Fewer Cores, More Hertz: Leveraging High-Frequency Cores in the OS Scheduler for Improved Application Performance

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

Fewer Cores, More Hertz: Leveraging High-Frequency Cores in the OS Scheduler for Improved Application Performance

Redha Gouicem and Damien Carver, *Sorbonne University, LIP6, Inria;* Jean-Pierre Lozi, *Oracle Labs;* Julien Sopena, *Sorbonne University, LIP6, Inria;* Baptiste Lepers and Willy Zwaenepoel, *University of Sydney;* Nicolas Palix, *Université Grenoble Alpes;* Julia Lawall and Gilles Muller, *Inria, Sorbonne University, LIP6*

https://www.usenix.org/conference/atc20/presentation/gouicern

This paper is included in the Proceedings of the 2020 USENIX Annual Technical Conference.

July 15–17, 2020

978-1-939133-14-4

Open access to the Proceedings of the 2020 USENIX Annual Technical Conference is sponsored by USENIX.

Fewer Cores, More Hertz: Leveraging High-Frequency Cores in the OS Scheduler for Improved Application Performance

Redha Gouicem, Damien Carver
Sorbonne University, LIP6, Inria

Jean-Pierre Lozi
Oracle Labs

Julien Sopena
Sorbonne University, LIP6, Inria

Baptiste Lepers, Willy Zwaenepoel
University of Sydney

Nicolas Palix
Université Grenoble Alpes

Julia Lawall, Gilles Muller
Inria, Sorbonne University, LIP6

Abstract

In modern server CPUs, individual cores can run at different frequencies, which allows for fine-grained control of the performance/energy tradeoff. Adjusting the frequency, however, incurs a high latency. We find that this can lead to a problem of *frequency inversion*, whereby the Linux scheduler places a newly active thread on an idle core that takes dozens to hundreds of milliseconds to reach a high frequency, just before another core already running at a high frequency becomes idle.

In this paper, we first illustrate the significant performance overhead of repeated frequency inversion through a case study of scheduler behavior during the compilation of the Linux kernel on an 80-core Intel® Xeon-based machine. Following this, we propose two strategies to reduce the likelihood of frequency inversion in the Linux scheduler. When benchmarked over 60 diverse applications on the Intel® Xeon, the better performing strategy, S_{move}, improves performance by more than 5% (at most 56% with no energy overhead) for 23 applications, and worsens performance by more than 5% (at most 8%) for only 3 applications. On a 4-core AMD Ryzen we obtain performance improvements up to 56%.

1 Introduction

Striking a balance between performance and energy consumption has long been a battle in the development of computing systems. For several decades, CPUs have supported Dynamic Frequency Scaling (DFS), allowing the hardware or the software to update the CPU frequency at runtime. Reducing CPU frequency can reduce energy usage, but may also decrease overall performance. Still, reduced performance may be acceptable for tasks that are often idle or are not very urgent, making it desirable to save energy by reducing the frequency in many use cases. While on the first multi-core machines, all cores of a CPU had to run at the same frequency, recent server CPUs from Intel® and AMD® make it possible to update the frequency of individual cores. This feature allows for much finer-grained control, but also raises new challenges.

One source of challenges in managing core frequencies is the *Frequency Transition Latency* (*FTL*). Indeed, transitioning a core from a low to a high frequency, or conversely, has an FTL of dozens to hundreds of milliseconds. FTL leads to a problem of *frequency inversion* in scenarios that are typical of the use of the standard POSIX `fork()` and `wait()` system calls on process creation, or of synchronization between lightweight threads in a producer-consumer application. The problem occurs as follows. First, a task T_{waker} running on core C_{waker} creates or unblocks a task T_{woken}. If the *Completely Fair Scheduler* (*CFS*), *i.e.*, the default scheduler in Linux, finds an idle core C_{CFS}, it will place T_{woken} on it. Shortly thereafter, T_{waker} terminates or blocks, because *e.g.*, it was a parent process that `fork`ed a child process and `wait`ed just afterwards, or because it was a thread that was done producing data and woke up a consumer thread as its last action before going to sleep. Now C_{waker} is idle and yet executing at a high frequency because it was running T_{waker} until recently, and C_{CFS}, on which T_{woken} is running, is likely to be executing at a low frequency because it was previously idle. Consequently, the frequencies at which C_{waker} and C_{CFS} operate are *inverted* as compared to the load on the cores. This frequency inversion will not be resolved until C_{waker} reaches a low frequency and C_{CFS} reaches a high frequency, *i.e.*, for the duration of the FTL. Current hardware and software DFS policies, including the `schedutil` policy [9] that was recently added to CFS cannot prevent frequency inversion as their only decisions consist in updating core frequencies, thus paying the FTL each time. Frequency inversion reduces performance and may increase energy usage.

In this paper, we first exhibit the problem of frequency inversion in a real-world scenario through a case study of the behavior of CFS when building the Linux kernel on a Intel® Xeon-based machine with 80 cores (160 hardware threads). Our case study finds repeated frequency inversions when processes are created through the `fork()` and `wait()` system calls, and our profiling traces make it clear that frequency inversion leads to tasks running on low frequency cores for a significant part of their execution.

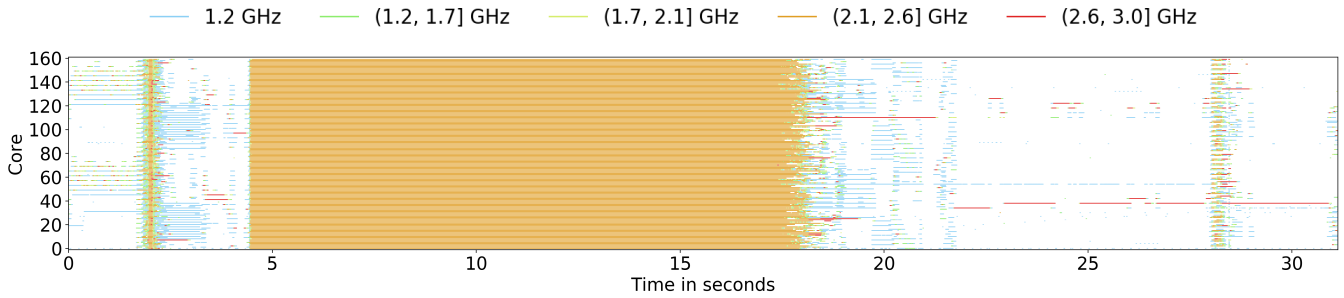


Figure 1: Execution trace when building the Linux kernel version 5.4 using 320 jobs.

Based on the results of the case study, we propose to address frequency inversion at the scheduler level. Our key observation is that the scheduler can avoid frequency inversion by taking core frequencies into account when placing a task on a core. For this, we propose and analyze two strategies. Our first strategy S_{local} is for the scheduler to simply place T_{woken} on C_{waker}, as frequency inversion involves a core C_{waker} that is likely at a high frequency, and may soon be idle. This strategy improves the kernel build performance. It runs the risk, however, that T_{waker} does not promptly terminate or block, causing a long wait before T_{woken} is scheduled. Accordingly, our second strategy S_{move} additionally arms a high-resolution timer when it places T_{woken} on C_{waker}, and if the timer expires before T_{woken} is scheduled, then T_{woken} is migrated to C_{CFS}, *i.e.*, the core CFS originally chose for it. Furthermore, even slightly delaying T_{woken} by placing it on C_{waker} is not worthwhile when C_{CFS} is above the minimum frequency. Thus, S_{move} first checks whether the frequency of C_{CFS} is above the minimum, and if so places T_{woken} on C_{CFS} directly.

The contributions of this paper are the following.

- The identification of the *frequency inversion* phenomenon, which leads to some idle cores running at a high frequency while some busy cores run at a low frequency for a significant amount of time.
- A case study, building the Linux kernel on an 80-core server, with independent per-core frequencies.
- Two strategies, S_{local} and S_{move}, to prevent frequency inversion in CFS. Implementing these policies only required minor code changes: 3 (resp. 124) lines were modified in the Linux kernel to implement S_{local} (resp. S_{move}).
- A comprehensive evaluation of our strategies on 60 diverse applications, including popular Linux benchmarks as well as applications from the Phoronix [23] and NAS [5] benchmark suites. The evaluation considers both the `powersave` CPU governor, which is currently used by default in Linux, and the experimental `schedutil` governor. It also considers two machines: a large 80-core Intel® Xeon E7-8870 v4 server and a smaller 4-core AMD® Ryzen 5 3400G desktop machine.

