Graphing Tools for Scheduler Tracing

Julia Lawall, Himadri Chhaya-Shailesh, Jean-Pierre Lozi, Gilles Muller

Graphing Tools for Scheduler Tracing

Julia Lawall, Himadri Chhaya-Shailesh, Jean-Pierre Lozi, Gilles Muller

RESEARCH REPORT

N° 9498

February 2023

Project-Team Whisper

ISSN 0249-6399 ISRN INRIA/RR--9498--FR+ENG

Graphing Tools for Scheduler Tracing

Julia Lawall,* Himadri Chhaya-Shailesh,* Jean-Pierre Lozi,*
Gilles Muller*

Project-Team Whisper

Research Report n° 9498 — February 2023 — 15 pages

Abstract: Understanding task-scheduler behavior can be important for understanding application performance. In this paper, we present some tools that we have developed to help understand task-scheduler behavior on highly multicore machines. Building on traces collected with `trace-cmd`, these tools produce graphs showing what tasks are running on what cores and showing overload situations. The tools thus make it possible to get an overview of the application execution, as well as to study specific execution intervals.

Key-words: Linux kernel, scheduling, trace-cmd

* Inria Paris, 2 rue Simone IFF, CS 42112 75589 Paris CEDEX 12, France

RESEARCH CENTRE
PARIS

2 rue Simone Iff - CS 42112
75589 Paris Cedex 12

Graphing Tools for Scheduler Tracing

Résumé : Comprendre le comportement d'un ordonnanceur des tâches peut être important pour comprendre les performances des applications. Dans cet article, nous présentons quelques outils que nous avons developpés pour aider à comprendre le comportement d'un ordonnanceur dans un environnement hautement multicœurs. S'appuyant sur les traces collectées avec `trace-cmd`, ces outils produisent des graphiques montrant quelles tâches s'exécutent sur quels cœurs et des situations de surcharge. Les outils permettent d'avoir une vue d'ensemble de l'exécution de l'application, et d'étudier l'exécution des intervalles spécifiques.

Mots-clés : Noyau Linux, ordonnancement, trace-cmd

Contents

1 Introduction

A task scheduler is the part of an operating system that places tasks on cores when a new task is created, when a task wakes up, or during load balancing. The task scheduler also selects the next task to run on a core when a core becomes idle. In terms of the Linux kernel, we are concerned with the implementation of the completely fair scheduler (CFS), for which the main files are `kernel/sched/core.c` and `kernel/sched/fair.c`. In this work, we focus on task placement. In practice, task placement can have a substantial impact on application performance. Poor placement choices can slow tasks down, and this slowdown can have a domino effect, when a delay in one task causes delays in other tasks. Accordingly, to understand the performance of an application that involves multiple tasks it is often essential to understand the task scheduler's behavior on the application.

This paper presents the tools `dat2graph` and `running_waiting` that we have designed and used extensively to understand task scheduler behavior. Both tools rely on traces created using `trace-cmd` [6], a front-end for `ftrace` [5] provided by the Linux kernel community for collecting and visualizing execution traces. `dat2graph` and `running_waiting` provide, respectively, a concise visualization of task activity on cores suitable for understanding scheduler behavior on large multicore machines, and a summary of the number of tasks that are running and that are waiting on runqueues, to highlight the occurrences of overloads. These tools produce `pdf` files that are easy to view, collect, compare, share, and publish. These tools were used in motivating and developing the Nest scheduler, published at EuroSys 2022 [4].

The rest of this paper is organized as follows. Section 2 describes some issues that can arise in scheduling that have an impact on application performance and that motivate the need for detailed tracing and visualization of scheduler behavior. Section 2 also presents the tools `trace-cmd` and `kernelshark` [3] used by the Linux kernel developer community for collecting and visualizing event traces. Section 3 presents `dat2graph` and `running_waiting` through the motivating example of a performance issue in the execution of the NAS benchmark suite [1]. Section 4 briefly describes the implementation of `dat2graph` and `running_waiting`, and provide links for the various software dependencies. Section 5 then concludes. The primary purpose of this work is to serve as a reference for those who would like to use `dat2graph` and `running_waiting`, and thus we do not survey other tools for visualizing scheduler behavior.

