

Graphing Tools for Scheduler Tracing

Julia Lawall, Inria

November 14, 2023

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.

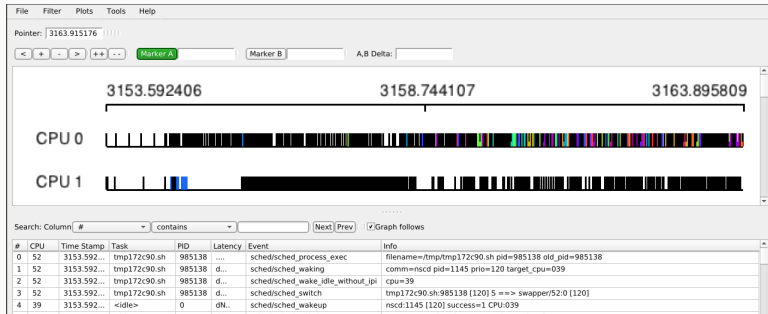
```
trace-cmd record -e sched -q -o trace.dat ./mycommand
```

Sample trace (trace-cmd report trace.dat):

```
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
```

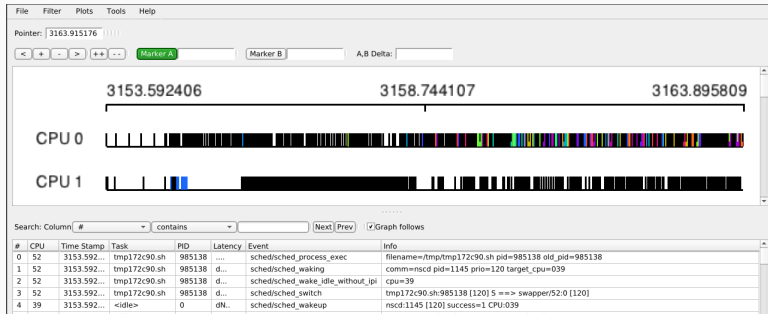
Some help available

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.

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.
- `stepper`: Step-by-step execution of all tasks on all cores.
- `hostguest`: Activity on vcpus + status of vcpus as running or waiting.

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.
- `stepper`: Step-by-step execution of all tasks on all cores.
- `hostguest`: Activity on vcpus + status of vcpus as running or waiting.

All publicly available.

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:

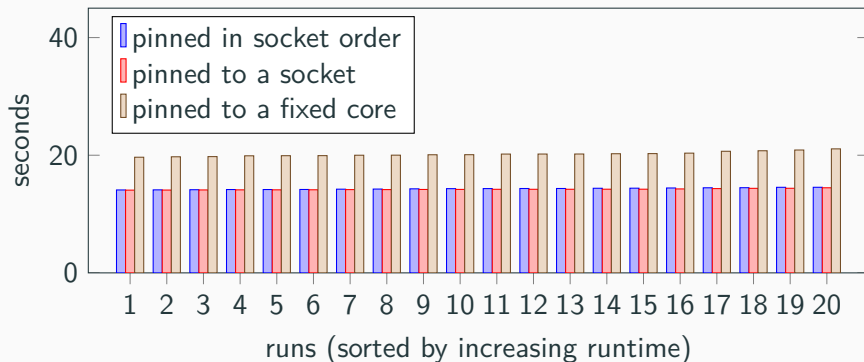
UA: “Unstructured Adaptive mesh, dynamic and irregular memory access”

- N tasks on N cores.

