Linux schedulers – overview continued

CFS, BFS, Deadline, MuQSS, etc.

What’s new in process scheduling?

Table of contents

Completely Fair Scheduler
Ingo Molnar, April 2007

Ingo Molnar, April 2007

I wrote the first line of code of the CFS patch this week, 8am Wednesday morning, and released it to lkml 62 hours later, 22pm on Friday.

I'd like to give credit to Con Kolivas for the general approach here: he has proven via RSDL/SD that 'fair scheduling' is possible and that it results in better desktop scheduling. Kudos Con!

The CFS patch uses a completely different approach and implementation from RSDL/SD. My goal was to make CFS's interactivity quality exceed that of RSDL/SD, which is a high standard to meet ☺

18 files changed, 1454 inserstions(+), 1133 deletions(-)

Schedulers: the plot thickens, https://lwn.net/Articles/230574/, J. Corbet, April 2007

The new scheduler which got into the game so abruptly caused a big storm on Linux mailing lists. Ultimately, CFS was incorporated into the **2.6.23 kerne**l, and Con Kolivas gave up work on the Linux kernel (temporarily).

Completely Fair Scheduler

- A completely new concept for the scheduler.
- Trying to implement **Processor Sharing** (there is no state in which one process has gotten more of the CPU than another, since all tasks are running on it at once, so all processes have a **fair share** of the CPU).
- On a real CPU **unfairness** will inevitably be **nonzero** when there are more tasks than CPUs. When one task is running on a CPU, this increases the amount of CPU time that the CPU **owes** to all other tasks. CFS schedules the task with the **largest unfairness** onto the CPU first.
- **Unfairness** is measured by the **virtual runtime**.
- In practice, the **virtual runtime** of a task is its **actual runtime normalized** to the total number of **running tasks**.
- Virtual time flows at a **priority-dependent** speed.
- Instead of queues of tasks – a **red-black tree**, sorted after the **virtual runtime** and the selection of the task, that run for the **shortest time** period.

Completely Fair Scheduler

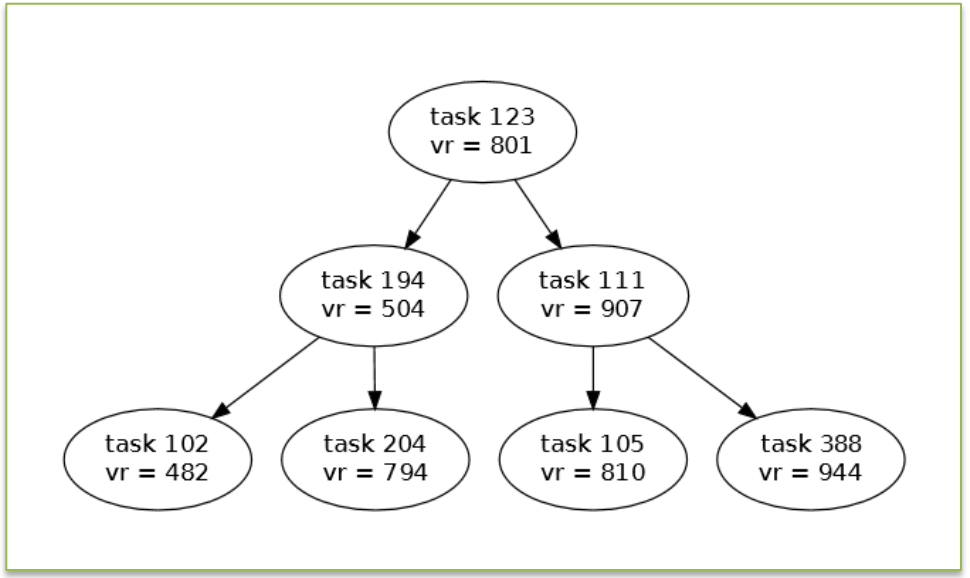
- It is also possible to group tasks and share processor time fairly among defined „**entities**” – **process groups**.
- When implementing CFS, the code was reorganized to separate sections responsible for the scheduling policy (the **struct sched_class** has been created, there is a pointer to this structure in **struct task_struct**).
- Like the O(1) scheduler, CFS maintains **separate data structures for each CPU**. This reduces the wait for the lock to be removed, but requires explicit processor load balancing.
- When a new task is created, it is assigned the minimum current vruntime (**min_vruntime**).

- With CFS, tasks have no concept of timeslice but rather run until they are no longer the most unfairly treated task. To reduce context switching overhead, CFS divides time into a minimum granularity.

```
struct task_struct  {
    ...
    const struct sched_class *sched_class;
    ...
}
```

Completely Fair Scheduler

Red-black tree
for CFS scheduler
process selection **O(1)** insertion
O(log(n))



- When a task has finished running on the CPU, all of the other tasks in the tree need to have their **unfairness** increase.
- To prevent having to update **all** of the tasks in the tree the scheduler maintains a per-task **vruntime** statistic.
- This is the **amount of total nanoseconds** that the task has spent **running** on a **CPU weighted** by its **niceness**.
- Thus, instead of updating all other tasks to be more **unfair** when a task has **finished** running on the CPU, we update the **leaving** task to be more fair than others by increasing its **virtual runtime**.
- The scheduler always selects the **most unfairly treated** task by selecting the task with the **lowest vruntime**.

http://cs.unm.edu/~eschulte/classes/cs587/data/bfs-v-cfs_groves-knockel-schulte.pdf

Completely Fair Scheduler

sched_latency_ns – the time when one era should take place, i.e. all tasks from the queue should be completed. The default is 20 ms.

sched_min_granularity_ns – minimum time, which on average should be given to a task in an era. The default is 4 ms.

epoch = max(sched_latency_ns; sched_min_granularity_ns * nr_running)

ideal_slice = epoch * (weight/∑weight)

Variable quantum lengths, but a fixed period of rotation of the era, guarantees small delays. Under heavy load, the quanta are not shortened below the minimum value, at the expense of responsiveness.

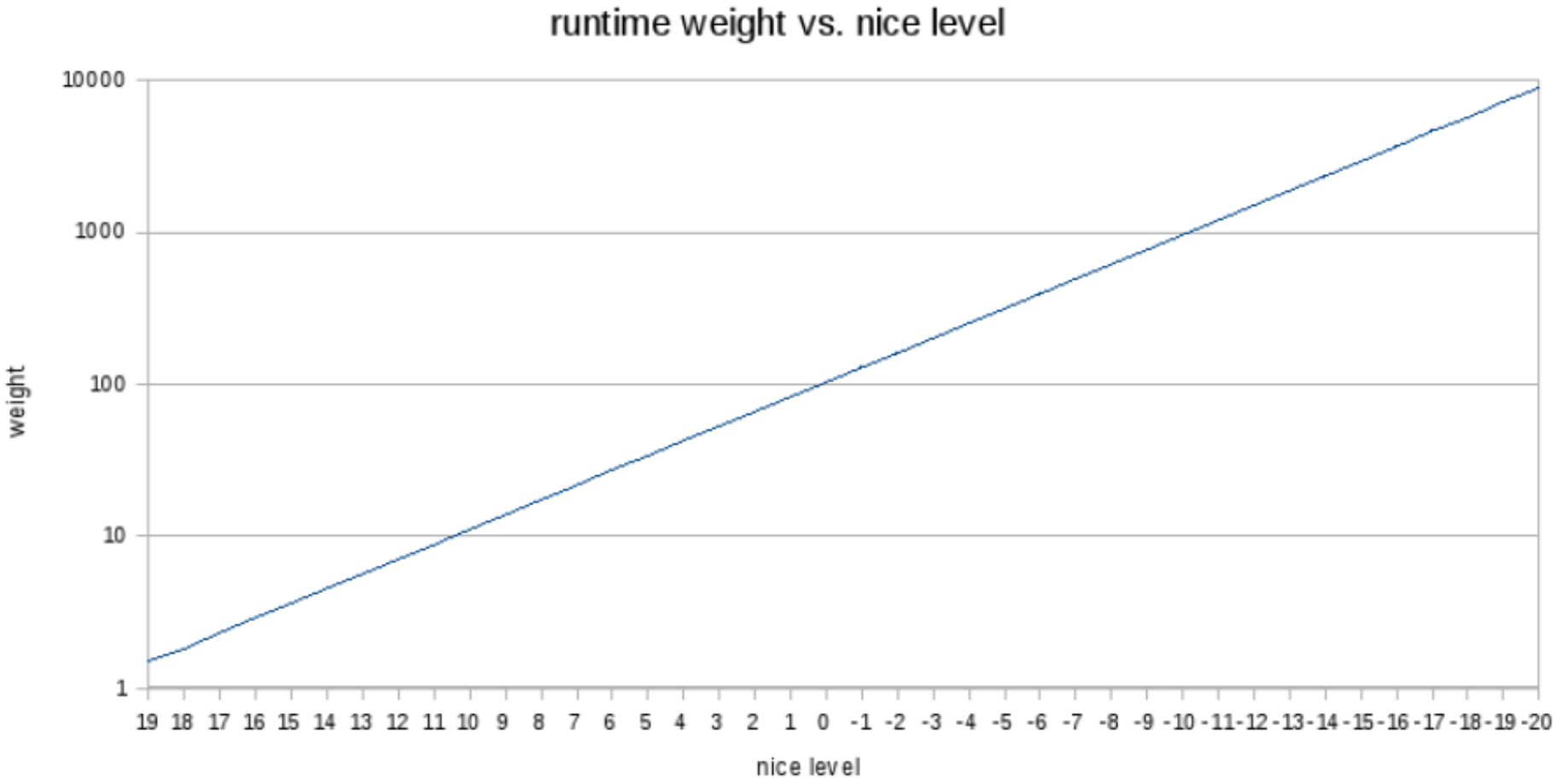
Processes with different **priorities** receive different **weights**. The virtual time is scaled with these weights – the scheduler takes into account the differences in *nice* values of processes. Priority weights are allocated as geometric progression:

prio_to_weight[n] ≈ prio_to_weight[n+1] * 1.25

Processes with a given priority difference will always receive CPU time in a constant proportion.

Completely Fair Scheduler

weights of processes



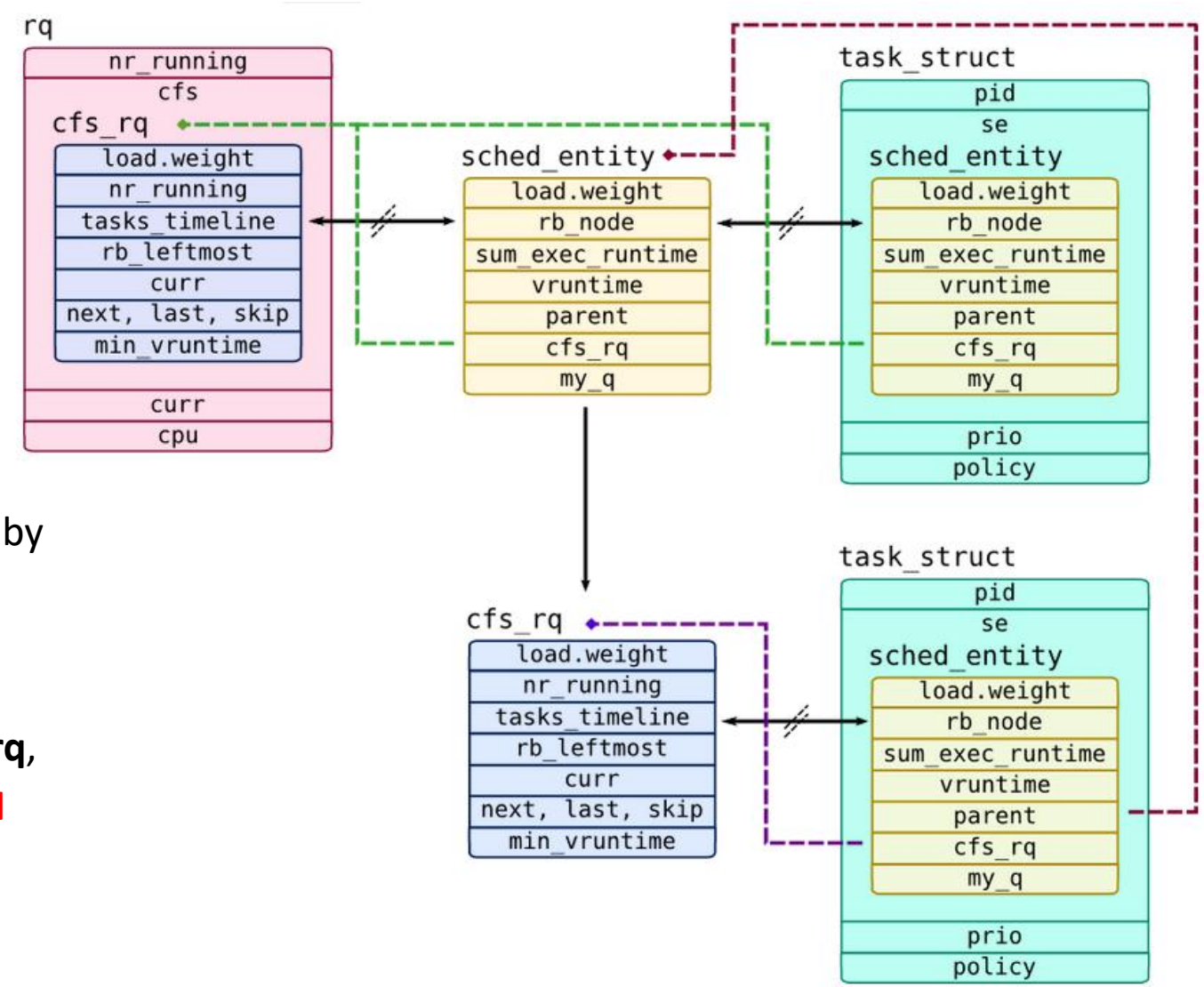
A process with *nice* -20 will get about 6000 times more CPU time than a process with *nice* 19 (for comparison: 160 time more than in the old scheduler)

CFS scheduler doesn't deal with **tasks**, but with **scheduler entities** of type **struct sched_entity**.

Sched entity may represent a **task** or a **queue** of **entities** of type **struct cfs_rq** (which is referenced by field **my_q**), thus allowing to build hierarchies of entities and allocate resources to task groups (**Cgroups)**.

Processor run queue, represented by type **struct rq**, contains field **cfs** which is instance of **struct cfs_rq** and contains queue of all high-level entities.

Each entity has **cfs_rq** pointer which points to CFS **runqueue** to which that entity belongs.



In this example processor run queue has two **scheduler entities**: one **CFS queue** with single task (which refers to top-level cfs_rq through parent pointer) in it and one **top-level task**.