Terminology Modern servers offer *hyperthreading*, allowing one physical core to run multiple instruction streams (hardware threads). Following the strategy used in the Linux kernel scheduler, we refer to each hardware thread independently as a *core*.

2 Background

We first present some scheduling issues that can affect application performance. We then present some tools that are commonly used in the Linux kernel community to understand scheduler behavior.

2.1 Scheduling issues

Work conservation Work conservation is the property that no core should be overloaded if any core is idle. Put another way, when a task becomes ready to run, either because it was just created or because it has been woken up, the scheduler should favor placing the task on an idle core, rather than on a core where another task is already running. While work conservation may seem obviously desirable, achieving it is challenging for a decentralized scheduler, such CFS,

```
 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
```

Figure 1: Extract of a scheduling trace captured with `trace-cmd`

as it requires probing other cores to see whether they are idle. Accordingly, CFS is not work conserving.

Locality Locality is the property that a task should run close to its data. Locality particularly has an impact on non-uniform memory access (NUMA) machines, where cores have faster access to local memory and slower access to memory on other NUMA nodes. Work conservation can conflict with locality: placing a task on a core that is currently idle but that is on a different NUMA node than the one containing the data that the task commonly accesses can slow down the application in the long term.

Core frequency The core frequency determines how fast instructions run on a core, and thus strongly determines application performance for CPU-bound tasks. Modern servers allow each physical core to have its own frequency. Modern Intel and AMD servers furthermore offer *turbo* frequencies, in which cores can run at a frequency higher than the nominal frequency, subject to thermal constraints. Consolidating tasks on a smaller number of cores, if this can be done without introducing overloads, allows the use of higher turbo frequencies, and thus enables applications to run faster [4].

2.2 `trace-cmd` and `kernelshark`

`Trace-cmd` [6] is a front end for `ftrace` [5] that allows collecting and displaying traces of various kinds of Linux kernel activity. A typical command line for tracing scheduler events is as follows:

```
trace-cmd -e sched -v -e sched_stat_runtime -q \
  -o trace.dat ./mycommand arg1 arg2 ...
```

`-e sched` indicates that all scheduling events should be traced. `-v -e sched_stat_runtime` indicates that statistics-collecting events should be ignored, as they are numerous and not relevant for understanding why tasks are placed on the various cores. `-q` requests quiet output. `-o trace.dat` indicates the name of the output file (`trace.dat` is the default). The rest of the line provides the command to run and its arguments. `trace-cmd` must be run as root.

The information in a `trace.dat` file can be extracted as

```
trace-cmd report trace.dat
```

A small extract of a typical trace is shown in Figure 1. Such a trace gives complete information about the scheduling decisions made by the kernel – what happens, at what time, involving what tasks, etc. Still, there are typically a huge number of events. The events are ordered chronologically, which can make it difficult to manually follow what each core or task is doing.

To help understand these traces, `kernelshark` [3] was developed. `Kernelshark` offers a graphical user interface with some bars on the top of the screen representing the cores and the tasks that they are running, and the textual trace below (Figure 2). It also offers the option to reorient the bars so that each bar represents a task, and then shows the cores on which the task is running. `Kernelshark` makes it easy to zoom in to a region of time, or to jump to specific events using the textual event trace. Still, the fairly spacious layout means that it can be hard to get a complete overview of the scheduler activity on a large multicore machine.