With the `powersave` governor on the server machine, we find that both S_{local} and S_{move} perform well overall: out of the 60 applications used in the evaluation, S_{local} and S_{move} improve the performance of 27 and 23 applications by more than 5% respectively, and worsen the performance of only 3 applications by more than 5%. In the best case, S_{local} and S_{move} improve application performance by 58% and 56% respectively with no energy overhead. However, S_{local} performs very poorly with two of the applications, even worsening performance by 80% in the worst case, which may not be acceptable for a general-purpose scheduler. S_{move} performs much better in the worst case: the increase in application execution time is only 8% and mitigated by a 9% improvement in terms of energy usage. Evaluation results with `schedutil` show that this governor does not address the frequency inversion issue, and exhibits several more cases in which S_{local} performs very poorly—while S_{move} again has much better worst-case performance. The evaluation on the desktop machine shows similar trends, albeit on a smaller scale. Again, S_{move} performs better than S_{local} on edge cases.

2 A Case Study: Building the Linux Kernel

We present a case study of the workload that led us to discover the frequency inversion phenomenon: building the Linux kernel version 5.4 with 320 jobs (`-j`) on a 4-socket Intel® Xeon E7-8870 v4 machine with 80 cores (160 hardware threads), with a nominal frequency of 2.1 GHz. Thanks to the Intel® SpeedStep and Turbo Boost technologies, our CPUs can individually vary the frequency of each core between 1.2 and 3.0 GHz. The frequency of the two hardware threads of a core is the same. In the rest of the paper, for simplicity, we use the term "core" for hardware threads.

Figure 1 shows the frequency of each core of the machine while the kernel build workload is running. This plot was produced with two tools that we have developed, SchedLog and SchedDisplay [10]. SchedLog collects the execution trace of an application with very low overhead. SchedDisplay produces a graphical view of such a trace. We have used SchedDisplay to generate all execution traces presented in this paper. SchedLog records the frequency information shown in

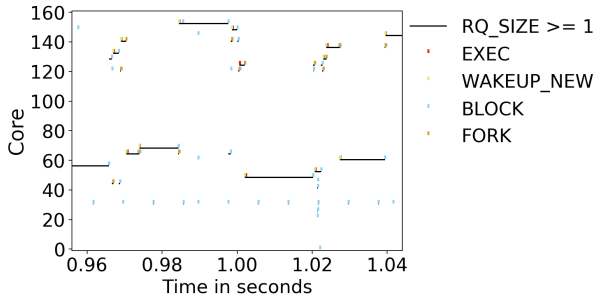
Figure 1 at each tick event (4ms in CFS). Consequently, the absence of a colored line in such traces means that ticks have been disabled by CFS on that core. CFS disables ticks on inactive cores to allow them to switch to a low-power state.

In Figure 1, we notice different phases in the execution. For a short period around 2 seconds, for a longer period between 4.5 and 18 seconds, and for a short period around 28 seconds, the kernel build has highly parallel phases that use all of the cores at a high frequency. The second of these three phases corresponds to the bulk of the compilation. In these three phases, the CPUs seem to be exploited to their maximum. Furthermore, between 22 and 31 seconds, there is a long phase of mostly sequential code with very few active cores, of which there is always one running at a high frequency. In this phase, the bottleneck is the CPU's single-core performance.

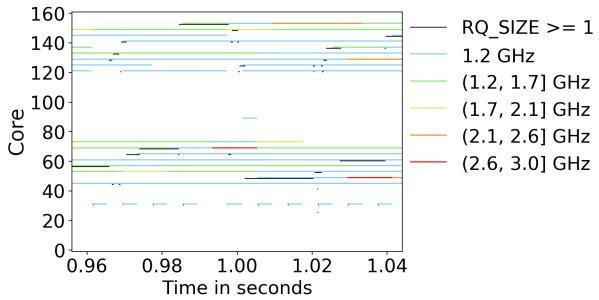
Between 0 and 4.5 seconds, and between 18 and 22 seconds however, there are phases during which all cores are used but they run at the CPU's lowest frequency (1.2 GHz). Upon closer inspection, these phases are actually mainly sequential: zooming in reveals that while all cores are used across the duration of the phase, only one or two cores are used at any given time. This raises two questions: why are so many cores used for a nearly sequential execution, and why are those cores running at such a low frequency.

We focus on the first couple of seconds where core utilization seems to be suboptimal. Zooming around 1 second, we first look at runqueue sizes and scheduling events, as illustrated in Figure 2a. We are in the presence of a pattern that is typical of mostly-sequential shell scripts: processes are created through the `fork()` and `exec()` system calls, and generally execute one after the other. These processes can easily be recognized on Figure 2a as they start with *WAKEUP_NEW* and *EXEC* scheduler events. After the process that runs on Core 56 blocks around the 0.96 s mark, three such short-lived processes execute one after the other on Cores 132, 140, and 65. After that, two longer-running ones run on Core 69 around the 0.98 s mark, and on Core 152 between the 0.98 s and 1.00 s mark. This pattern goes on for the entire duration of the execution shown in Figure 2a, with tasks created one after the other on Cores 148, 125, 49, 52, 129, 156, 60 and finally 145.

Looking at the core frequencies in the same part of the execution, as illustrated by Figure 2b, gives us a hint as to why cores are running slowly in this phase: there seems to be a significant delay between the time when a task starts running on a core, and the time when the core frequency starts increasing. For instance, between 1.00 s and 1.02 s, the task on Core 49 runs at a low frequency, and only when it is over at around 1.04 s does the frequency of the core rise to its maximum—before starting to decrease again almost instantly as the hardware notices that no task is running anymore on that core. The same issue can be observed shortly before 1.00 s on Core 152, and around 0.98 s on Core 69. In this last example, the core's frequency was even on a downward slope when the



(a) Scheduler events.



(b) Core frequencies.

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

task started, and the frequency keeps going down even after the task ended before finally increasing again around 1.00 s. It appears that in the considered phase of the execution, the FTL is much higher than the duration of the tasks. Since tasks that follow each other tend to be scheduled on different cores, they are likely to always run at a low frequency as most cores are idle most of the time in this phase of the execution.

To confirm our intuition about the FTL, we develop a fine-grained tool [1] to monitor the frequency of a single core around the execution of an isolated busy loop, using the `powersave` governor. As shown in Figure 3, the task runs for 0.20 s, as illustrated by the *start* and *end* vertical lines in the figure. It takes an FTL of 29 ms for the core to go from its minimum frequency of 1.25 GHz to its maximum frequency of 3.00 GHz in order to accommodate the task. When the task ends, it takes approximately 10 ms for the core to go back to its initial frequency, but the duration of the FTL is compounded by the fact that the frequency tends to bounce back several times for around 98 ms before stabilizing at the core's lowest frequency. These measurements are consistent with our interpretation of Figure 2b: a FTL of several dozens of milliseconds is significantly longer than the execution of the tasks that are visible in the figure, as the longest task runs for around 20 ms between the 1.00 s and 1.02 s marks. Note that the duration of the FTL is mainly due to the time for the hardware to detect a load change and then decide to change the frequency. Previous work [22] shows that the actual latency for the core to change its frequency is only

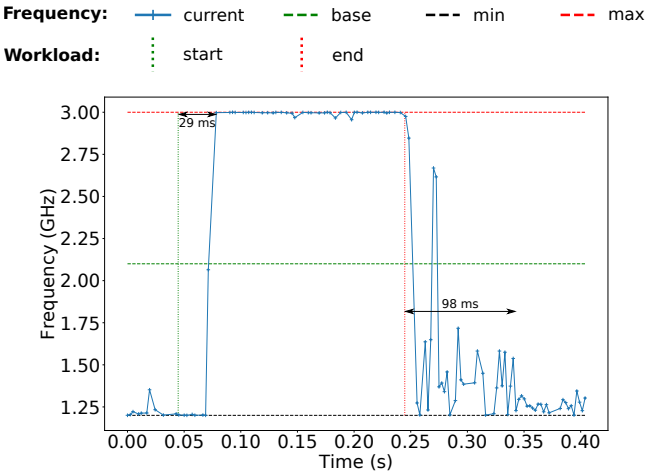


Figure 3: FTL for the Xeon E7-8870 v4 CPU.

tens of microseconds on Intel® CPUs.