Getting to know the benchmark

A common approach for scientific code is to pin tasks to cores:

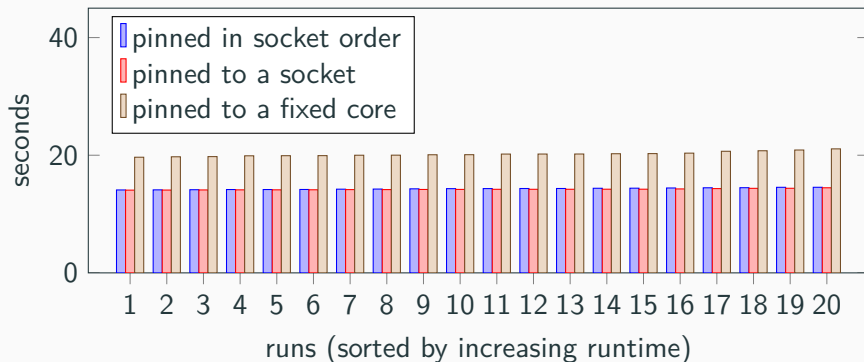
(All tests on a 4-socket machine with 128 hardware threads.)



Getting to know the benchmark

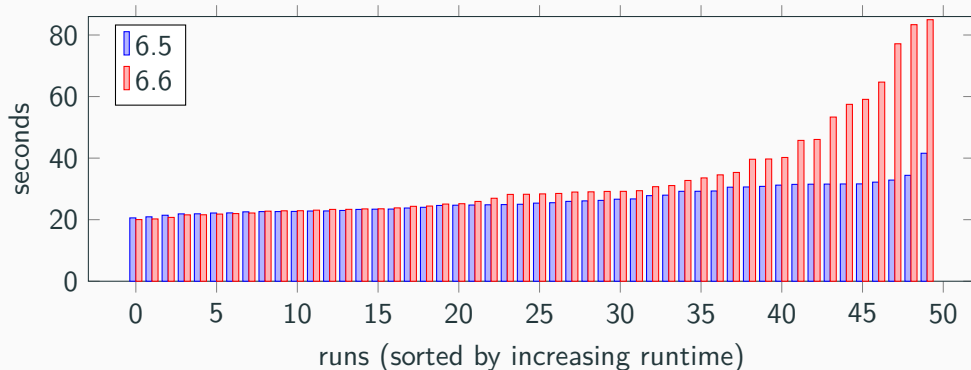
A common approach for scientific code is to pin tasks to cores:

(All tests on a 4-socket machine with 128 hardware threads.)

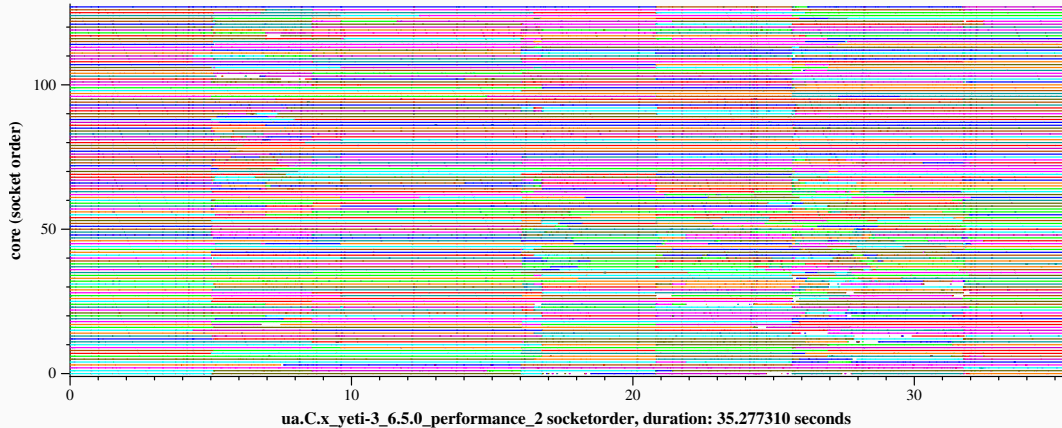


Suggests that memory locality impacts performance.