Source: https://myaut.github.io/dtrace-stap-book/kernel/sched.html

Completely Fair Scheduler

SKIP

- CFS doesn't allocate timeslices. Instead it accounts total time which task had spend on CPU and saves it to **sum_exec_runtime** field.
- When task is dispatched onto CPU, its **sum_exec_runtime** is saved into **prev_sum_exec_runtime**, so calculating their difference will give time period that task spent on CPU since last dispatch. **sum_exec_runtime** is expressed in nanoseconds but it is not directly used to measure task's runtime.
- To implement priorities, CFS uses task **weight** (in field **load.weight**) and divides runtime by tasks weight, so tasks with higher weights will advance their runtime meter (saved into **vruntime** field) slower.
- Tasks are sorted according to their **vruntime** in a red-black tree called **tasks_timeline**, while left-most task which has lowest **vruntime** of all tasks and saved into **rb_leftmost**.
- CFS has special case for tasks that have been woken up. Because they can be sleeping too long, their **vruntime** may be too low and they will get unfairly high amount of CPU time. To prevent this, CFS keeps **minimum** possible **vruntime** of all tasks in **min_vruntime** field, so all waking up tasks will get **min_vruntime** minus a predefined "**timeslice**" value.
- CFS also have a **scheduler buddies** – helper pointers for a dispatcher:
  - **next** – task that was recently awoken,
  - **last** – task that recently was evicted from CPU and
  - **skip** – task that called **sched_yield()** giving CPU to other entities.

Source: https://myaut.github.io/dtrace-stap-book/kernel/sched.html

Completely Fair Scheduler

easy exchange of schedulers or modularization?

Con Kolivas accused CFS of having used his scheduler modularization ideas included in the **plugsched** framework (enabling easy exchange of schedulers), although they were previously criticized.

Linus Torwalds (https://yarchive.net/comp/linux/security.html)

The arguments that ‘servers’ have a different profile than ‘desktop’ is pure and utter garbage, and is perpetuated by people who don’t know what they are talking about (...) Yes, there are differences in tuning, but those have nothing to do with the basic algorithm. They have to do with goals and trade-offs, and most of the time we should aim for those things to **auto-tune**.

Ingo refuted Con's allegations that his proposed reorganization of the scheduler code was not intended to allow the interchangeability of schedulers, but only to improve the quality of the code.

Con Kolivas returns in 2009 in FAQ about BFS

Why „Brain Fuck”?

Because ...

... it throws out everything about what we know is good about how to design a modern scheduler in scalability.

... it's so ridiculously simple.

... it performs so ridiculously well on what it's good at despite being that simple.

... it's designed in such a way that mainline would never be interested in adopting it, which is how I like it.

... it will make people sit up and take notice of where the problems are in the current design.

... it **throws out the philosophy that one scheduler fits all** and shows that you can do a -lot- better with a **scheduler designed for a particular purpose**. I don't want to use a steamroller to crack nuts.

... it actually means that more CPUs means better latencies.

... I must be fucked in the head to be working on this again.

I'll think of some more becauses later.

Con Kolivas – on lwn.net (and wiki) about BFS

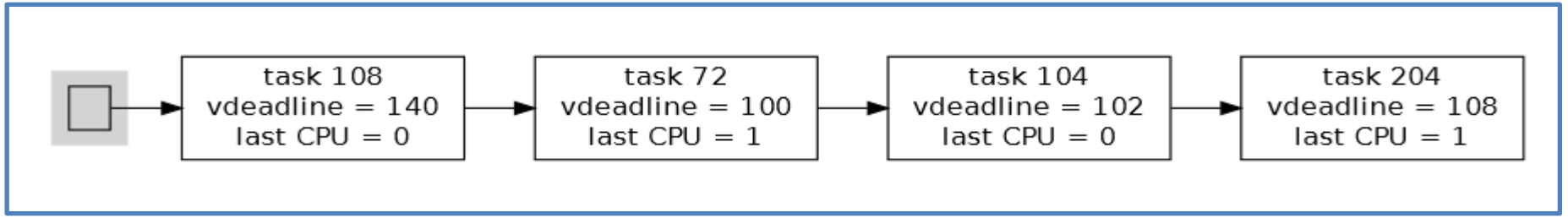
The main focus of BFS is to achieve excellent desktop **interactivity** and **responsiveness** without heuristics and tuning knobs that are difficult to understand, impossible to model and predict the effect of, and when tuned to one workload cause massive detriment to another.

BFS is best described as a **single runqueue**, O(n) lookup, **earliest effective virtual deadline first** design.

The reason for going back to a single runqueue design is that once **multiple runqueues** are introduced, per-CPU or otherwise, there will be **complex interactions** as each runqueue will be responsible for the scheduling latency and fairness of the tasks only on its own runqueue, and to achieve **fairness** and **low latency** across **multiple CPUs**, any advantage in throughput of having CPU local tasks causes other disadvantages.

A significant feature of BFS is that all accounting is done purely based on **CPU used** and **nowhere is sleep time used in any way** to determine entitlement or interactivity.

Con Kolivas – on lwn.net (i na wiki) about BFS



The task put into the queue receives a time quantum of **rr_interval** and **deadline**

jiffies + (prio_ratio * rr_interval)

where **prio_ratio**, like the weights in CFS, depends geometrically on the priority.

If the calculated deadline is earlier than the deadline of the process performed on one of the processors, then the new process immediately preempts it.

The system selects the task by reviewing the entire list of tasks in **O(n)** (!). If it encounters a task with an **expired deadline**, it immediately runs it. Otherwise, the one with the **nearest deadline** begins to run.

Simple additional mechanisms that favor tasks on the same processor as they were previously run.

Only two configuration parameters:

rr_interval – time quantum, default 6 ms,

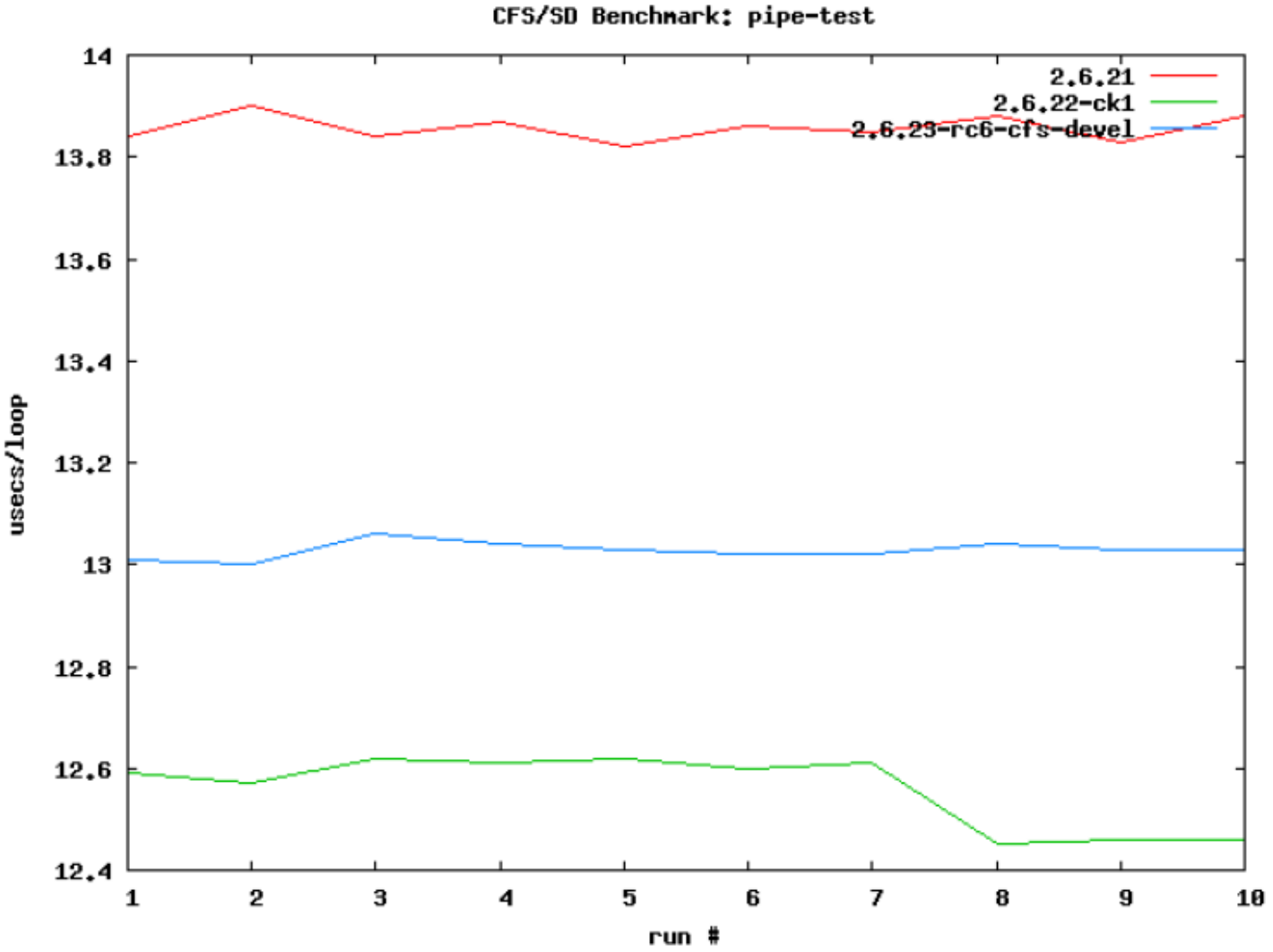
iso_cpu – percentage of processor time that user processes simulating RT tasks can take up at maximum, default 70%.

SKIP

Comparing performance of Linux schedulers

Experiments by Rob Hussey (2007)

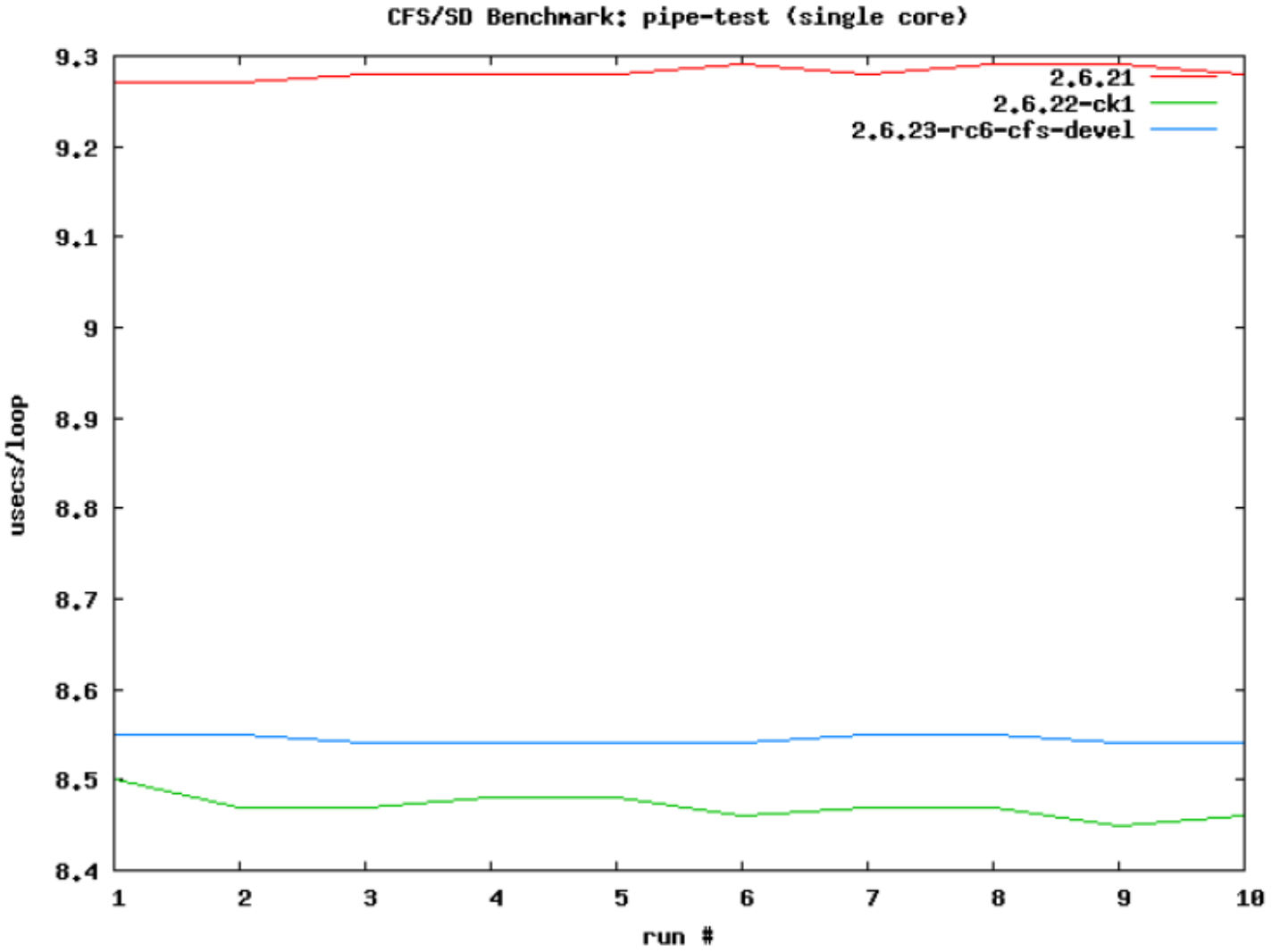
pipe-test – two processes exchanging messages through pipe



http://ck.vds.kolivas.narkive.com/eFcZA61H/scheduler-benchmarks-a-follow-up

Experiments by Rob Hussey (2007)