Coming back to Figure 2a, the phenomenon we have been observing is the following. Computations in the (near) sequential phases of the build of Linux are launched sequentially as processes through the `fork()` and `wait()` system calls, and the execution of these computations is shorter than the FTL. Consequently, cores speed up after they have performed a computation, even though at that point, computation has moved to newly `fork`ed processes, which are likely to run on cores that were not recently used if the machine is not very busy. Indeed, CFS often selects different cores for tasks to wake up on, and if most cores are idle, it is likely that the selected cores were not used recently, and therefore run at a low frequency. The tasks that initiated the `fork()` perform `wait()` operations shortly afterwards, which means that the frequency increase they initiated is mostly wasted. We are in the presence of recurring *frequency inversion*, which is caused by a very common scenario: launching a series of sequential processes, as is commonly done in a shell script.

Sequential creation of processes through the `fork()` and `wait()` system calls is not the only cause of recurring frequency inversion. This phenomenon can also occur with lightweight threads that unblock and block each other, as is common in producer-consumer applications. Indeed, the CFS code that selects a core for a new task to wake up on is also used to select a core for already existing tasks that wake up. Note that CFS does not use different code paths depending on the type of task, namely, a process or a thread.

3 Strategies to Prevent Frequency Inversion

Since frequency inversion is the result of scheduling decisions, we believe it must be addressed at the scheduler level. In our experience, every change to the scheduler may have unpredictable consequences on some workloads, and the more complex the change, the less predictable the consequences. Therefore, proposing extensive or complex changes to the scheduler, or a complete rewrite, would make it unclear where performance gains come from. Striving for minimal, simple changes allows for an apples-to-apples comparison with CFS.

We propose two strategies to solve the frequency inversion problem. The first one is a simple strategy that offers good performance but suffers from large performance degradations in some scheduling scenarios. The second solution aims to have the same benefits as the first solution while minimizing worst cases at the expense of some simplicity.

3.1 Placing Threads Locally

The first strategy that we propose to prevent frequency inversion is S_{local}: when a thread is created or unblocked, it is placed on the same core as the process that created or unblocked it. In the context of creating a single process through the `fork()` and `wait()` system calls, this strategy implies that the created process is more likely to run on a high-frequency core, as the frequency of the core may already be high due to the activity of the parent. Furthermore, the duration in which there are two processes running on the same core will be limited, if the parent process calls `wait()` shortly afterwards. In the context of a producer-consumer application, when a producer thread wakes up a consumer thread, this strategy again implies that the consumer thread is more likely to run on a high-frequency core, and the duration in which there are two processes running on the same core will again be limited, if the last action of the producer is to wake up the consumer before blocking or terminating.

However, there are cases in which S_{local} might hurt performance: if the task that created or woke another task does not block or terminate quickly afterwards, the created or woken task will wait for the CPU resource for a certain period of time. This issue is mitigated by the periodic load balancer of the Linux scheduler that will migrate one of the tasks to another less loaded core. However, waiting for the next load balancing event might be quite long. In CFS, periodic load balancing is performed hierarchically, with different periods: cores in the same cache domain are more frequently balanced than cores on different NUMA nodes. These periods can vary from 4 to hundreds of milliseconds on large machines.

S_{local} significantly changes the behavior of CFS by fully replacing its thread placement strategy. Additionally, the aforementioned shortcomings make it a high risk solution for certain workloads. Both issues make this solution unsatisfactory given the prerequisites that we previously set.

3.2 Deferring Thread Migrations

In order to fix core oversubscription without waiting for periodic load balancing, we propose a second strategy, S_{move}. With vanilla CFS, when a thread is created or woken, CFS

CPU vendor	Intel®	AMD®
CPU model	Xeon E7-8870 v4	Ryzen 5 3400G
Cores (SMT)	80 (160)	4 (8)
Min freq	1.2 GHz	1.4 GHz
Base freq	2.1 GHz	3.7 GHz
Turbo freq	3.0 GHz	4.2 GHz
Memory	512 GB	8 GB
OS	Debian 10 (buster)	Arch Linux

Table 1: Configurations of our experimental machines.

decides on which core it should run. S_{move} defers the use of this chosen core to allow waking threads to take advantage of a core that is more likely to run at a high frequency.

Let T_{woken} be the newly created or waking task, C_{waker} the core where task T_{waker} that created or woke T_{woken} is running and C_{CFS} the destination core chosen by CFS. The normal behavior of the scheduler is to directly enqueue task T_{woken} into C_{CFS}'s runqueue. We propose to delay this migration to allow T_{woken} to be more likely to use a high-frequency core if C_{CFS} is running at a low frequency. First, if C_{CFS} is running at a frequency higher than the CPU's minimum one, we enqueue T_{woken} in C_{CFS}'s runqueue. Otherwise, we arm a high-resolution timer interrupt that will perform the migration in D μs and we enqueue T_{woken} into C_{waker}'s runqueue.The timer is cancelled if T_{woken} is scheduled on C_{waker}.

The rationale behind S_{move} is that we want to avoid waking low frequency cores if the task can be performed quickly when placed locally on a core that is likely to run at a high frequency. Indeed, T_{waker} is running at the time of the placement, meaning that C_{waker} is likely to run at a high frequency. The delay D can be changed at run time by writing to a parameter file in the `sysfs` pseudo file system. We have chosen a default value of 50 μs, which is close to the delay between a fork and a wait system call during our Linux kernel build experiments. We have found that varying the value of this parameter between 25 μs and 1 ms has insignificant impact on the benchmarks used in Section 4.

4 Evaluation

This section aims to demonstrate that our strategies improve performance on most workloads, while not degrading energy consumption. We run a wide range of applications from the Phoronix benchmark suite [23], the NAS benchmark suite [5], as well as other applications, such as `hackbench` (a popular benchmark in the Linux kernel scheduler community) and `sysbench OLTP` (a database benchmark). These experiments are run on a server-grade 4-socket NUMA machine with an 80-core Intel® CPU and on a desktop machine equipped with a 4-core AMD® CPU (Table 1). Both CPUs can select independent frequencies for each core[1] We have implemented S_{local} and S_{move} in the latest LTS kernel, Linux 5.4, released in November 2019 [3], and compare our strategies to Linux 5.4.

Implementing S_{local} (resp. S_{move}) only required modifying 3 (resp. 124) lines in CFS. We run all experiments 10 times. Energy consumption is evaluated on both machines using the Intel® RAPL [19] feature, which measures the energy consumption of the CPU socket and the DRAM. The performance results are those reported by each benchmark, and thus they involve different metrics, such as execution time, throughput, or latency, with inconsistent units. For better readability, all the following graphs show the improvement in terms of performance and energy usage compared to the mean of the runs with CFS. Therefore, higher is always better, regardless of the measured unit. The mean of the results for CFS is displayed on top of the graph for all benchmarks with the benchmark's unit.

