Graphing Tools for Scheduler Tracing

Julia Lawall, Inria

February 5, 2023

What is a task scheduler?

An important part of the Linux kernel:

- Places tasks on cores on fork, wakeup, or load balancing.
- Selects a task on the core to run when the core becomes idle.
- `kernel/sched/core.c`, `kernel/sched/fair.c`

What is a task scheduler?

An important part of the Linux kernel:

- Places tasks on cores on fork, wakeup, or load balancing.
- Selects a task on the core to run when the core becomes idle.
- `kernel/sched/core.c`, `kernel/sched/fair.c`

We are interested in task placement in this talk.

How can a task scheduler impact application performance?

- A scheduler has to make decisions.
- Poor decisions can slow tasks down, sometimes in the long term.

Issues: Work conservation

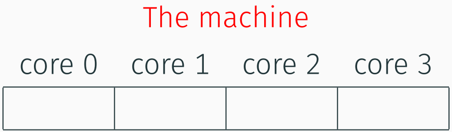
The machine

Issues: Work conservation

The machine

Where to put waking task **T1**?

Issues: Work conservation



Where to put waking task **T1**?

- Maybe anywhere is fine...

Issues: Work conservation

The machine

core 0	core 1	core 2	core 3
T1			

Where to put waking task **T1**?

- Maybe anywhere is fine...

Issues: Work conservation

The machine

core 0	core 1	core 2	core 3
T1			

Where to put waking task **T2**?

Issues: Work conservation

The machine

core 0	core 1	core 2	core 3
T1			

Where to put waking task **T2**?

- Core 1, core 2, or core 3 might be fine.
- Core 0 would not be a good choice.

Issues: Work conservation

The machine

core 0	core 1	core 2	core 3
T1		**T2**	

Where to put waking task **T2**?

- Core 1, core 2, or core 3 might be fine.
- Core 0 would not be a good choice.

Work conservation: No core should be overloaded if any core is idle.

Issues: Locality

A two-socket machine

core 0	core 1	core 2	core 3
T1			

Issues: Locality

A two-socket machine

core 0	core 1	core 2	core 3
T1			

Where to put waking task **T2**?

- Core 1 is good if **T2** has previously allocated memory on that socket.
- Core 1 is good if **T2** communicates a lot with **T1**.
- Core 2 or Core 3 could cause slowdowns.

A challenge

- The task scheduler can have a large impact on application performance.
- But the task scheduler is buried deep in the OS...

A challenge

- The task scheduler can have a large impact on application performance.
- But the task scheduler is buried deep in the OS...
- How to understand what the task scheduler is doing?

Some help available

`trace-cmd`: Collects ftrace information, including scheduling events.

```
trace-cmd -e sched -q -o trace.dat ./mycommand
```

Sample trace:

```
C1 CompilerThre-166659 [026]  9539.524366: sched_wakeup: C1 CompilerThre:166654 [120] success=1 CPU:062
          <idle>-0     [062]  9539.524369: sched_switch: swapper/62:0 [120] R ==> C1 CompilerThre:166654 [120]
C1 CompilerThre-166659 [026]  9539.524369: sched_switch: C1 CompilerThre:166659 [120] S ==> swapper/26:0 [120]
           java-166654 [062]  9539.524372: sched_waking: comm=C1 CompilerThre pid=166660 prio=120 target_cpu=028
```

Some help available

`kernelshark`: Graphical front end for `trace-cmd` data.

Some help available

`kernelshark`: Graphical front end for `trace-cmd` data.

Hard to get an overview, of *e.g.* 128 cores.

Our target: Large multicore servers

Goals for a trace-visualization tool:

- See activity on all cores at once.
- Produce files that can be shared (pdfs).
- Caveat: Interactivity (e.g., zooming) completely abandoned.

Our tools

- `dat2graph`: Horizontal bar graph showing what is happening on each core at each time.
- `running_waiting`: Line graph of how many tasks are running or waiting on a runqueue at any point in time.

Both publicly available.