Figure 2: Kernelshark screenshot

3 Approach

We now present `dat2graph` and `running_waiting` and illustrate their use. As a motivating example, we take the execution of the UA benchmark of the NAS parallel benchmark suite [1]. The NAS parallel benchmark is composed of various kernels of scientific applications. The UA benchmark is described by the NAS parallel benchmark website as "Unstructured Adaptive mesh, dynamic and irregular memory access."[1] We use the `C` (mid-sized) dataset. For our purposes, the important features of UA are that the benchmark uses N tasks on N cores and involves frequent synchronization. Our tests are performed on a 128-core four-socket Intel 6130 server, using the powersave-active power governor. This work led us to propose Linux kernel commit `d8fcb81f1acf`, which first appeared in mainline release v5.11. Our experiments use the snapshot of the kernel just prior to the integration of commit `d8fcb81f1acf`, obtained with `git checkout d8fcb81f1acf^`.

Given that UA involves N tasks on N cores, it should be possible to run the benchmark such that each task starts running on its own core, and remains on that core throughout its execution. Indeed, one can pin the tasks to cores to achieve this effect. Our investigation using `dat2graph` and `running_waiting` helps understand why execution with the Linux kernel CFS scheduler has a different behavior.

3.1 Initial experiments

Our first experiment is to run the UA benchmark 10 times. The obtained running times, sorted by increasing running time, are shown in Figure 3. The longest running time is more than 40% longer than the shortest. In particular, the three running times at the far right are significantly longer than the others.

Using `dat2graph`, we can start to see what happens. We first use `trace-cmd` to make traces of 5 additional runs. Here too there are performance variations. We then use `dat2graph` to visualize the fastest and slowest runs, which are 2 and 5, using the following command line:

```
dat2graph --after-sleep --socket-order --forked \
  ua.C.x_yeti-1_5.10.0beforemypatch_2.dat \
  ua.C.x_yeti-1_5.10.0beforemypatch_5.dat
```

The argument `--after-sleep` is used because we have added a sleep of one second before running the application, to avoid interference between the setup of `trace-cmd` and the application execution. `dat2graph` skips the sleep portion of the trace. The argument `--socket-order` is used to order the cores such that cores on the same socket are adjacent, rather than using the

[1] https://www.nas.nasa.gov/software/npb.html

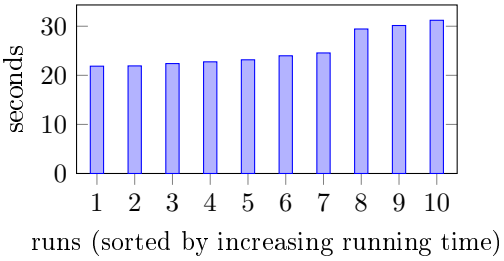

Figure 3: Running time of UA

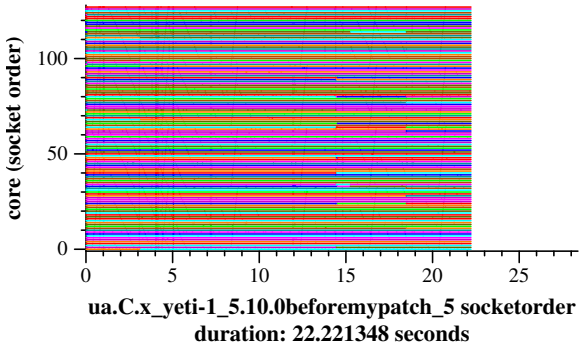

Figure 4: Trace of a fast UA run

round-robin numbering used by Intel servers.[2] The argument `--forked` indicates that only tasks created after the start of the trace, *i.e.*, the application tasks, should be included, for readability. The remaining arguments are the trace files to visualize. Multiple trace files can be provided on the command line, in which case all graphs are scaled to the running time of the slowest run.

The graph for the fastest run is shown in Figure 4.[3] In this graph, the x-axis is the elapsed time in seconds, the y-axis is the cores, and the colors represent (non uniquely) the PIDs. A line starts when a task starts running, and ends when the task sleeps or is preempted. Mostly tasks stay on the same cores and use all the cores all the time. There are, however, a few points at which tasks move around, near 3 seconds and near 15 seconds.

