

Fork/Wait and Multicore Frequency Scaling: a Generational Clash

Damien Carver, Redha Gouicem, Jean-Pierre Lozi, Julien Sopena, Baptiste Lepers, Willy Zwaenepoel, Nicolas Palix, Julia L. Lawall, Gilles Muller

▶ **To cite this version:**

Damien Carver, Redha Gouicem, Jean-Pierre Lozi, Julien Sopena, Baptiste Lepers, et al.. Fork/Wait and Multicore Frequency Scaling: a Generational Clash. 10th Workshop on Programming Languages and Operating Systems, Oct 2019, Huntsville, Canada. pp.53-59, 10.1145/3365137.3365400 . hal-02349987v1

HAL Id: hal-02349987

https://inria.hal.science/hal-02349987v1

Submitted on 5 Nov 2019 (v1), last revised 31 Jan 2020 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Fork/Wait and Multicore Frequency Scaling: a Generational Clash

Damien Carver, Redha Gouicem, Julien Sopena,
Julia Lawall, Gilles Muller
first.last@lip6.fr
Sorbonne Université, LIP6, Inria

Jean-Pierre Lozi
jean-pierre.lozi@oracle.com
Oracle Labs

Baptiste Lepers, Willy Zwaenepoel
first.last@sydney.edu.au
University of Sydney

Nicolas Palix
nicolas.palix@imag.fr
Université Grenoble Alpes

Abstract

The complexity of computer architectures has risen since the early years of the Linux kernel: Simultaneous Multi-Threading (SMT), multicore processing, and frequency scaling with complex algorithms such as Intel® Turbo Boost have all become omnipresent. In order to keep up with hardware innovations, the Linux scheduler has been rewritten several times, and many hardware-related heuristics have been added. Despite this, we show in this paper that a fundamental problem was never identified: the POSIX process creation model, *i.e.*, `fork/wait`, can behave inefficiently on current multicore architectures due to frequency scaling. We investigate this issue through a simple case study: the compilation of the Linux kernel source tree. To do this, we develop SchedLog, a low-overhead scheduler tracing tool, and SchedDisplay, a scriptable tool to graphically analyze SchedLog's traces efficiently.

We implement two solutions to the problem at the scheduler level which improve the speed of compiling part of the Linux kernel by up to 26%, and the whole kernel by up to 10%.

ACM Reference Format:
Damien Carver, Redha Gouicem, Julien Sopena, Julia Lawall, Gilles Muller, Jean-Pierre Lozi, Baptiste Lepers, Willy Zwaenepoel, and Nicolas Palix. 2019. Fork/Wait and Multicore Frequency Scaling: a Generational Clash. In *Proceedings of 10th Workshop on Programming Languages and Operating Systems (PLOS '19).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Over the past decade, computer architectures have grown increasingly complex. It is now common, even for affordable machines, to sport multiple CPUs with dozens of hardware threads. These hardware threads are sometimes linked together through hyperthreading and different levels of shared caches, their memory latency and bandwidth is non-uniform due to NUMA and a complex network of interconnect links, and they run at different—yet non-independent—frequencies.

In order to tackle increased hardware complexity, the Linux scheduler has had to evolve. It was rewritten twice: in 2003, the O(1) scheduler was introduced, and in 2007, the Completely Fair Scheduler (CFS) replaced it. Since then, a myriad of heuristics has been added. A large body of research has focused on improving scheduling on modern multicore architectures, often focusing on locality and NUMA issues [13, 19].

Despite these efforts, recent works have hinted that there may still be significant room for improvement. Major performance bugs have gone unnoticed for years in CFS [20], despite the widespread use of Linux. Furthermore, while ULE, the scheduler of FreeBSD, does not outperform CFS on average, it does outperform it significantly on many workloads [7], for reasons that are not always well understood. With currently available tools, studying scheduler behavior at a fine grain is cumbersome, and there is a risk that the overhead of monitoring will interfere with the behavior that is intended to be observed. Indeed, it is rarely done, either by Linux kernel developers or by the research community. However, understanding scheduler behavior is necessary in order to fully exploit the performance of multicore architectures, as most classes of workloads trust the scheduler for task placement.