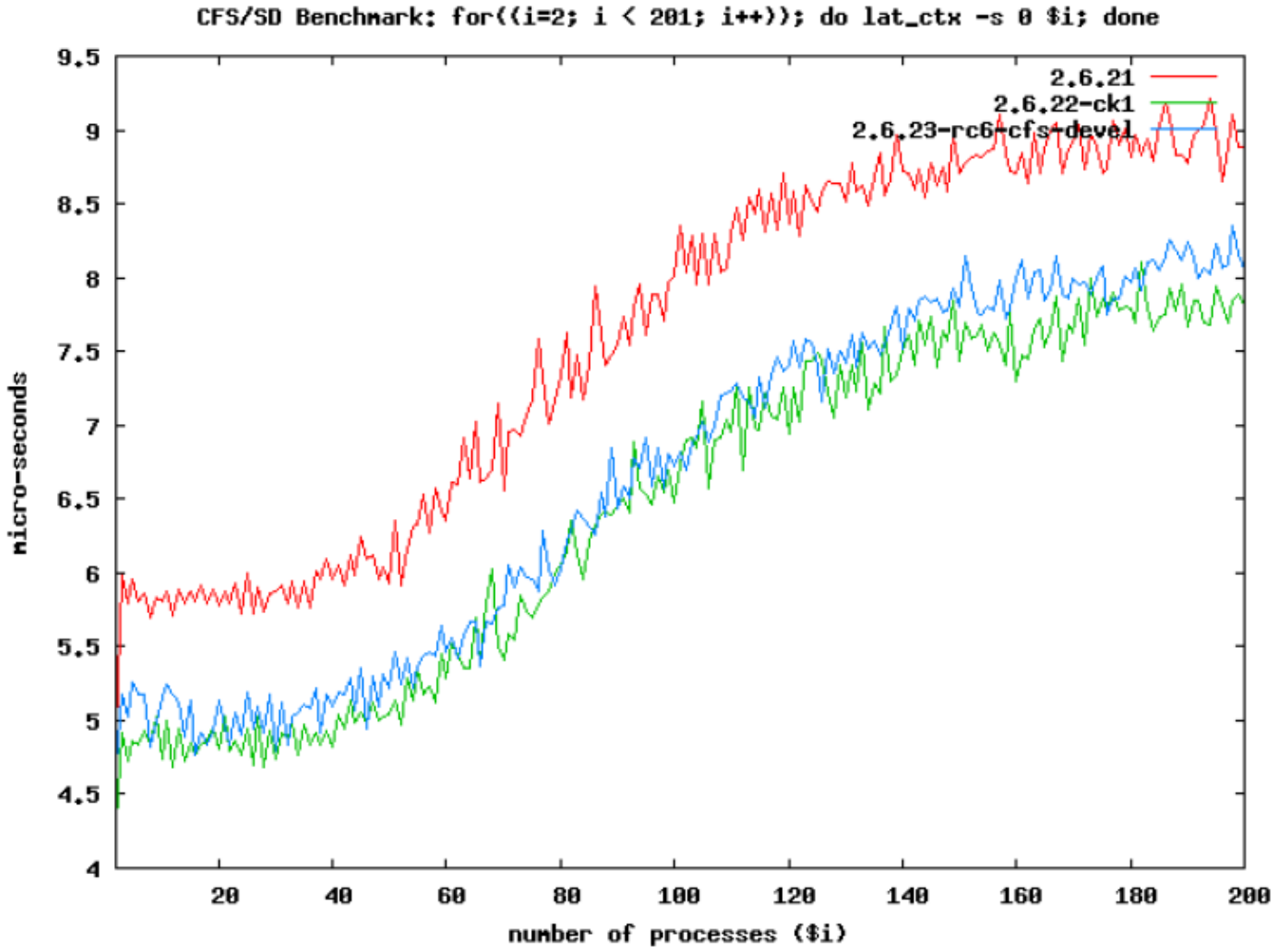
pipe-test – two processes exchanging messages through pipe



http://ck.vds.kolivas.narkive.com/eFcZA61H/scheduler-benchmarks-a-follow-up

Experiments by Rob Hussey (2007)

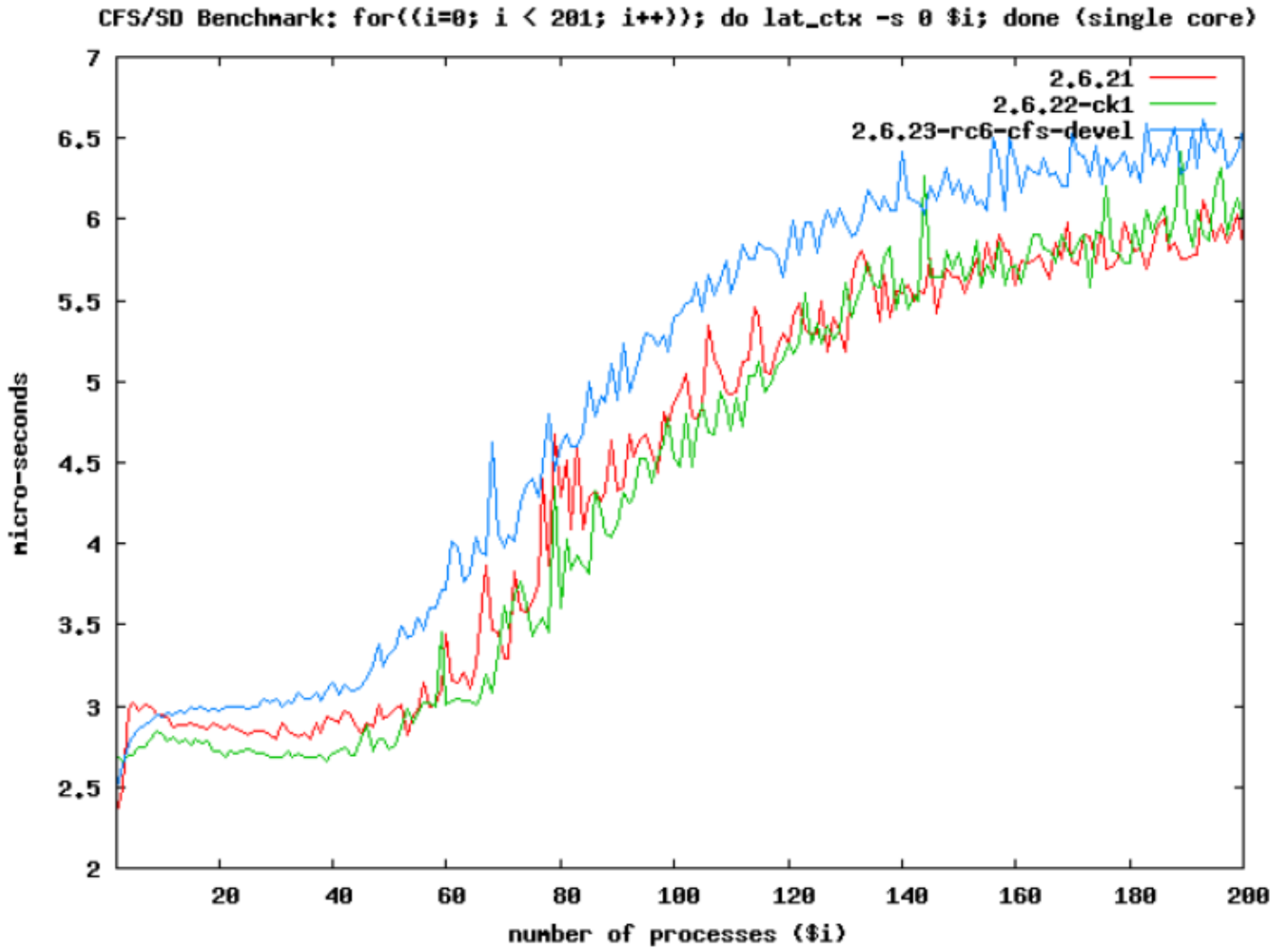
lat_ctx – processes connected by pipes into a ring, transmit messages, intensively use the processor after receiving the message, and then send the message on



http://ck.vds.kolivas.narkive.com/eFcZA61H/scheduler-benchmarks-a-follow-up

Experiments by Rob Hussey (2007)

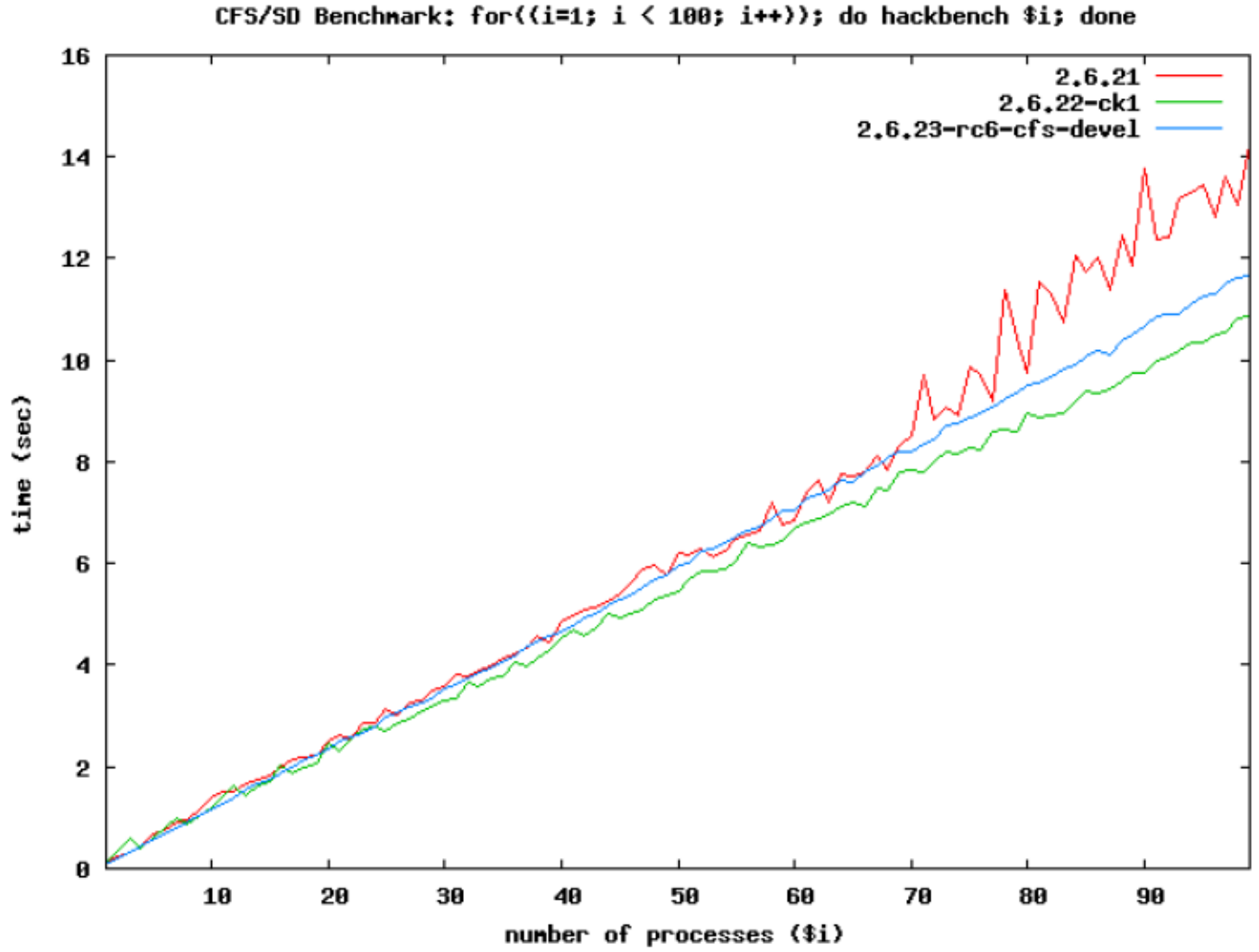
lat_ctx – processes connected by pipes into a ring, transmit messages, intensively use the processor after receiving the message, and then send the message on



http://ck.vds.kolivas.narkive.com/eFcZA61H/scheduler-benchmarks-a-follow-up

Experiments by Rob Hussey (2007)

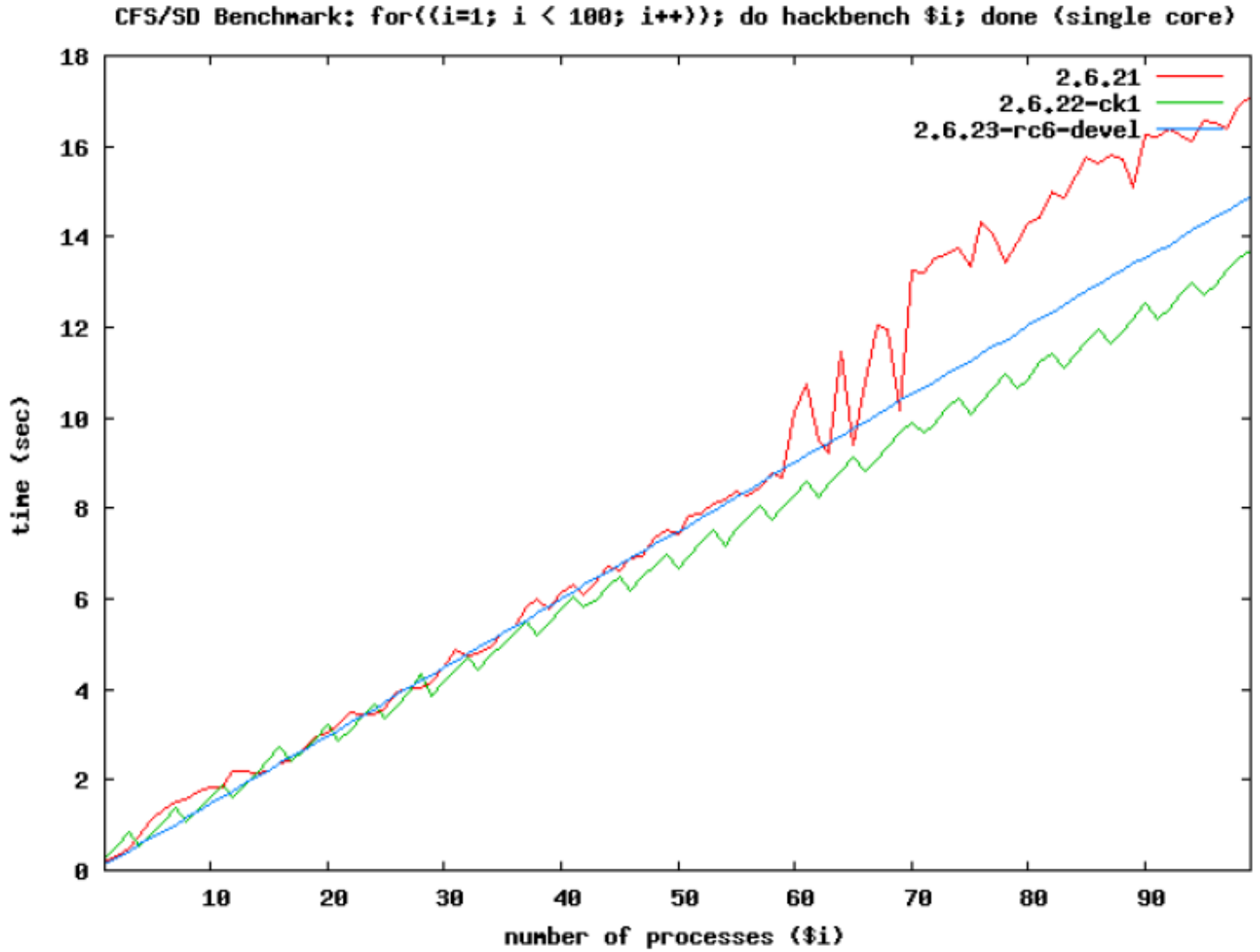
hackbench – creates sending and receiving groups. Each sending group process sends a pipe/socket message to each receiving group process



http://ck.vds.kolivas.narkive.com/eFcZA61H/scheduler-benchmarks-a-follow-up

Experiments by Rob Hussey (2007)

hackbench – creates sending and receiving groups. Each sending group process sends a pipe/socket message to each receiving group process