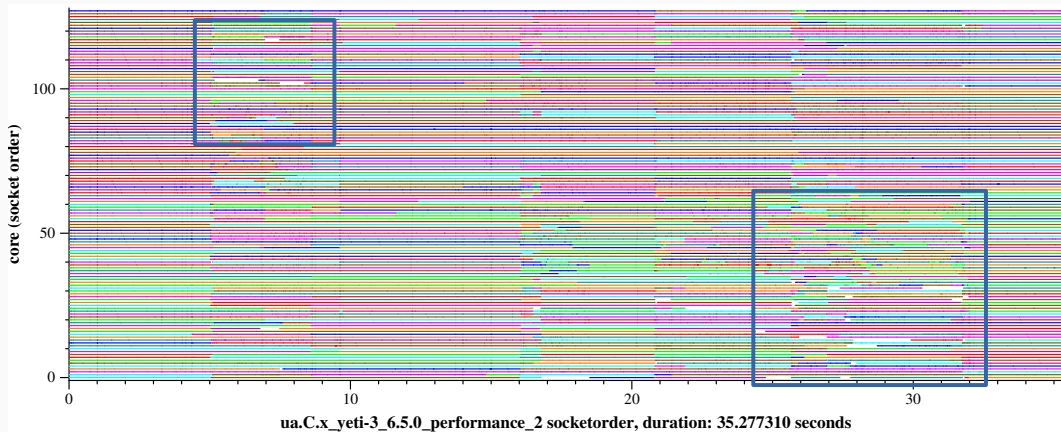
UA without pinning, before and after EEVDF (Linux v6.5 and v6.6)



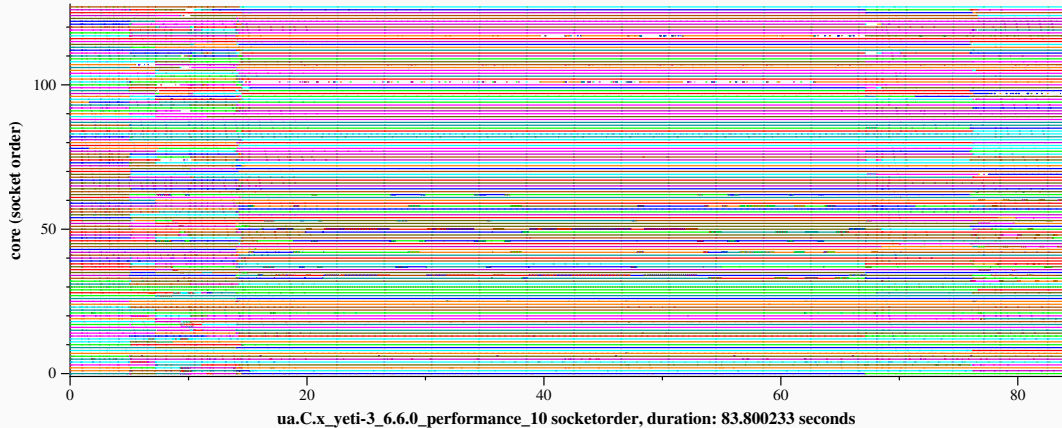
A slow run in v6.5 (dat2graph)



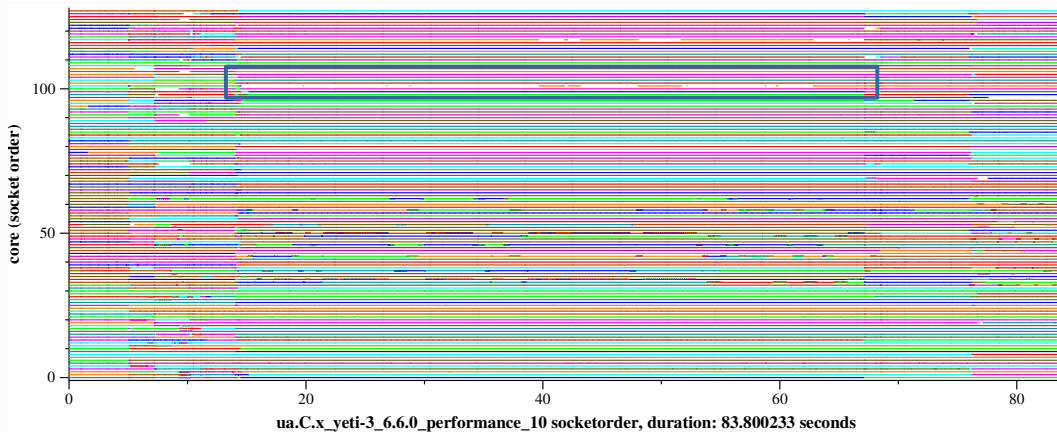
A slow run in v6.5 (dat2graph)