The graph for the slowest run is shown in Figure 5. In contrast to the graph in Figure 4, tasks move around more, perhaps losing locality. Even worse, there are gaps, particularly between 15 and 20 seconds, where one or more cores are doing nothing for long periods of time.

Gaps in the execution can arise because tasks have nothing to do, *i.e.*, if one task is ahead of the others and is waiting for the others to complete their work. Gaps can also arise when there are overloads, *i.e.*, multiple tasks have been placed on the same core, where they have to wait until CFS preempts the running task, and other cores lie empty. The graph produced by `dat2graph` shows only running tasks, not waiting ones, and thus is not sufficient to distinguish between these cases. We have experimented with graphs that show how many runnable tasks

[2] Using the `--socket-order` argument requires modifying the function `reorder` of `dat2graph2.ml` to reflect the topology of the machine.

[3] All graphs are slightly touched up to fit the two-column format.

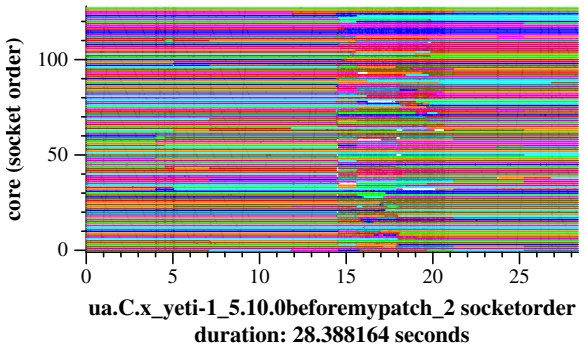

Figure 5: Trace of a slow UA run

there are on each core. Nevertheless, in understanding traces on large machines and where such overloads are fairly rare, we have found it most useful to just have a summary of the number of running and waiting tasks.

Such a summary is produced by `running_waiting`, with the following command line:

```
running_waiting --forked --after-sleep \
  ua.C.x_yeti-1_5.10.0beforemypatch_2.dat
```

The argument `--forked` limits the graph to the tasks created by the application. The argument `--after-sleep` has the same meaning as for `dat2graph`. The remaining argument is the trace file of interest. Multiple trace files can be provided.

The graph produced by `running_waiting` for the slow run of UA is shown in Figure 6. The x-axis is the time and the y-axis is the number of tasks. The height of the green line shows the number of running tasks. The height of the red line shows the number of runnable tasks, *i.e.*, both the tasks that are running and the tasks that are waiting in run queues. If the red line is visible, *i.e.*, higher than the green line, some runnable task is waiting in a runqueue, *i.e.*, there is an overload. The black horizontal dotted line indicates the number of cores on the machine. If the red line is above the black dotted line, then there are more tasks than cores, and overloads are inevitable. If the red line is visible but below the black dotted line, it means that there is a work conservation problem; some cores are overloaded while other cores are idle. Figure 6 shows that UA in the slow run has a persistent failure of work conservation between 15 and 20 seconds.

3.2 Zooming in, step 1

To understand the scheduler behavior, we create new traces that focus on a particular region of the execution time. For this, we focus on the fast run. That run is more uniform, making it easier to identify the points at which there may be scheduling issues.

Figure 7 shows a trace focusing on the time from 3 to 3.2 seconds, made using the following command line:

```
dat2graph --after-sleep --socket-order --min 3 \
  --max 3.2 --target ua \
  ua.C.x_yeti-1_5.10.0beforemypatch_5.dat
```

The arguments `--min` and `--max` show the starting and ending times to be shown in the graph, respectively. The argument `--target ua` indicates that the lines corresponding to the UA ap-