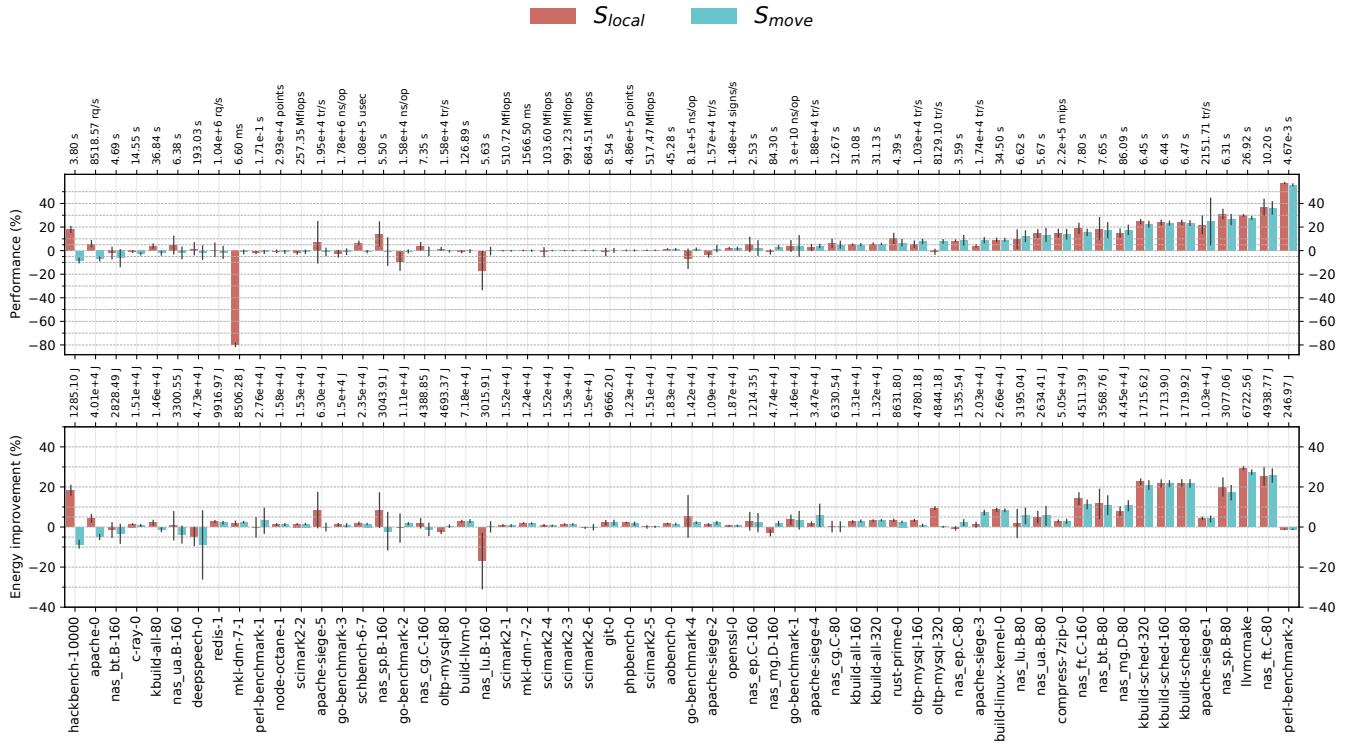
In Linux, frequency is controlled by a subsystem called a *governor*. On modern Intel® hardware, the `powersave` governor delegates the choice of the frequency to the hardware since it can perform more fine-grained adjustments. The hardware frequency-selection algorithm tries to save energy with a minimal impact on performance. The hardware estimates the load of a core based on various heuristics such as the number of retired instructions. This is the default governor for Intel® hardware on most Linux distributions. The `schedutil` governor, in development by the Linux community since Linux 4.7 (July 2016), tries to give control back to the operating system. It uses the internal data of the kernel scheduler, CFS, to estimate the load on each core, and changes the frequency accordingly. Two other governors, `performance` and `ondemand`, are available in Linux but are of no interest to us: the former runs all cores at the highest frequency, thus disabling dynamic scaling, while the latter is not supported on modern Intel® processors. To demonstrate that our work is orthogonal to the used governor, we evaluate our strategies using both `powersave` and the `schedutil`.

We first present the complete results on the Intel® server and summarize the results on the AMD® desktop machine. We then revisit our kernel build case study and study some worst case results (`mkl`, `hackbench`). Finally, we discuss the overhead of our S_{move} strategy.

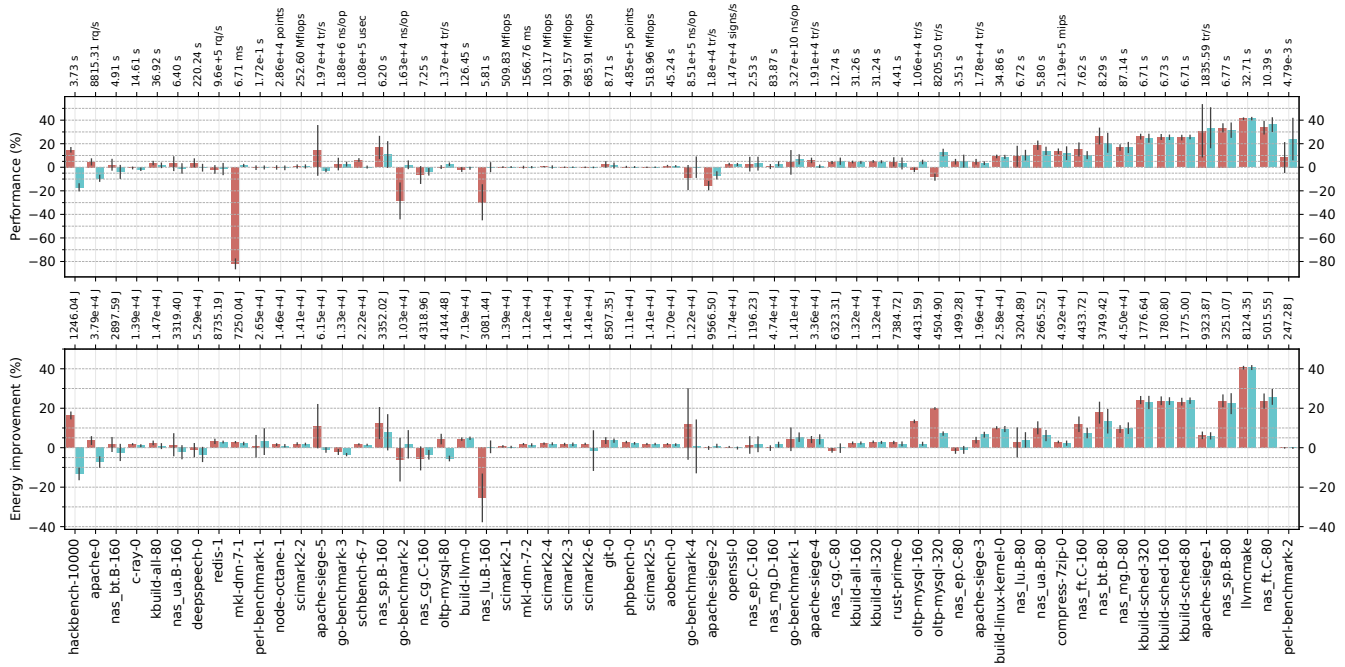
4.1 Execution Using `powersave`

We first consider the execution under `powersave`. Figure 4a shows the improvement in terms of performance and energy consumption of S_{local} and S_{move} as compared to CFS. We consider that improvements or deteriorations that do not exceed 5% to be on par with CFS.

[1] This is different from turbo frequencies: many desktop and laptop CPUs have per-core DFS in order to support turbo frequencies, but in practice, all cores not using the turbo range run at the same frequency.



(a) Comparison with CFS using the `powersave` governor.



(b) Comparison with CFS using the `schedutil` governor.

Figure 4: Performance and energy consumption improvement w.r.t. Linux 5.4 on the server machine (higher is better).

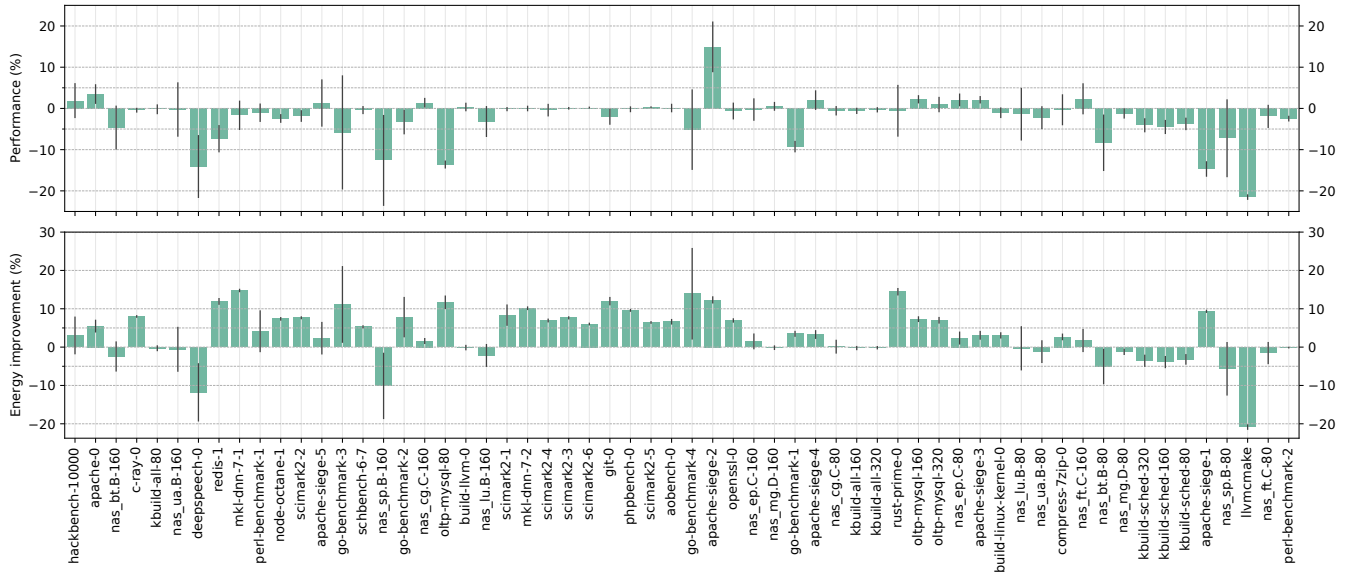


Figure 5: Performance of `schedutil` compared to `powersave` with CFS on the server machine.