A slow run in v6.6 (dat2graph)



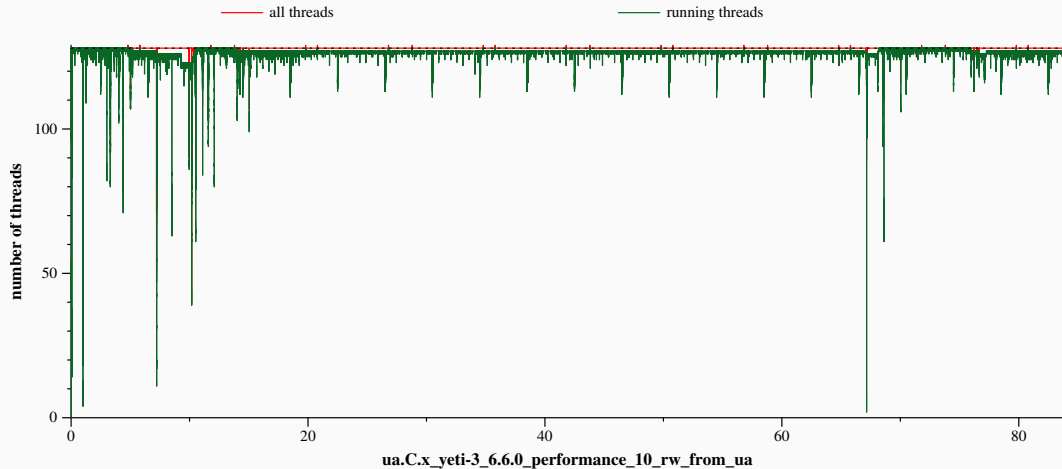
A slow run in v6.6 (dat2graph)



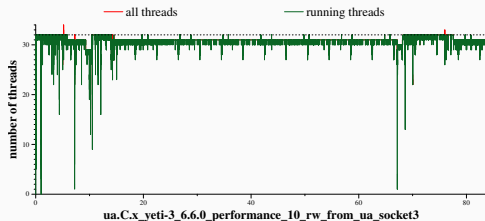
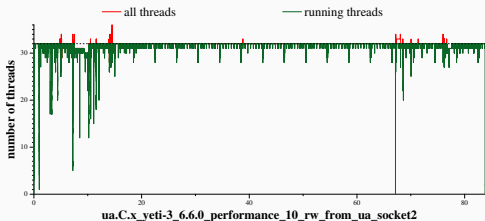
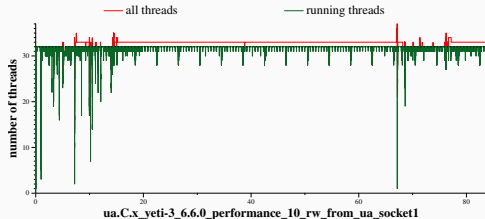
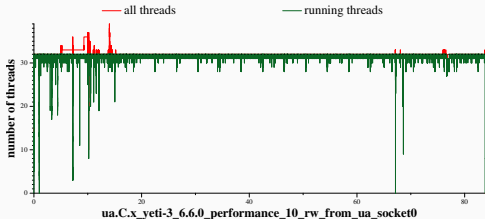
The impact of gaps depends on the reason why they are present:

- Tasks have nothing to do.
 - No performance impact.
- Some cores are **overloaded** while others are idle (work conservation issue).
 - Potential slowdown.