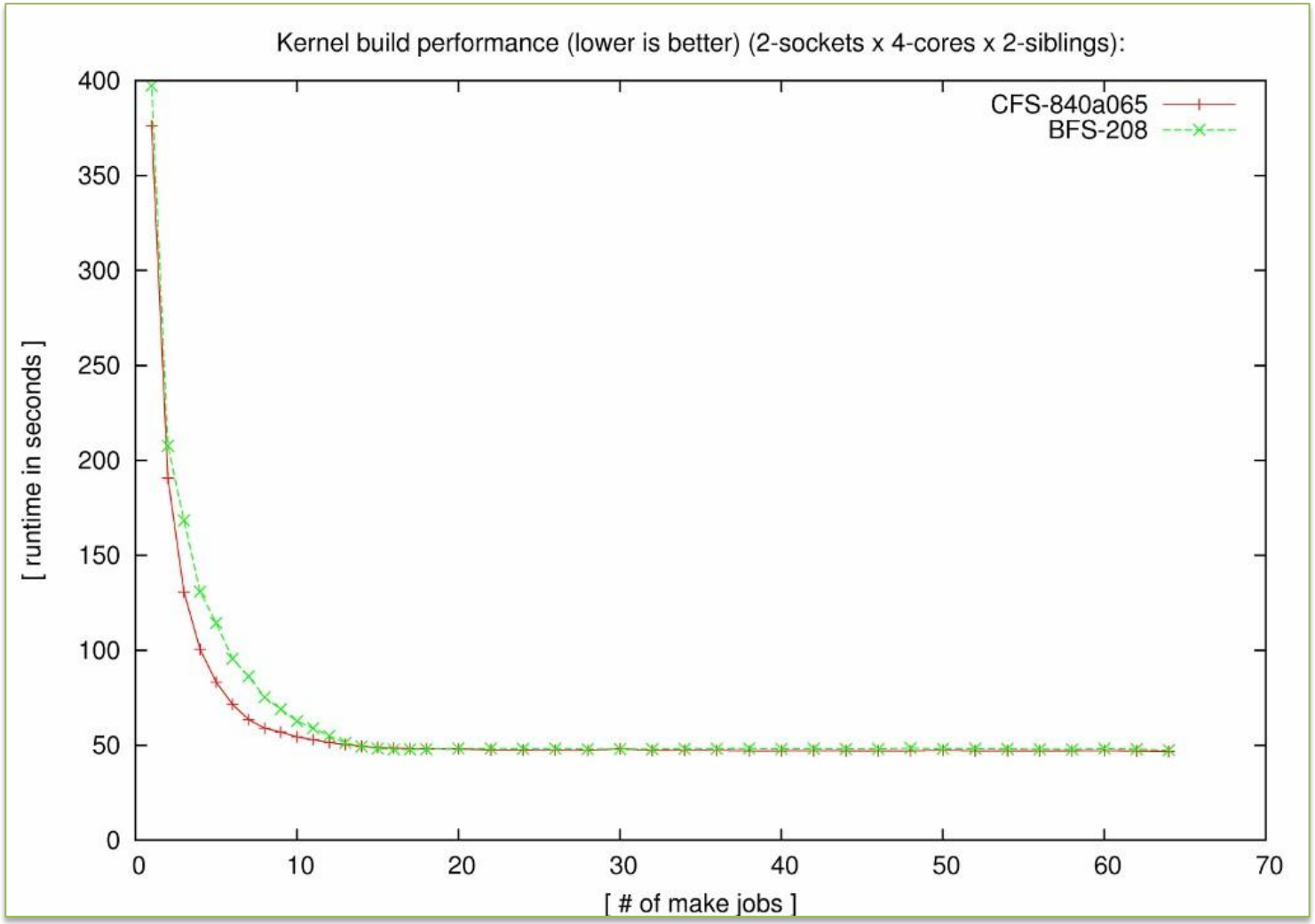
http://ck.vds.kolivas.narkive.com/eFcZA61H/scheduler-benchmarks-a-follow-up

Experiments by Ingo Molnar (2009-09-06)

- First and foremost, let me say that I'm happy that you are hacking the Linux scheduler again.
- **It's perhaps proof that hacking the scheduler is one of the most addictive things on the planet** ;-)
- I understand that BFS is still early code and that you are not targeting BFS for mainline inclusion – but BFS is an interesting and bold new approach, cutting a _lot_ of code out of kernel/sched*.c, so it raised my curiosity and interest :-)
- The testbox i picked fits into the upper portion of what I consider a sane range of systems to tune for – and should still fit into BFS's design bracket as well according to your description: it's a dual quad core system with hyperthreading.

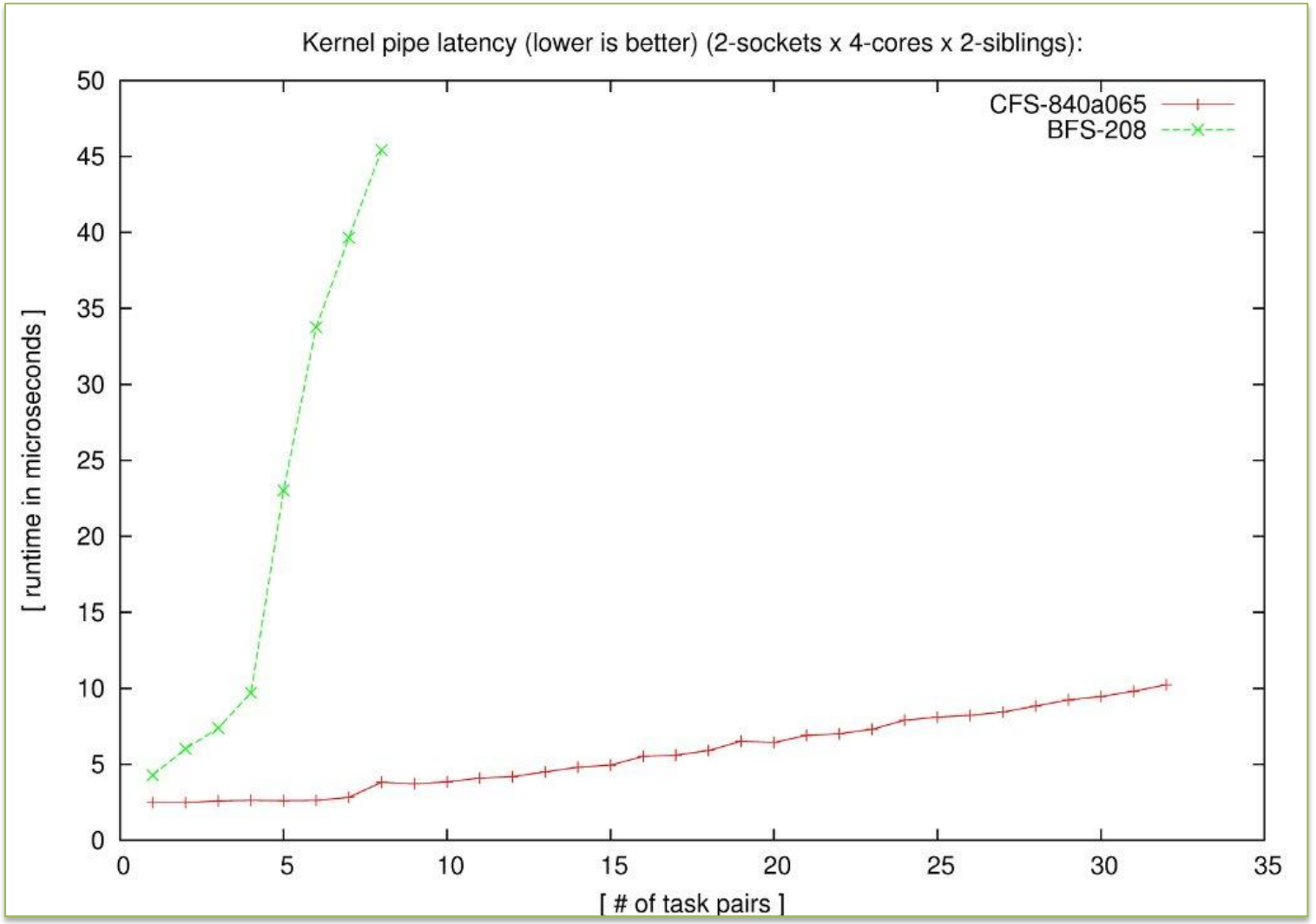
http://marc.info/?l=linux-kernel&m=125227082723350&w=2

Experiments by Ingo Molnar (2009-09-06)



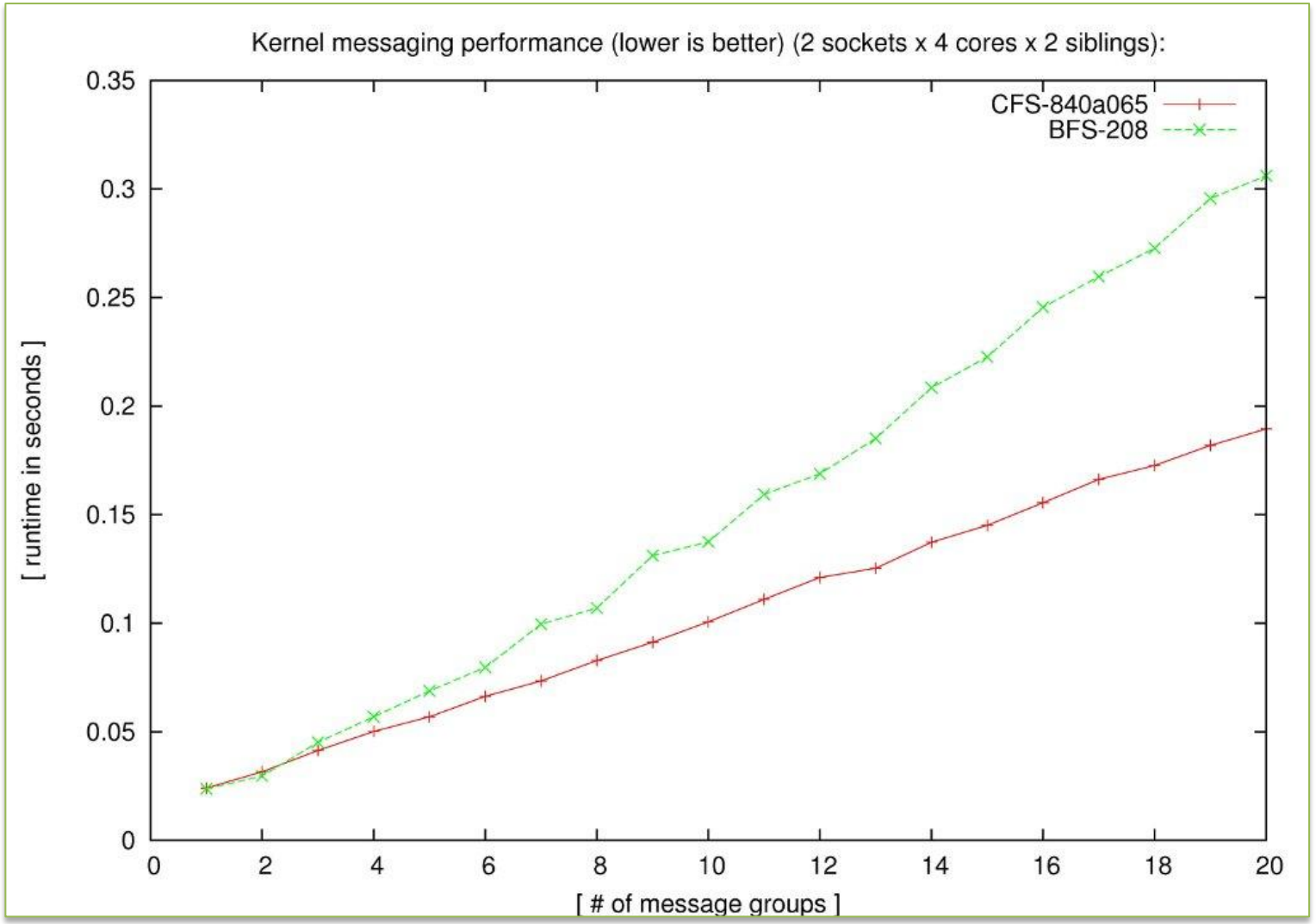
http://marc.info/?l=linux-kernel&m=125227082723350&w=2

Experiments by Ingo Molnar (2009-09-06)



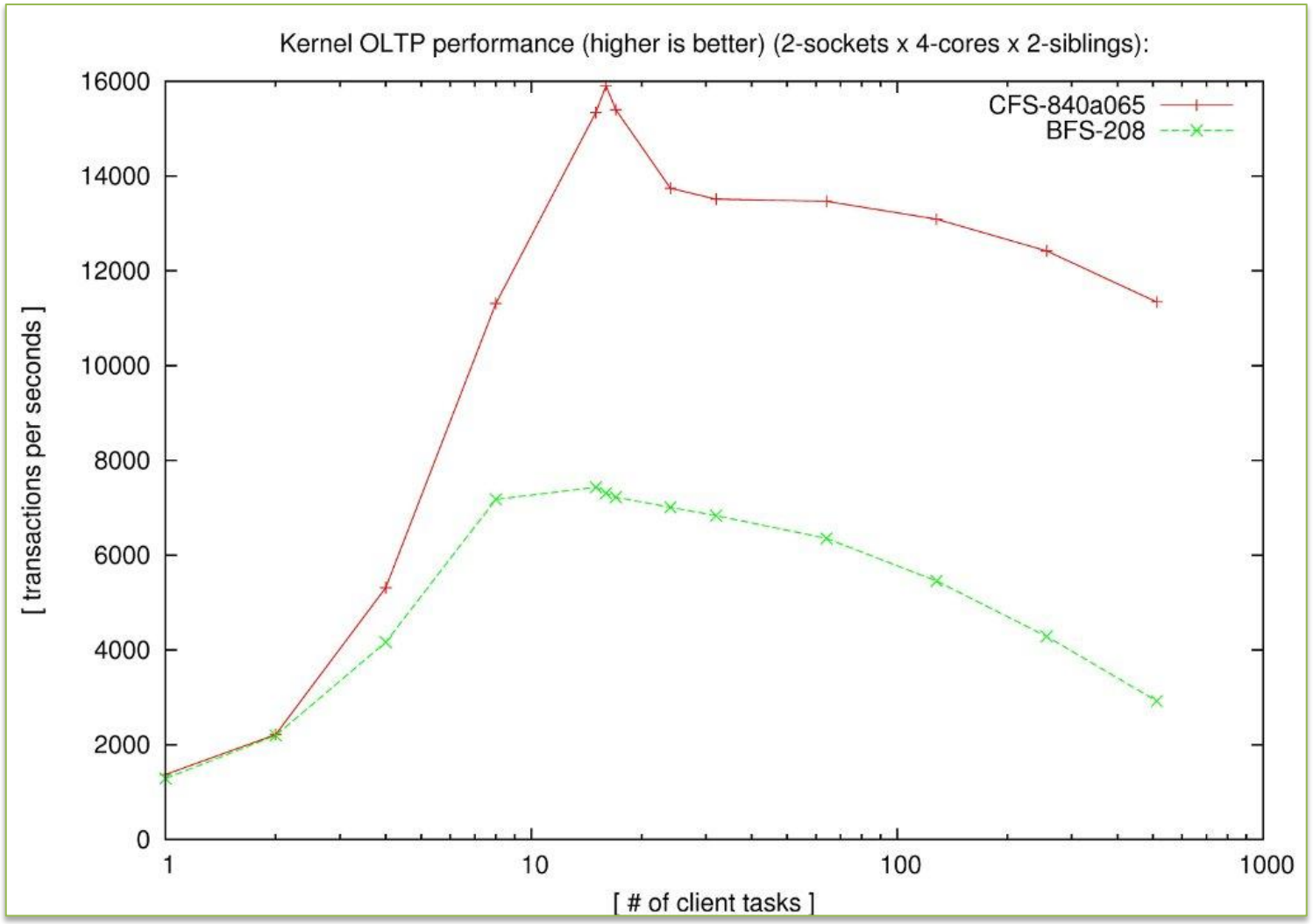
http://marc.info/?l=linux-kernel&m=125227082723350&w=2

Experiments by Ingo Molnar (2009-09-06)



http://marc.info/?l=linux-kernel&m=125227082723350&w=2

Experiments by Ingo Molnar (2009-09-06)



http://marc.info/?l=linux-kernel&m=125227082723350&w=2

Experiments by Ingo Molnar (2009-09-06)

- Alas, as it can be seen in the graphs, I can not see **any BFS performance improvements**, on this box.
- In the **kbuild** test BFS is showing significant weaknesses up to 16 CPUs.
- BFS performed very poorly in the **pipe** test: at 8 pairs of tasks it had a runtime of 45.42 seconds - while sched-devel finished them in 3.8 seconds.
- **Messaging**: mainline sched-devel is significantly faster for smaller and larger loads as well. With 20 groups mainline ran 61.5% faster.
- **OLTP peformance**: for sysbench OLTP performance sched-devel outperforms BFS on each of the main stages.
- General interactivity of BFS seemed good to me - except for the pipe test when there was significant lag over a minute. I think it's some starvation bug, not an inherent design property of BFS, so I'm looking forward to re-test it with the fix.
- I agree with the general goals described by you in the BFS announcement - **small desktop systems matter more than large systems**. We find it critically important that the mainline Linux scheduler performs well on those systems too.

http://marc.info/?l=linux-kernel&m=125227082723350&w=2

Con replies (7 September 2009) ...