Motivating example (a commit in Linux 5.11)

```
commit d8fcb81f1acf651a0e50eacecca43d0524984f87
Author: Julia Lawall <Julia.Lawall@inria.fr>
Date:   Thu Oct 22 15:15:50 2020 +0200

sched/fair: Check for idle core in wake_affine
...

diff --git a/kernel/sched/fair.c b/kernel/sched/fair.c
--- a/kernel/sched/fair.c
+++ b/kernel/sched/fair.c
@@ -5813,6 +5813,9 @@ wake_affine_idle(int this_cpu, int prev_cpu, int sync)
        if (sync && cpu_rq(this_cpu)->nr_running == 1)
                return this_cpu;

+       if (available_idle_cpu(prev_cpu))
+               return prev_cpu;
+
        return nr_cpumask_bits;
 }
```

Example

NAS benchmark suite: “The NAS Parallel Benchmarks (NPB) are a small set of programs designed to help evaluate the performance of parallel supercomputers. The benchmarks are derived from computational fluid dynamics (CFD) applications...”

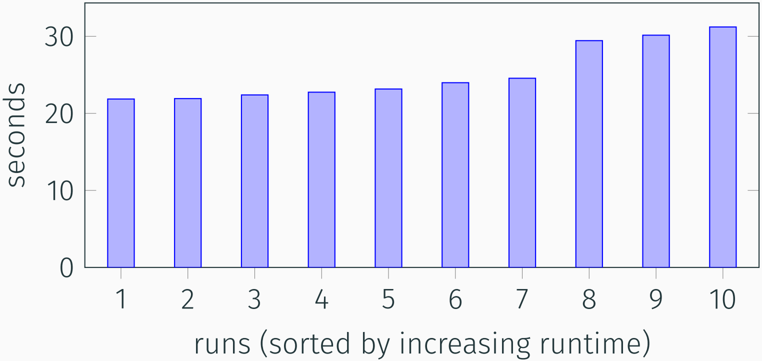
Our focus:

UA: “Unstructured Adaptive mesh, dynamic and irregular memory access”

- N tasks on N cores.

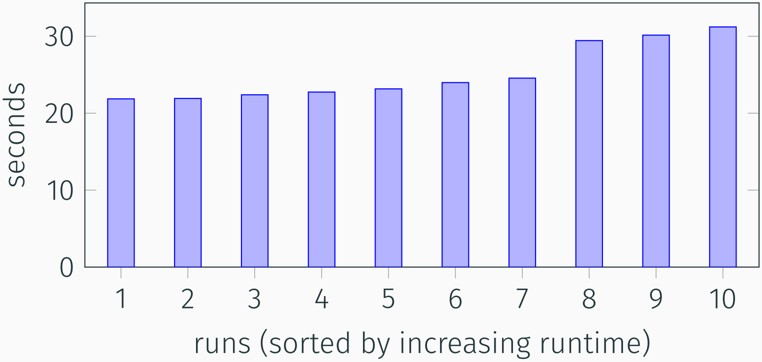
UA runtimes prior to my patch

4-socket, 128 core, Intel 6130.



UA runtimes prior to my patch

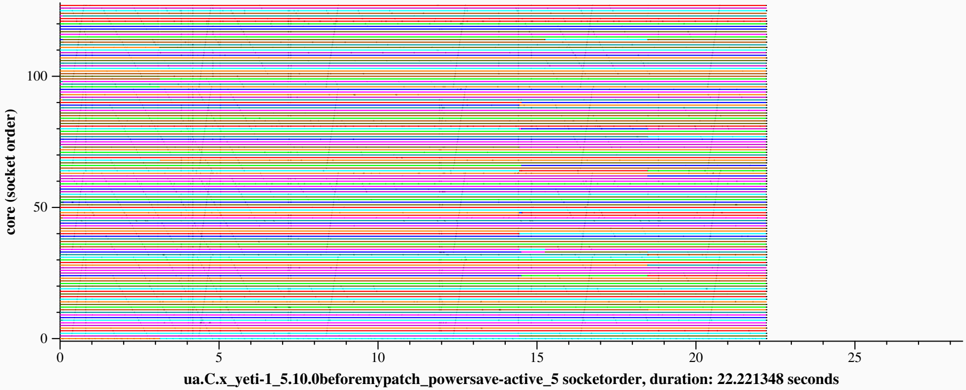
4-socket, 128 core, Intel 6130.



Why so much variation?

