OS Scheduling with Nest: Keeping Threads Close Together on Warm Cores

Anonymous Author(s)
Submission Id: Paper 406

Abstract

To best support highly parallel applications, Linux's CFS scheduler tends to spread threads across the machine on thread creation and wakeup. It has been observed, however, that in a server environment, such a strategy leads to threads being placed on long-idle cores that are running at lower frequencies, which reduces performance, and to threads being distributed across sockets, which consumes more energy. In this paper, we propose to exploit the principle of core reuse, by constructing a nest of cores to be used for thread scheduling in priority, thus obtaining higher frequencies and using fewer sockets. We implement the Nest scheduler in the Linux kernel. While performance and energy usage are comparable to CFS for highly parallel applications, for a variety of applications that use fewer threads than cores, Nest gives performance improvements of up to almost 2x and can reduce energy usage.

ACM Reference Format:
Anonymous Author(s). 2021. OS Scheduling with Nest: Keeping Threads Close Together on Warm Cores. In *Proceedings of EuroSys 2022: The European Conference on Computer Systems (EuroSys '22).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

The primary goal of an operating system (OS) thread scheduler is to allocate threads to cores in a way that maximizes application performance. A well-known desirable property is *work conservation*, *i.e.*, if a thread is placed on a core that is not idle, then no idle core should be available [10, 11]. However, in choosing a core for a thread, it is also important to consider whether the chosen core will allow the thread to access needed (hardware) resources efficiently. The performance that a thread can achieve is determined in part by the *frequency* of the chosen core [6]. On modern CPUs, core frequencies may vary significantly, as individual cores can adjust their frequency independently. Nevertheless, the Linux kernel's default CFS scheduler does not take core frequency into account. Placing threads on cores in a way that causes higher frequencies to be used can improve performance.

We consider scheduling on large multicore servers. Large multicore servers are today becoming more accessible and affordable. Such machines can be used for traditional high-performance computing, where applications are often designed to decompose to the number of cores available, so that threads are pinned to cores, making scheduling irrelevant. But multicore servers can also be used as computing resources for applications that are demanding in terms of compute cycles, memory, or disk requirements. Such applications rely on the OS thread scheduler for thread placement. The number of cores required may vary from few to moderate to many, and back, across the course of the application. To get the best performance, the OS thread scheduler must be able to optimally adapt to all of these situations.

Modern servers offer "turbo" frequencies [1, 8] allowing cores to run at a frequency higher than the nominal frequency. Various turbo frequencies are available, depending on the number of active cores on the socket, to respect thermal constraints. The choice of frequency is determined jointly by the software and the hardware. The software, typically an OS kernel-level power governor, suggests boundaries, and then the hardware chooses a frequency for a core within these boundaries according to the degree of activity observed on the core and the number of cores on the same socket that are currently active. In order to obtain the highest possible frequencies, it is thus necessary to minimize the number of cores used ("keeping threads close together") and ensure a sustained activity ("keeping cores warm").

In this paper, we propose the thread scheduler Nest, designed according to the principles *reuse cores* and *keep cores warm*. To increase core reuse, Nest tries to place threads within a set of recently used cores (the *nest*). To keep cores warm, with Nest, the idle process spins on a newly idle core for a short period, to encourage the hardware to keep the frequency high. Finally, when a thread must be placed outside the nest because another thread is using its previous core, Nest remembers this previous core, and attempts to return the thread to the same core in the nest the next time a core is chosen for the thread. On purely sequential applications and applications with as many or more threads than cores, Nest performs similarly to CFS. Nest is particularly beneficial for applications with a moderate number of effective concurrent threads, and where threads fork, block, and terminate frequently, resulting in many thread placements.

We implement Nest within the Linux kernel, by modifying the CFS scheduler. Recent work has proposed frameworks for pluggable schedulers, at the kernel level [10] and at user level [7]. The Linux kernel community seems opposed to pluggable schedulers.[1] While Nest could be implemented at

[1]"I absolutely *detest* pluggable schedulers." – Linus Torvalds [17]

user level, it relies on a fast turnaround between selection of a core and placement of a thread on that core, which might not be achievable in that setting.

Our contributions are as follows:

- We motivate two new principles for thread schedulers: reuse cores, and keep the cores warm and an implementation of these principles in the Linux Kernel.
- We show performance improvements with Nest on a wide range of multicore benchmarks, on four 2- or 4-socket multicore machines, including improvements on a 4-socket Intel Xeon 6130 of more than 20% on 8% of our more than 200 Phoronix multicore tests.
- We show that these performance improvements can also reduce CPU power usage by up to 20%, on our software configuration benchmark.
- Nest often achieves performance comparable to or better than the Linux kernel's *performance* power governor, which requests the use of at least the nominal frequency, while using the *schedutil* power governor, which allows the use of lower frequencies in periods of light activity.

The rest of this paper is organized as follows. Section 2 presents background about the Linux kernel's CFS scheduler and power management in the Linux kernel. Section 3 presents Nest, and Section 4 presents our implementation of Nest in the Linux kernel. Section 5 evaluates Nest on multicore benchmarks. Finally, Section 6 presents some related work and Section 7 concludes.

Terminology. We say *concurrent threads* for the set of threads that are running at a given point in time. This does not include any threads that are sleeping or waiting to run. We evaluate Nest on Intel servers, which offer simultaneous multithreading, where multiple virtual cores share a single physical core. For simplicity, we refer to the number of cores on a machine as the number of virtual cores. We say that one core is a *hyperthread* of another core if both share the same physical core. Our target hardware offers only two cores per physical core.

2 Background

We present Linux's Completely Fair (CFS) scheduler (Linux v5.9), that Nest extends, as well as a recent scheduler, S_{move}, that also targets better use of core frequencies. We then present Linux's power governors that influence the frequency changes at the hardware level.

2.1 Linux's CFS scheduler

CFS uses a collection of heuristics to place threads on cores on thread fork, thread exec, thread wakeup, and load balancing. Fork and wakeup are most relevant to Nest. We present the most commonly used heuristics, which rely on the Linux *scheduling domains*. Fork and wakeup represent around 1300 lines of code, out of the more than 11K lines of code in `fair.c`, the main file implementing CFS.

Scheduling domains. The Linux kernel scheduler views the available CPUs according to a hierarchy of *domains*, organized into levels: typically NUMA (all cores on the machine), SMP (cores sharing a last-level cache), and SMT (hyperthreads sharing the same physical core), from highest to lowest. Each core is associated with the sequence of domains that contain it. Each domain refers to a list of *groups*, which comprise the list of cores associated with each of its child domains.