Hard to keep a project under wraps and get an audience at the same time, it is. I do realise it was inevitable LKML would invade my personal space no matter how much I didn't want it to, but it would be rude of me to not respond.

/me sees Ingo run off to find the right combination of hardware and benchmark to prove his point.

[snip lots of bullshit meaningless benchmarks showing how great cfs is and/or how bad bfs is, along with telling people they should use these artificial benchmarks to determine how good it is, demonstrating yet again why benchmarks fail the desktop]

I'm not interested in a long protracted discussion about this since I'm too busy to live linux the way full time developers do, so I'll keep it short, and perhaps you'll understand my intent better if the FAQ wasn't clear enough.

Do you know what a normal desktop PC looks like? No, a more realistic question based on what you chose to benchmark to prove your point would be: Do you know what normal people actually do on them?

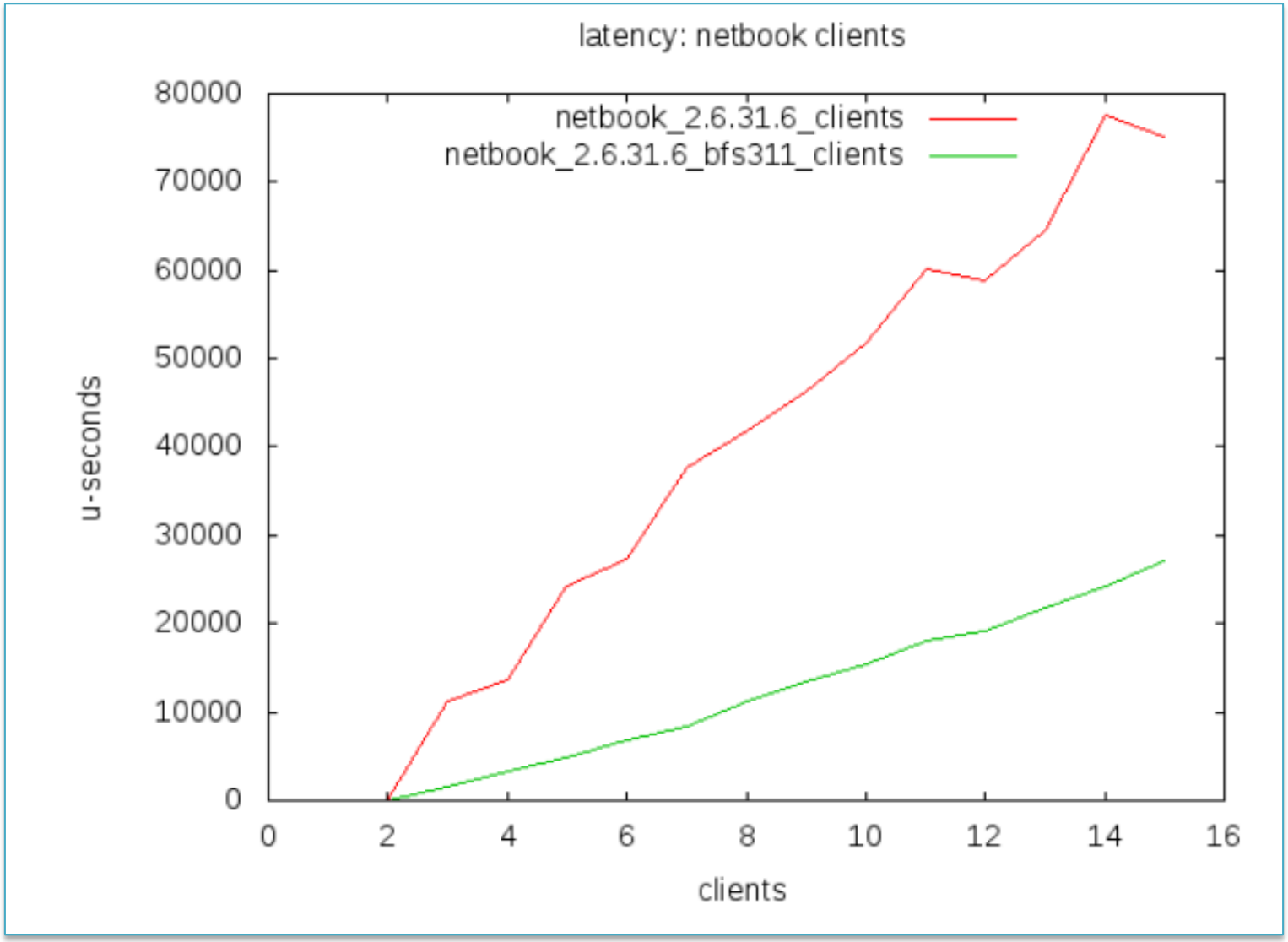
Feel free to treat the question as rhetorical.

Regards, -ck

/me checks on his distributed computing client's progress, fires up his next H264 encode, changes music tracks and prepares to have his arse whooped on quake live.

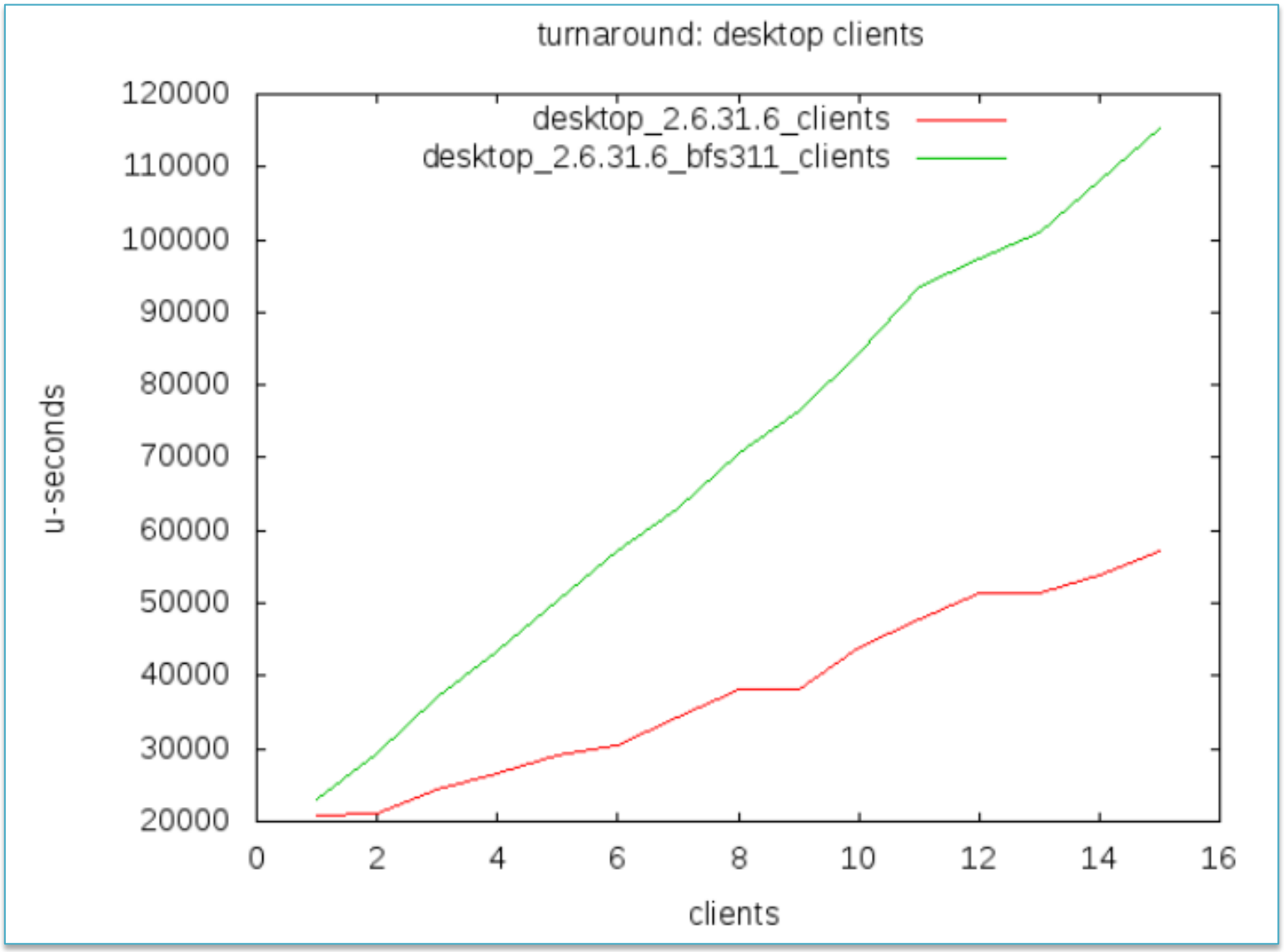
https://lwn.net/Articles/351504/

Experiments by T. Groves, J. Knockel, E. Schulte (2009-12-11)



http://cs.unm.edu/~eschulte/classes/cs587/data/bfs-v-cfs_groves-knockel-schulte.pdf

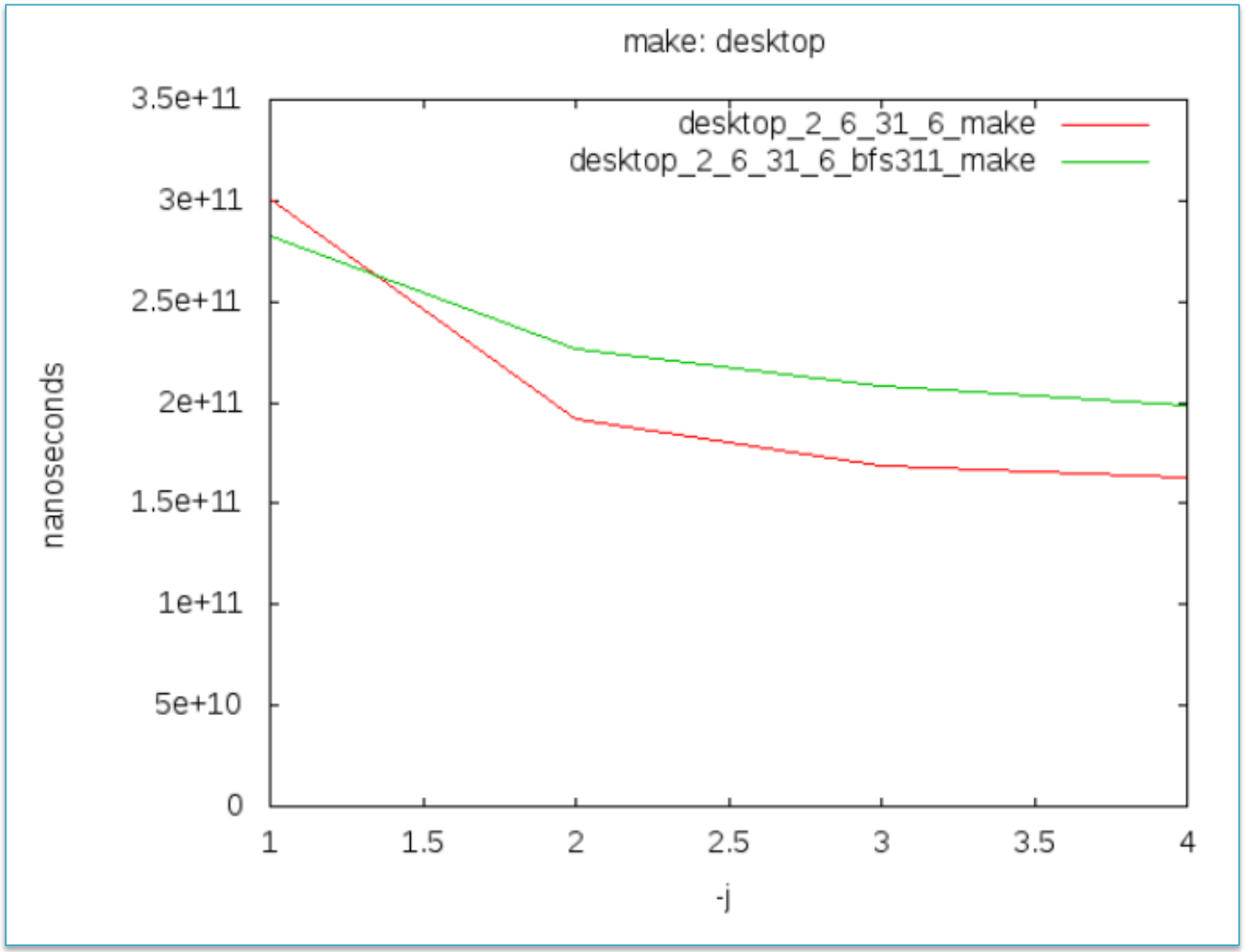
Experiments by T. Groves, J. Knockel, E. Schulte (2009-12-11)



http://cs.unm.edu/~eschulte/classes/cs587/data/bfs-v-cfs_groves-knockel-schulte.pdf

Experiments by T. Groves, J. Knockel, E. Schulte (2009-12-11)