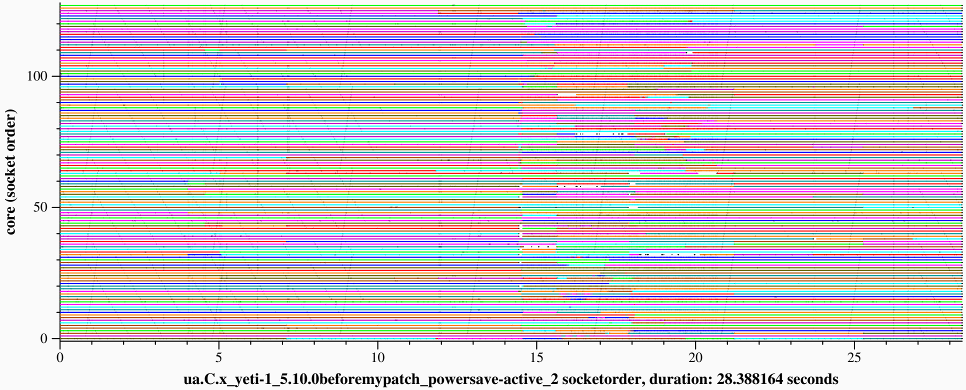
UA with dat2graph

A fast run (`dat2graph2 --socket-order ua..._5.dat`).



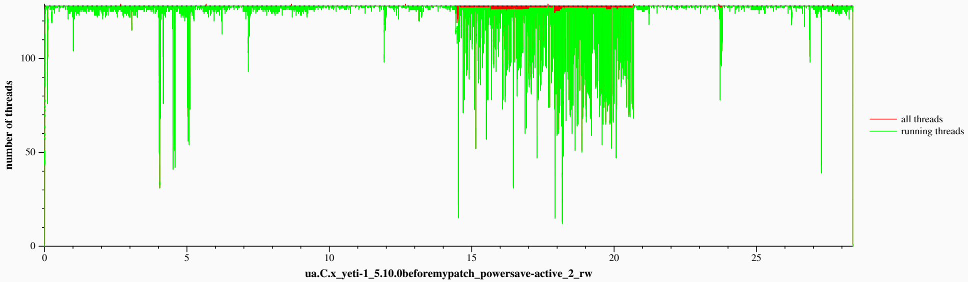
UA with dat2graph

A slow run (`dat2graph2 --socket-order ua..._2.dat`).



UA with dat2graph

Another perspective on the slow run.

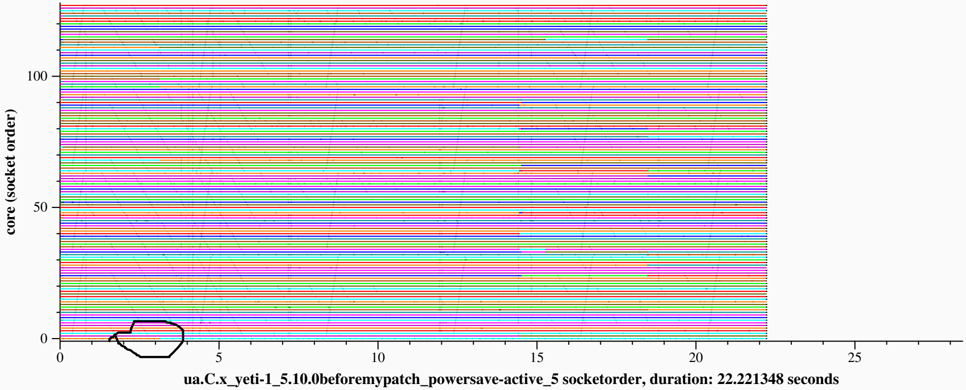


The problem

- Tasks are moving around.
- Some cores are overloaded, so tasks run less often.

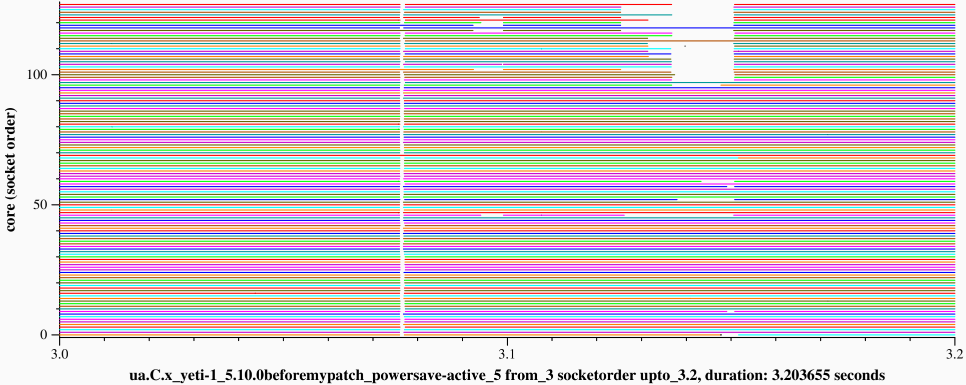
The fast run revisited

Tasks move around sometimes, for example around 3 seconds:

