Graphing Tools for Tracing Task Schedulers: The Quest for a DSL

Julia Lawall, Inria

April 5, 2023

What is a task scheduler?

- Places tasks on cores at task 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?

- Places tasks on cores at task 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 the performance of an application?

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

How can a task scheduler impact the performance of an application?

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

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`: What is running on each core, at each time.
- `running_waiting`: How many tasks are running or waiting, at each time.

Our tools

- `dat2graph`: What is running on each core, at each time.
- `running_waiting`: How many tasks are running or waiting, at each time.
- Lots of other special-purpose things... (hence DSL potential).

Example of debugging a scheduling problem

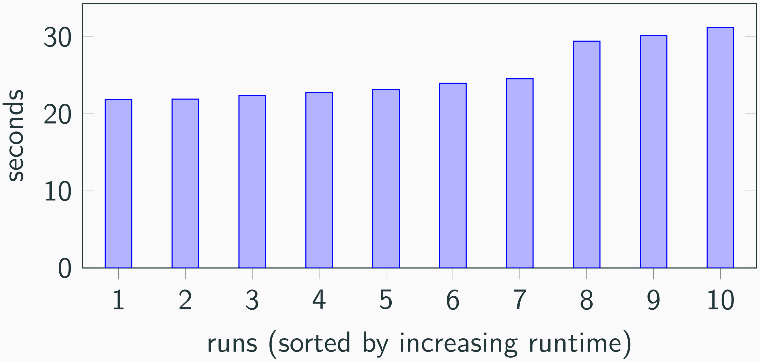
NAS benchmark suite: “The NAS Parallel Benchmarks (NPB) are a small set of programs designed to help evaluate the performance of parallel supercomputers. The benchmarks are derived from computational fluid dynamics (CFD) applications...”

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

- N tasks on N cores.

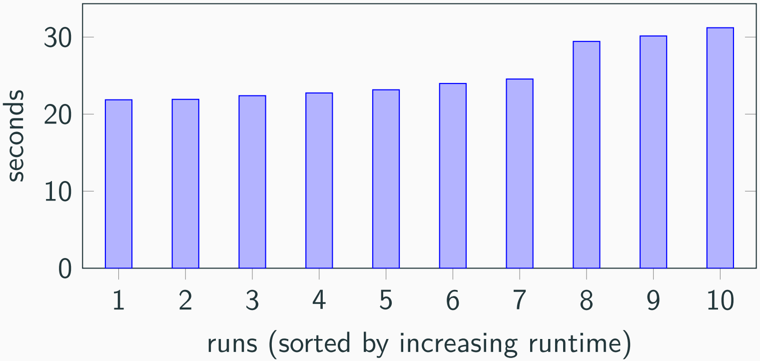
UA runtimes

4-socket, 128 core, Intel 6130.



UA runtimes

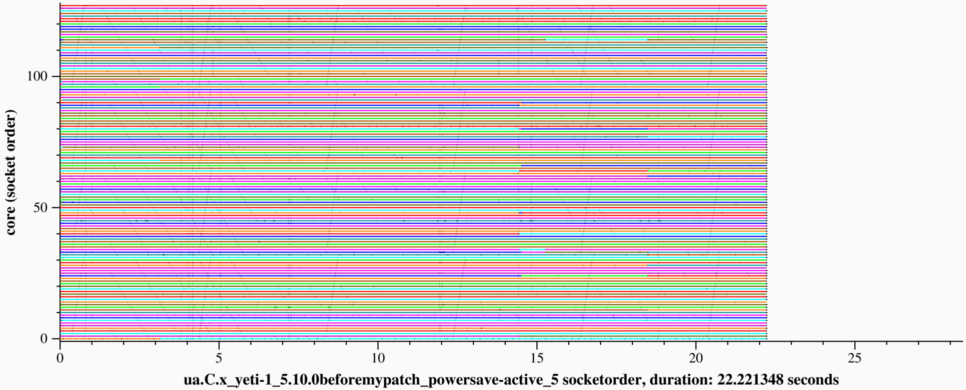
4-socket, 128 core, Intel 6130.