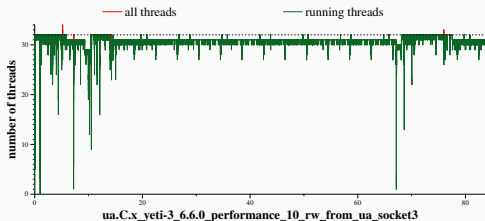
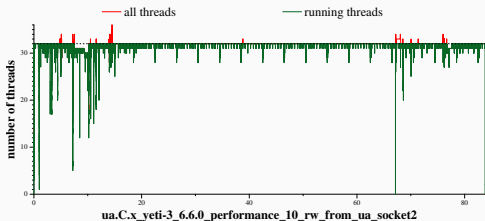
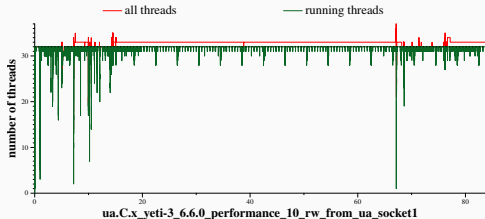
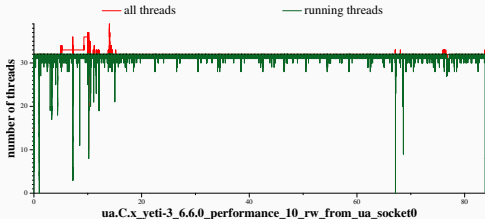
Detecting overloads: Focus on UA threads (running_waiting)



Detecting overloads: Focus on UA threads, by socket (running_waiting)

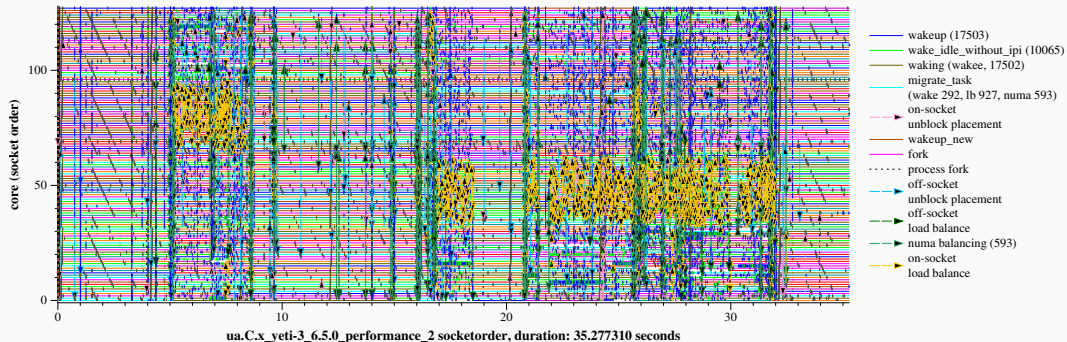


Detecting overloads: Focus on UA threads, by socket (running_waiting)