In this paper, we show that CFS suffers from a fundamental performance issue that directly stems from the POSIX model of creating processes, *i.e.*, `fork/wait`, on multicore architectures with frequency scaling enabled. Consider a common case where a parent process forks a child and waits for the result, in a low concurrency scenario, for example in the case of a typical shell script. If there are idle cores, CFS will choose one of them for the child process. As the core has been idle, the child process will likely start to run at a low frequency. On the other hand, the core of the parent has seen recent activity. The hardware's frequency scaling algorithm will likely have increased its frequency, even if, due to a wait, the core is now idle.

We expose the performance issue through a case study of Kbuild, *i.e.*, building all or part of the Linux kernel. We conduct the study using our own dedicated tracing tool for the Linux scheduler, SchedLog, that focuses on recording scheduling events with very low overhead. We then visualize

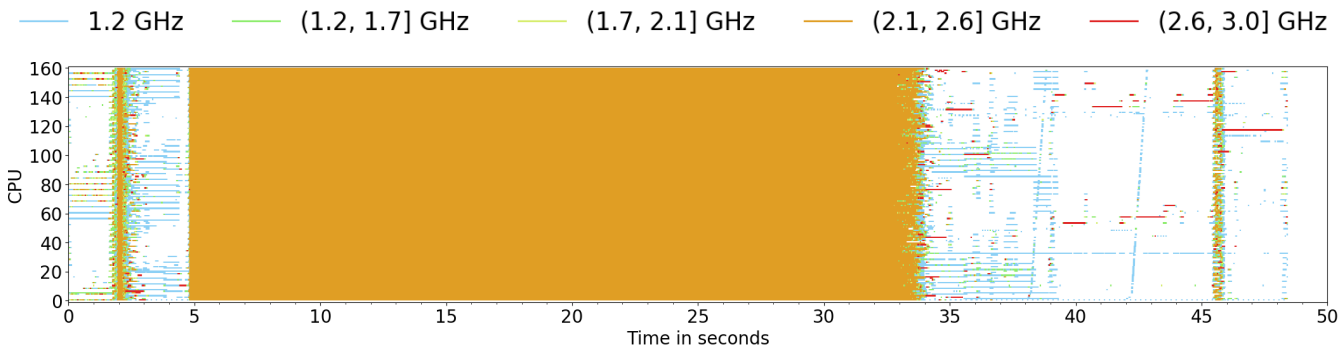


Figure 1. Execution trace: building the Linux kernel scheduler using 320 jobs with CFS.

these events with SchedDisplay, a highly configurable visualization tool we developed, to understand the scheduler's behavior. With our newly learnt knowledge, we then implement two solutions for the performance issues that improve performance by up to 26% on Kbuild. Finally, we propose alternative solutions that aim to be more efficient, and to reduce the likelihood of worsening performance on other workloads.

The contributions of this paper are the following:

- The identification of a performance issue in CFS that directly stems from the `fork/wait` process creation model of POSIX.
- A case study of the performance of the Linux scheduler using a workload that is representative of running compilation tasks in a large C project: Kbuild.
- SchedLog, a low-overhead tracing tool that focuses on scheduling events.
- SchedDisplay, a highly configurable graphical tracing tool for SchedLog traces to ease the detection of poor scheduler behavior.

The rest of this paper is organized as follows. Section 2 shows, through a case study of Kbuild, that the `fork/wait` process creation model of POSIX can behave inefficiently on current multicore architectures. Section 3 describes our graphical tracing tool for the Linux scheduler. Section 4 discusses solutions to the performance problem and presents some that we implemented. Section 5 presents related work. Finally, Section 6 concludes.

2 Kbuild and the Performance of Fork/Wait

We analyze the performance of Kbuild on a 4-socket Xeon E7-8870 machine (80 cores/160 hardware threads) with 512 GiB of RAM, running Linux 4.19 in Debian *Buster*. The CPUs have a frequency range of 1.2-2.1 GHz, and can reach up to 3.0 GHz with Intel® Turbo Boost.

Figure 1 shows a trace from our graphical tool with the frequency of the active cores at each scheduler tick (every 4 ms) when compiling the whole Linux kernel. An initial kernel compilation (*i.e.*, subsequent to `make clean`) performs a number of preparatory activities, then compiles the kernel source files, and finally performs some cleanup. Around 2 seconds there is a short burst of activity, performing an initial build across the source tree, where all of the cores are used and the cores reach a high frequency (2.1 – 2.6 GHz; they do not reach the highest frequencies provided by Turbo Boost due to safeguards in Turbo Boost to prevent overheating). Likewise, from 5 to 34 seconds, all of the cores are again used and reach a high frequency, to compile the major part of the kernel source code. There appears to be little room for improvement in these regions. On the other hand, the regions at the beginning and end of the graph show a moderate number of cores used, almost all running at the lowest frequency, around 1.2 GHz. Studying the trace at a higher degree of magnification shows that the code executed here is nearly sequential, typically alternating between one and two active cores at a time. This raises the question of why so many cores are used for so little work, and why are they not able to reach a higher frequency when they are occupied. We are in the presence of a performance bug.