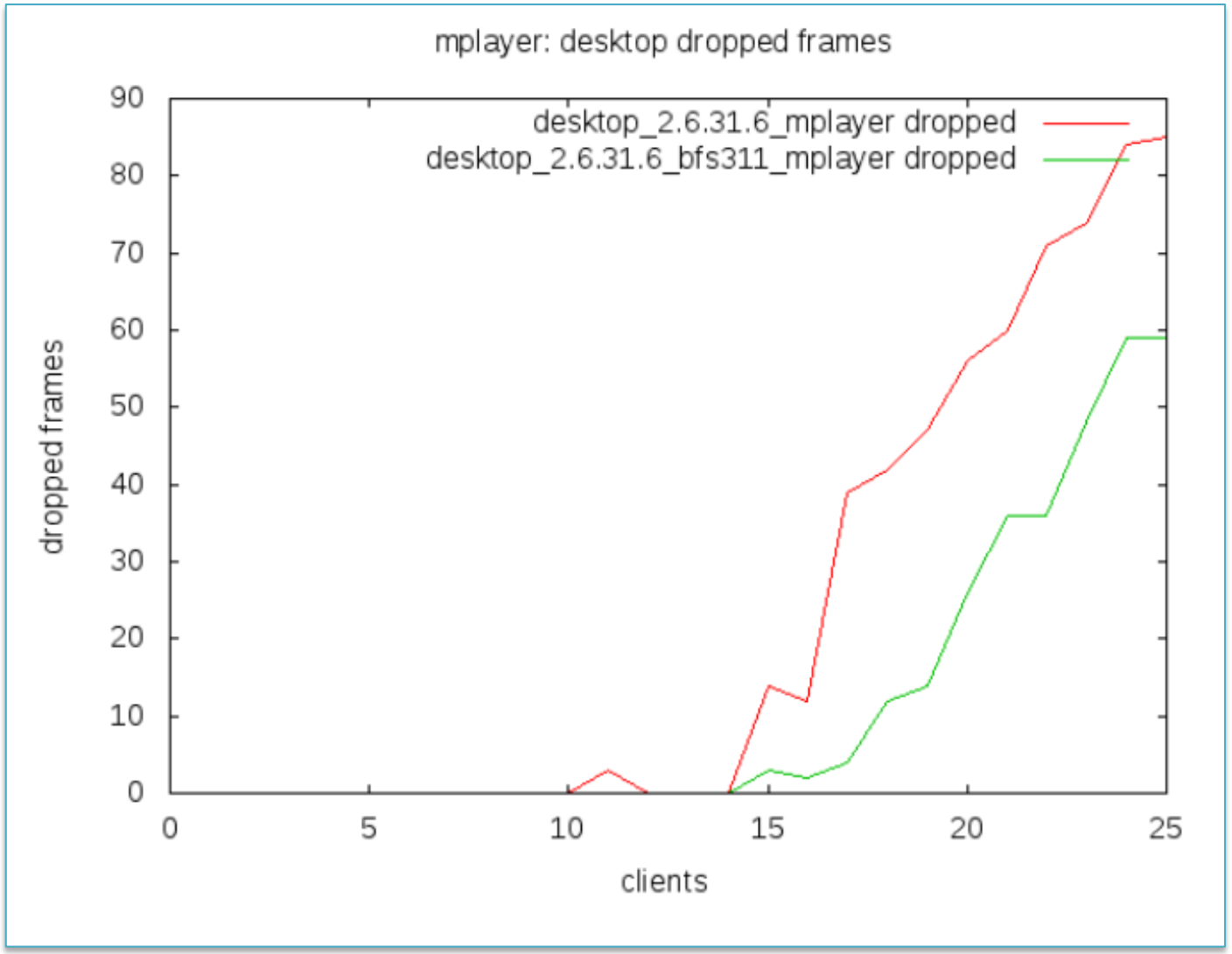
Turnaround time



http://cs.unm.edu/~eschulte/classes/cs587/data/bfs-v-cfs_groves-knockel-schulte.pdf

Experiments by T. Groves, J. Knockel, E. Schulte (2009-12-11)

Interactivity



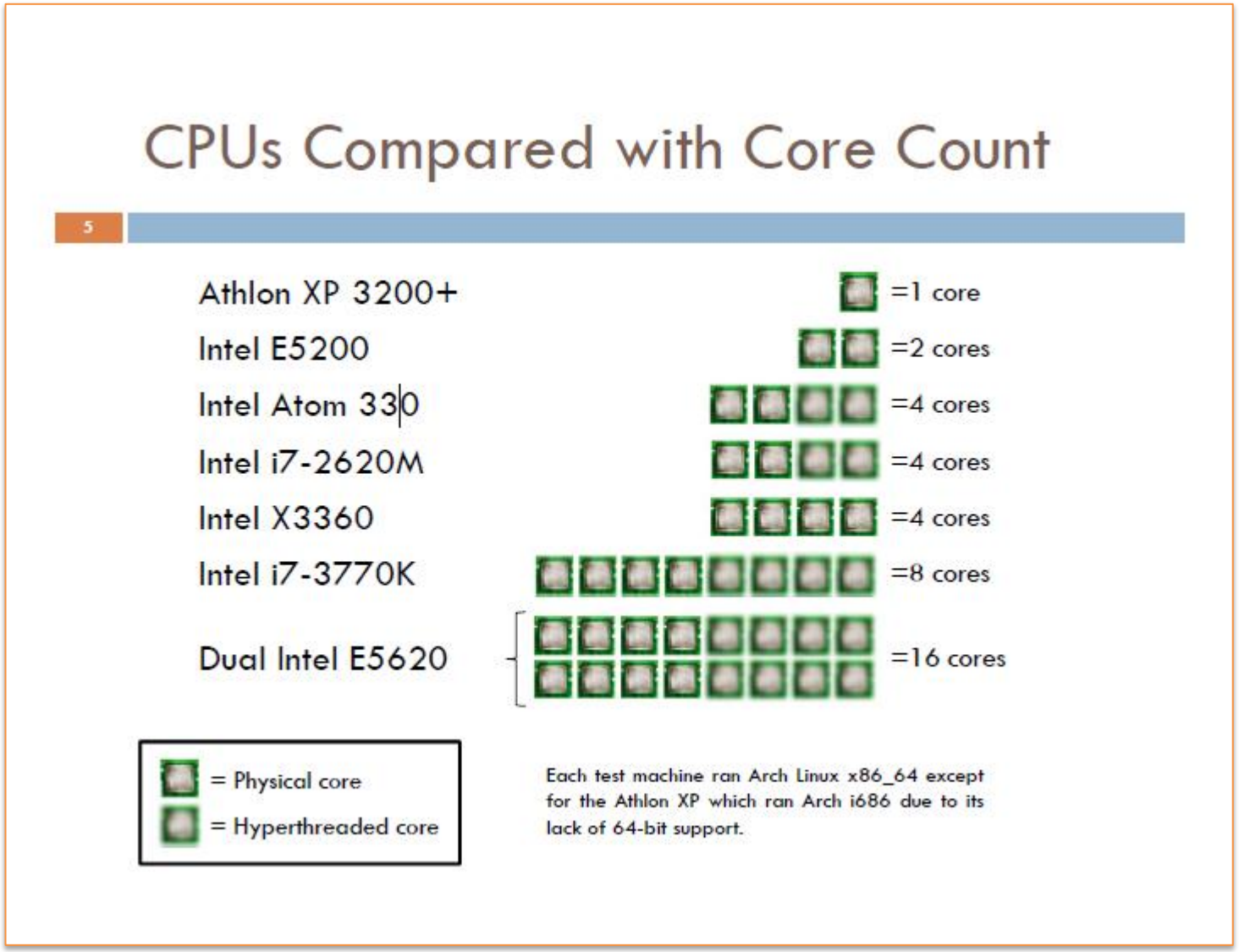
http://cs.unm.edu/~eschulte/classes/cs587/data/bfs-v-cfs_groves-knockel-schulte.pdf

T. Groves, J. Knockel, E. Schulte Conclusions

- The results indicate that CFS outperformed BFS with minimizing turnaround time but that BFS outperformed CFS for minimizing latency. This indicates that BFS is better for interactive tasks that block on I/O or user input and that CFS is better for batch processing that is CPU bound.
- Many distros like Ubuntu already have separate kernel packages for desktops and servers optimized for those common use cases. To improve the average desktop experience, distros could patch their kernel to use the BFS scheduler. If desktop users do perform a lot of batch processing, **distros could provide two different kernel packages alternatives**.

http://cs.unm.edu/~eschulte/classes/cs587/data/bfs-v-cfs_groves-knockel-schulte.pdf

CPU schedulers compared – Graysky, 2012



http://repo-ck.com/bench/cpu_schedulers_compared.pdf

CPU schedulers compared – Graysky, 2012

Compression Benchmark Results

7

CPU	Average Time (sec)		CK Kernel is…	
	Vanilla	*CK-Patched*	*Difference*	*Result*
AMD Athlon XP	558.3320	**547.7577**	-10.5753	1.9 % faster
Intel E5200	166.3141	**165.7622**	-0.5519	0.3 % faster
Intel Atom 330	470.0283	**454.8185**	-15.2098	3.2 % faster
Intel i7-2620M	81.2978	**79.4898**	-1.808	2.2 % faster
Intel X3360	68.2635	**67.7987**	-0.4648	0.7 % faster
Intel i7-3770K	35.2904	**34.3310**	-0.9594	2.7 % faster
Dual Intel E5620	**27.7919**	28.2716	+0.4797	1.7 % slower

Small (1-3 %) efficiency/speed gains were observed almost universally across the test systems when timing compression using lrzip. The notable exception being the multi-socket machine which was around 2 % slower.

http://repo-ck.com/bench/cpu_schedulers_compared.pdf

CPU schedulers compared – Graysky, 2012

Make Benchmark Results

8

CPU	Average Time (sec)		CK Kernel is…	
	Vanilla	*CK-Patched*	*Difference*	*Result*
AMD Athlon XP	1,120.6486	**1,095.6486**	-25.4671	2.3 % faster
Intel E5200	374.0274	**366.0912**	-7.9362	2.1 % faster
Intel Atom 330	1,568.5016	**1,546.4804**	-22.0212	1.4 % faster
Intel i7-2620M	192.5477	**190.6712**	-1.8765	1.4 % faster
Intel X3360	127.6179	**127.2340**	-0.3839	0.3 % faster
Intel i7-3770K	68.9835	**67.9671**	-1.0164	1.5 % faster
Dual Intel E5620	74.1218	**68.2665**	-5.8553	7.9 % faster

Small to moderate (2-8 %) efficiency/speed gains were observed across all test systems with the gcc-based endpoint. Of note is the multi-socket "dual quad" machine which saw the largest boost using the bfs of nearly 8 %.

http://repo-ck.com/bench/cpu_schedulers_compared.pdf

CPU schedulers compared – Graysky, 2012

Video Benchmark Results

9

CPU	Average Time (sec)		CK Kernel is…	
	Vanilla	*CK-Patched*	*Difference*	*Result*
AMD Athlon XP	**380.8340**	386.4206	+5.5866	1.5 % slower
Intel E5200	102[3183	**99.8744**	-2.4439	2.4 % faster
Intel Atom 330	471.0781	**443.4450**	-27.6331	5.9 % faster
Intel i7-2620M	50.1631	**48.7489**	-1.4142	2.8 % faster
Intel X3360	39.3656	**37.6724**	-1.7232	4.4 % faster
Intel i7-3770K	19.6863	**18.1044**	-1.5819	8.0 % faster
Dual Intel E5620	30.9037	**30.7141**	-0.1896	0.6 % faster

Small to moderate (2-8 %) efficiency/speed gains were observed almost universally across the test systems when timing ffmpeg video encoding. Here the oldest CPU showed a slight decrease in speed of around 1.5 %.

http://repo-ck.com/bench/cpu_schedulers_compared.pdf

CPU schedulers compared – Graysky, 2012

Conclusion

10

In addition to the primary design goals of the bfs, increased desktop interactivity and responsiveness, kernels patched with the ck1 patch set including the bfs ***outperformed*** the vanilla kernel using the cfs at nearly all the performance-based benchmarks tested. Further study with a larger test set could be conducted, but based on the small test set of 7 PCs evaluated, these increases in process queuing, efficiency/speed are, on the whole, independent of CPU type (mono, dual, quad, hyperthreaded, etc.), CPU architecture (32-bit and 64-bit), and of CPU multiplicity (mono or dual socket).

Moreover, several "modern" CPUs (Intel C2D and Ci7) that represent common workstations and laptops, consistently outperformed the vanilla kernel at all benchmarks. Efficiency and speed gains were small to moderate.

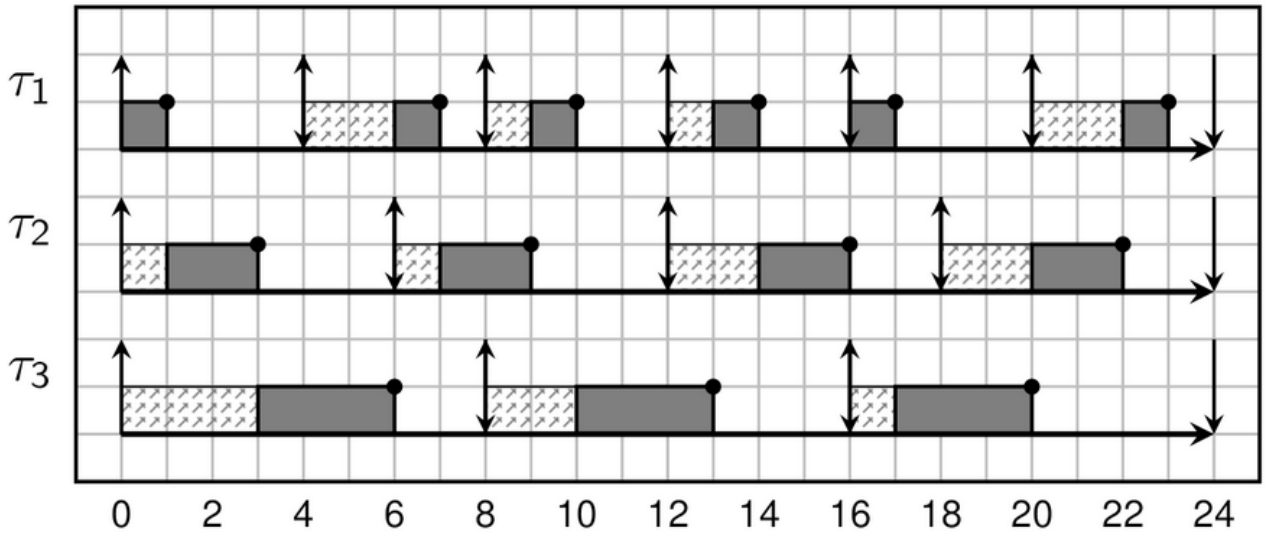
Feel free to contact the author with questions, suggestions, or rants: graysky AT archlinux DOT us

http://repo-ck.com/bench/cpu_schedulers_compared.pdf

The Deadline scheduler

Worst-case execution time

Task	Runtime (WCET)	Period
T_1	1	4
T_2	2	6
T_3	3	8



https://lwn.net/Articles/743740/

it is not possible to use a fixed-priority scheduler to schedule this task set while meeting every deadline; regardless of the assignment of priorities, one task will not run in time to get its work done.

Deadline scheduling gets away with the notion of process priorities. Instead, processes provide three parameters: runtime, period, and deadline. A SCHED_DEADLINE task is guaranteed to receive "**runtime**" microseconds of execution time every "**period**" microseconds, and these "runtime" microseconds are available within "**deadline**" microseconds from the beginning of the period. The task scheduler uses that information to run the process with the **earliest deadline first** (EDF).

