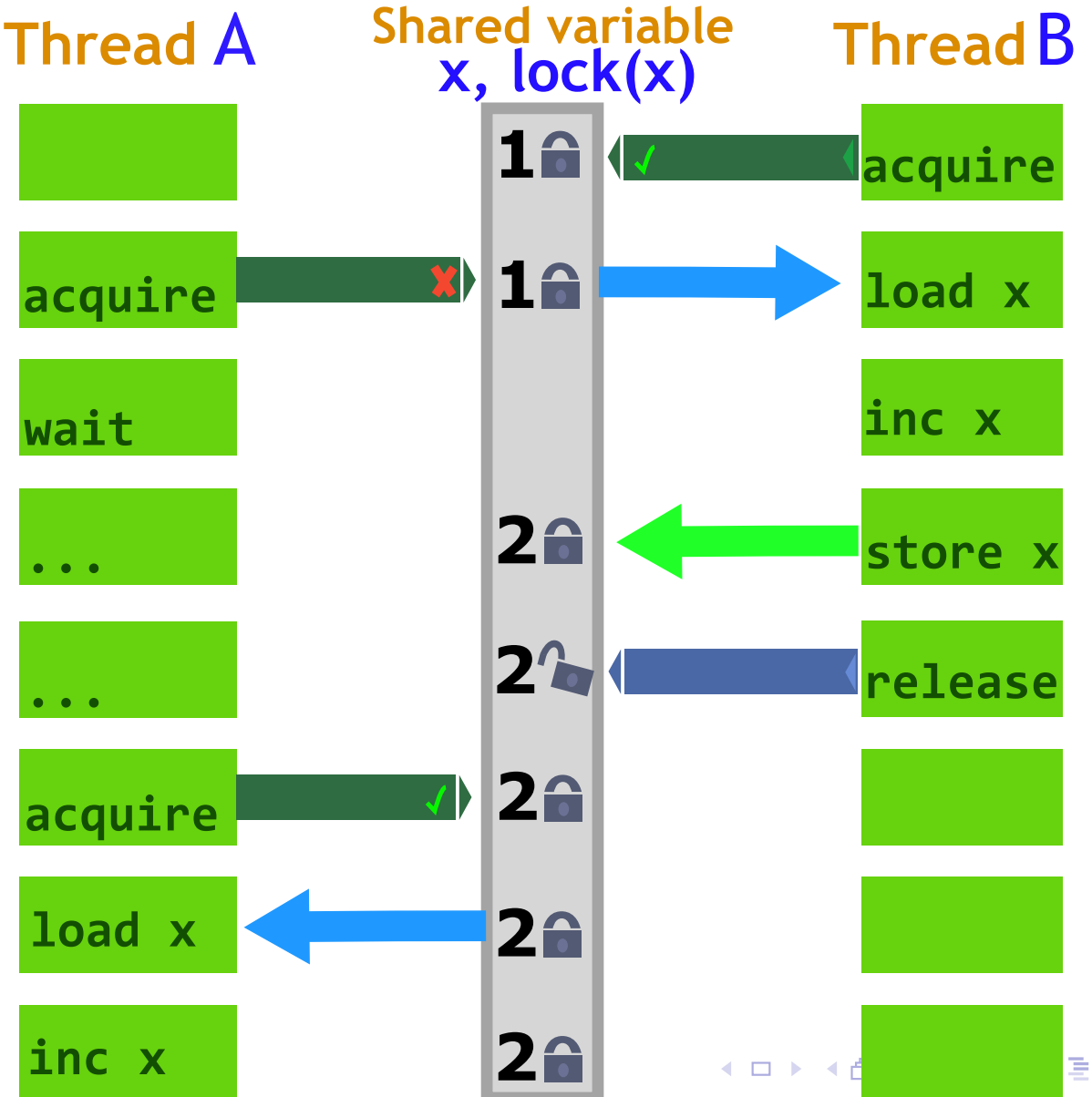


Synchronicity and Locks

- 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



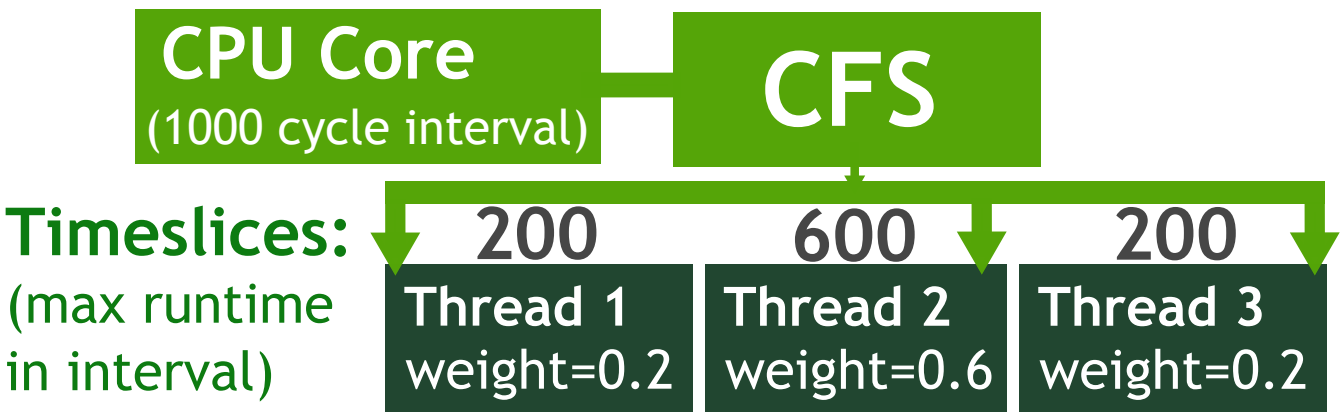
Outline

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[1]
- Monitors the number of cycles that the running thread receives and switches it out when it exceeds its timeslice



[1] Priority (PR) and niceness (NI) values are responsible for determining process priority on Linux. We won't get into that here.

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 the processor's threads should have access to

- Compiled code, Data
- Handles (references) for files and sockets
- Process control block (Important logistical information)

Thread State

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

- Run-time stack
- Copy of core's registers from when the thread was last active

Important note: Process state contains much more data

Cache

- Cache is the fastest form of memory
- 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