Why so much variation?

UA with dat2graph

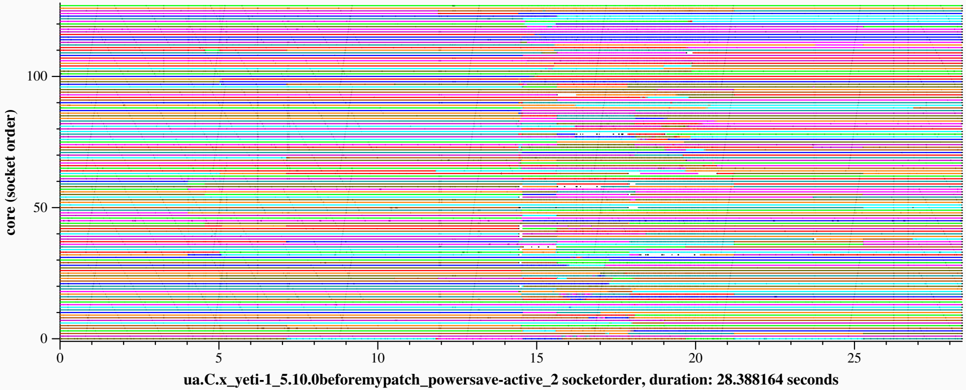
A fast run (`dat2graph --socket-order ua..._5.dat`).



Colored horizontal lines indicate running UA tasks. Colors chosen by pids.

UA with dat2graph

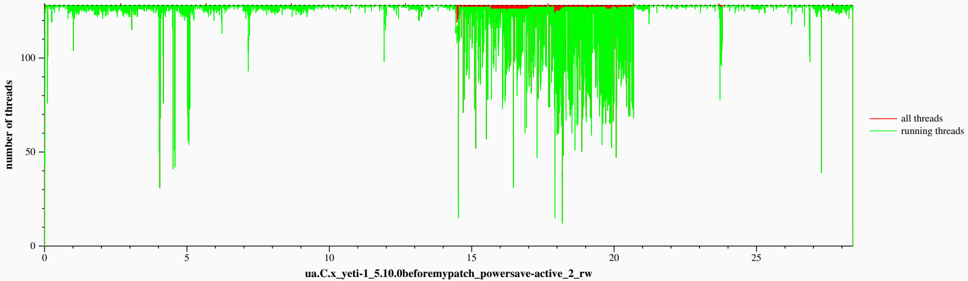
A slow run (`dat2graph --socket-order ua..._2.dat`).



White gaps indicate idleness.

UA with running_waiting

Another perspective on the slow run.

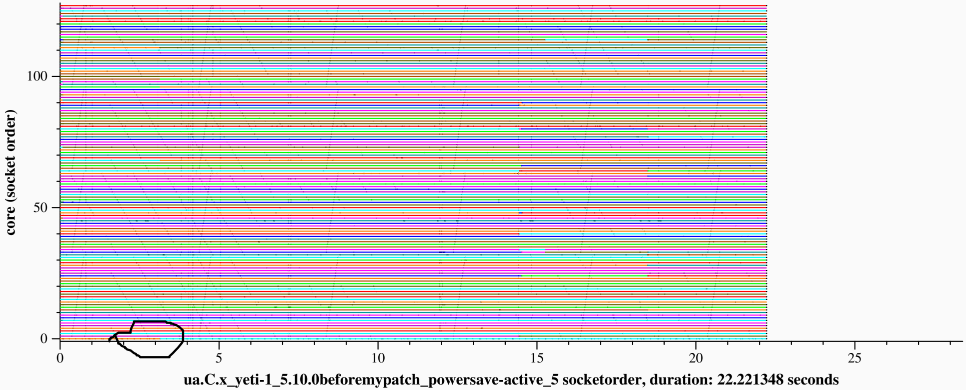


The height of the green line is the number of running tasks.