3 A Tracing Tool for CFS

In this section, we describe a very low overhead tool that we have developed for tracing scheduler behavior. In the next section, we describe how we have used this tool to understand the Kbuild performance bug and to evaluate our proposed solutions.

3.1 Collecting scheduling events

In order to study the Linux scheduler, we need to understand its behavior. To this end, we must collect the list of scheduling events that happen at runtime, such as thread creations, clock ticks and thread migrations. In order to avoid impacting the application's or the scheduler's behavior, the event collection mechanism should have nearly no overhead. Perf [24] is a widely used tool for kernel tracing, but we have found that it incurs a 3% overhead on average when tracing scheduling

events. Indeed, Perf appears to collect more information than necessary. For example, Perf collects 104 bytes of information for each wakeup event, while we have found only 20 bytes to be sufficient for understanding scheduler behavior. We have thus developed SchedLog, a tracing tool that reduces both the amount of information collected and the memory footprint so as to obtain a nearly negligible overhead.

SchedLog records events in preallocated private per-CPU ring buffers. We have found that tracing the compilation of the whole kernel with 320 threads only uses 82 MB of memory in the ring buffers: SchedLog can thus record long and complex executions in a reasonably sized buffer. If a buffer overflows, SchedLog continues recording new events by overwriting the older ones and keeping track of the number of lost events. This optimistic behavior removes a lot of sanity checks, thus minimizing the cost of recording events.

In the ring buffers, each entry occupies a fixed size (24 bytes), allowing us to implement each ring buffer as an array. An entry contains:

- an 8-byte timestamp with nanosecond precision. This timestamp is recorded with the per-CPU local clock to minimize overhead.
- the PID of the process executing the scheduling event (4 bytes).
- a 4-byte event ID.
- an 8-byte field used to store additional event-specific information.

Examples of information stored in the last field are: the source and destination CPUs of a migrated thread during a `migrate` event; the current hardware frequency of the CPU for a `tick` event; the first 8 characters of the new name of a process, obtained from the `comm` field of the structure representing a task in the kernel, for an `exec` event.

The ring buffers are dumped at the end of the execution for offline analysis. SchedLog can record millions of scheduling events per second with very little overhead. On the Kbuild use-case presented in this paper, the overhead was under 0.01%.

3.2 Displaying scheduling events

Manually exploring and spotting bottlenecks in the traces is quite challenging. First, it requires dealing with a huge mass of information. Second, the information of interest is not known in advance and becomes clear only when making progress in understanding the application/scheduler behavior and interaction. Therefore, we have also developed a trace visualization tool, SchedDisplay, that can be configured using user-defined scripts. SchedDisplay is based on two major python libraries: Bokeh [6], which enables the interactive visualization of data in modern web browsers, and Datashader [1], which enables the rasterization of large amounts of data on the web server. SchedDisplay represents a scheduling event at a given time on a given CPU as a vertical segment. It also uses horizontal segments to represent intervals having a particular property.

First, SchedDisplay can be configured to enrich SchedLog entries with information that SchedLog omitted for efficiency. For example, the user can add to every event a new `name` field that is mapped to the name of the process concerned by the event (i.e. its `comm`) using the event's PID and the previous `fork` and `exec` events of this PID. The user can then use this process name as a tag for filtering events. Another scripting example is the possibility to create events that span a range of time. For this, a script can search for different occurrences of a given event on a given core, and create pairs of related events. Then, horizontal lines can be drawn to connect the two events in a pair.

Second, SchedDisplay selects subsets of segments according to user-defined queries. A query is a tree of logical operators on any field (either available in the trace, or added during the decoration step). The tool then attaches user-defined properties such as color, line width and label, to the selected segments.