Fork. Starting from the highest domain, CFS first searches for the least loaded associated group, and then for the least loaded core within that group. The load, of a group or a core, is characterized by a variety of criteria, including the number of idle cores, the recent load on the cores in the domain, the expected time to wake from idle states, and the new thread's NUMA preferences. Cores are numbered by successive integers. The search for a core within a group is carried out in numerical order, modulo the number of cores available, starting from the core performing the fork. When a core is selected, the search repeats with the child domain containing the chosen core.

The fact that the search for a core is carried out across the entire machine favors work conservation. On the other hand, the fact that the search is carried out in a fixed order means that recently used cores may be overlooked. The fact that recent load is taken into account also hinders reuse, as it disfavors idle cores that have been recently used.

Wakeup. CFS first selects a *target core*, that is either the woken thread's previous core or the core performing the wakeup. Heuristics are used to choose between them, including core idleness, the type of wakeup, and the recent load on the core. CFS then searches for an idle core among the cores in the SMP domain of the target. First it searches for a core such that both the core and its hyperthread are idle. If none is found, it searches through a few cores to find one that is idle. If this search also fails, CFS checks the hyperthread of the target. If all previous searches fail, CFS selects the target.

Wakeup is not work conserving, as it only considers the cores associated with a single SMP domain, and it only makes a limited effort to find an idle core therein. Wakeup traverses the cores in a fixed order, and thus it may overlook recently used idle cores. On the other hand, it does not take recent load into account in the final core choice, and thus recently used idle cores are not disfavored.

2.2 The S_{move} scheduler, targeting core frequency

Gouicem *et al.* [6] identified the problem of *frequency inversion*, in the common case where a thread T_{parent} forks or wakes another thread T_{child} and then immediately sleeps to wait for T_{child}'s results. T_{parent}'s core is likely running at a high frequency, while CFS will place T_{child} on an idle core if one is available. T_{parent} will thus be delayed until T_{child} completes on an initially low-frequency core, while T_{parent}'s former high frequency core is available. Gouicem *et al.* showed that frequency inversion incurs a slowdown on a variety of applications.

Gouicem *et al.* proposed the scheduler S_{move} that places T_{child} on the core of T_{parent}, allowing T_{child} to benefit from that core's high frequency. If T_{parent} does not immediately block or other threads are waiting, T_{child} may then incur high latency. Accordingly, S_{move} only makes this placement when the core chosen by CFS has a low frequency and sets a timer for T_{child}, to move T_{child} to the core chosen by CFS if it is not scheduled on the core of T_{parent} within a brief delay. When the timer expires, however, S_{move} does nothing to ensure that the thread ends up on a core with a high frequency.

2.3 Linux's power governors

The scheduler has no control over core frequencies. Instead, core frequency results from an interplay between the Linux power governor and the hardware. The governor sets the bounds in which the frequency should vary and can make suggestions about what frequency should be used. The hardware combines the information from the governor with its observations about the current activity on the core and the core's socket, and chooses a frequency for the core accordingly. The power governor has a significant impact on performance for many applications. Thus, when we refer to a scheduler, we refer to the used governor as well. We consider the *performance* and *schedutil* governors, which are available on most Linux systems and represent distinct strategies.

Performance. *Performance* requests that the hardware use the nominal frequency of the machine. The hardware can still freely choose between the nominal frequency and the turbo frequencies. The *performance* governor gives threads high performance, but misses the potential energy savings that can be obtained by running non-demanding threads at lower frequencies.

Schedutil. *Schedutil* takes into account information from the scheduler about recent thread activity, to attempt to reconcile performance and energy usage. It allows the machine to use its full range of frequencies. When *schedutil* observes that the threads on a core have a high recent CPU utilization, it suggests to the hardware to increase the frequency.

3 The Nest Approach

The key idea behind Nest is the use of a *nest*, defining a limited set of recently used cores to consider in high priority when placing a thread. By limiting activity to a small number of cores, Nest encourages the hardware to choose a high core frequency. The challenge in creating a scheduler around this idea is to design heuristics that properly dimension the

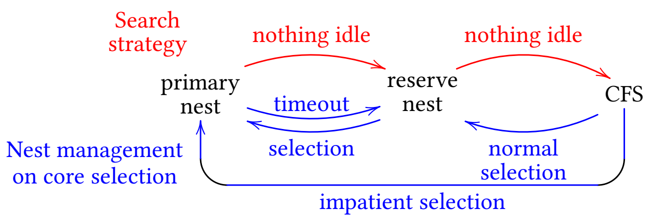


Figure 1. Nest management

nest according to applications' current needs. A nest that is too large will result in thread dispersal, replicating the dispersal problem of CFS. A nest that is too small will result in threads competing for the same cores, inducing overloads.

3.1 Building the nest

As shown in Figure 1, Nest keeps track of two sets of cores (nests), to consider in high priority for thread placement. Cores in the *primary nest* have been used recently and are expected to be useful in the near future. Cores in the *reserve nest* were previously in the primary nest but have not been used recently and thus are considered to be less likely to be used in the near future, or have recently been selected by CFS and have not yet proved their necessity for the current set of threads.

The top of Figure 1 (red arrows) indicates the core-search heuristic. For a forking or waking thread, Nest first searches for an idle core in the primary nest, then if none is found it searches for an idle thread in the reserve nest. If that also fails, then it falls back on CFS. The search in the primary nest starts at the thread's previous core (or the core of the parent in the case of a fork), to reduce the risk of collision with concurrent forks and wakeups on other cores. The search in the reserve nest, which is expected to be accessed less often, starts from a fixed core, chosen arbitrarily as the core of the thread that initiates the use of Nest, to reduce thread dispersal. In both cases, the search first considers cores in the same socket as the thread's previous core (or the core of the parent in the case of a fork), before considering cores in the other sockets. This heuristic reduces the number of used sockets, thus increasing the chance of leaving some sockets completely idle and saving energy. Unlike CFS, Nest selects any core that is found to be currently idle, without taking into account recent load, in order to favor core reuse. Recent load may suggest that the thread causing that load may soon be scheduled again, but this is highly application dependent, and Nest address the problem in a different way (Section 3.3). Also unlike CFS, Nest does not take into account activity on hyperthreads. Nevertheless, all cores that newly enter the nests are initially chosen by CFS when there are no idle cores in the nests, and thus they inherit CFS's selection criteria.

The bottom of Figure 1 (blue arrows) indicates how cores move between the nests, to allow the nest size to adapt to the number of concurrent threads. If a core is selected from the reserve nest, the core is promoted to the primary nest. If a core is selected when nest falls back to CFS, in the normal case it is placed in the reserve nest, if space is available. On the other hand, if a core in the primary nest has not been used for some time (at most P_{remove} ticks), it is demoted to the reserve nest (*nest compaction*) as soon as a thread tries to use it. Nest compaction is not applied to the reserve nest, due to its bounded size. Finally, if a thread terminates on a core and the core becomes idle, then the core is considered no longer useful and it is immediately demoted from the primary nest to the reserve nest.