Performance. Both S_{local} and S_{move} perform well overall with respectively 27 and 23 out of 60 applications outperforming CFS. The best results for these policies are seen, as expected, on benchmarks that extensively use the `fork/wait` pattern, and therefore exhibit a large number of frequency inversions. In the best case, S_{local} and S_{move} gain up to 58% and 56% respectively on `perl-benchmark-2`, that measures the startup time of the `perl` interpreter. This benchmark benefits greatly from avoiding frequency inversions since it mostly consists of `fork/wait` patterns. In terms of performance losses, both strategies deteriorate the performance of only 3 applications, but on very different scales. S_{local} deteriorates `mkl-dnn-7-1` by 80% and `nas_lu.B-160` by 17% while S_{move} has a worst case deterioration of 8.4% on `hackbench`.

Energy consumption. Overall, both S_{local} and S_{move} improve energy usage. Out of our 60 applications, we improve energy consumption by more than 5% for 16 and 14 applications, respectively, compared to CFS. Most of the improvements are seen on benchmarks where performance is also improved. In these cases, the energy savings are likely mostly due to the shorter execution times of the applications. However, we also see some improvements on applications where the performance is on par with that on CFS. This is due to the fact that we avoid waking up cores that are in low power states, therefore saving the energy necessary to power up and run those cores. In terms of loss, S_{local} consumes more energy than CFS on only one application, `nas_lu.B-160`. This loss is explained by the bad performance of S_{local} on this application. This benchmark's metric is its execution time, and increasing the execution time without correspondingly reducing the frequency increases the energy consumption. S_{move} consumes more energy than CFS on two applications: `hackbench`, because of the performance loss, and `deepspeech` that has too high a standard deviation for its results to have significance.

Overall score. To compare the overall impact of our strategies, we compute the geometric mean of all runs, where each run is normalized to the mean result of CFS. S_{move} has a performance improvement of 6%, a reduction in energy usage of 3% and an improvement of 4% with both metrics combined. S_{local} has similar overall scores (always 5%), but its worst cases suggest that S_{move} is a better option for a general-purpose scheduler. These small differences are expected because most of the applications we evaluate perform similarly with CFS and with our strategies. We also evaluate the statistical significance of our results with a t-test. With p-values of at most $3 \cdot 10^{-20}$, we deem our results statistically significant.

4.2 Execution Using `schedutil`

Next, we consider execution under the `schedutil` governor. As a baseline, Figure 5 first shows the performance and energy improvements of the `schedutil` governor compared to the `powersave` governor with CFS. Overall, we observe that the `schedutil` governor deteriorates the performance of most applications while improving energy usage. This indicates that this new governor is more aggressive in terms of power savings than the one implemented in hardware. We omit raw values since they are already available in Figures 4a and 4b. Figure 4b then shows the improvement in terms of performance and energy consumption of our strategies compared to CFS, when using the `schedutil` governor.

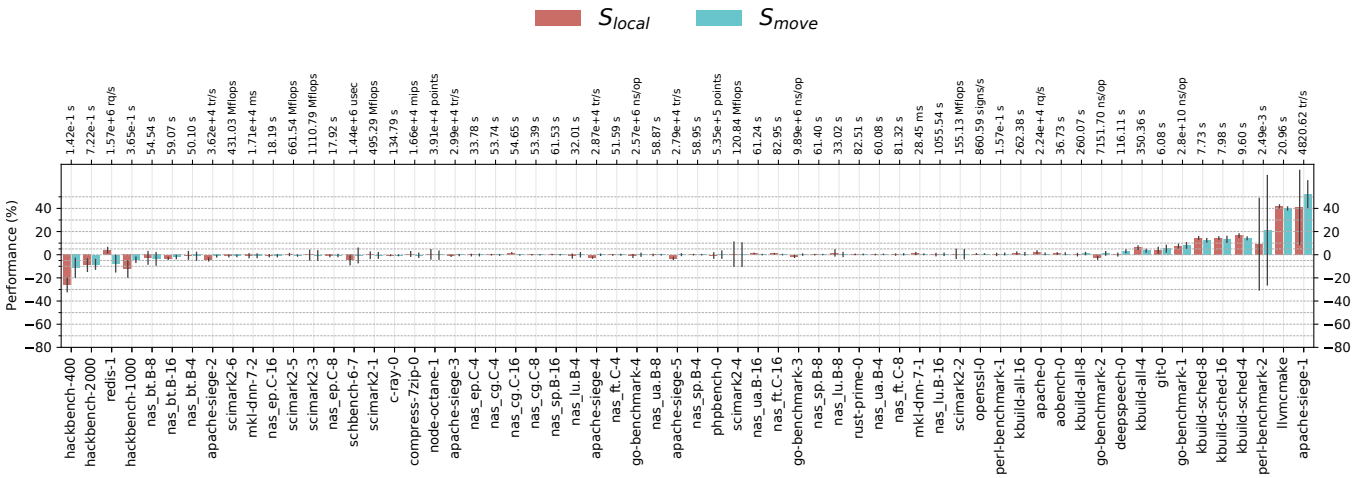


Figure 6: Performance improvement w.r.t. Linux 5.4 on the desktop machine (higher is better).

Performance. S_{local} and S_{move} outperform CFS on 22 and 20 applications out of 60 respectively. The applications concerned are the same that were improved with the `powersave` governor. In terms of performance losses, however, S_{local} is more impacted by the `schedutil` governor than S_{move}, with 7 applications performing worse than CFS versus only 2.

Energy consumption. The overall improvement in terms of energy usage of `schedutil` with CFS would suggest that we might see the same trend with S_{local} and S_{move}. And indeed, the results are quite similar to what we observe with the `powersave` governor.

Overall score. The geometric means with this governor are the following for `schedutil` and S_{move}: 6% for performance, 4% for energy and 5% with both metrics combined. S_{local} has similar results (2%, 6% and 4% respectively), but the worst cases are still too detrimental for a general-purpose scheduler. These results are also statistically significant with p-values of at most $3 \cdot 10^{-20}$.

4.3 Evaluation on the Desktop Machine

We also evaluate our strategies on the smaller 4-core AMD® desktop CPU presented in Table 1. In contrast to Intel® CPUs, the `powersave` governor on AMD® CPUs always uses the lowest available frequency, making it unusable in our context. We therefore use the `schedutil` governor on this machine.

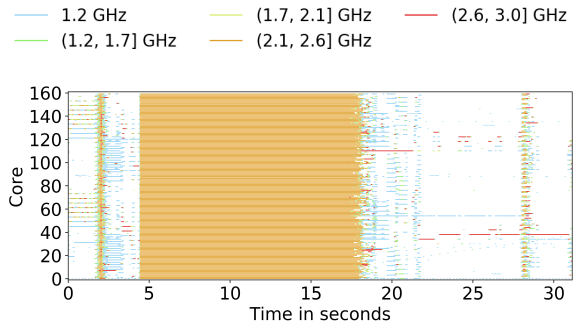
As shown in Figure 6, we observe the same general trend as on our server machine. S_{local} and S_{move} behave similarly when there is improvement, and S_{move} behaves better on the few benchmarks with performance degradation. We measure at worst an 11% slowdown and at best a 52% speedup for S_{move}, with an aggregate performance improvement of 2%. Additionally, S_{move} improves the performance of 7 applications by more than 5% while only degrading the performance of 4 applications at the same scale. The S_{local} strategy gives the same results regarding the number of improved and degraded applications, but suffers worse edge cases. Its best performance improvement is 42% while its worst deterioration is 25%, with an aggregate performance improvement of 1%. We conclude that even if there is no major global improvement, S_{move} is still a good strategy to eliminate frequency inversions on machines with smaller core counts. Our performance results are statistically significant, with p-values of $5 \cdot 10^{-4}$ for S_{move} and $3 \cdot 10^{-2}$ for S_{local}.

In terms of energy consumption, both S_{local} and S_{move} seem to have little to no impact as compared to CFS. However, the measures we were able to gather with all three strategies had a large variance that we did not observe on our Intel® CPU. We suspect that this is due to the energy-related hardware counters available on AMD® processors or the lack of good software support for these counters.