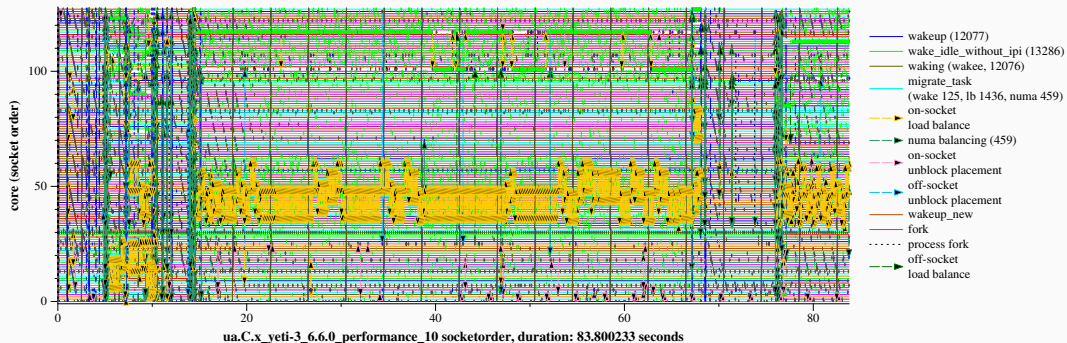


Maybe NUMA balancing is the culprit?

NUMA balancing in v6.5 (dat2graph --events)

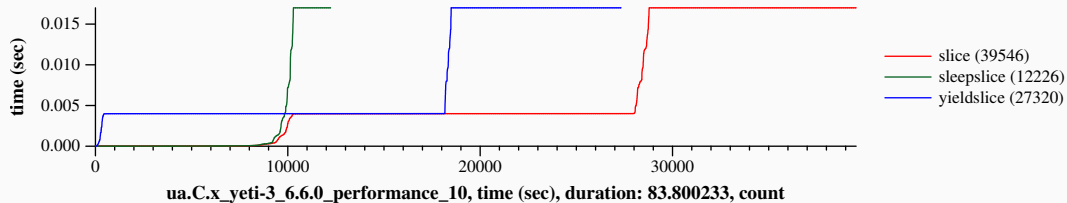
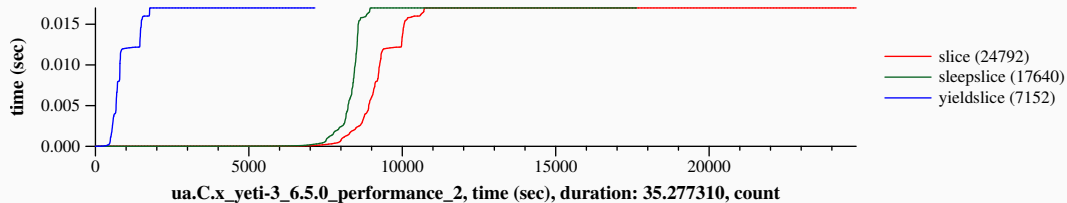


NUMA balancing in v6.6 (dat2graph --events)



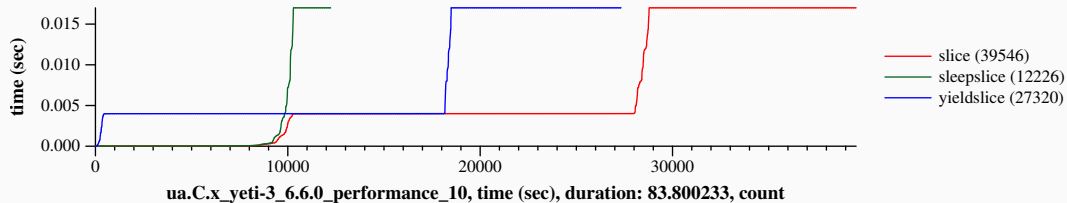
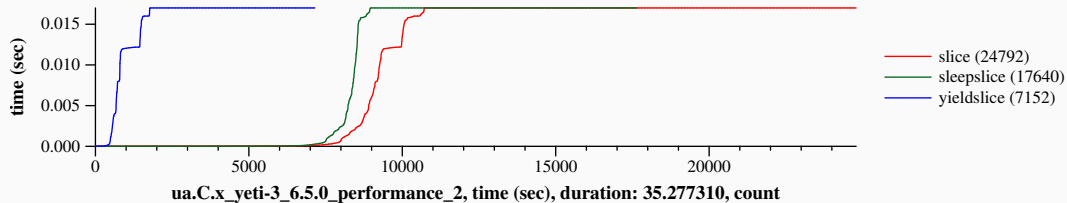
What's new with EEVDF?