It may occur that too many threads are trying to share the primary nest. In this case, a thread may bounce between cores, if it often wakes up to find that the core on which it was running previously is occupied by another thread. If a thread finds that its previous core is not idle $R_{impatient}$ times, then the thread is labelled as *impatient.* To place such a thread, Nest does not explore the primary nest, but rather turns to the reserve nest and potentially CFS. In either case, the chosen core is directly added to the primary nest.

3.2 Keeping the cores in the nest warm

Our goal of keeping cores at a high frequency is hampered by the fact that threads often briefly pause, for synchronization or to wait for I/O. These pauses may cause the hardware to decrease the frequency, which is not desirable if the core will be frequently used. Accordingly, we modify the idle process, to spin for up to a small number of ticks (S_{max}) when a thread blocks. The spin continually checks whether there is a thread on the core's hyperthread; if one appears the spinning stops immediately, leaving it to the thread running on the hyperthread to keep the core warm.

3.3 Returning to the nest

It is desirable for a waking thread to return to its previous core; moving the thread to another core may in turn cause another thread to find its previous core occupied, triggering a cascade of thread migrations. At the same time, to keep the nest small and improve core reuse, we do not want to permanently place a thread on a core outside the primary nest just because the thread happens to wake up at the same time as *e.g.*, a brief daemon thread. Accordingly, in addition to recording the core used on the previous execution of the thread, as done by CFS, Nest also records the core used on the execution before that, creating a history of size 2. If the cores used in the two previous executions are the same, that is, if the core has once succeeded in returning to its previous core, the thread is considered to be *attached* to that core. The first choice of a thread is always the core that it is attached to, if any, if that core is in the primary nest and is idle. A thread can even reclaim a core that is in the primary nest and is eligible for nest compaction, as long as no other thread has tried to use the core in the meantime.

3.4 Other issues

The primary nest is most effective when its size corresponds to the number of threads that are trying to run concurrently, which means that work conservation (no thread waiting on a busy core when some core is idle) is an important property. To reduce wakeup latency, CFS is not work-conserving on thread wakeups: it only considers the previous socket of the waking thread and the socket of the waker as possible sources of idle cores. In the context of Nest, such a strategy hinders the primary nest in reaching the optimal size. To propagate threads more quickly to cores where they can run with minimal interference from other threads, Nest enhances the degree of work conservation of CFS. Specifically, Nest extends the CFS core selection on thread wakeup by examining all of the sockets, if searching the socket chosen by CFS does not result in an idle core.

The Linux kernel places a forked or waking thread on a core in two steps: first selecting a core, and then adding the thread to the chosen core's run queue. The absence of a global lock in the first step means that cores that are forking or waking threads at the same time can choose the same core for their threads, leading to overload and degrading performance. This problem arises with CFS, but is exacerbated with Nest, due to the smaller number of cores considered. To prevent such a collision, Nest associates a flag with each run queue, indicating whether it has placed a thread has been placed on the corresponding core. This flag is checked using a compare-and-swap instruction, ensuring that the Nest heuristics place at most one thread on a given core. This optimization could be applied to CFS independently of Nest, but we expect less impact, as the problem is more rare.

4 Implementation

We have implemented Nest in Linux v5.9.[2] The implementation involves adding around 500 lines of code, across 6 files, with most of the changes in the implementation of CFS (kernel/sched/fair.c) and the scheduler core (kernel/sched/-core.c). For simplicity in our prototype, we appropriate an existing system call to allow a thread to indicate that it, and all of its children, should be scheduled with Nest.

Implementing Nest requires choosing values for the various thresholds mentioned in Section 3. Table 1 shows the values used. Notably, P_{remove}, the delay before a core is ejected from the primary mask, is small so that the primary mask can closely track the current number of runnable threads, up to the number of cores on the machine. S_{max}, the spinning duration, is also small to reduce the chance that spinning on

[2]Linux v5.9 was the latest kernel version at the time of our initial experiments. We also ported Nest to Linux v5.12, in around 1 hour. However, we later learned that the Linux kernel developers had found some errors in the Linux v5.12 code controlling thread placement, and returned to Linux v5.9 as a more accurate implementation of the CFS policy.

Table 1. Chosen values of the NEST parameters

Parameter	Description	Value
P_{remove}	Delay before removing an idle core from the primary nest	2 ticks
R_{max}	Maximum number of cores in the reserve nest	5
$R_{impatient}$	Number of successive placement failures tolerated before trying to expand the primary nest	2
S_{max}	Maximum spin duration	2 ticks

idle cores delays active cores on the same socket in reaching the highest turbo frequencies.

5 Evaluation

Evaluating a scheduler requires testing its impact on applications that exhibit a high diversity of thread behaviors. To comprehensively evaluate NEST, we compare its behavior to that of CFS on a wide range of benchmarks, including applications that are nearly single threaded, that use one thread per core, and that have more variable thread behaviors. NEST particularly targets applications in which the number and set of concurrent threads varies over time. We aim to show that NEST improves the performance of such applications and has negligible impact (±5%) on others. We evaluate both performance and energy consumption.

5.1 Evaluation Setting

The baseline for our experiments is the Linux kernel v5.9's CFS scheduler using the *schedutil* power governor.

Hardware. Table 2 describes the machines used. These reflect three generations of Intel Xeon architectures, released between 2016 and 2019, with 2 and 4 sockets, and 16 or 20 cores per socket. The machines have been chosen to illustrate the evolution in performance and power management, and to illustrate a variety of machine classes. We consider 2-socket 64-core machines from different generations (Skylake and Cascade Lake), 4-socket machines of different generations (Broadwell and Skylake), and the same model of Skylake with 2 and 4 sockets. Table 3 indicates the turbo frequencies that can be achieved on these machines, according to the number of active cores on a given socket.

All of our test machines have multiple sockets, amounting to multiple NUMA nodes. We have not made any effort to control the memory allocation across these NUMA nodes.

Measurements. For performance tests, we perform two warmup runs and then average over 10 runs. As we observe low standard deviations in the relevant performance results, frequency traces are collected from one run. Power measurements are over 30 runs, to reduce their standard deviation. The bar graphs that compare an average value for a given scheduler to the corresponding average value for CFS-schedutil, include error bars. These represent the standard deviation of the improvement, computed by comparing each value obtained by the given scheduler with the average value obtained for CFS-schedutil. All speedups are normalized such that 0 represents identical performance, percentages greater than 0 represent improvements, and percentages below 0 represent degradations.