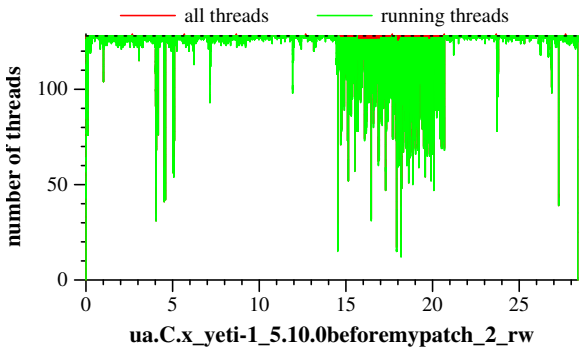

Figure 6: Tasks running and waiting in the slow UA run

plication (*i.e.*, having a command name starting with `ua`) should be colored according to the corresponding task's PID. The lines corresponding to other tasks are then colored black.

The zoomed graph shows that even the fast run has gaps. We study further the leftmost gap, which involves almost all of the cores, between 3.07 and 3.08 seconds. Figure 8 shows a further zoom into the trace. This zoom reveals a few scattered black lines, shortly before the gap. These are not the UA application. To find out what they represent, we can use the option `--color-by-command` of `dat2graph`. This option colors the lines according to the command name of the running task. The resulting graph is shown in Figure 9. The color-by-command graph also provides some information about the first and last point of execution of each command (1.6038-3.0765 for the command `ua.C.x`, representing the UA application; here, this value is truncated by the `--max` argument), the overall execution time of the command (72.40 seconds for `ua.C.x`), the number of context switches to the command (134 for `ua.C.x`), and the number of PIDs associated with the command (128 for `ua.C.x`).

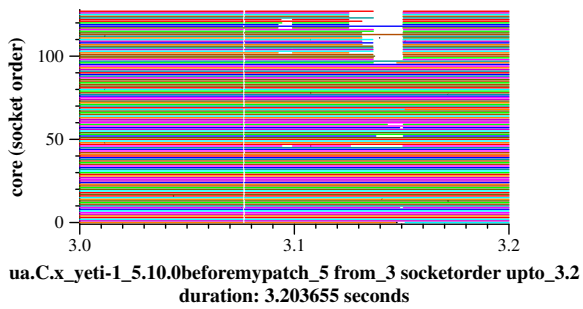

Figure 7: Zoom into the time around 3 sec. in the fast run

The color-by-command graph reveals that the black lines are some kernel daemons, `kcompactd` and `kworker`. These for example perform the computation associated with memory compaction, interrupts or other kernel functions. They have high priority and preempt the running task. This preemption causes some of the application tasks to fall behind, and then other tasks that depend on them fall behind as well. Most tasks finish at one point (the left side of the gap),

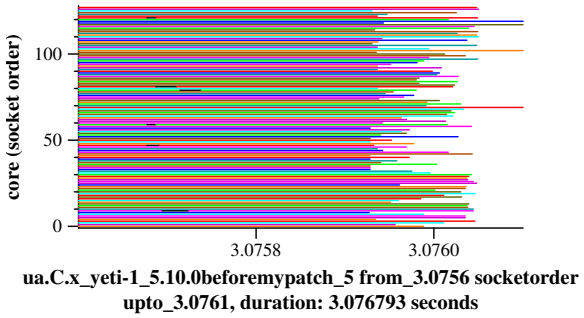

Figure 8: Zoom into the leftmost gap in the fast run

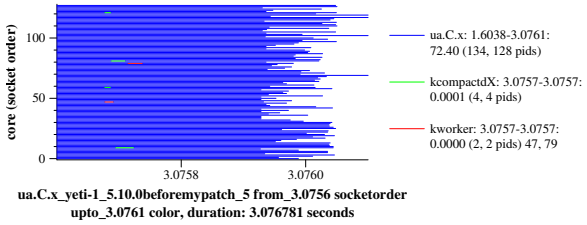

Figure 9: Zoom into the gap on almost all cores in the fast run, colored by the command name

but there are a few stragglers. Still, according to colors around the black lines on the left side of Figure 8, the application tasks that are preempted by the kernel daemons return to run on the same core when the kernel daemon completes, so once the tasks all complete their current work and resynchronize (*i.e.*, at the right side of the gap), the execution should continue normally.