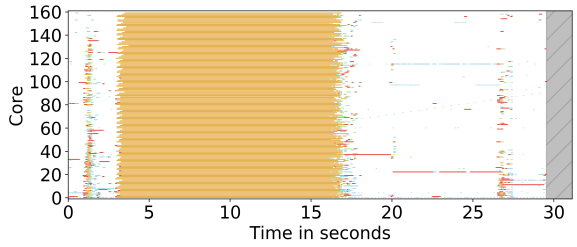
4.4 In-Depth Analysis

We now present a detailed analysis of specific benchmarks that either performed particularly well or particularly poorly with our solutions. In this section all traces were obtained with the `powersave` governor.

`kbuild` Figure 7 shows the execution of the build of the Linux kernel as presented in the case study, with CFS (top) and S_{move} (bottom). During the mostly sequential phases with multiple cores running at a low frequency on CFS (0-2 s, 2.5-4.5 s, 17-22 s), S_{move} uses fewer cores at a higher frequency. This is mainly due to the `fork()`/`wait()` pattern: as the waker thread calls `wait()` shortly after the `fork()`, the S_{move} timer does not expire and the woken threads remain on the local core running at a high frequency, thus avoiding fre-

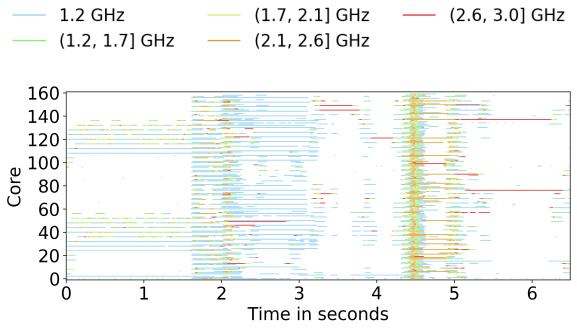


(a) CFS

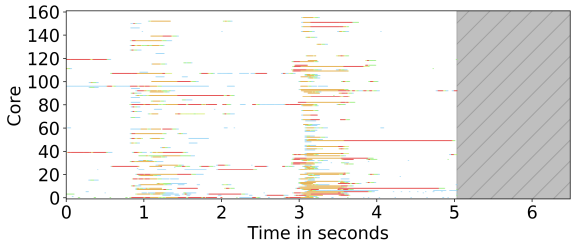


(b) S_{move}

Figure 7: Execution trace when building the Linux kernel version 5.4 using 320 jobs.



(a) CFS



(b) S_{move}

Figure 8: Execution trace when building the `sched` directory of the Linux kernel version 5.4 using 320 jobs.

quency inversion. As a result, for example, the phase before the long parallel phase is executed in 4.4 seconds on CFS and in only 2.9 seconds with S_{move}.

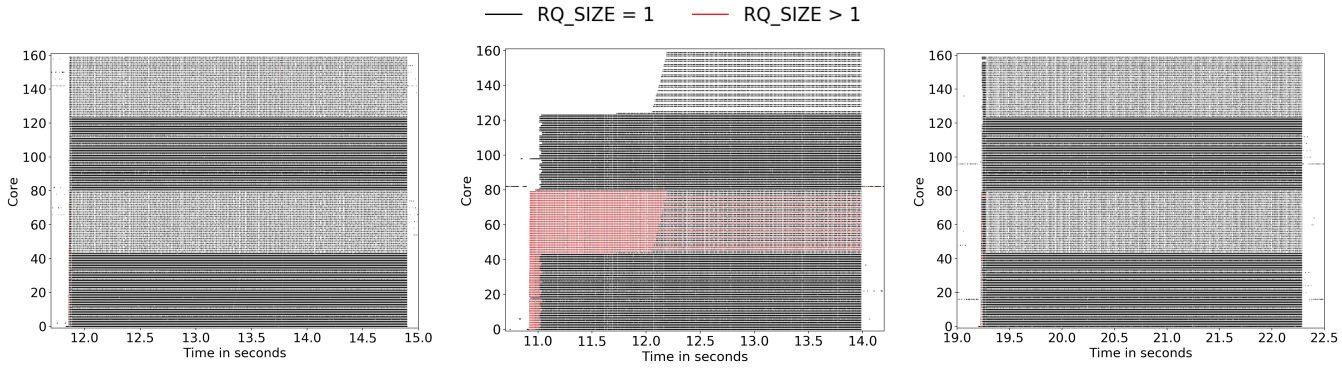
To understand the impact of S_{move} better, Figure 8 shows the `kbuild-sched-320` benchmark, which builds only the scheduler subsystem of the Linux kernel. Here, the parallel phase is much shorter than with a complete build, as there are fewer files to compile, making the sequential phases of the execution more visible. Again, we see that fewer cores are used, at a higher frequency.

`mkl` The `mkl-dnn-7-1` benchmark is the worst-case scenario for S_{local}: all threads keep blocking and unblocking and therefore avoid periodic load balancing and continue returning to the same set of cores. Thus, threads that are sharing a core with another thread will tend to remain there with the S_{local} strategy. Figure 9 shows the number of threads on the runqueue of each core with all three schedulers with the `powersave` governor. A black line indicates that there is one thread in the runqueue, and a red line indicates that there is more than one. CFS spreads the threads on all cores rapidly, and achieves a balanced machine with one thread per core in less than 0.2 seconds. On the other hand, S_{local} tries to maximize core reuse and oversubscribes 36 cores. This leads to never using all cores, achieving at most 85% CPU utilization with multiple cores overloaded. This is a persistent violation of the *work-conservation* property, as defined by Lozi *et al.* [21], *i.e.*, no core is idle if a core has more than one thread in its runqueue.

Interestingly, in our experiment, the balancing operations that spread threads are due to system or daemon threads (e.g. `systemd`) that wake up and block immediately, thus triggering an idle balancing from the scheduler. On a machine with nothing running in the background, we could have stayed in an overloaded situation for a long period of time, as ticks are deactivated on idle cores, removing opportunities for periodic balancing. We can see the same pattern on `nas-lu.B-160`, another benchmark that does not work well with S_{local}. S_{move} solves the problem by migrating, after a configurable delay, the threads that overload cores to available idle cores.

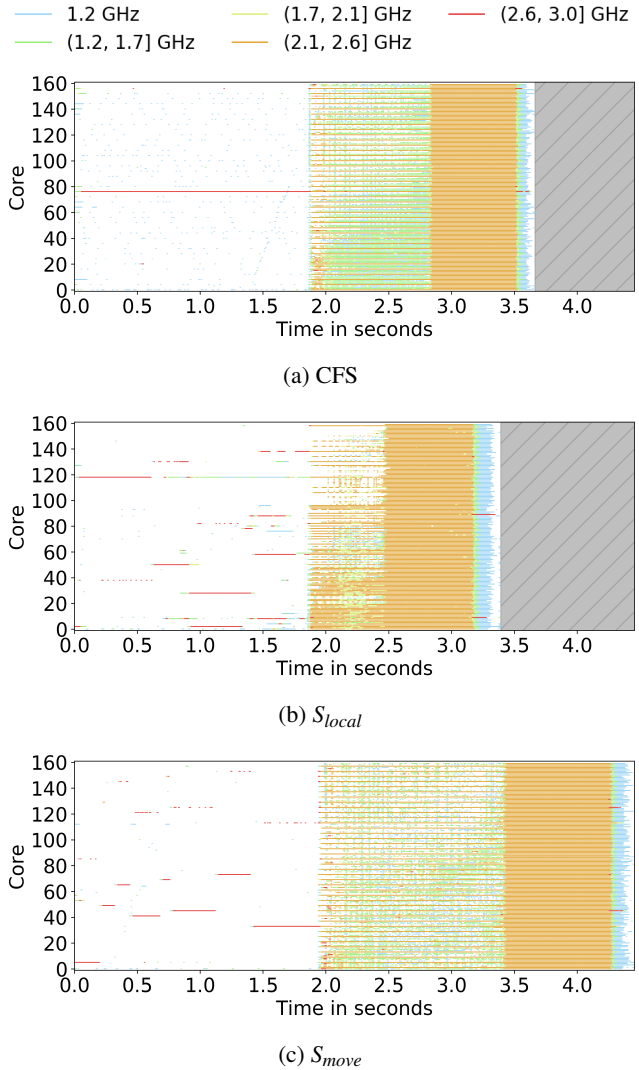
`hackbench` The `hackbench-10000` benchmark is the worst application performance-wise for the S_{move} strategy. This micro-benchmark is particularly stressful for the scheduler, with 10,000 running threads. However, the patterns exhibited are interesting to better understand the shortcomings of S_{move} and give insights on how to improve our strategies.