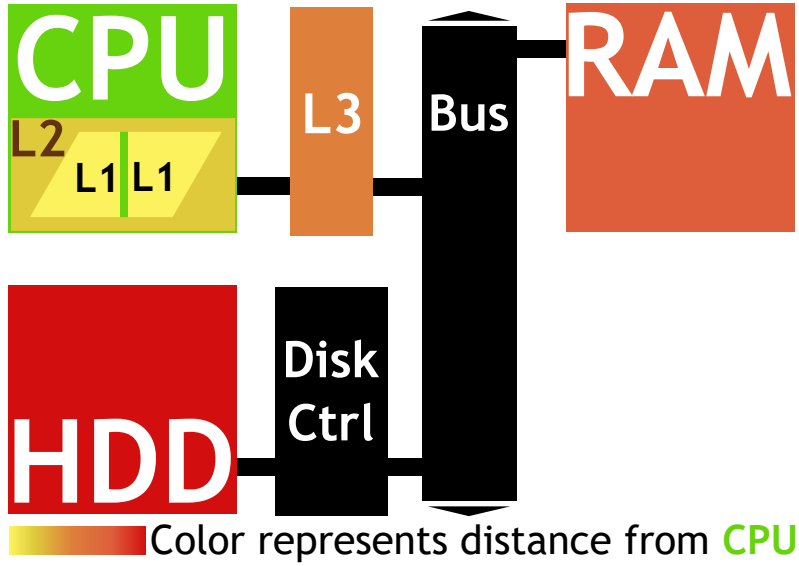


Figure: Distance of various forms of memory from CPU

Cache

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

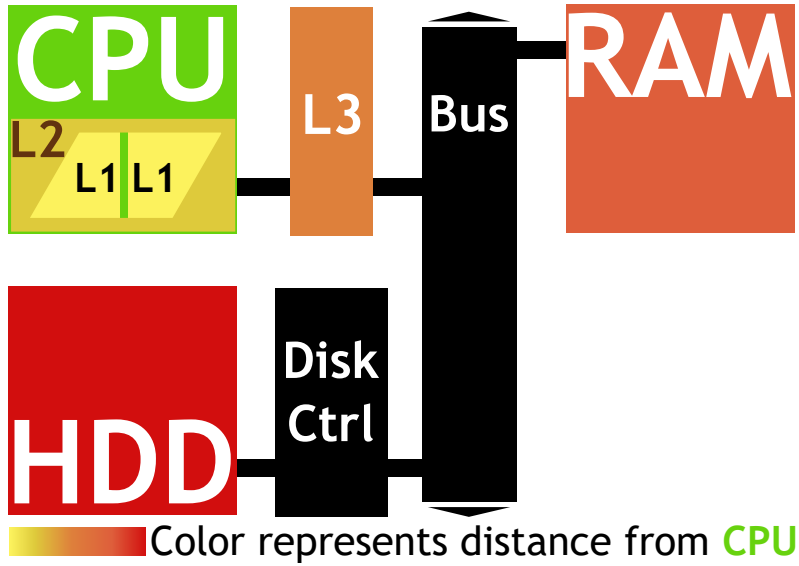


Figure: Distance of various forms of memory from CPU

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
- Explaining load-balancing depends on scheduling domains and groups. Don't have time to cover in this talk

Outline

Bugs everywhere!

Two of four bugs from Lozi et al. [2]:

- The Overload-on-Wakeup bug
- The Missing Scheduling Domains bug

- The Scheduling Group Construction bug
- The Group Imbalance bug

(These two heavily depend on definition of load balancing, scheduling domains, and scheduling groups)

The Overload-on Wakeup bug

- Optimization in thread wakeup core where threads awakened by other threads are placed on the same core in hope to increase cache locality.
- However, it does this disregarding whether the thread should start on an idle core instead!
- This problem only occurs in environments where threads sleep frequently, such as database systems.

The Overload-on Wakeup bug

The fix for this bug was to modify thread wakeup code:

When a thread awakens by another, if the same core isn't busy it will go to that core, otherwise it goes to an idle core

Since this bug strongly effected database systems, Lozi et al. ran a database benchmark for a popular proprietary database called TPC-H.

Bug Fixes	TPC-H request #18	Full TPC-H benchmark
None	55.9s	542.9s
Overload-on-Wakeup	43.5s (-22.2%)	471.1s (-13.2%)

Table: "Impact of the bug fix for Overload-on-Wakeup on a popular commercial database (values averaged after five runs.)"

The Missing Scheduling Domains bug

- Bug was *already fixed*, but regressed during a refactor on Linux kernel version 3.19+
- Occurs rarely, when a core is disable and re-enabled
- Bug causes system to misrepresent amount of groups of cores available to perform load-balancing on, meaning load-balancing would never happen!
- This meant that threads and processes would stay where they were
- Reintroducing the removed line fixed the problem