3.3 Zooming in, step 2

We next consider a region slightly to the right of the previously considered gap, starting at around 3.148 seconds. The graph is shown in Figure 10. Here there are more gaps, in particular on the fourth socket, *i.e.*, in the topmost 32 cores. On core 0, just to the left of 3.148 seconds, there is another black line, this one representing a `kworker`. Prior to the black line is an orange line, representing a UA task that was preempted. Shortly afterwards, this orange task appears on core 96. This behavior is characteristic of load balancing.

To the scheduler, moving the orange task from core 0 (socket 0) to core 96 (socket 3) appears to be beneficial, because socket 3 is currently largely idle. But overall the application has N tasks on N cores; the tasks that were on socket 3 will wake up at some time in the future, either leading to an overload or triggering further migrations. In our case, there is the following series of task migrations:

1. The UA task (orange) on core 0 (socket 0) is load balanced at 3.147741 seconds to core 96 (socket 3).

2. The task previously on core 96 wakes up at 3.150725 seconds and is placed on core 99 (also

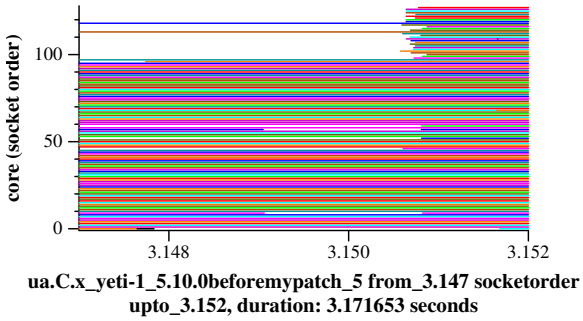

Figure 10: Zoom into the region starting at 3.147 seconds

on socket 3).

3. The task previously on core 99 wakes up at 3.150745 seconds and is placed on core 100 (also on socket 3).

4. The task previously on core 100 wakes up at 3.150757 seconds and is placed on core 111 (also on socket 3).

The initial load balancing migration moves a task between sockets and can introduce both locality and work conservation issues. The remaining migrations are within a socket, but can still introduce locality issues if there is any locally cached data.

3.4 Zooming in, step 3

Our final investigation is of the color change on core 68, from cyan to orange, shortly before 3.152 seconds (right side of Figure 10). Unlike the other cases we have considered, this perturbation in the UA behavior occurs instantaneously, with no intervention from a kernel thread. This behavior is characteristic of the case where there are multiple tasks on one core, and they switch between each other according to their time slices. We confirm this with `running_waiting`, as shown in Figure 11, where in this region there is a red line above the green line. The number of running tasks is one less than the number of cores, and thus one task is waiting in a runqueue.

We investigate further why the cyan and orange tasks are on the same core, specifically why the orange task is placed on core 68 where the cyan task is already running. Figure 12 shows an extract of the activity on cores 68 and 111.[4] The orange task was previously running on core 111. Shortly after 3.15 seconds, at the point marked by $\times$, the `trace-cmd` trace reveals that the cyan task, running on core 68, wakes the orange task. The scheduler, however, does not place the orange task on its previous core, even though that core is currently idle, but rather on the core of the waker, *i.e.*, core 68. This placement is a work conservation violation.

Understanding the reason for this placement requires more information about the task placement strategy of CFS. When a task wakes up, the scheduler first chooses a *target core*, which is the core from which the search begins for an idle core. This search is furthermore limited to the socket of the target core. The target core is either the core where the task ran previously or the core of the waker. When, as here, the core of the waker is not idle (because it is running

[4] This graph was constructed manually from the output of `dat2graph`.