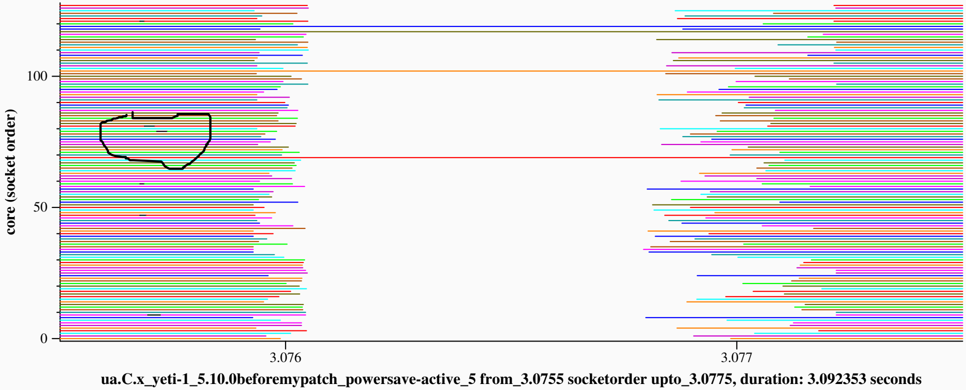


Zooming in

```
dat2graph2 --socket-order --min 3 --max 3.2 --target ua
ua.C.x_yeti-1_5.10.0beforemypatch_powersave-active_5.dat
```

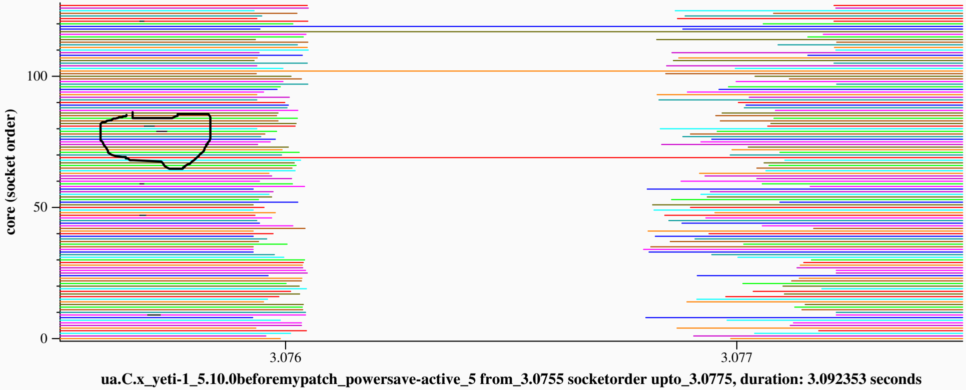


Focusing on the first gap



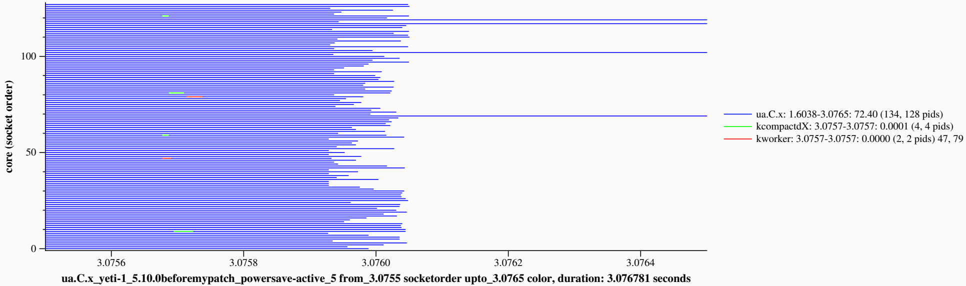
Focusing on the first gap

What are the black lines?