The height delta of the red line indicates the number of waiting tasks (overload).

The fast run revisited

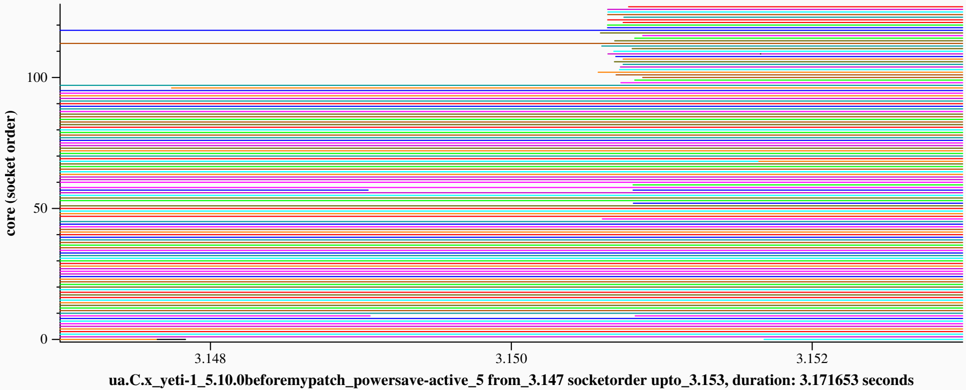
Tasks move around sometimes, for example around 3 seconds:



Change of color indicates a context switch.

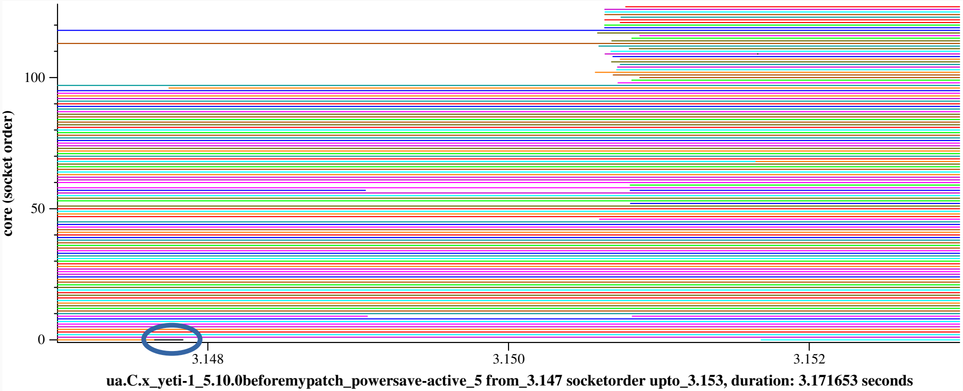
Zooming in

`dat2graph --target ua --min 3.147 --max 3.153 ... ua...dat`



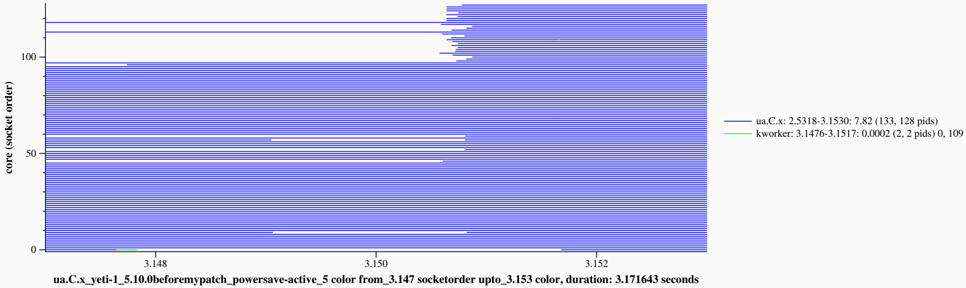
Zooming in

```
dat2graph --target ua --min 3.147 --max 3.153 ... ua...dat
```