5.2 Software configuration test suite evaluation

Typical software configuration code is based on shell scripts, and forks off hundreds or even thousands of threads, many running alone and with a short lifespan, to test various conditions and to attempt to run programs that the software may require. Developers may run the configuration script frequently, to test the software in different environments, and thus care about its performance. Continuous integration systems may run the configuration script repeatedly. The frequent forking of short-lived threads that mostly run alone makes software configuration an ideal case for NEST.

Case study. We first illustrate the behavior of CFS on the first 0.3 seconds of the configuration of LLVM using `cmake`, for compilation using Ninja. Figure 2(a) shows the activation of the various threads and their frequencies with CFS on the 5218.[3] Starting from the thread in the lower left, threads are forked and are placed on cores with increasing core numbers. This even occurs when cores closer to the starting thread are idle; CFS prefers a fully idle core to one that has recently been used. The threads end up dispersed across 8 cores. Even though only one or two threads run at a time, the processor is not able to react quickly enough to the change of core activity, and the cores stay in the lower turbo range. Eventually, CFS starts using the cores near the initial one again, and the pattern repeats. This pattern makes up around half of the execution; the remainder involves longer-running non-concurrent threads. In contrast, as shown in Figure 2(b), NEST places the threads on only two cores, which mostly stay at the highest frequencies.

Performance analysis. As there is no software configuration benchmark, we use the configuration scripts provided with the software in the Phoronix Timed Code Compilation Test Suite [16]. We consider only software where the configuration scripts are generated using the commonly used tools autotools and cmake, which clearly separate configuration from the rest of the build. Figure 3 shows the speedups achieved with NEST. Figure 4 shows the distribution of frequencies. NEST reduces the number of cores used, such that the highest frequencies are almost always used. Speedups compared to CFS-schedutil exceed 5% except on NodeJS. NodeJS configuration only performs one operation, so there is little room for improvement. The greatest speedup with NEST-schedutil, on the E7-8870 v4, is 35%.

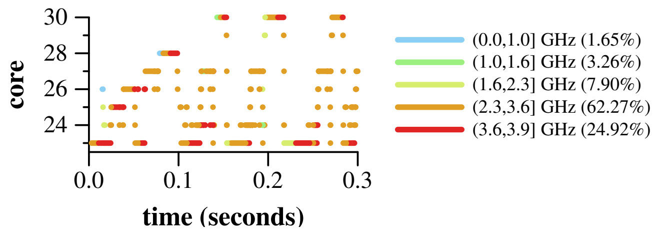
[3]Cores have been renumbered such that adjacent cores on the same socket have adjacent numbers, for readability.

Table 2. Hardware characteristics

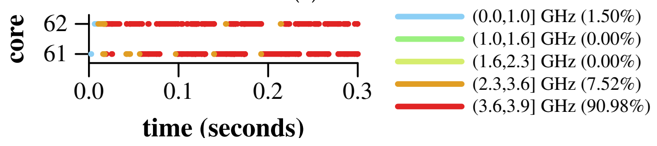
CPU	Microarchitecture	# cores	Min freq	Max freq	Max turbo	Power management
Intel Xeon E7-8870 v4	Broadwell	4x20x2=160	1.2GHz	2.1 GHz	3.0GHz	Enhanced Intel SpeedStep
Intel Xeon Gold 6130	Skylake	2x16x2 = 64	1.0GHz	2.1GHz	3.7GHz	Intel Speed Shift
Intel Xeon Gold 6130	Skylake	4x16x2 = 128	1.0GHz	2.1GHz	3.7GHz	Intel Speed Shift
Intel Xeon Gold 5218	Cascade Lake	2x16x2 = 64	1.0GHz	2.3GHz	3.9GHz	Intel Speed Shift

Table 3. Available turbo frequencies, in terms of number of cores used on a given socket

	1	2	3	4	5-8	9-12	13-16	17-20
E7-8870 v4	3.0	3.0	2.8	2.7	2.6	2.6	2.6	2.6
6130	3.7	3.7	3.5	3.5	3.4	3.1	2.8	–
5218	3.9	3.9	3.7	3.7	3.6	3.1	2.8	–



(a) CFS



(b) Nest

Figure 2. Core frequency trace for LLVM configuration for build with Ninja, when using the schedutil governor, on a 64-core, 2-socket, Intel Xeon Gold 5218

We next compare the speedups by governor. On the recent 6130 and 5218 machines, Nest gives about the same speedup with both *schedutil* and *performance*, and gives a much greater speedup than CFS-performance. CFS-performance does not give any speedup on these machines because CFS-schedutil already reaches the turbo frequencies. Unlike Nest, *performance* alone does nothing to reduce the number of used cores. On the older E7-8870 v4 machine, Nest-schedutil gives less speedup than CFS-performance or Nest-performance. Indeed, with *schedutil*, whenever there are gaps in the computation, this machine is prone to using subturbo frequencies, even when very few cores are used. By forcing the minimal frequency to be the nominal frequency, *performance* increases the likelihood that cores will reach and remain at the turbo frequencies. Nest-performance almost always achieves more speedup than CFS-performance, because CFS uses too many cores.

Next, we compare the speedups across machines. The speedups are comparable for the two 6130 machines. Indeed, the computation fits into a single socket, making the number of sockets irrelevant. The 5218, which is faster than the 6130, shows both better performance with Nest and better speedup than observed on the 6130 machines. Finally, the older E7-8870 v4 is in all cases the slowest, but it achieves the greatest speedup (up to 35%).

At the highest frequencies, cores consume more energy than at lower frequencies, and thus the Wattage observed when using Nest is greater than that observed when using CFS. However, Nest reduces the execution time, thus reducing the application's CPU energy consumption by up to 19% (Figure 5). Overall, Nest provides both a consistent speedup and energy savings by ensuring that the computation remains on a small number of cores.

Note that on the Intel Xeon while the hardware sets the core frequencies up and down, the CPU power consumption is determined by the consumption of the highest frequency core on the socket. Furthermore, if any core on the machine is active, all of the sockets remain in a high state of availability in case of accesses to their associated memory. Thus, while some power can be saved by concentrating all of the threads on a single socket, the greatest CPU power savings can only be achieved by reducing the running time of the application.

Impact of Nest features. To understand the impact of the various features of Nest, we perform a small ablation study, removing the features one by one. We likewise consider multiplying each of the parameters shown in Table 1 by 2 or 10. On these variants, we test llvm_ninja and mplayer configuration, using schedutil. The only change in performance is a degradation for mplayer of 4-6% for the 6130 and 5218 machines, and of 16% for the E7-8870 v4, as compared to Nest-schedutil, if we remove the reserve nest. The reserve nest allows the primary nest to remain small, but allows some extra cores to be chosen just because they are idle, rather than taking into account recent load as done by CFS.