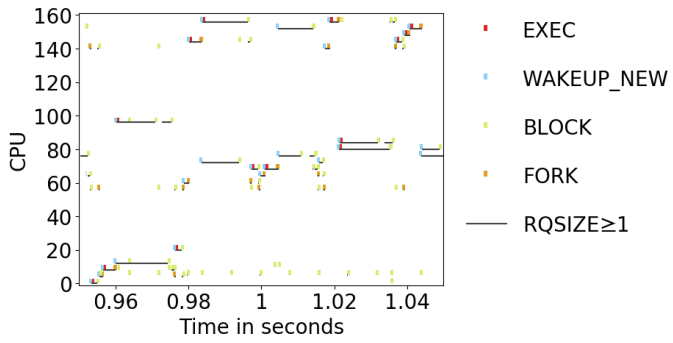
Finally, SchedDisplay offers two display modes. When the user wants a global view, *e.g.*, to detect application phases, the tool computes an image where the color of individual pixels is an aggregate of the color of the segments represented by the pixel. When the user needs a more precise representation of events, he may zoom in and specify a starting time and a duration to study parts of the trace in more detail. In the latter case, for efficiency, SchedDisplay does not compute the segments that are located outside of the local view. Also the user can obtain detailed information on SchedLog entries (PID, command, etc.) by hovering the mouse over segments.

4 Kbuild Revisited

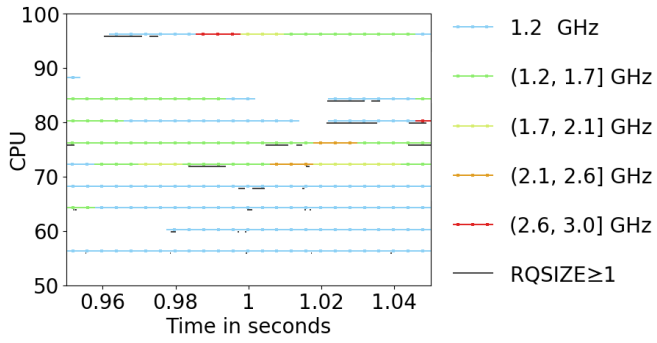
We now use our tools to study in more detail the execution of Kbuild, and then propose some changes to the scheduler that can improve the utilization of higher frequencies and reduce the Kbuild execution time.

4.1 Identifying the problem

We use SchedLog to collect the trace of the kernel build. We then use SchedDisplay to study the execution of the various processes forked by Kbuild on their various cores, focusing on the first couple of seconds where the core utilization seems to be suboptimal. Zooming in, as shown in Figure 2a, we observe that frequently one process runs for a short amount of time (horizontal segment annotated with "RQSIZE $\geq$ 1"), and then another process (or two) starts on another core(s). See for example, the horizontal segments in the lower left of Figure 2a. It remains to understand why this is undesirable behavior. A potential target is the cache, but this is unlikely to be relevant to a job like Kbuild that composes different programs (`gcc`, `as`, shell, etc.). We then



(a) Scheduler events.



(b) Frequencies.

Figure 2. Zoom over a sparse region of Figure 1.

consider another way in which cores can differ, which is by their frequency. Extending the tick event to record the frequency of the associated core shows the behavior presented in the introduction and shown in Figure 2b. The core frequency only increases after a delay that amounts to several ticks. The computations in the (near) sequential phases of Kbuild are shorter than this delay. Thus, cores speed up after they have performed some computation, but at that point, computation has moved to other processes, running on recently idle and thus slower cores.

4.2 Patching CFS

Based on the above analysis, a solution to improve the performance of Kbuild is for the scheduler to find high frequency cores for any forked and waking processes. Our observation is that due to the nature of a software build, processes in Kbuild that fork other processes, mostly then wait for their results, rather than continuing execution. The core that runs the parent process is thus likely at or soon to be at a high frequency, and is also soon to be idle, making it a desirable core to run a child. A similar situation occurs when the parent wakes after a child has terminated; the core of the child has seen recent execution and is thus likely at or soon to be at a high frequency, while also currently being underutilized.

One solution would be to provide more information about the application behavior to the scheduler by extending the `fork` system call with a flag indicating whether the parent process will immediately wait. With this information, at the time of the `fork`, the scheduler can detect whether the core of the parent process will soon be idle (modulo other process migrations). The scheduler could place the child on the core of the parent only in this case. More generally, as child processes can have different execution times, the application could exploit such a `fork` variant to indicate to the scheduler which is the best child to collocate with the parent, to maximize the likelihood that the core will reach a high frequency within the child's computation. Extending `fork`, however, would amount to breaking POSIX, and would require substantial changes in user-level software to be effective in practice.