Color by command

```
dat2graph2 --socket-order --min 3.0755 --max 3.0765
--color-by-command
ua.C.x_yeti-1_5.10.0beforemypatch_powersave-active_5.dat
```



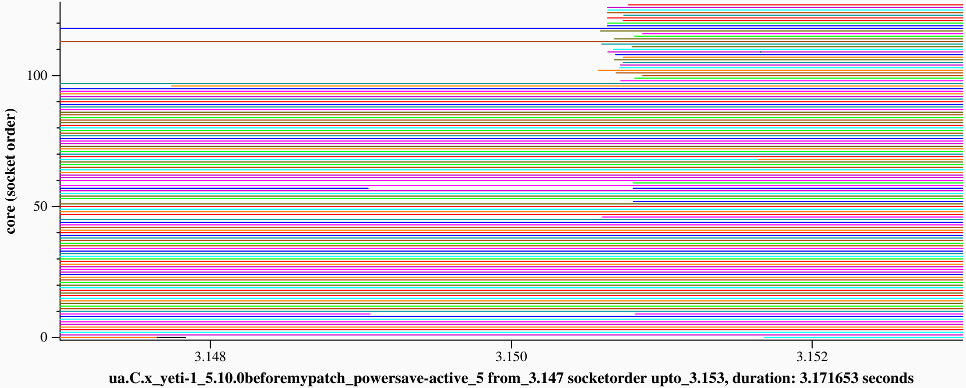
Assessment

- Kernel threads show up from time to time, to provide needed services.
- Having high priority, they preempt the running task.
- Some tasks get behind, leading to gaps until resynchronization.
- No application-application overloads introduced.

Assessment

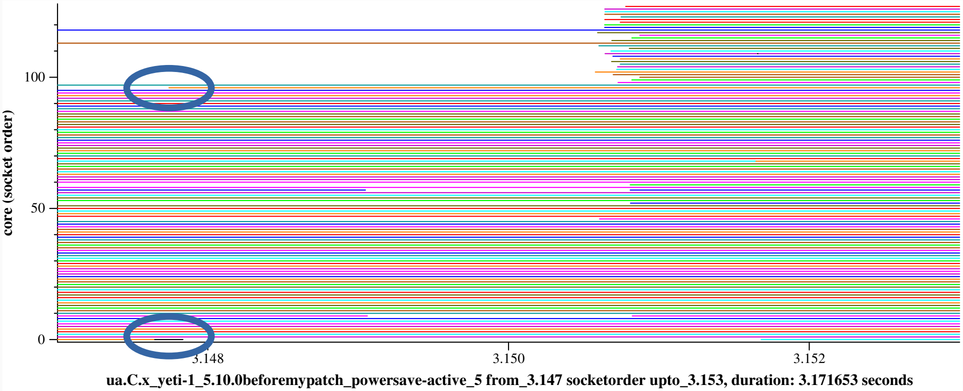
- Kernel threads show up from time to time, to provide needed services.
- Having high priority, they preempt the running task.
- Some tasks get behind, leading to gaps until resynchronization.
- No application-application overloads introduced.
- Life goes on...

Moving a bit to the right



Load balancing

Pid 12569 gets load balanced from core 0 to core 96 (off socket).



A cascade of migrations

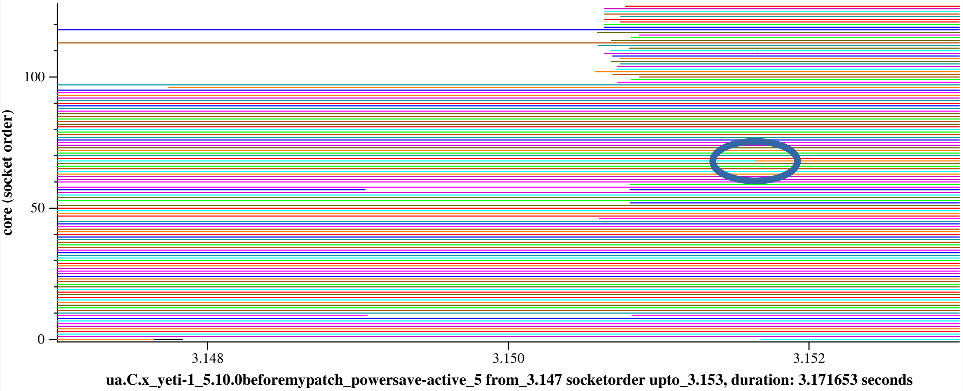
- 12569 gets load balanced from core 0 to core 96.
- 12561 wakes for core 96 but is moved to core 99.
- 12564 wakes for core 99 but is moved to core 100.
- 12568 wakes for core 100 but is moved to core 111.