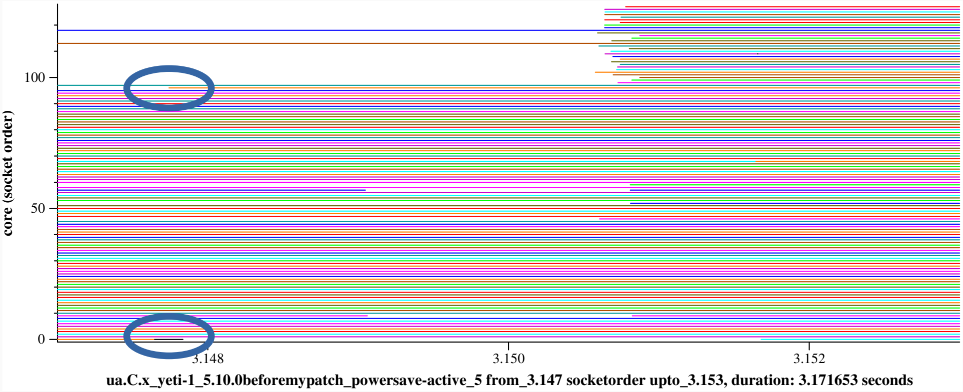
Color by command: understanding the black line

`dat2graph --socket-order --min 3.147 --max 3.153 --color-by-command ua...dat`



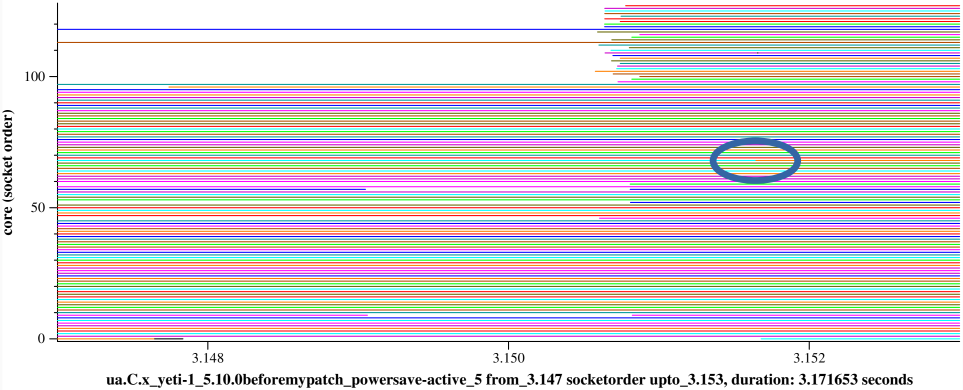
Conclusion: Load balancing

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

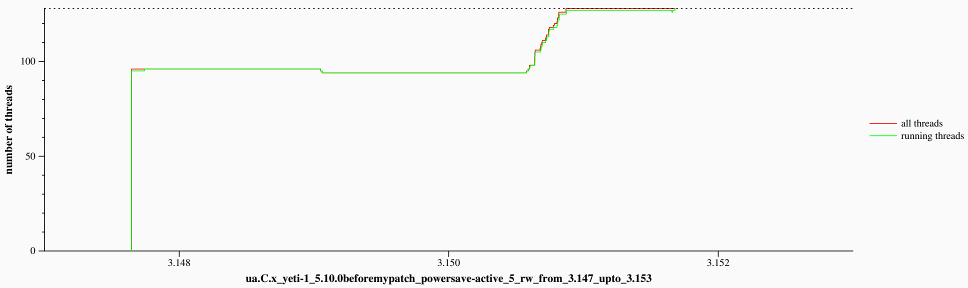


Another anomaly

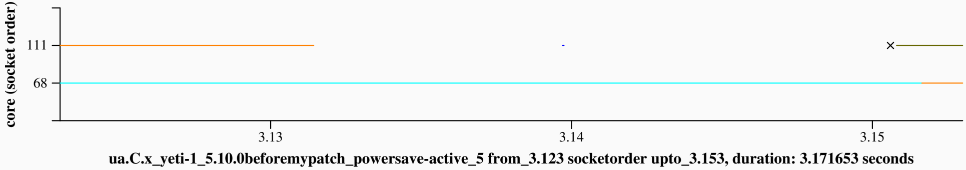
UA-UA overload (no black line)