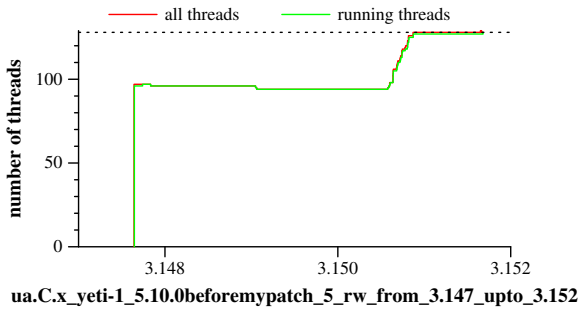

Figure 11: Running and waiting threads near the overload on core 68

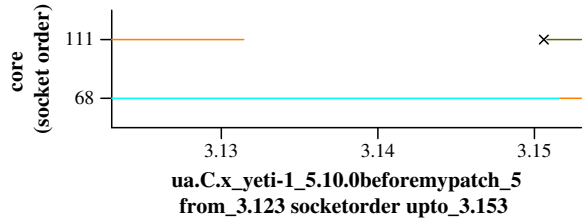

Figure 12: The cores involved in the overload between the cyan task and the orange task

the waker), the choice is made by comparing the load averages of the two cores (the function `wake_affine_weight` in `fair.c`). The load average reflects not only the current load on the core, but also the loads of tasks that have recently run on the core. In our case, there is a small black mark on core 111 near 3.14 seconds, representing the execution of a `kworker`. In this case, the influence of this `kworker` is sufficient to cause the waker core to be chosen as the target. At this point, there is also a task running on every core of the socket of the waker, and thus CFS places the waking task on the target core (*i.e.*, the waker core) itself.

3.5 Epilogue: A patch for the Linux kernel

To address the issue leading to the overload of the cyan and orange tasks, we proposed the patch shown in Figure 13 to the Linux kernel scheduler developers. This patch ensures that if the core where the task ran previously is currently idle, then that core will be chosen as the target core. If this core remains idle throughout the core selection process, the task will then be placed on its previous core. The running times of UA after applying this patch are shown in Figure 14. While the running times vary somewhat, the overheads of over 40% have been eliminated, resulting in a maximum overhead between the longest and the shortest of the 10 runs of 21%. The patch first appeared in a Linux kernel release in v5.11 (commit id `d8fcb81f1acf`).

In general, the change has the impact of keeping tasks more often on their previous cores when possible, and thus keeping them on their previous socket, which can improve locality. Figure 15 compares the execution of the benchmark `h2` of the DaCapo [2] benchmark suite for Java before (left) and after (right) applying the patch (the graphs were created with the `--forked` option to highlight the `h2` behavior). We show the slowest of 5 executions in each case. Each

```
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;
 }
```

Figure 13: Patch ensuring that a task will be placed on its previous core, if that core is idle

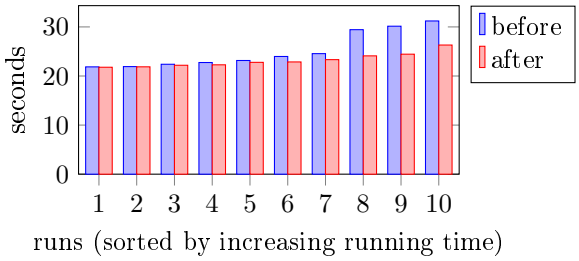

Figure 14: Running time of UA before and after applying commit `d8fcb81f1acf`

graph represents 10 runs of the `h2` application. Without the patch, the tasks often spread out across the machine, and the runs where the tasks spread out more have longer running times, suggesting a locality issue. With the patch, almost all `h2` tasks remain on the same socket (the small amount of activity on the other sockets mostly involves the JIT compilation and garbage collection tasks, as can be seen using `--color-by-command`). Over the 5 executions, the running times range from 81 to 105 seconds without the patch and from 63 to 69 seconds with the patch, exhibiting both an improvement in running time and a reduction in variability.