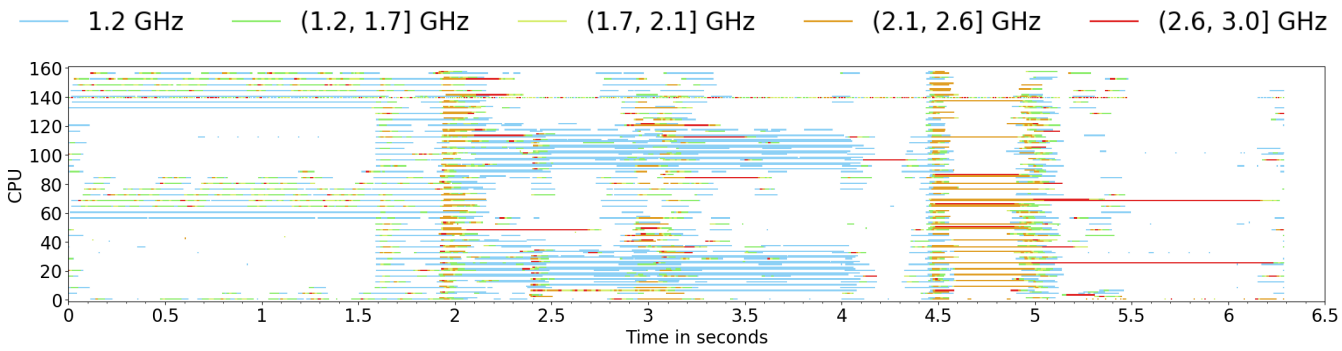
We consider two scheduling strategies that do not require modification to the POSIX API. These strategies tweak the thread placement strategy on a process fork or wakeup to avoid unnecessary frequency throttling. The first strategy, on a fork, places the child on the same core as the parent, in the case when there is no process other than the parent on that core. Likewise, this strategy places a waking process on the core where it was executing previously if there is at most one process on that core. When these criteria are not satisfied, this strategy behaves the same as CFS. The second strategy places all children on the same core as the parent, and likewise, always returns a waking process to the core where it ran previously. This strategy relies on load balancing to disperse the children if the core becomes overloaded. We apply these strategies throughout the execution of Kbuild, including both the mostly sequential and the highly concurrent phases.

4.3 Experiments

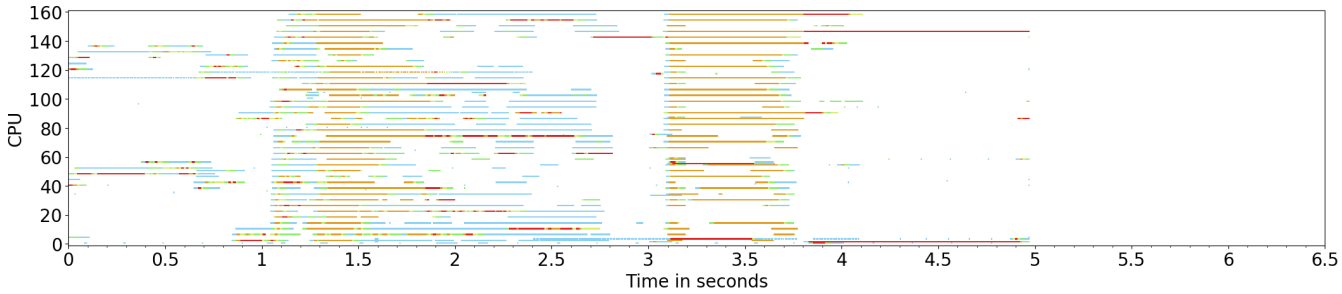
To illustrate the performance of our strategies, we consider building only the kernel scheduler (the `kernel/sched/` directory) with 32 jobs, which entails less computation than a full kernel build, thus resulting in more readable traces. Figure 3a shows the effect of CFS. The trace has a similar structure to that of the full kernel build in Figure 1. In particular, in the sequential phases, many cores are used and these cores rarely reach the higher frequencies, and do so only for short periods. Even the concurrent phases do not always use the higher frequencies available.