Comparison with S_{move}. Figure 3 shows the speedup of Nest-schedutil and S_{move}-schedutil as compared to CFS-schedutil. While Nest gives speedups of up to over 20% on the 6130's and the 5218, and even higher speedups on the E7-8870 v4, the speedup of S_{move} on the 6130's and the 5218 is always under 5%, except on the configuration of LLVM, where it reaches 9%. S_{move} tentatively places the forked or woken thread on the same core as the parent only when the frequency observed at the last clock tick of the core chosen by CFS is low. On the 6130's and the 5218, when a core

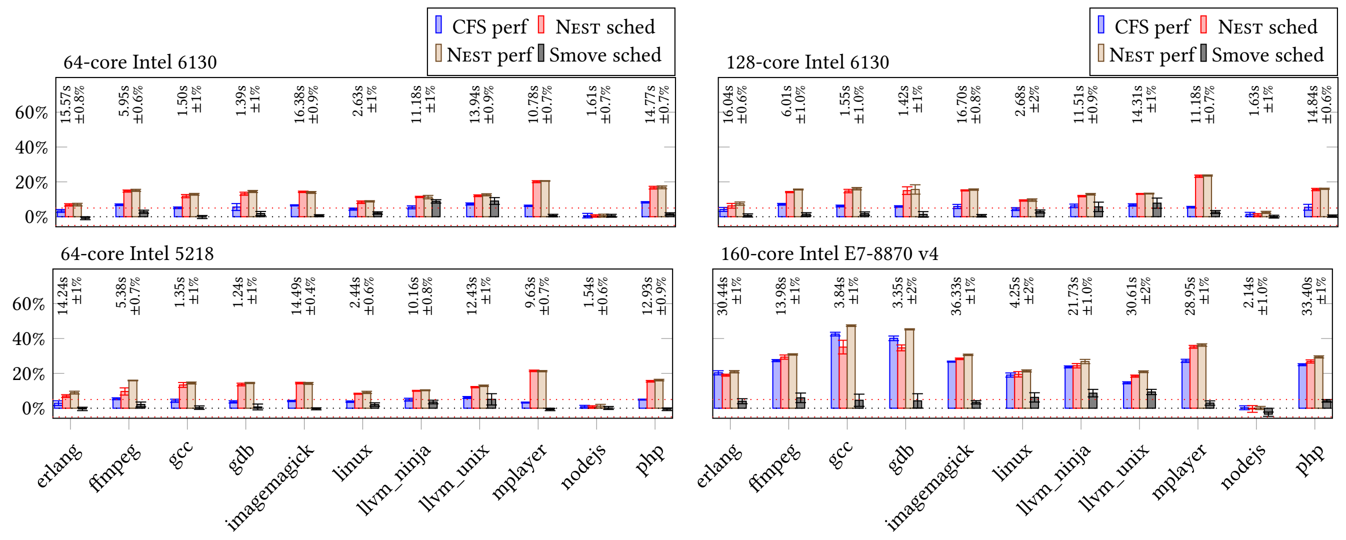


Figure 3. Configuration tests, speedup as compared to CFS schedutil

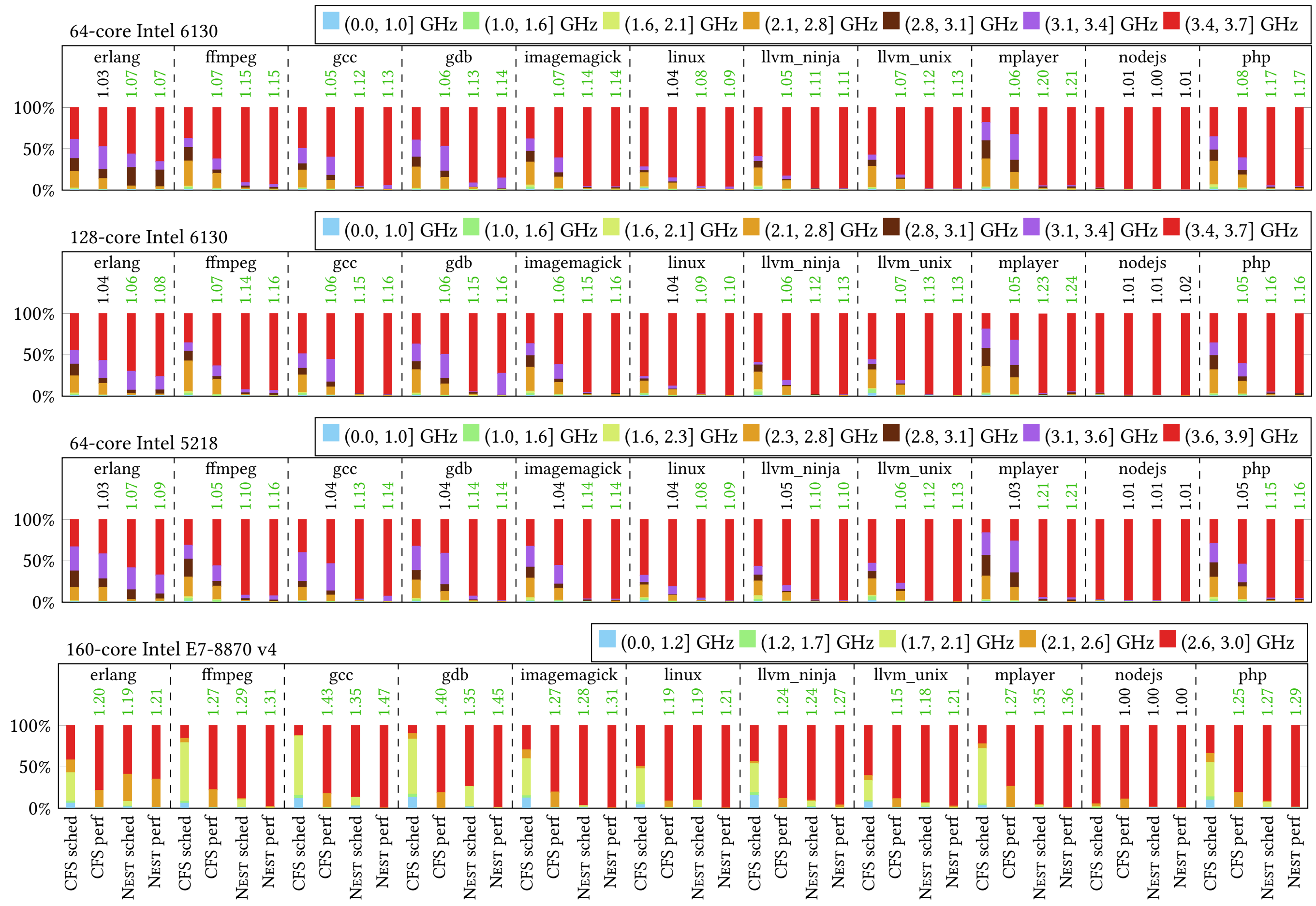


Figure 4. Configuration tests, frequency distribution. The numbers above the bars indicate the speedup as compared to CFS schedutil. Green numbers indicate a speedup of more than 5%.

