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

core 0  core 1  core 2  core 3

Where to put waking task T1?

• Maybe anywhere is fine...

Core 0 would not be a good choice.

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

Locality:
Issues: Work conservation

The machine

| core 0 | core 1 | core 2 | core 3 |

Where to put waking task $T_1$?

- Maybe anywhere is fine...
- Core 0 would not be a good choice.
- Work conservation: No core should be overloaded if any core is idle.

Locality: 4
Issues: Work conservation

The machine

<table>
<thead>
<tr>
<th>core 0</th>
<th>core 1</th>
<th>core 2</th>
<th>core 3</th>
</tr>
</thead>
</table>

Where to put waking task $T_1$?

- Maybe anywhere is fine...
**Where to put waking task T1?**

- Maybe anywhere is fine...
### Issues: Work conservation

The machine

<table>
<thead>
<tr>
<th>core 0</th>
<th>core 1</th>
<th>core 2</th>
<th>core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>T1</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Where to put waking task **T2**?
Issues: Work conservation

The machine

<table>
<thead>
<tr>
<th></th>
<th>core 0</th>
<th>core 1</th>
<th>core 2</th>
<th>core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Where to put waking task T2?

- Core 1, core 2, or core 3 might be fine.
- Core 0 would not be a good choice.
Where to put waking task $T_2$?

- 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

<table>
<thead>
<tr>
<th>core 0</th>
<th>core 1</th>
<th>core 2</th>
<th>core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Where to put waking task?

- 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.

Locality: Tasks should be near their data.
Locality: Tasks sharing data should be on the same socket.
# Issues: Locality

A two-socket machine

<table>
<thead>
<tr>
<th>core 0</th>
<th>core 1</th>
<th>core 2</th>
<th>core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Where to put waking task \textbf{T2}?

- Core 1 is good if \textbf{T2} has previously allocated memory on that socket.
- Core 1 is good if \textbf{T2} communicates a lot with \textbf{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?
trace-cmd: Collects ftrace information, including scheduling events.

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

Sample trace:

<table>
<thead>
<tr>
<th>Process</th>
<th>Event</th>
<th>Time</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1 CompilerThre-166659 [026]</td>
<td>sched_wakeup: C1 CompilerThre:166654 [120] success=1 CPU:062</td>
<td>9539.524366</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;idle&gt;-0 [062]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C1 CompilerThre-166659 [026]</td>
<td>sched_switch: swapper/62:0 [120] R ==&gt; C1 CompilerThre:166654 [120]</td>
<td>9539.524369</td>
<td></td>
</tr>
<tr>
<td>java-166654 [062]</td>
<td>sched_waking: comm=C1 CompilerThre pid=166660 prio=120 target_cpu=028</td>
<td>9539.524372</td>
<td></td>
</tr>
</tbody>
</table>
Some help available

**kernelshark**: Graphical front end for *trace-cmd* data.
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 d8fc81f1ac651a0e50eacecca43d0524984f87
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;
}
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.

Why so much variation?
UA runtimes prior to my patch

4-socket, 128 core, Intel 6130.

Why so much variation?
A fast run (`dat2graph2 --socket-order ua..._5.dat`).
A slow run (dat2graph2 --socket-order ua..._2.dat).
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:

![Graph showing tasks moving around](image-url)
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

What are the black lines?

ua.C.x_yeti-1_5.10.0beforemypatch_powersave-active_5 from_3.0755 socketorder upto_3.0775, duration: 3.092353 seconds

core (socket order)

0 50 100

3.076 3.077

ua.C.x_yeti-1_5.10.0beforemypatch_powersave-active_5 from_3.0755 socketorder upto_3.0775, duration: 3.092353 seconds
Focusing on the first gap

What are the black lines?

ua.C.x_yeti-1_5.10.0beforemypatch_powersave-active_5 from_3.0755 socketorder upto_3.0775, duration: 3.092353 seconds
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

ua.C.x: 1.6038-3.0765: 72.40 (134, 128 pids)
kcompactdX: 3.0757-3.0757: 0.0001 (4, 4 pids)
kworker: 3.0757-3.0757: 0.0000 (2, 2 pids) 47, 79
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.
• 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

ua.C.x_yeti-1_5.10.0beforemypatch_powersave-active_5 from_3.147 socketorder upto_3.153, duration: 3.171653 seconds
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)
Understanding the source of the overload

- 12655 on core 68 wakes 12549 for core 111
• 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).
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

runs (sorted by increasing runtime)

seconds

before
after
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