A cascade of migrations

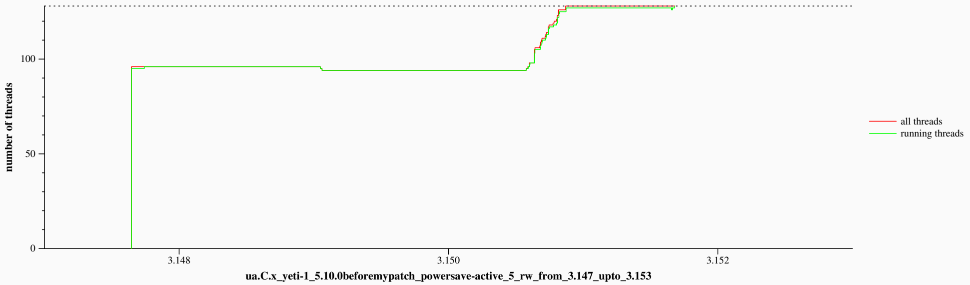
- 12569 gets load balanced from core 0 to core 96.
- 12561 wakes for core 96 but is moved to core 99.
- 12564 wakes for core 99 but is moved to core 100.
- 12568 wakes for core 100 but is moved to core 111.
- Each task finds a place on the fourth socket, but one too many tasks want to be placed there.

Overload

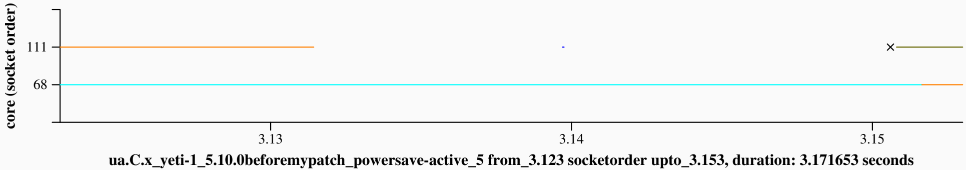
UA-UA overload (no black dot)



Running-waiting view

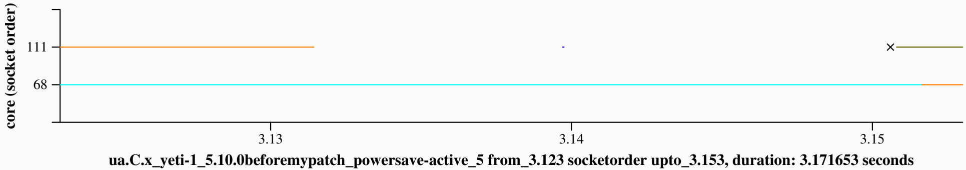


Understanding the source of the overload



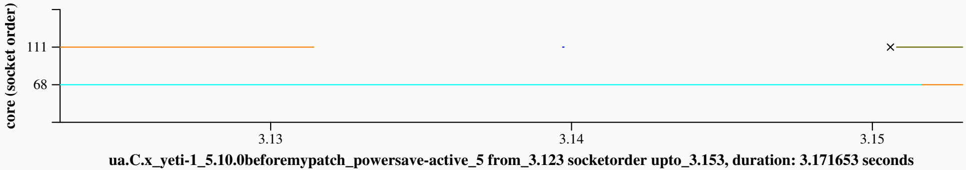
- 12655 on core 68 wakes 12549 for core 111

Understanding the source of the overload



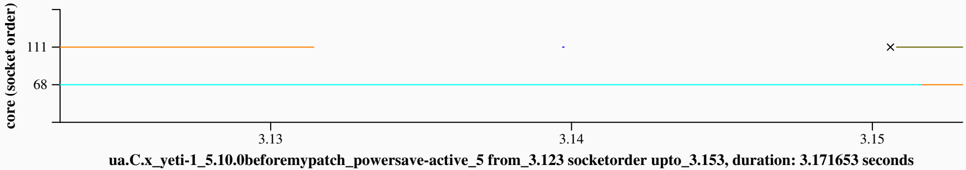
- 12655 on core 68 wakes 12549 for core 111
- 111 is idle!