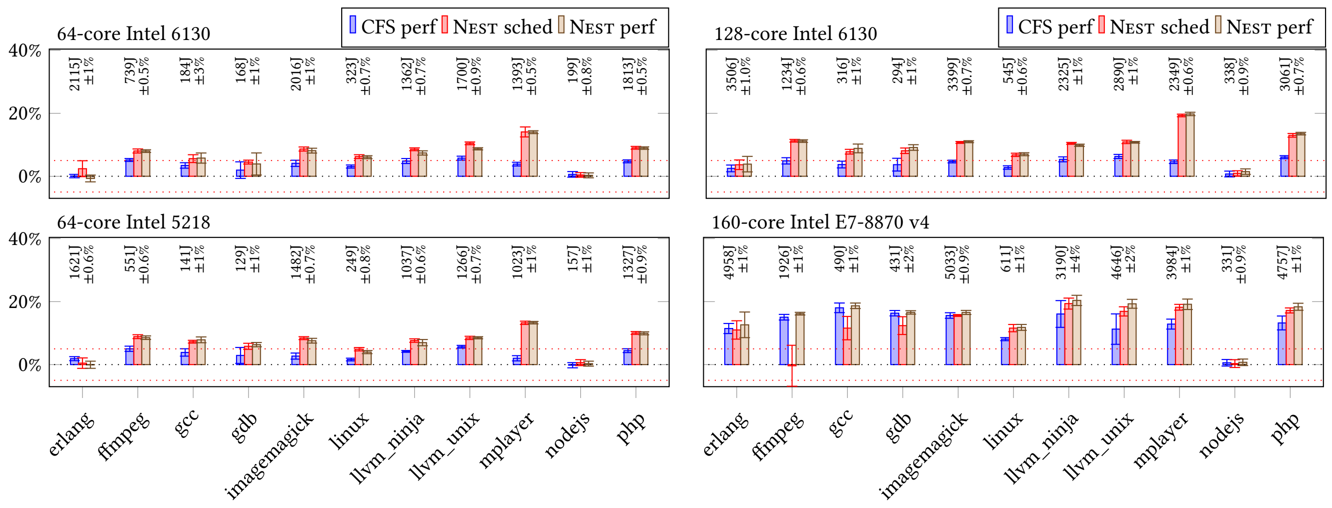


Figure 5. Configuration tests, reduction in CPU power consumption as compared to CFS-schedutil

becomes idle there is often no clock tick that observes a low frequency. Thus, S_{move} considers that the core chosen by CFS is still at its previous high frequency, making S_{move}'s placement heuristic unnecessary. S_{move} gives slightly higher speedups on the E7-8870 v4, but they remain far from those observed with Nest.

Software configuration is a best case for S_{move}, because it heavily involves single-threaded execution, with one thread handing off control to another. As S_{move} does not perform as well as Nest even in this scenario, we do not include S_{move} in our remaining evaluations.

5.3 DaCapo benchmark suite evaluation

The DaCapo benchmark suite [3] comprises a number of real-world Java applications. We use both the original version and the *evaluation* version that contains more recent applications.[4] We give the latter applications names ending in "-eval". We omit the benchmarks that do not run on our Debian OpenJDK 11.0.2.[5] To avoid interference of the JIT compiler and the garbage collector, the DaCapo benchmark developers propose that only the last of N runs be considered. We choose instead to report on the complete execution of 10 runs, to cover a wider range of behaviors. Each test runs the application 10 times. Over 10 tests we have very small standard deviations (most $\leq$ 2%).

Performance analysis. The DaCapo benchmarks include both single-threaded and highly multicore applications. The results with Nest (Figure 6) range from a 7-8% degradation with the single-threaded benchmarks fop and luindex on the E7-8870 v4 (the only degradations of more than 5%) to a speedup of more than 40%. Nest-schedutil achieves the highest speedups on h2, tradebeans, and graphchi-eval. These applications also run significantly faster on the two-socket machines than on the four-socket 6130. We focus on h2.

On the four-socket 6130, with CFS-schedutil, h2 application threads are distributed across most of the cores of one socket, and additionally typically across 0-2 other sockets. Despite this large core use, after the JIT compilation, h2 mostly has only five or fewer concurrent threads. Nest-schedutil concentrates the post-JIT computation on a single socket and on around 10 cores. This gives higher frequencies: with CFS-schedutil, over 26% of the execution time is in subturbo frequencies, while with Nest-schedutil, this is the case for only 5% of the execution time. With Nest-schedutil, 67% of the execution time is in the low and medium turbo frequencies, while this is only 49% in the case of CFS-schedutil. These higher frequencies result in a large speedup.

CFS-performance mostly eliminates the use of the subturbo frequencies, but the frequencies remain more in the bottom half of the turbo range, due to the large number of used cores. The speedup is thus lower: 30% with CFS-performance and 45% with Nest-schedutil. Nest-performance further consolidates the computations on a small number of cores, thus increasing the use of the low and middle turbo frequencies to 76% of the execution time. The speedup of Nest-performance over CFS-schedutil is 51%.

On the smaller two-socket 6130 and 5218 machines, for h2, CFS-schedutil still uses most of the cores on one socket and often uses cores on the other socket. But the time in the subturbo frequencies slightly decreases (from 26% to 22%) and the time in the turbo frequencies, including the highest turbo frequencies, slightly increases, improving the CFS-schedutil performance as compared to the four-socket 6130

[4]Jar files dacapo-9.12-MR1-bach.jar and dacapo-evaluation-git+309e1fa.jar.
[5]Of the original DaCapo benchmarks, batik and eclipse crash. Tomcat gives a failure result. Tradesoap crashes intermittently. Some of these problems are documented here: https://github.com/eclipse-openj9/openj9/issues/4859. h2o-eval goes into an infinite loop on some runs.