Figure 3b then shows the frequency trace of the build of the kernel scheduler with 32 jobs using our first strategy, where on a fork the first child is placed on the same core as the parent. During the initial phase, the build now never uses more than 5 CPUs at a time, and these CPUs run at a higher frequency. This phase's duration drops by a third, and the overall CPU usage is also better, leading to energy savings. The build also consumes 20% less energy.[1] The second phase

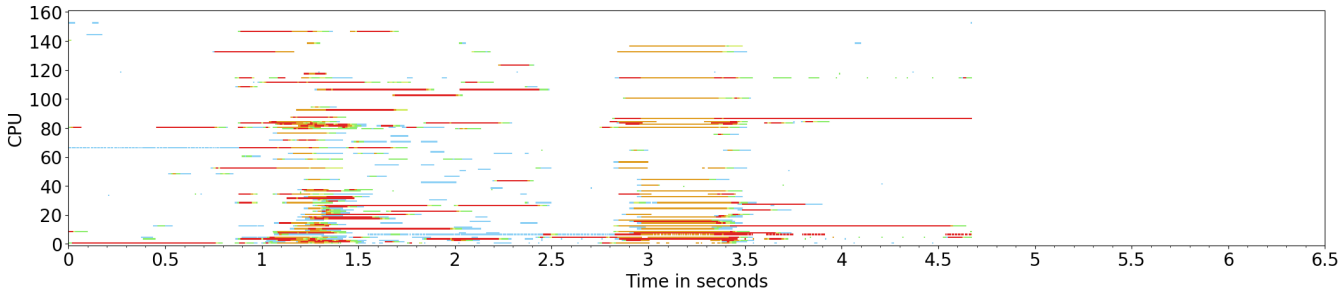
[1]We measure the energy consumption of the CPU packages and of the DRAM using hardware counters.



(a) With CFS.



(b) With our first strategy for patching CFS.



(c) With our second strategy for patching CFS.

Figure 3. Execution traces: building the Linux kernel scheduler using 32 jobs.

that exhibited frequency issues also benefits from our first strategy in the same proportions.

Figure 3c then shows the frequency trace of the build of the kernel scheduler with 32 jobs using our second strategy, where on a fork all children are placed on the same core as the parent. We observe that CPUs running at low frequencies have almost completely disappeared. Overall, there are also fewer CPUs in use concurrently, which allows the CPUs to run at maximum frequency for long periods of time via Intel® Turbo Boost (*e.g.,* between 1 and 1.5 seconds). As compared to our first strategy (Figure 3b), this strategy further reduces slightly the duration of the mostly sequential phases that ran at low frequency with the default Linux scheduler. Compared to our first strategy, we use even fewer CPUs during the parallel phases, and those used run at higher frequencies.

4.4 Interaction with Intel® Turbo Boost

As our experiments have been carried out on Intel® CPUs, we consider the possible interaction between our proposed scheduling strategies and Intel® Turbo Boost. Turbo Boost makes it possible for a limited set of cores to run at frequencies higher than their nominal one, while throttling other cores. Threads running on these throttled cores might slow down the entire application and hurt performance (e.g. barriers).

In practice, the frequency trace in Figure 1 using CFS, shows that it is possible to have all cores running at a frequency that, while not being the maximum (the (2.6,3.0] GHz range shown in red) is still greater than the nominal frequency of the machine (2.1 GHz). Furthermore, Figure 3c shows that our second strategy can reduce the number of

cores used in the concurrent case. In practice, as shown by Figure 3c, multiple cores run at the highest frequency even in concurrent phases, thanks to Turbo Boost.

5 Related Work

Profiler design. Various profilers have been implemented to help understand the performance of applications on Linux. Most used is the Perf tool [14, 15, 32]. Perf allows developers to collect hardware- and software-related events and offers tools to visualize scheduling decisions, via the `perf sched` command. While useful to understand analyze the behavior of basic scheduler workloads, Perf's performance overhead is high on real-world workloads such as Kbuild. Mollison et al. [22] do regression testing for schedulers. They target real-time schedulers and only study a subset of scheduling-related events that would not be sufficient to understand to the `fork/wait` issue. Lozi. et al. [21] analyze work-conservation bugs in Linux. They develop a basic tracing tool that only records runqueue sizes and thread loads, not scheduling events.

Altman et al. [4] profile idle time in applications and analyze dependencies that result in threads waiting for others threads. Various tools have been proposed to understand sources of latency in applications [10, 16, 18]. To the best of our knowledge, none of these tools analyzes the impact of the scheduler on performance.