This benchmark has three phases: thread creation, communication and thread termination. Figure 10 shows the frequency of all cores during the execution of `hackbench` with CFS, S_{local} and S_{move}. The first phase corresponds to the first two seconds on all three schedulers. A main thread creates 10,000 threads with the `fork()` system call, and all child threads immediately wait on a barrier. With CFS, child threads are placed on idle cores that become idle again when the threads arrive at the barrier. This means that all cores remain



(a) CFS (b) S_{local} (c) S_{move}

Figure 9: Number of threads per core during the execution of `mkl-dnn-7-1`.



(a) CFS

(b) S_{local}

(c) S_{move}

Figure 10: Core frequency when executing `hackbench`.

mostly idle. This also leads to the main thread remaining on the same core during this phase. However, S_{local} and S_{move} place the child threads locally, causing oversubscription of the main thread's core and migrations by the load balancer. The main thread itself is thus sometimes migrated from core to core. When all threads are created, the main thread releases the threads waiting on the barrier and waits for their termination, thus beginning the second phase. During this phase, the child threads communicate by reading and writing in pipes. CFS tries to even out the load between all cores, but its heuristics give a huge penalty to migrations across NUMA nodes, so a single node runs at a high frequency (cores 0, 4, 8, etc. share the same node on our machine) while the others have little work to perform and run at lower frequencies. This phase finishes at 2.8 seconds. The remainder of the execution is the main thread reaping its children and terminating.

S_{local} packs threads aggressively, leading to long runqueues in the second phase, and therefore facilitating load balancing across nodes because of the large induced overload. However, S_{local} still does not use all cores, mainly avoiding running on hyperthreaded pairs of cores (cores n and $n+80$ are hyperthreaded on our machine). S_{local} runs the second phase faster than CFS, terminating it at 2.5 seconds, because it uses half of the cores at a high frequency all the time, and many of the other cores run at a medium frequency.

On the other hand, S_{move} performs poorly in the second phase, completing it at 3.4 seconds. The behavior seems very close to that of CFS, with one core out of four running at a high frequency. However, S_{move} results in more idleness or low frequency on the other cores. This is due to S_{move} placing threads locally: many threads contend for the local core; some are able to use the resource while others are migrated when the timer interrupt is triggered. The delays cause idleness compared to CFS, and the migrations leave cores idle, lowering their frequency compared to S_{local}. Additionally, when threads are migrated because of timers expiring, they are all placed on the same core, and oversubscribe it. For `hackbench`, choosing

the middle ground is the worst strategy. We can also note that load balancing is not able to mitigate this situation because of the high volatility of this workload. This problem was also demonstrated by Lozi et al. [21] on a database application.

This `hackbench` setup is an extreme situation that is unlikely to happen in real life, with a largely overloaded machine (10,000 threads) and a highly volatile application. This microbenchmark is only interesting to study the behavior of our strategies. Still, overall, S_{move} gives better performance than S_{local}.

4.5 Scheduling Overhead of S_{move}

S_{move} is more complex than S_{local}, and so we analyze its overhead as compared to CFS, as an upper bound for our strategies. We identify two possible sources of overhead: querying frequency and using timers.

First, we evaluate the cost of querying the core frequency. Querying the frequency of a core mostly consists in reading two hardware registers and performing some arithmetic operations, as the current frequency is the division of these two registers times the base frequency of the CPU. Even though this is a very small amount of computation compared to the rest of the scheduler, we minimize it furthermore by querying this information at every tick instead of every time it is needed. In our benchmarks, we notice no difference in performance with or without querying the frequency at every tick.

Second, we evaluate the cost of triggering a large number of timers in the scheduler. To do so, we run `schbench` on two versions of Linux: the vanilla 5.4 kernel and a modified version with timers armed under the same condition as S_{move}. Here, however, the timer handler does not migrate the thread as in S_{move}. We choose `schbench` because it performs the same workload as `hackbench` but provides, as a performance evaluation, the latencies of the messages sent through pipes instead of the completion time. Table 2 shows the results of this benchmark. Overall, the 99.5^{th} percentile of latencies is the same for both versions of the kernel, except for 256 threads where timers have a negative impact. We can also observe that the number of timers triggered increases with the number of threads but drops after 256 threads. This behavior is expected: more threads means more wake-ups, but when the machine starts being overloaded, all cores run at high frequencies, and the timers are less frequently armed. This tipping point arrives around 256 threads because `schbench` threads constantly block, meaning that typically fewer than 160 threads are runnable at a time.

Threads	Latency		Timers triggered
	vanilla	with timers	
64	78	77	2971
128	86	84	13910
192	119	144	63965
256	2292	3188	93001
512	36544	36544	512
768	60224	60480	959
1024	76416	76928	1290

Table 2: `schbench` latencies (99.5^{th} percentile, in μsec) and number of timers triggered.

5 Discussion

As previously stated, our proposed solutions S_{local} and S_{move} are purposefully simple. We now discuss other more complex solutions to the frequency inversion problem.

High frequency pool. A possible solution would be to keep a pool of cores running at a high frequency even though no thread is running on them. This would allow threads to be placed on an idle core running at a high frequency instantaneously. This pool could, however, waste energy and reduce the maximal frequency attainable by busy cores, which diminishes when the number of active cores increases.

Tweaking the placement heuristic. We could add a new frequency heuristic to the existing placement strategy. However, the tradeoff between using a core running at a higher frequency and e.g., cache locality is not clear, and may vary greatly according to the workload and the architecture.

Frequency model. The impact of the frequency of one core on the performance of other cores is hardware-specific. If the scheduler were to take frequency-related decisions, it would also need to account for the impact its decision would have on the frequency of all cores. Such models are not currently available, and would be complicated to create.

6 Related Work

Dynamic frequency scaling. Using DFS to reduce energy usage has been studied for over two decades. Weiser *et al.* [33] were the first to propose to adjust the frequency of the CPU according to its load, with the aim to maximize the *millions of instructions per joule* metric. Following this, in the early 2000s, Chase *et al.* [11] as well as Elnozahy *et al.* [17] proposed to reduce the frequency of underutilized servers in farms that exhibit workload concentration. Bianchini and Rajamony summarized these early works in a survey from 2004 [6]. Nowadays, on the hardware side, most CPUs support DFS, with the most recent series having elaborate hardware algorithms that are able to dynamically select very different frequencies for cores on the same chip, with technologies such as Enhanced Intel SpeedStep® [2] and AMD® SenseMI [4]. Despite this

shift of DFS logic from the software side to the hardware side in recent years, the decision to develop the experimental `schedutil` [9] governor in Linux was based on the idea that software still has a role to play in DFS, as it knows better the load being executed. Similarly, our strategies show that the software placing tasks on high-frequency cores can be more efficient than waiting for the hardware to increase the frequency of cores after task placement, due to the FTL.

Tracking inefficient scheduler behavior. Perf [15, 16, 32], which is provided with the Linux kernel, supports monitoring scheduler behavior through the `perf sched` command. While `perf sched` makes it possible to analyze the behavior of the scheduler on simple workloads with good accuracy, it has significant overhead on the Linux kernel build and other real-world workloads. Lozi et al. [21] identify performance bugs in the Linux scheduler. To analyze them, they write a basic profiler that monitors, for each core, the number of queued threads and the load. Their basic profiler does not monitor scheduling events. SchedLog and SchedDisplay [10], which we use in this paper, make it possible to record relevant information about all scheduler events with low overhead, and to efficiently navigate through the large amount of recorded data with a powerful and scriptable graphical user interface.

Mollison et al [25] apply regression testing to schedulers. Their focus is limited to real-time schedulers, and they do not take DFS into account. More generally, there has been an ongoing effort to test and understand the impact of the Linux scheduler on performance. Since 2005, the LKP project [12] has focused on hunting performance regressions, and a myriad of tools that make it possible to identify performance bugs in kernels have been proposed by the community [7, 18, 26, 28]. The focus of these tools, however, is to detect slowdowns inside the kernel code, and not slowdowns in application code that were caused by decisions from the kernel. Consequently, they are unable to detect poor scheduling behavior.