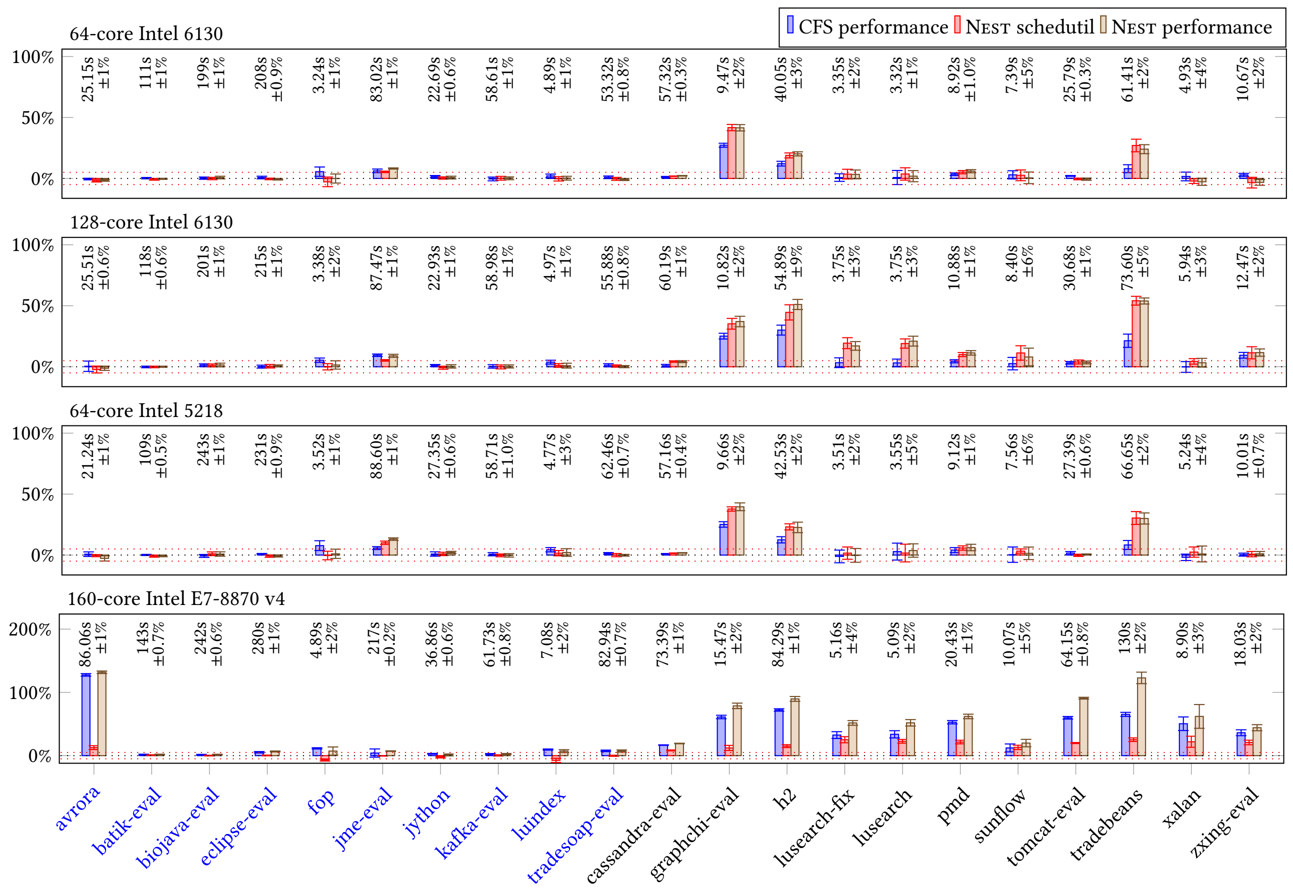


Figure 6. DaCapo tests, speedup as compared to CFS schedutil. Blue applications involve only one or a few threads.

case. Nest-schedutil again concentrates the computations onto a single socket, increasing use of the turbo frequencies, and giving a 19% and 23% speedup on the 6130 and the 5218, respectively. On these machines, CFS-performance also gives a speedup, but it is 12% and 13%, respectively, and thus again smaller than the speedup obtained by Nest-schedutil.

The thread placement of h2 on the older four-socket E7-8870 v4 is similar to that of the two-socket 6130 and 5218. Nest again concentrates the threads on around 10 cores of a single socket. But the performance improvement is lower. Indeed, on the E7-8870 v4, Nest-schedutil has little impact on the frequency for many of the DaCapo benchmarks. These benchmarks involve threads that frequently sleep for brief periods, which causes the E7-8870 v4 to use the lowest frequencies. The *performance* governor requests use of the nominal frequency, which causes the E7-8870 v4 to use the lower turbo frequencies for such applications (likewise for the single-threaded avrora). As Nest-performance also reduces the number of cores used for h2, it gives a larger speedup than CFS-performance.

An alternative to minimizing the number of sockets could be to spread the threads out across multiple sockets, to limit the number of active cores on each socket, thus giving each the maximum frequency. Indeed, particularly with the E7-8870 v4 that offers few turbo frequency levels, we have observed that such distribution with CFS on the DaCapo benchmarks can result in some cores having a higher frequency than with Nest. In those cases, however, the impact is not great enough to give a signficant performance improvement.

Impact of Nest features. We study the performance of h2, graphchi-eval, and tradebeans if we remove some of the Nest features. Spinning has the greatest impact on these applications; removing it gives a degradation of 11-19% on the two-socket 6130 and 5218, and a degradation of up to 24% on the four-socket 6130. The favoring of cores in the primary nest that are on the same socket as the core previously used by a waking thread or currently used by the parent of a forking thread also has a 5-12% impact. Tradebeans, on the other hand, is degraded by the "impatient" feature; its performance improves by removing this feature, and the improvement goes up to 22% if the feature is removed in conjunction with the "attached" feature and the favoring of the previous core. The "impatient" feature detects cases where the nest contains too few cores, which can result in