The Deadline scheduler

- /Documentation/echeduler/sched-deadline.rst.
- SCHED_DEADLINE in Wikipedia.
- Deadline scheduling: coming soon?, Jonathan Corbet, December 2013.
- Deadline scheduling in the Linux kernel, J.Lelli, C. Scordino, L. Abeni, D.Faggioli, Software – Practice& Experience, vol. 46 (6), June 2016.

  Container-Based Real-Time Scheduling in the Linux Kernel, L. Abeni, A. Balsini, T. Cucinotta, EWiLi'18, October 4th, 2018.
- Deadline scheduling part 1 — overview and theory, D. Oliveira, 2018.
- Deadline scheduler part 2 — details and usage, D.Oliveira, 2018.
- https://lwn.net/Kernel/Index/#Realtime-Deadline_scheduling.
- SCHED_DEADLINE desiderata and slightly crazy ideas, D. Oliveira, J. Lelli, DevConf.cz, January 2020.
- Capacity awareness for the deadline scheduler, M. Rybczyńska, May 2020.

  The current implementation of the deadline scheduler does not work well on asymmetric CPU configurations like Arm's big.LITTLE. Dietmar Eggemann posted a patch set to address this problem by adding the notion of **CPU capacity** (the number of instructions that can be executed in a given time) and taking it into account in the admission-control and task-placement algorithms.

The MuQSS CPU scheduler

- Based on https://lwn.net/Articles/720227/ by Nur Hussein, April 2017.
- Original message from Con Kolivas : http://ck-hack.blogspot.com/2016/10/muqss-multiple-queue-skiplist-scheduler.html announcing patch for Linux 4.7 (October 2016).

- **MuQSS – The Multiple Queue Skiplist Scheduler** (pronounced *mux*)
- The main goal is to tackle 2 major **scalability limitations** in BFS:
  - The **single runqueue** which means all CPUs would fight for lock contention over the one runqueue (problems start when the **number of CPUs increases** beyond 16),
  - The **O(n) look up** which means linear increase in overhead for task lookups as **number of processes increases**. Also, iterating over a linked list led to cache-thrashing behavior.

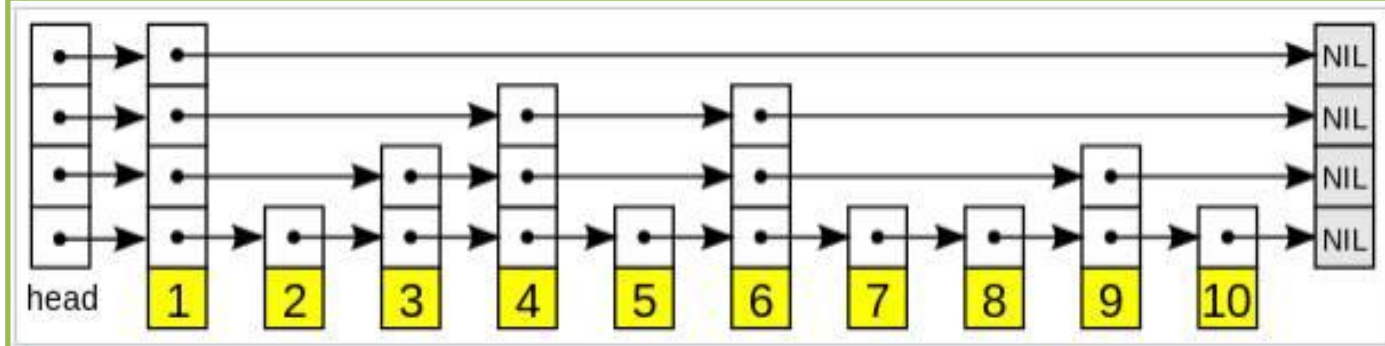
The MuQSS CPU scheduler

MuQSS is BFS with **multiple run queues**, **one per CPU**.

The queues have been implemented as **skip lists**.

Wikipedia, https://en.wikipedia.org/wiki/Skip_list

Skip list is a data structure that allows **O(log n) search** complexity as well as **O(log n) insertion** complexity within an ordered sequence of **n** elements.



A schematic picture of the skip list data structure. Each box with an arrow represents a pointer and a row is a linked list giving a sparse subsequence; the numbered boxes (in yellow) at the bottom represent the ordered data sequence. Searching proceeds downwards from the sparsest subsequence at the top until consecutive elements bracketing the search element are found.

The MuQSS CPU scheduler

Virtual deadline is calculated as in BFS, **niffies** are used instead of **jiffies** (nanosecond-resolution monotonic counter)

virtual_deadline = niffies + (prio_ratio * rr_interval)

The scheduler can find the next eligible task to run in **O(1)**, insertion is done in **O(log n)**.

The scheduler will use a **non-blocking** "trylock" attempt when popping the chosen task from the relevant run queue, but will move on to the next-nearest deadline on another queue if it fails to acquire the queue lock (no lock contention among different CPU queues).

Only 3 configuration parameters:

rr_interval – CPU quantum, which defaults to 6 ms,

interactive – a tunable to toggle the deadline behavior. If disabled, searching for the next task to run is done independently on each CPU, instead of across all CPUs.

iso_cpu – percentage of CPU time, across a rolling five-second average, that isochronous tasks (SCHED_ISO) will be allowed to use.

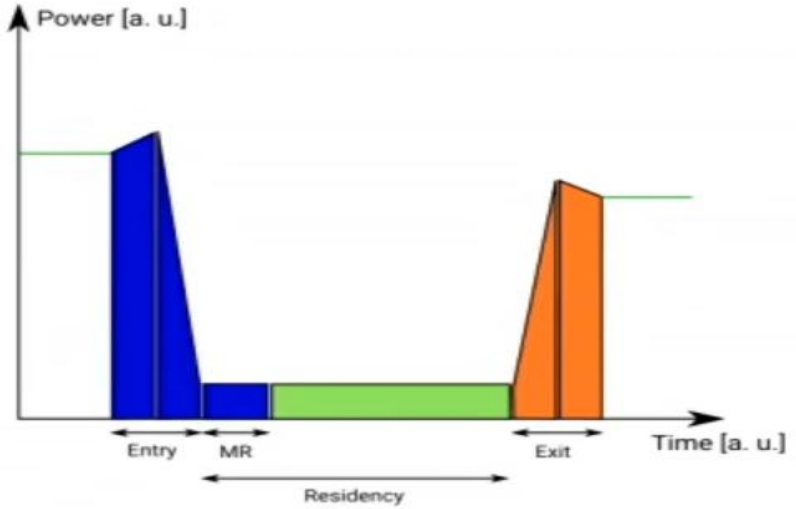
CPU Idle Loop

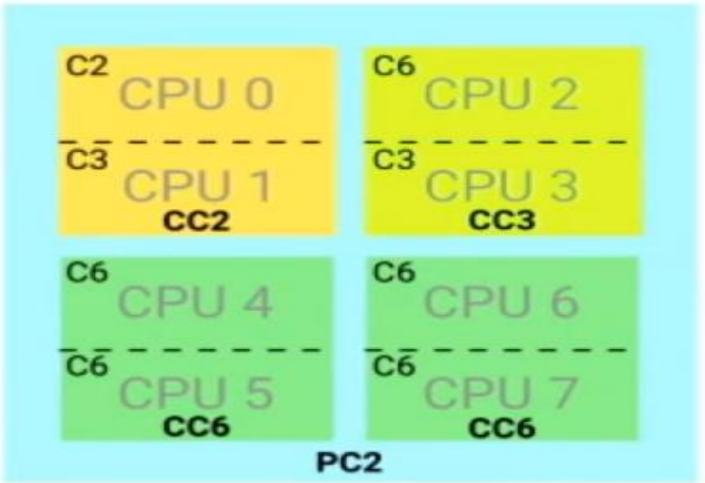
- [CPU Idle Loop Rework](), Rafael J. Wysocki (Intel), 2018.

  [What's a CPU to do when it has nothing to do?](), Tom Yates, October 2018.

  Although increasingly deep idle states consume decreasing amounts of power, they have increasingly large costs to enter and exit. It is in the kernel's best interests to predict how long a CPU will be idle before deciding how deeply to idle it. This is the job of the **idle loop**. The scheduler then calls the **governor**, which does its best to predict the appropriate idle state to enter.

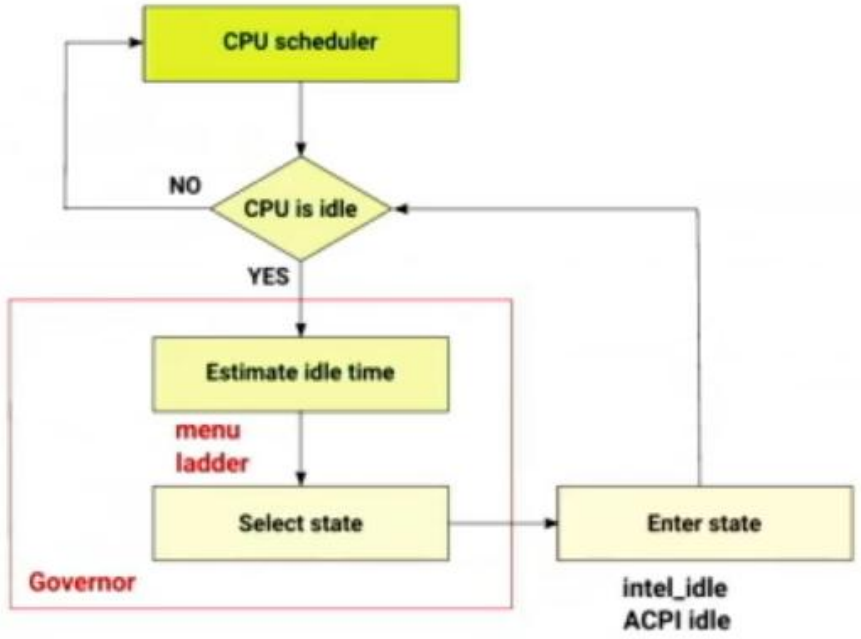
  He reworked the idle loop for kernel 4.17 so that the decision about **stopping the tick** is taken *after* the governor has made its recommendation of the idle state.



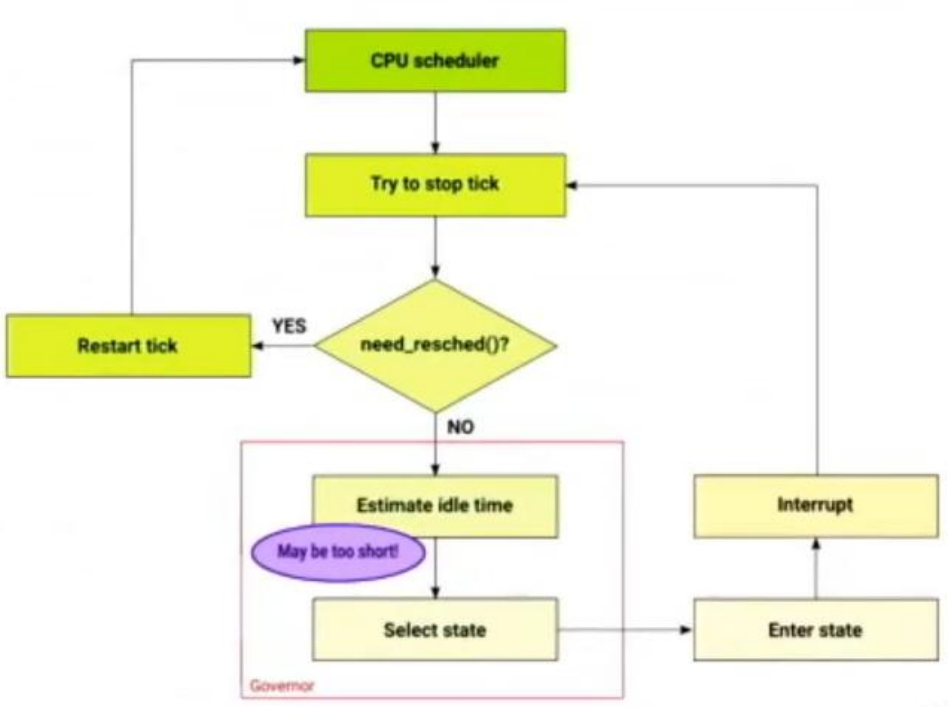


CPU Idle Loop

CPU Idle Loop Rework, Rafael J. Wysocki (Intel), 2018.



High-level CPU idle time management control flow



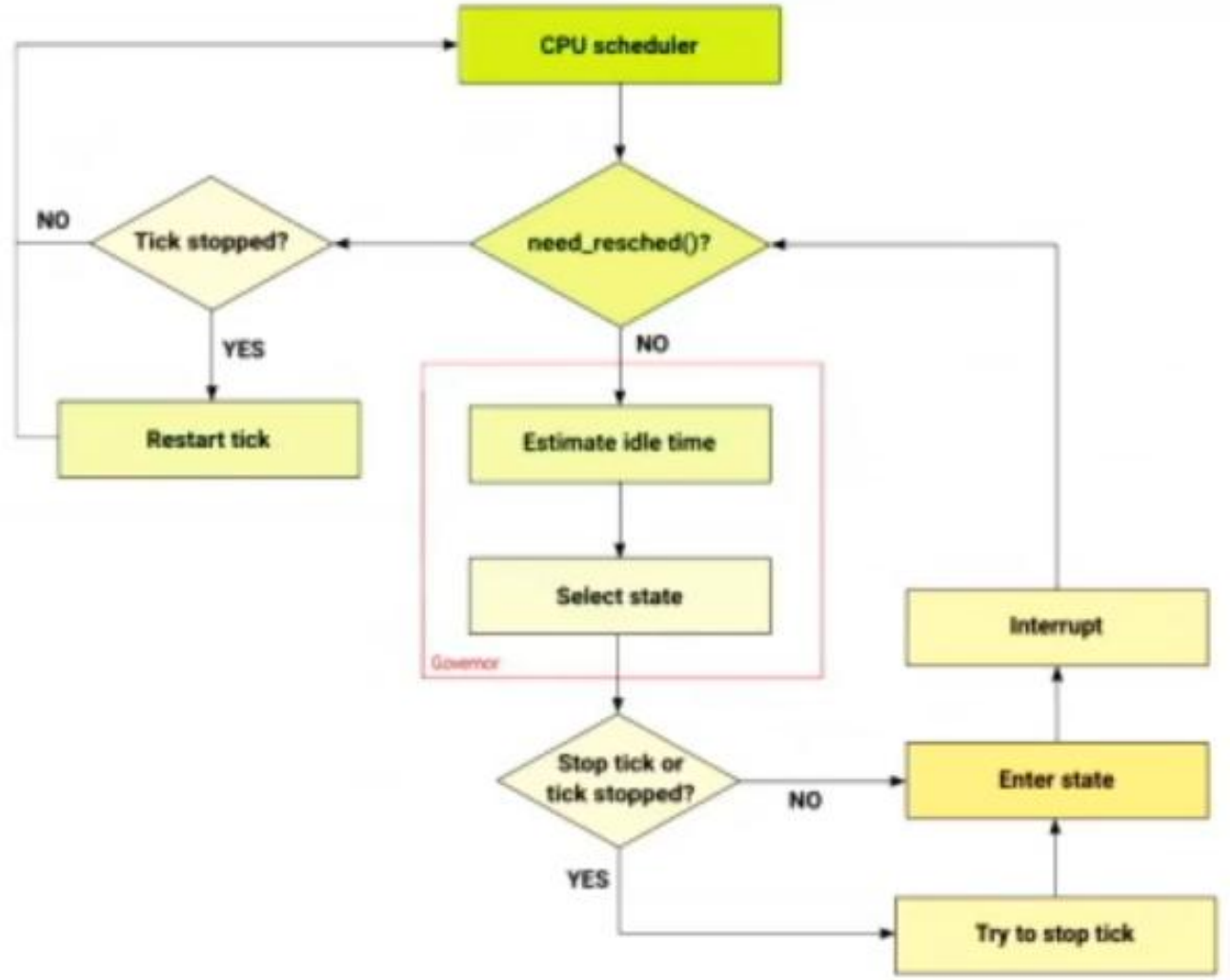
Original idle loop design issue

```
jmd@students$ cat /sys/devices/system/cpu/cpuidle/current_governor_ro
menu
```