Improving scheduler behavior. Most previous work focuses on improving general-purpose OS scheduling with new policies that improve a specific performance metric, such as reducing contention over shared resources [31, 35], optimizing the use of CPU caches [29, 30], improving NUMA locality [8, 14] or minimizing idleness [20]. These papers systematically disable DFS in their experiments. Merkel *et al.* [24] propose a scheduling algorithm that avoids resource contention by co-scheduling applications that use complementary resources. They reduce contention by lowering the frequency of cores that execute inauspicious workloads. Zhang et al. [34] propose a scheduling policy for multi-core architectures that facilitates DFS, although their main focus is reducing cache interference. They only consider per-chip DFS, as per-core DFS was not commonplace at the time.

Linux kernel developers have recently focused on DFS and turbo frequencies [13], as it was discovered that a short-lived jitter process that runs on a previously idle core can make that core switch to turbo frequencies, which can in turn reduce the frequencies used by other cores—even after the jitter process completes. To solve this issue, a patch [27] was proposed to explicitly mark jitter tasks. The scheduler then tries to place these marked tasks on cores that are active and expected to remain active. In contrast, the frequency inversion issue we identified is not specifically caused by turbo frequencies: it can occur with any DFS policy in which different cores may run at different frequencies.

Child runs first. CFS has a feature that may seem related to our solutions: `sched_child_runs_first`. At thread creation, this feature assigns a lower *vruntime* to the child thread, giving it a higher priority than its parent. If CFS places the thread on the same core as its parent, the thread will preempt the parent; otherwise, the thread will just run elsewhere. This feature does not affect thread placement and thus cannot address the frequency inversion problem. Using this feature in combination with S_{move} would defeat S_{move}'s purpose by always canceling the timer. The strategy would resemble S_{local}, except that the child thread would always preempt its parent.

7 Conclusion

In this paper, we have identified the issue of frequency inversion in Linux, which occurs on multi-core CPUs with per-core DFS. Frequency inversion leads to running tasks on low-frequency cores and may severely degrade performance. We have implemented two strategies to prevent the issue in the Linux 5.4 CFS scheduler. Implementing these strategies required few code changes: they can easily be ported to other versions of the Linux kernel. On a diverse set of 60 applications, we show that our better solution, S_{move}, often significantly improves performance. Additionally, for applications that do not exhibit the frequency inversion problem, S_{move} induces a penalty of 8% or less with 3 of the evaluated applications. As independent core frequency scaling becomes a standard feature on latest generation processors, our work will target a larger number of machines.

In future work, we want to improve thread placement in the scheduler by including the cores' frequencies directly in the placement algorithm. This improvement will need to account for various parameters such as architecture-specific DFS, simultaneous multi-threading and maintaining cache locality.

Acknowledgments and Availability

This work is supported in part by Oracle donation CR 1930. We would also like to thank the anonymous reviewers and our shepherd, Heiner Litz, for their feedback.

S_{local} and S_{move} patches for Linux 5.4 are available at: https://gitlab.inria.fr/whisper-public/atc20.

References

[1] frequency_logger. https://github.com/rgouicem/frequency_logger.

[2] Intel®. Frequently Asked Questions about Enhanced Intel SpeedStep® Technology for Intel® Processors. https://www.intel.com/content/www/us/en/support/articles/000007073/processors.html.

[3] Linus Torvalds' official git repository. https://github.com/torvalds/linux.

[4] AMD®. SenseMI Technology. https://www.amd.com/en/technologies/sense-mi.

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

[6] Ricardo Bianchini and Ram Rajamony. Power and energy management for server systems. *Computer*, 37(11):68–76, 2004.

[7] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of Linux scalability to many cores. In *OSDI*, pages 86–93, Vancouver, BC, Canada, 2010.

[8] Timothy Brecht. On the importance of parallel application placement in NUMA Multiprocessors. In *USENIX SEDMS*, San Diego, CA, USA, 1993.

[9] Neil Brown. Improvements in CPU frequency management. https://lwn.net/Articles/682391/.

[10] Damien Carver, Redha Gouicem, Jean-Pierre Lozi, Julien Sopena, Baptiste Lepers, Willy Zwaenepoel, Nicolas Palix, Julia Lawall, and Gilles Muller. Fork/wait and multicore frequency scaling: a generational clash. In *PLOS*, pages 53–59, Huntsville, ON, Canada, 2019. ACM.

[11] Jeffrey S Chase, Darrell C Anderson, Prachi N Thakar, Amin M Vahdat, and Ronald P Doyle. Managing energy and server resources in hosting centers. *ACM SIGOPS operating systems review*, 35(5):103–116, 2001.

[12] Tim Chen, Leonid I Ananiev, and Alexander V Tikhonov. Keeping kernel performance from regressions. In *Linux Symposium*, pages 93–102, Ottawa, ON, Canada, 2007.

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

[14] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic management: a holistic approach to memory placement on NUMA systems. In *ASPLOS*, pages 381–394, Houston, TX, USA, 2013.

[15] Arnaldo Carvalho de Melo. Performance counters on Linux. In *Linux Plumbers Conference*, Portland, OR, USA, 2009.

[16] Arnaldo Carvalho de Melo. The new Linux 'perf' tools. In *Slides from Linux Kongress*, Nuremberg, Germany, 2010.

[17] EN Mootaz Elnozahy, Michael Kistler, and Ramakrishnan Rajamony. Energy-efficient server clusters. In *PACS*, pages 179–197, Cambridge, MA, USA, 2002. Springer.

[18] Ashif S. Harji, Peter A. Buhr, and Tim Brecht. Our troubles with Linux and why you should care. In *APSys*, pages 1–5, Shanghai, China, 2011.

[19] Intel®. Intel® and 64 and IA-32 architectures software developer's manual. *Volume 3B: System programming Guide, Part 2, Chapter 14.9*, page 5, 2011.

[20] Baptiste Lepers, Redha Gouicem, Damien Carver, Jean-Pierre Lozi, Nicolas Palix, Maria-Virginia Aponte, Willy Zwaenepoel, Julien Sopena, Julia Lawall, and Gilles Muller. Provable multicore schedulers with ipanema: application to work conservation. In *EuroSys*, pages 3:1–3:16, Heraklion, Greece, 2020. ACM.

[21] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The Linux scheduler: a decade of wasted cores. In *EuroSys*, pages 1–16, London, UK, 2016.

[22] Abdelhafid Mazouz, Alexandre Laurent, Benoît Pradelle, and William Jalby. Evaluation of CPU frequency transition latency. *Comput. Sci. Res. Dev.*, 29(3-4):187–195, 2014.

[23] Phoronix Media. Phoronix test suite – Linux testing & benchmarking platform, automated testing, open-source benchmarking. http://www.phoronix-test-suite.com/.

[24] Andreas Merkel, Jan Stoess, and Frank Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. In *EuroSys*, pages 153–166, Paris, France, 2010. ACM.

[25] Malcolm S Mollison, Björn Brandenburg, and James H Anderson. Towards unit testing real-time schedulers in LITMUS^{RT}. In *OSPERT*, Stuttgart, Germany, 2009.

[26] Sharon E. Perl and William E. Weihl. Performance assertion checking. In *SOSP*, pages 134–145, Asheville, NC, USA, 1993.

[27] Parth Shah. TurboSched: A scheduler for sustaining turbo frequencies for longer durations, June 25, 2019. https://lkml.org/lkml/2019/6/25/25.

[28] Kai Shen, Ming Zhong, and Chuanpeng Li. I/O system performance debugging using model-driven anomaly characterization. In *FAST*, pages 309–322, San Francisco, CA, USA, 2005.

[29] David Tam, Reza Azimi, and Michael Stumm. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *EuroSys*, pages 47–58, Lisbon, Portugal, 2007.

[30] Lingjia Tang, J. Mars, Xiao Zhang, R. Hagmann, R. Hundt, and E. Tune. Optimizing Google's warehouse scale computers: The NUMA experience. In *HPCA*, pages 188–197, Shenzhen, China, 2013.

[31] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *EuroSys*, pages 1–17, Bordeaux, France, 2015.

[32] Vincent M Weaver. Linux perf_event features and overhead. In *FastPath*, pages 80–87, Austin, TX, 2013.

[33] Mark Weiser, Brent B. Welch, Alan J. Demers, and Scott Shenker. Scheduling for reduced CPU energy. In *OSDI*, pages 13–23, Monterey, CA, USA, 1994.

[34] Xiao Zhang, Sandhya Dwarkadas, and Rongrong Zhong. An evaluation of per-chip nonuniform frequency scaling on multicores. In *USENIX ATC*, Berkeley, CA, USA, 2010.

[35] Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Computing Surveys (CSUR)*, 45(1):4, 2012.