Study of `h2` with `running_waiting` furthermore reveals that only a handful of tasks execute at any point in time. Consolidating these tasks on fewer cores of the socket gives a further performance improvement by allowing the use of the highest turbo frequencies. Such consolidation was proposed in the work on Nest [4], targeting Linux v5.9, where Nest additionally provided an alternate means to reduce the tendency of the tasks to spread across the sockets of the machine.

4 Implementation

`dat2graph` is implemented as around 1200 lines of OCaml code, and `running_waiting` is implemented as around 500 lines of OCaml code. They jointly rely on an additional 2000 lines of dedicated library code. They are available at:

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

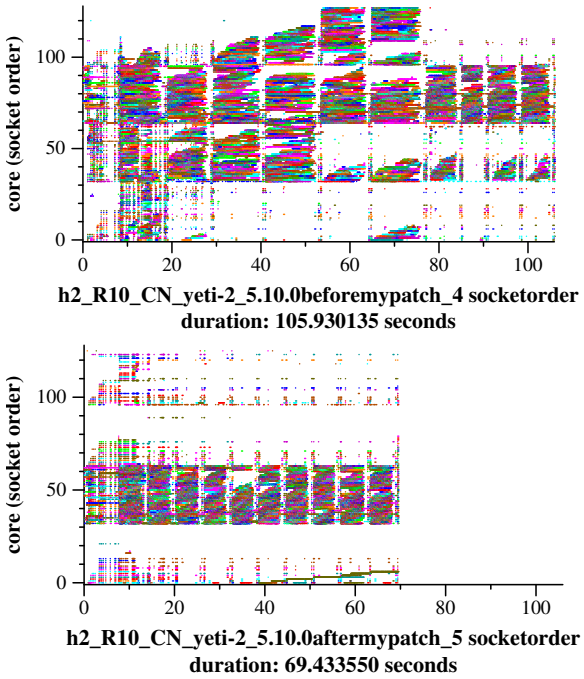

Figure 15: Execution of `h2` before and after application of commit `d8fcb81f1acf`

`dat2graph` and `running_waiting` require the installation of `libtraceevent`, `libtracefs` and `trace-cmd`, which are available at:

```
https://git.kernel.org/pub/scm/libs/libtrace/libtraceevent.git
https://git.kernel.org/pub/scm/libs/libtrace/libtracefs.git
https://git.kernel.org/pub/scm/utils/trace-cmd/trace-cmd.git
```

5 Conclusion

In conclusion, understanding scheduler behavior requires understanding not just what is the running time of the application, but also understanding how the scheduler behaves on the individual tasks of the application. We have also found that different tools can give complementary information. In particular, while `running_waiting` gives very coarse-grained information, knowing how many cores are used and whether tasks are waiting can give some useful big-picture information about the application and its scheduling behavior. In future work, we will consider how to make the tools more configurable, and whether it is possible to improve the graph generation time, ideally, to the point of reintroducing some degree of interactivity.

Acknowledgements. Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr). This work is supported in part by Oracle donations CR 2961 and CR 4201. We would like to thank Steve Rostedt for timely help with `trace-cmd`.

References

[1] D. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS parallel benchmarks summary and preliminary results. In *Supercomputing*, pages 158–165, Seattle, WA, USA, 1991.

[2] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, pages 169–190, Oct. 2006.

[3] https://kernelshark.org/.

[4] J. Lawall, H. Chhaya-Shailesh, J. Lozi, B. Lepers, W. Zwaenepoel, and G. Muller. OS scheduling with Nest: keeping tasks close together on warm cores. In *EuroSys*, pages 368–383. ACM, 2022.

[5] S. Rostedt. ftrace - function tracer. https://docs.kernel.org/trace/ftrace.html.

[6] https://www.trace-cmd.org/.

RESEARCH CENTRE
PARIS

2 rue Simone Iff - CS 42112
75589 Paris Cedex 12

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399