Running-waiting view



Understanding the source of the overload



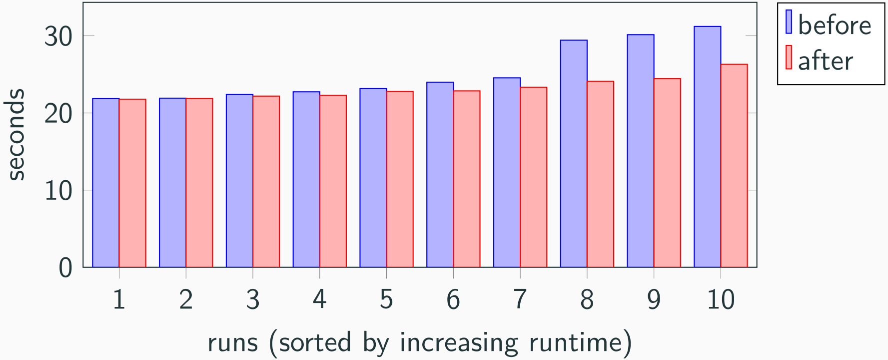
- 12655 on core 68 wakes 12549 for core 111 (different sockets)
- CFS first chooses a “target”, between the previous core and the waker core.
- 68 is chosen, due to the recent activity on 111.
- There are no idle cores on the socket of 68, resulting in an overload.

A patch

```diff
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



Assessment

Multiple kinds of graphs were useful to understand the problem:

- dat2graph: Which task is running, when, on what core?
- dat2graph –color-by command:
  Which application is running, when, on what core?
- running_waiting: How many tasks are running and waiting?

More complex options:

- What is the frequency of each core, and what application is currently running at that frequency?

Original implementation

```
match l with
  Parse_line.Sched_switch(fromcmd,frompid,reason,tocmd,topid) ->
    (if not !fast_freq && tracking frompid fromcmd requested_pids time firstmatch
    then
      switchfrom base inkvm index time core frompid fromcmd corestate
        freqtrace hoststate pending mapping
        startpoint);
    (if not !fast_freq && tracking topid tocmd requested_pids time firstmatch
    then
      switchto inkvm index time core topid tocmd corestate
        freqstate freqtrace hoststate pending mapping
        first_appearance startpoint);
| Parse_line.Sched_wakeup(cmd,pid,prevcpu,cpu)    -> ...
| Parse_line.Sched_wakeup_new(cmd,pid,parent,cpu) -> ...
| Parse_line.Sched_process_exec(cmd,oldcmd,pid,oldpid) ->
    (if tracking oldpid oldcmd requested_pids time firstmatch
    then (* pid as is before exec *)
      switchfrom base inkvm index time core oldpid oldcmd corestate
        freqtrace hoststate pending mapping startpoint);
    (if !forked
    then Hashtbl.add requested_pids pid ()
    else pid_transition oldpid cmd pid requested_pids);
    if tracking pid cmd requested_pids time firstmatch
    then
      switchto inkvm index time core pid cmd corestate
        freqstate freqtrace hoststate pending mapping
        first_appearance startpoint
```

Issues

Code duplication due to similar events:

- sched_switch vs. sched_process_exec
- sched_wakeup vs. sched_wakeup_new

Issues

Code duplication due to similar events:

- sched_switch vs. sched_process_exec
- sched_wakeup vs. sched_wakeup_new

Many data structures:

- Data structures to record the current state.
- Data structures to collect historical traces.

Issues

Code duplication due to similar events:

- sched_switch vs. sched_process_exec
- sched_wakeup vs. sched_wakeup_new

Many data structures:

- Data structures to record the current state.
- Data structures to collect historical traces.

Difficult to customize for specific purposes.

Towards a DSL...

Initial steps

Libraries

- Shared parser, shared graph printer.
- Shared utilities.

Initial steps

Libraries

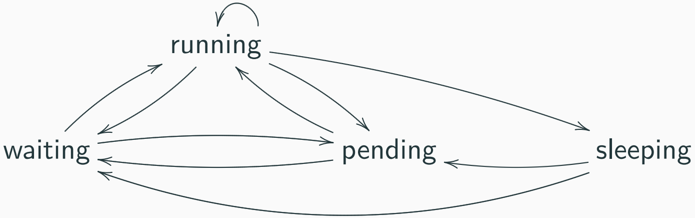
- Shared parser, shared graph printer.
- Shared utilities.

To use the libraries

- Copy-pasteable typical implementation

A small insight

Mostly, we care about task states, not about state transitions



Handlers for entering or leaving a state.

Simplified dat2graph