Time slices are now mostly one tick.



What's new with EEVDF?

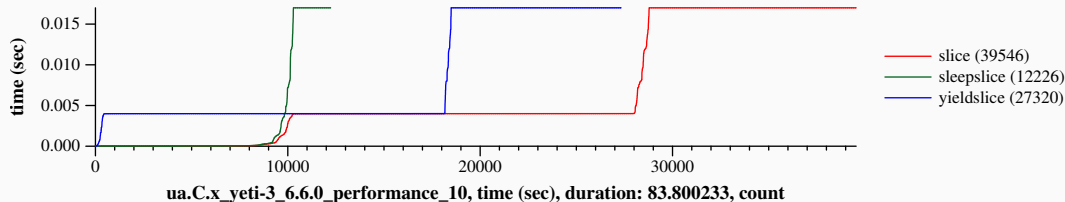
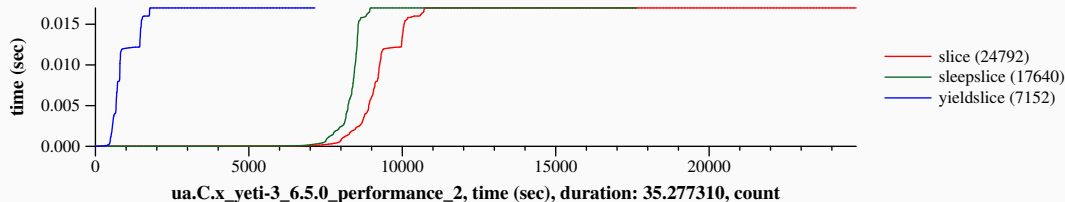
Time slices are now mostly one tick.



Does it matter that they are all one tick?

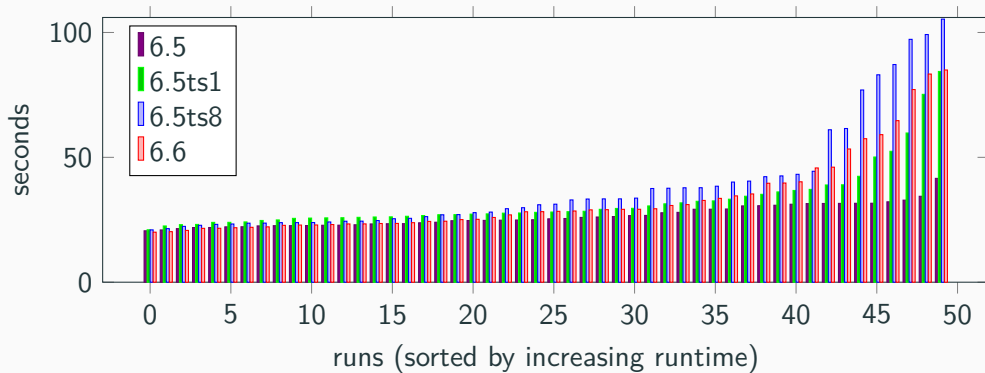
What's new with EEVDF?

Time slices are now mostly one tick.