More generally, testing the impact of kernels on performance is an ongoing research effort. The Linux Kernel Performance project [11] was started in 2005 to find performance regressions, and numerous tools have been proposed to find performance-related bugs in kernels [8, 17, 25, 27]. These works focus on timing issues in the kernel (e.g., functions taking too long to complete) and cannot be used to understand scheduler-related performance issues.

Impact of the scheduler. The influence of general-purpose OS schedulers on performance has been extensively studied. Most previous work focuses on implementing new generic scheduling policies that improve a specific hardware-related performance metric: locality for NUMA machines [9, 13], cache re-use [28, 29], or reducing contention on shared resources [31, 35]. We are not aware of recent research on general-purpose OS schedulers that focuses on hardware metrics that are related to frequency scaling. In 2010, Zhang et al. [34] proposed a simple scheduling policy for multicore architectures that reduced cache interference. They argued that their approach also facilitated frequency scaling, but their focus was only on per-chip frequency as back then, per-core frequency scaling was not as efficient or commonplace. In the context of mobile devices, some more recent research has focused on energy-aware scheduling [30, 33].

Recently, the interaction between frequency scaling and scheduling has drawn attention in the Linux kernel developer community [12], specifically in relation to turbo frequencies. It has been observed that a short-lived jitter process that starts on an idle core can lead that core to eventually use turbo frequencies. If this core causes the number of cores using turbo frequencies to exceed the number allowed to avoid overheating, other cores will be forced to reduce their frequency, even if the jitter process has already completed and its core is idle. A patch set [26] has been proposed to ensure that explicitly marked jitter tasks are only placed on cores that are already active and are expected to remain active, to avoid increasing the number of active cores. In contrast, the `fork/wait` issue we have identified applies to all forms of frequency scaling, whether or not turbo frequencies are used.

Another approach to enhance scheduler performance consists in strongly coupling the scheduler and applications. Scheduler activations [5] map user-level threads to kernel-level threads in order to tweak the kernel scheduler's decisions from userspace. Rinnegan [23] provides kernel support to help applications make informed thread placement on heterogeneous architectures. These approaches usually require changes in applications or in the kernel API.

6 Conclusion

In this paper, we have identified a performance issue caused by the `fork/wait` model on multicore architectures due to frequency scaling. Thanks to their flexibility and low overhead, our SchedLog and SchedDisplay tools were crucial to finding and understanding the performance issue. We propose and evaluate two scheduling strategies targeting the behaviors we observe in our case study Kbuild. The two solutions we propose can be further improved. In future work, as a first step, we will try to use a strategy to locate high-frequency cores. And as a second step, we will target more architectures and applications, and design a general-purpose, frequency-aware scheduling algorithm for multicore architectures. We are making SchedLog and SchedDisplay publicly available [2, 3].

References

[1] Datashader. http://datashader.org/.

[2] SchedDisplay. https://github.com/carverdamien/SchedDisplay.

[3] SchedLog. https://github.com/carverdamien/SchedLog.

[4] Altman, E., Arnold, M., Fink, S., and Mitchell, N. Performance analysis of idle programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (New York, NY, USA, 2010), OOPSLA '10, ACM, pp. 739–753.

[5] Anderson, T. E., Bershad, B. N., Lazowska, E. D., and Levy, H. M. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems (TOCS) 10*, 1 (1992), 53–79.

[6] Bird, S., Canavan, L., Hulsey, C., Rudiger, M. P. P., and de Ven, B. V. Welcome to Bokeh, 2013. https://bokeh.pydata.org/en/latest/.

[7] Bouron, J., Chevalley, S., Lepers, B., Zwaenepoel, W., Gouicem, R., Lawall, J., Muller, G., and Sopena, J. The battle of the schedulers:

FreeBSD ULE vs. Linux CFS. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018.* (2018), pp. 85–96.

[8] Boyd-Wickizer, S., Clements, A. T., Mao, Y., Pesterev, A., Kaashoek, M. F., Morris, R., and Zeldovich, N. An analysis of Linux scalability to many cores. In *OSDI* (2010).

[9] Brecht, T. On the importance of parallel application placement in NUMA Multiprocessors. In *USENIX SEDMS* (1993).

[10] Chanda, A., Cox, A. L., and Zwaenepoel, W. Whodunit: Transactional profiling for multi-tier applications. In *EuroSys* (2007), pp. 17–30.

[11] Chen, T., Ananiev, L. I., and Tikhonov, A. V. Keeping kernel performance from regressions. In *Linux Symposium* (2007), vol. 1, pp. 93–102.