Understanding the source of the overload



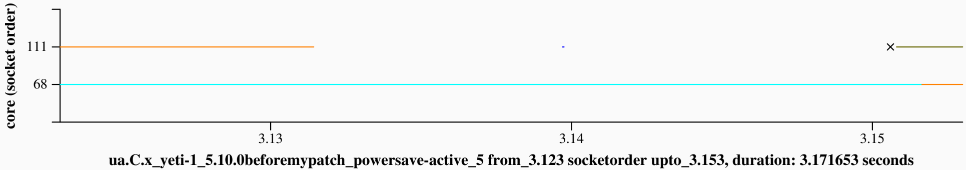
- 12655 on core 68 wakes 12549 for core 111
- 111 is idle!
- But 12549 is placed on core 111, where it has to wait for 12655

Understanding the source of the overload



- 12655 on core 68 wakes 12549 for core 111
- 111 is idle!
- But 12549 is placed on core 111, where it has to wait for 12655
- Huhhh???? (Remember work conservation).

Understanding the source of the overload

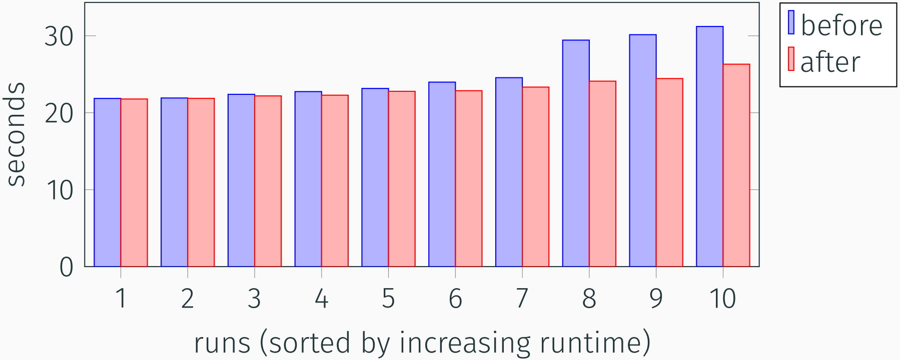


- 111 is idle when 12655 wakes, but it was used by a kworker recently.
- The load average is non zero.
- The scheduler prefers to put 12655 on the socket of the waker.
- This socket is all full, so there is an overload (12655 has to wait).

Back to the patch

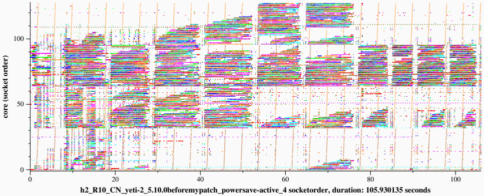
```
diff --git a/kernel/sched/fair.c b/kernel/sched/fair.c
--- a/kernel/sched/fair.c
+++ b/kernel/sched/fair.c
@@ -5813,6 +5813,9 @@ wake_affine_idle(int this_cpu, int prev_cpu, int sync)
        if (sync && cpu_rq(this_cpu)->nr_running == 1)
                return this_cpu;

+       if (available_idle_cpu(prev_cpu))
+               return prev_cpu;
+
        return nr_cpumask_bits;
 }
```

Benefit on UA

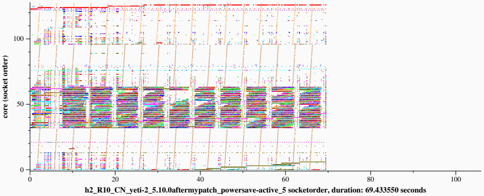


Benefit on another application

h2: part of the DaCapo Java benchmark suite.



before the patch (81-105sec)



after the patch (63-69 sec)

Conclusion

- Understanding scheduler behavior requires studying precise scheduling actions.
- Different perspectives provide complementary information.
- Some tools that I have found useful for large multicore machines:
  - `dat2graph2`: Who is running, when and where?
  - `running_waiting`: How many tasks are running, how many are waiting?
- Future work: Faster graph generation? More configurability?

Conclusion

- Understanding scheduler behavior requires studying precise scheduling actions.
- Different perspectives provide complementary information.
- Some tools that I have found useful for large multicore machines:
  - `dat2graph2`: Who is running, when and where?
  - `running_waiting`: How many tasks are running, how many are waiting?
- Future work: Faster graph generation? More configurability?

https://gitlab.inria.fr/schedgraph/schedgraph.git