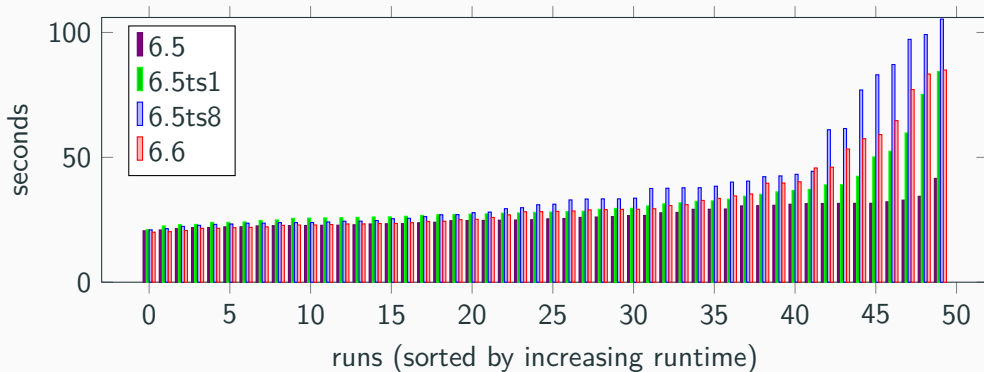


Does it matter that they are all one tick? Or that they are all the same?

v6.5, v6.5 with fixed time slices, and v6.6



v6.5, v6.5 with fixed time slices, and v6.6



And that's where the story ends...

- What goes wrong with UA?

- What goes wrong with UA?
- What information is useful to visualize?
- How to organize it?
- How to visualize new information (e.g., timeslice)?

A proposal

dat2graph:

An edge between task start and stop.

```
@dat2graph@
comm c1, c2, c3;
pid p1, p2 != 0, p3;
time t1, t2;
core c;
@@

sched_switch(c1,p1,_,c2,p2)@t1@c
...
sched_switch(c2,p2,_,c3,p3)@t2@c
==> edge(point(t1,c),point(t2,c),color(p2))
```


A proposal

dat2graph:

An edge between task start and stop.

```
@dat2graph@
comm c1, c2, c3;
pid p1, p2 != 0, p3;
time t1, t2;
core c;
@@

sched_switch(c1,p1,_,c2,p2)@t1@c
...
sched_switch(c2,p2,_,c3,p3)@t2@c
==> edge(point(t1,c),point(t2,c),color(p2))
```

timeslice:

Collect time between task start and stop.

```
@timeslice@
comm c1, c2, c3;
pid p1, p2 != 0, p3;
time t1, t2;
core c;
reason r;
@@

sched_switch(c1,p1,_,c2,p2)@t1@c
...
sched_switch(c2,p2,r,c3,p3)@t2@c
==> collect(r, t2 - t1)
```

A proposal

running_waiting: Keep a count of running tasks.

```
@running_waiting@
```

```
comm c; pid p; time t;
```

```
@@
```

```
sched_switch(_,0,_,c,p)@t
```

```
=> running[t] = ++running_count
```

```
@running_waiting@
```

```
comm c; pid p; time t;
```

```
@@
```

```
sched_switch(c,p,_,_,0)@t
```

```
=> running[t] = --running_count
```

A proposal

running_waiting: Keep a count of running tasks.

```
@running_waiting@
```

```
comm c; pid p; time t;
```

```
@@
```

```
sched_switch(_,0,_,c,p)@t
```

```
=> running[t] = ++running_count
```

```
@running_waiting@
```

```
comm c; pid p; time t;
```

```
@@
```

```
sched_switch(c,p,_,_,0)@t
```

```
=> running[t] = --running_count
```

Waiting is more complex, because a task can be waiting for multiple reasons.

An alternative

Focus on task states, rather than events.

```
@running_waiting@
```

```
comm c; pid p != 0; time t;
```

```
@@
```

```
In(running(c,p))@t
```

```
    waiting[t] = ++waiting_count;
```

```
@running_waiting@
```

```
comm c; pid p != 0; time t;
```

```
@@
```

```
Out(running(c,p))@t
```

```
    waiting[t] = --waiting_count;
```

```
@running_waiting@
```

```
comm c; pid p; time t;
```

```
@@
```

```
In(waiting(c,p))@t
```

```
@running_waiting@
```

```
comm c; pid p; time t;
```

```
@@
```

```
Out(waiting(c,p))@t
```

- `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.
- `stepper`: Step-by-step execution of all tasks on all cores.
- `hostguest`: Activity on vcpus + status of vcpus as running or waiting.

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