[12] Corbet, J. TurboSched: the return of small-task packing. *Linux Weekly News* (July 1, 2019). https://lwn.net/Articles/792471/.

[13] Dashti, M., Fedorova, A., Funston, J., Gaud, F., Lachaize, R., Lepers, B., Quema, V., and Roth, M. Traffic management: a holistic approach to memory placement on NUMA systems. In *ASPLOS* (2013), pp. 381–394.

[14] de Melo, A. C. Performance counters on Linux. In *Linux Plumbers Conference* (2009), vol. 118.

[15] de Melo, A. C. The new Linux 'perf' tools. In *Slides from Linux Kongress* (2010), vol. 18.

[16] Fürlinger, K., and Gerndt, M. ompP: A profiling tool for OpenMP. In *International Workshop on OpenMP* (2005), Springer, pp. 15–23.

[17] Harji, A. S., Buhr, P. A., and Brecht, T. Our troubles with Linux and why you should care. In *Proceedings of the Second Asia-Pacific Workshop on Systems* (2011), APSys '11, pp. 2:1–2:5.

[18] Joukov, N., Traeger, A., Iyer, R., Wright, C. P., and Zadok, E. Operating system profiling via latency analysis. In *Proceedings of the 7th symposium on Operating systems design and implementation* (2006), USENIX Association, pp. 89–102.

[19] Lepers, B., Quéma, V., and Fedorova, A. Thread and memory placement on NUMA systems: asymmetry matters. In *USENIX ATC* (2015), pp. 277–289.

[20] Lozi, J.-P., David, F., Thomas, G., Lawall, J., and Muller, G. Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)* (Boston, MA, 2012), USENIX, pp. 65–76.

[21] Lozi, J.-P., Lepers, B., Funston, J., Gaud, F., Quéma, V., and Fedorova, A. The Linux scheduler: a decade of wasted cores. In *EuroSys* (2016), ACM, pp. 1:1–1:16.

[22] Mollison, M. S., Brandenburg, B., and Anderson, J. H. Towards unit testing real-time schedulers in LITMUS^{RT}. In *Proceedings of the 5th Workshop on Operating Systems Platforms for Embedded Real-Time Applications* (2009), OSPERT.

[23] Panneerselvam, S., and Swift, M. Rinnegan: Efficient resource use in heterogeneous architectures. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation* (2016), ACM, pp. 373–386.

[24] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org.

[25] Perl, S. E., and Weihl, W. E. Performance assertion checking. In *SOSP* (1993), pp. 134–145.

[26] Shah, P. TurboSched: A scheduler for sustaining turbo frequencies for longer durations, June 25, 2019. https://lwn.net/ml/linux-kernel/20190625043726.21490-1-parth@linux.ibm.com/.

[27] Shen, K., Zhong, M., and Li, C. I/O system performance debugging using model-driven anomaly characterization. In *FAST* (2005), pp. 309–322.

[28] Tam, D., Azimi, R., and Stumm, M. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *EuroSys* (2007), pp. 47–58.

[29] Tang, L., Mars, J., Zhang, X., Hagmann, R., Hundt, R., and Tune, E. Optimizing Google's warehouse scale computers: The NUMA experience. In *High Performance Computer Architecture (HPCA2013)* (Feb 2013), pp. 188–197.

[30] Tseng, P.-H., Hsiu, P.-C., Pan, C.-C., and Kuo, T.-W. User-centric energy-efficient scheduling on multi-core mobile devices. In *Proceedings of the 51st Annual Design Automation Conference* (2014), ACM, pp. 1–6.

[31] Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., and Wilkes, J. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems* (2015), ACM, p. 18.

[32] Weaver, V. M. Linux perf_event features and overhead. In *The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath* (2013), vol. 13.

[33] Yu, K., Han, D., Youn, C., Hwang, S., and Lee, J. Power-aware task scheduling for big.LITTLE mobile processor. In *2013 International SoC Design Conference (ISOCC)* (2013), IEEE, pp. 208–212.

[34] Zhang, X., Dwarkadas, S., and Zhong, R. An evaluation of per-chip nonuniform frequency scaling on multicores. In *USENIX ATC* (Berkeley, CA, USA, 2010).

[35] Zhuravlev, S., Saez, J. C., Blagodurov, S., Fedorova, A., and Prieto, M. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Computing Surveys (CSUR) 45*, 1 (2012), 4.