```
let ops = Hashtbl.create 7

let init_ops running_trace =

  let starting time core pid = do_open running_trace time core pid in

  let ending time core pid   = do_close running_trace time core pid in

  Hashtbl.add ops (MR.Src MR.Running) ending;
  Hashtbl.add ops (MR.Dst MR.Running) starting
```

Simplified overload counter

```
let ovd_ops = Hashtbl.create 7

let overload_init_ops overload (_,(_,wait),_) =
  let find trace core = try Hashtbl.find trace core with _ -> [] in

  let starting time core pid =
    let hostcore = Array.get mapping core in
    let n =
      match find overload hostcore with
        Open(t1,v) :: _ -> v
      | _ -> List.length (Array.get wait core) - 1 in
    do_close overload time hostcore n;
    do_open overload time hostcore (n+1) in

  let ending time core pid =
    let hostcore = Array.get mapping core in
    let n =
      match find overload hostcore with
        Open(t1,v) :: _ -> v
      | _ -> List.length (Array.get wait core) + 1 in
    do_close overload time hostcore n;
    do_open overload time hostcore (n-1) in

  Hashtbl.add ovd_ops (MR.Src MR.Waiting) ending;
  Hashtbl.add ovd_ops (MR.Dst MR.Waiting) starting
```

An idea for a DSL

```
Edge on --exec {
        pid in running -> color(pid) @ pid.core
}

Edge on --color-by-commmand {
        pid in running -> color(pid.cmd) @ pid.core
}

Edge on --sockets {
        pid in running -> target(pid.cmd) -> color(socket(pid.core)) @ pid
}

Edge on --mfreq {
        pid in running ->
           print in "arch_scale_freq_tick: freq %d" ->
           pid.core = print.core ->
           color(print.$1) @ print.core @ 0
        pid in running -> color(pid) @ pid.core @ 1
}
```

Questions

How to find a syntax for such a DSL?

- Sufficiently expressive?
- Sufficiently user friendly?

How to increase expressivity?

- Reflection on events or internal data structures?
- Reflection on the underlying programming language?

Conclusion

- Understanding scheduler traces can be important to understanding application performance.
- Existing solutions are rigid and processing trace data is complex.
- Maybe a DSL can help...

Conclusion

- Understanding scheduler traces can be important to understanding application performance.
- Existing solutions are rigid and processing trace data is complex.
- Maybe a DSL can help...

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

An idea for a DSL

```
Line on --overload {
        pid in running v pid in waiting ->
           red @ sizeof(running) + sizeof(waiting) @ 0
        pid in waiting -> green @ sizeof(waiting) @ 1
```