CPU Idle Loop

Actual / Predicted	Long Idle	Short Idle
Long Idle	WIN	LOSS
Short Idle	LOSS	LOSS

Short idle duration prediction problem

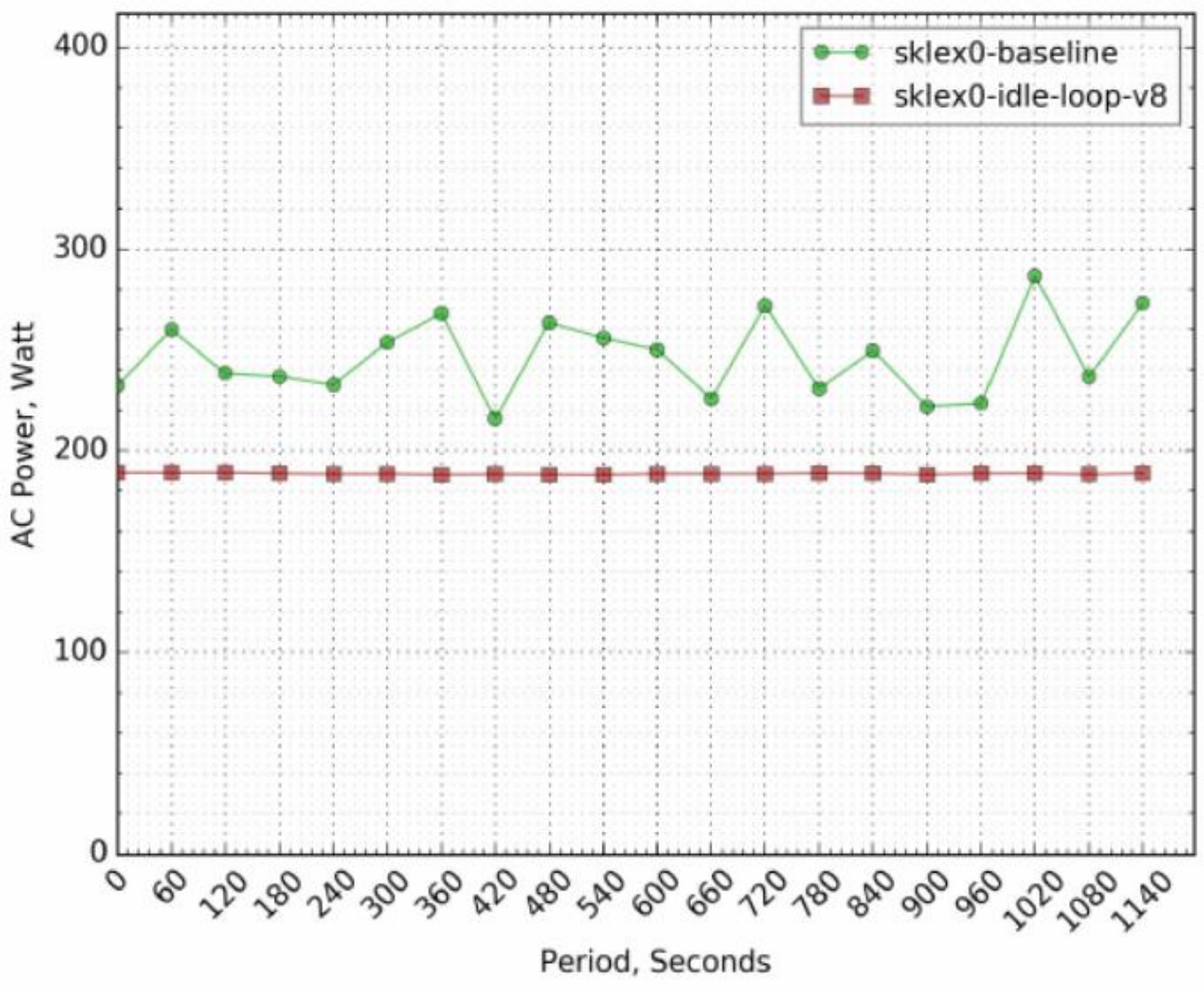


Redesigned idle loop (Linux* 4.17 and later)

[CPU Idle Loop Rework](), Rafael J. Wysocki (Intel), 2018.

Idle power (Intel OTC Server Power Lab)

The green line is with the old idle loop, the red is with the new: power consumption is less under the new scheme, and moreover it is much more predictable than before.



CPU Idle Loop Rework, Rafael J. Wysocki (Intel), 2018.

CPU Idle Loop

- Improving idle behavior in tickless systems, Marta Rybczyńska, December 2018.

  Linux currently provides two cpu idle governors, **ladder** and **menu**. Wysocki implemented the third, **timer events oriented** (TEO) – similar to menu, but takes into cosideration different factors.

- Fixing SCHED_IDLE, Viresh Kumar, November 2019 – The 5.4 kernel release includes a few improvements to the existing SCHED_IDLE scheduling policy that can help users improve the scheduling latency of their high-priority (interactive) tasks if they use the SCHED_IDLE policy for the lowest-priority (background) tasks.

Scheduling – thermal pressure

Telling the scheduler about thermal pressure, Marta Rybczyńska, May 2019.

Even with radiators and fans, a system's **CPUs** can **overheat**. When that happens, the kernel's thermal governor will **cap the maximum frequency** of that CPU to allow it to cool. The scheduler, however, is **not aware that the CPU's capacity** has changed; it may schedule more work than optimal in the current conditions, leading to a performance degradation.

The solution adds an interface to inform the scheduler about thermal events so that it can assign tasks better and thus improve the overall system performance.

The term **thermal pressure** means the difference between the maximum processing capacity of a CPU and the currently available capacity, which may be reduced by overheating events.

The two approaches, the **thermal pressure approach** and **energy-aware scheduling** (**EAS**), have different scope: thermal pressure is going to work better in asymmetric configurations where capacities are different and it is more likely to cause the scheduler to move tasks between CPUs.

The two approaches should also be independent because thermal pressure should work even if EAS is not compiled in.

Energy Aware Scheduling (EAS)

- Researched since 2013.
- Energy Aware Scheduling (EAS) on ARM wiki.

  It is an enhancement to Linux power management, unifying CPU power control under the Linux kernel. EAS extends the Linux kernel scheduler to make it fully **aware of the power/performance capabilities of the CPUs** in the system, to optimize energy consumption for advanced multi-core SoCs including **big.LITTLE**. With EAS, the Linux kernel will use the task load and a CPU 'Energy Model' to control task placement to select the optimal CPU to run on.

  Arm, Linaro and key partners are contributing jointly to the development of EAS.

  EAS is an example of a scheduler that considers cores differently.
- Energy Aware Scheduling on kernel.org
- Energy-Aware Scheduling Project on linaro.org.
- An Unbiased Look at the Energy Aware Scheduler, Vitaly Wool, Embedded Linux Conference, 2018.
- 2019: added to **Linux 5.0**.

Energy Aware Scheduling (EAS)

- Evaluating vendor changes to the scheduler, Jonathan Corbet, May 2020.

  The benchmark results for each of these patches were remarkably similar. They all tended to **hurt performance** by 3-5% while **reducing energy** use by 8-11%.

- Saving frequency scaling in the data center, J. Corbet, May 2020.

  Frequency scaling — adjusting a CPU's operating frequency to save power when the workload demands are low — is common practice across systems supported by Linux. It is, however, viewed with some suspicion in data-center settings, where power consumption is less of a concern and there is a strong emphasis on **getting the most performance** out of the hardware.

- Imbalance detection and fairness in the CPU scheduler, J. Corbet, May 2020.
- Power Management and Scheduling in the Linux Kernel, (OSPM Summit) IV edition, May 2020.

Scheduling – Arm big.LITTLE CPU chip

Scheduling for asymmetric Arm systems, Jonathan Corbet, November 2020.

The **big.LITTLE architecture** placed **fast** (but power-hungry) and **slower** (but more power-efficient) CPUs in the same **system-on-chip** (**SoC**); significant scheduler changes were needed for Linux to be able to properly distribute tasks on such systems.

Putting tasks on the wrong CPU can result in poor performance or excessive power consumption, so a lot of work has gone into the problem of **optimally distributing workloads** on big.LITTLE systems.

When the scheduler gets it wrong, though, performance will suffer, but things will still work.

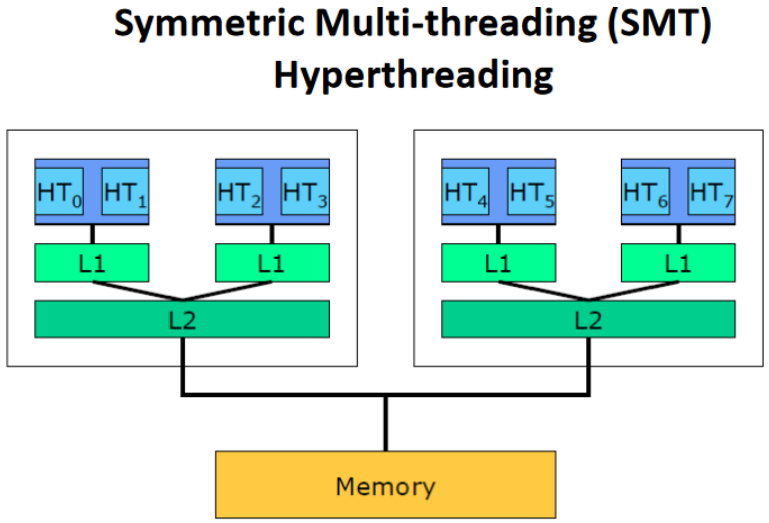
Future Arm designs, include systems where some CPUs can run **both 64-bit and 32-bit** tasks, while others are **limited to 64-bit tasks** only. The result of an incorrect scheduling choice is no longer a matter of performance; it could be catastrophic for the workload involved.

What should happen if a 32-bit task attempts to run on a 64-bit-only CPU?

- Kill the task or
- recalculate the task's CPU-affinity mask?

Cortex A57/A53 MPCore big.LITTLE CPU chip

Core scheduling



Core scheduling, Jonathan Corbet, February 2019.

SMT (**simultaneous multithreading**) increases performance by turning one physical CPU into two virtual CPUs that share the hardware; while one is waiting for data from memory, the other can be executing. Sharing a processor this closely has led to security issues and concerns for years, and many security-conscious users disable SMT entirely.

On kernels where core scheduling is enabled, a **core_cookie** field is added to the task structure. These cookies are used to define the trust boundaries; two processes with the same cookie value trust each other and can be allowed to run simultaneously on the same core. (Peter Zijlstra)

Completing and merging core scheduling, Jonathan Corbet, May 2020.

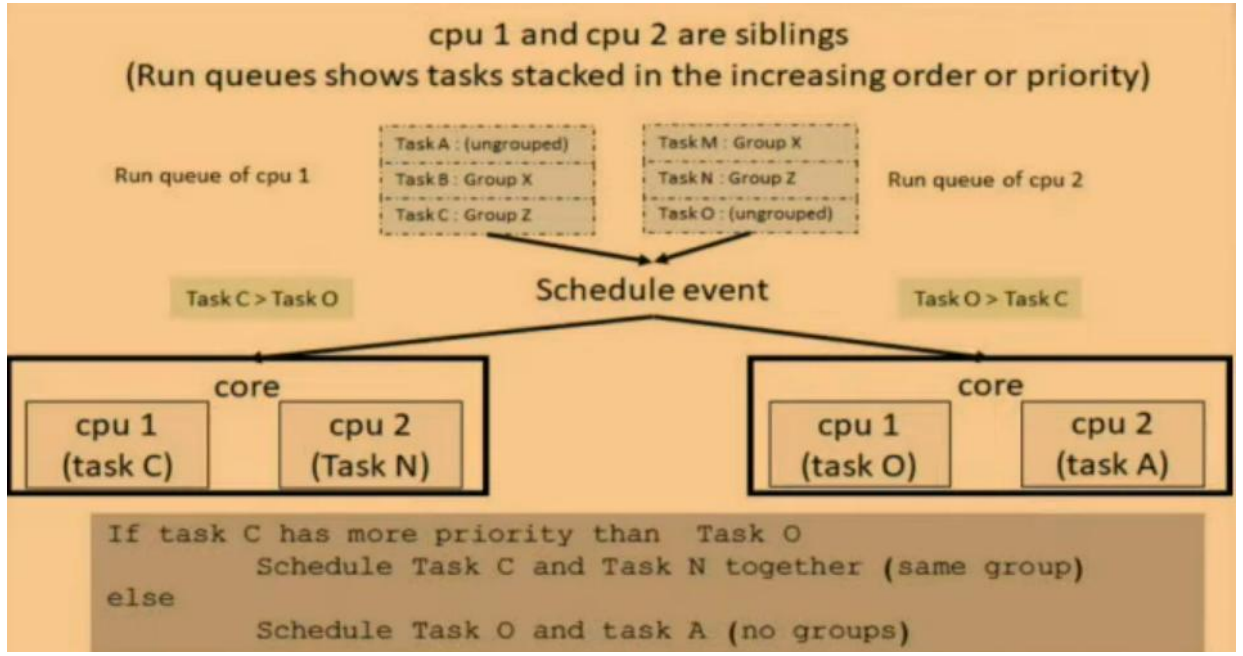
A set of virtualization tests showed the system running at **96%** of the performance of an unmodified kernel with **core scheduling enabled**; the 4% performance hit hurts, but it's far better than the **87%** performance result measured for this workload with **SMT turned off** entirely.

The all-important kernel-build benchmark showed almost no penalty with core scheduling, while turning off SMT cost 8%.

Core scheduling

Core Scheduling Looks Like It Will Be Ready For Linux 5.14 To Avoid Disabling SMT/HT, Michael Larabel, May 2021.

Core scheduling should be effective at mitigating user-space to user-space and user-to-kernel attacks when the functionality is properly used. But the default kernel policy will not change over how tasks are scheduled but is up to the administrator for identifying tasks that can or cannot share CPU resources.



https://www.youtube.com/watch?v=_8_xUf47-jE

Conclusions

Scheduler performance varies dramatically according to hardware and workload, and as a result we strongly encourage Linux distributions to take an increased level of responsibility for selecting appropriate default schedulers that best suit the intended usage of the system.