The Missing Scheduling Domains bug

- Lozi et al. used a group of computational applications that heavily utilize threads called the *NAS Parallel Benchmark (NPB)*, developed by NASA
- They disabled and re-enabled a core before each run of the benchmark

Application	Time w/ bug (sec)	T
bt	122	
cg	134	
ep	72	
ft	110	
is	283	
lu	2196	
mg	81	
sp	109	
ua	906	

Table: "Impact of the bug fix for Missing Scheduling Domains on the NPB. From Lozi et al. [2]"

That's all for the CFS, now on to some new schedulers!

- Shuffler
- FLSCHED

They both aim to solve the same problem

Problem: Parallel programs[2] whose threads share memory on Linux systems running the CFS experience repeated high waiting times on the acquisition of locks, which impedes efficiency.

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

[2] Reminder: All we *need* to know for this talk about parallel programs is that they heavily utilize threads

Shuffler

(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)
ONE REQUIREMENT TO RULE THEM ALL EFFICIENCY!!!

Aims to improve the efficiency of scheduling on linux by gutting the requirements and features of the scheduler. A lockless[3] thread scheduler.

Hey wait. Removing locks in the scheduler itself doesn't necessarily mean that program's threads within the scheduler won't contend for eachother? **How does it allocate threads?** It's more efficient though.

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

Outline

Conclusions

- Added developmental plasticity to N-gram GP using Incremental Fitness-based Development (IFD).
- IFD consistently improved N-gram GP performance on suite of test problems.
- “Knocking out” IFD shows it’s valuable in all phases, even if it wasn’t used earlier in a run.
- IFD generates more complex, less converged probability tables.
- IFD generates more modules/loops & uses more low-probability paths.
- Currently exploring applications to dynamic environments.

Thanks!

Thank you for your time and attention!

Contact:

- mcphee@morris.umn.edu
- http://www.morris.umn.edu/~mcphee/

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.