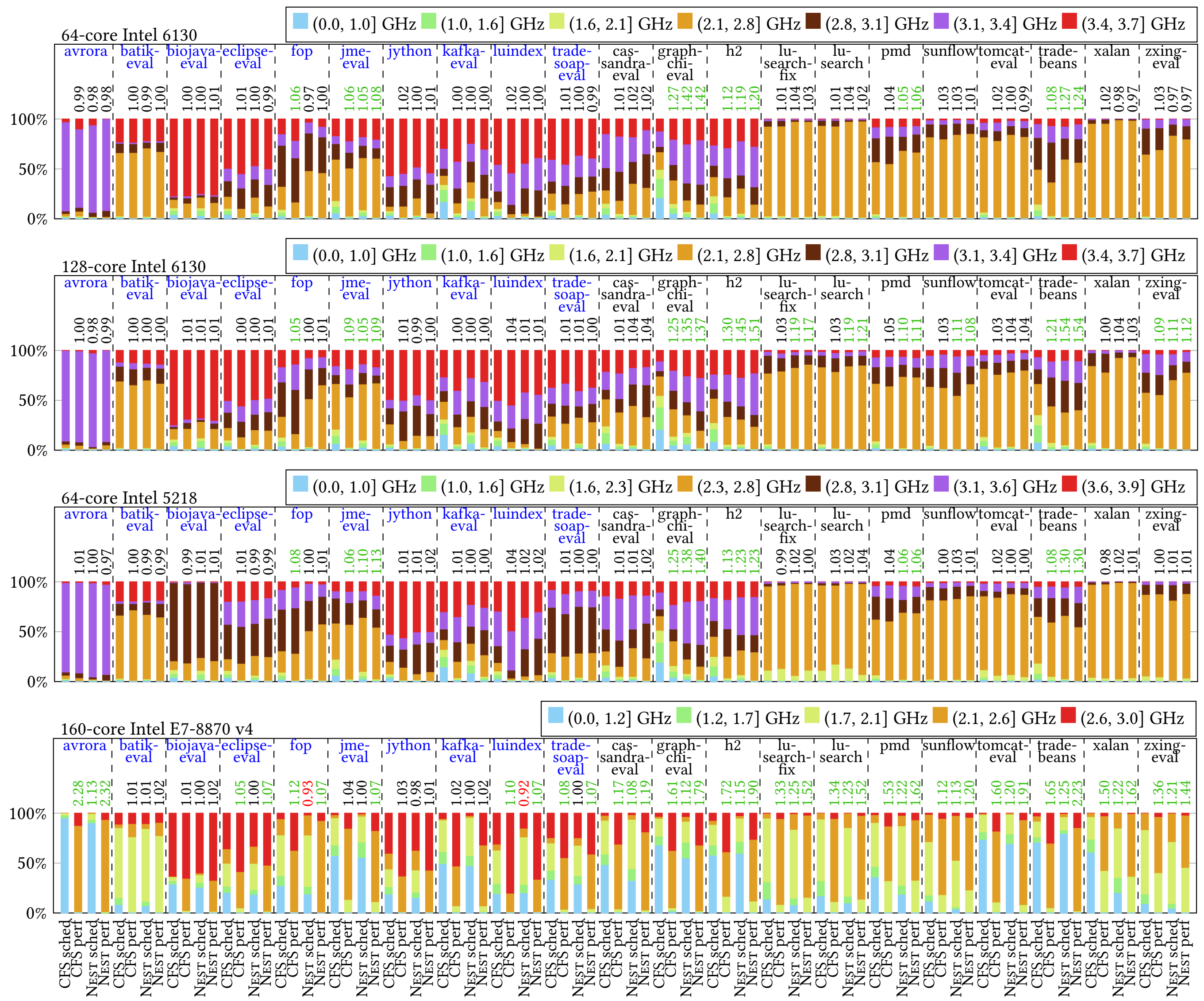


Figure 7. DaCapo tests, frequency distribution. The numbers above the bars indicate the speedup as compared to CFS schedutil. Green numbers indicate a speedup of more than 5%. Red numbers indicate a degradation of more than 5%.

some threads leaking into a second socket when the number of concurrently running threads is close to the number of cores on a single socket.

5.4 NAS Parallel Benchmarks evaluation

The NAS Parallel Benchmarks [2] are a collection of HPC kernels. We use version 3.4 with OpenMP.[6] For conciseness, we show only the results for the "C" datasets, the largest of the standard test problems.[7] These benchmarks each involve one thread per core. In the optimal case, each thread is immediately placed on its own core at the time of fork, and remains there throughout its execution. The challenge for Nest is to allow that to happen, by increasing the size of the primary nest to the number of cores on the machine.

As shown in Figure 8, on the two-socket 6130 and 5218, CFS and Nest have essentially the same performance. Typically, both schedulers place each thread on its own core as it is forked, the threads remain synchronized as they execute, and there is little movement between cores. For most of the tests, the cores remain in the low turbo frequencies; higher frequencies are not possible because all cores are active.

The performance on the four-socket 6130 is more variable, with high speedups with both Nest-schedutil and Nest-performance on BT (30%), LU (157% and 183%, respectively),

[6]https://github.com/mbdevpl/nas-parallel-benchmarks.git, tag v3.4
[7]We omit the benchmark DC that targets computational grids and is thus out of the scope of scheduling for individual servers.

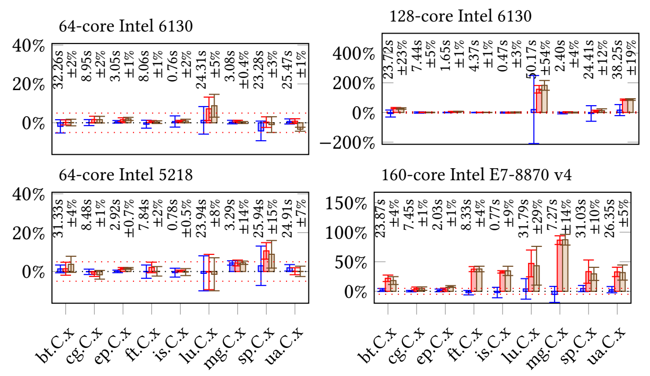


Figure 8. NAS tests, speedup as compared to CFS schedutil. The schedulers are color-coded as Figures 3, 5 and 6.

and UA (186% and 187%, respectively). These results are difficult to interpret, due to the high standard deviation of CFS-schedutil on these benchmarks, up to 54% in the case of LU. When CFS-schedutil has a standard deviation of under 10%, all of the schedulers give essentially the same performance.

On the four-socket E7-8870 v4, Nest provides substantial speedups, from 18% on BT (Nest performance) to over 80% on MG, on all tests except CG and EP. Lepers *et al.* [10] observed that on highly multicore machines where cores do not reach the highest frequencies quickly, CFS's fork is not able to place each NAS thread on its own core. This causes overloads that then must be gradually resolved via load balancing. Nest is more aggressively work conserving than CFS on thread wakeup. This feature allows the primary nest to quickly reach a size close to the number of concurrent threads. But it has the side effect of improving the performance of the NAS benchmarks on the E7-8870 v4 as compared to CFS. Nest-schedutil without work conservation on wakeups gives about the same performance on BT and MG as CFS-schedutil.

The results on all four machines show overall that the nest does not get in the way of highly parallel applications.

5.5 Phoronix multicore suite evaluation

We conclude with the Phoronix multicore suite [15]. Multicore applications have a wide variety of behaviors, and thus large-scale testing of a scheduler is necessary to achieve adoption. The Phoronix multicore suite comprises 90 benchmarks, each involving one or more tests, for a total of up to 224 tests that we were able to run. This is a much larger test set than considered by recent scheduling papers [4–6, 14].

The Phoronix tests involve various metrics; we say "speedup" to indicate an improvement in the metric value. Most tests (Table 4) are unaffected by Nest, with a speedup of ±5%. For all architectures, at least 6% of the tests have a speedup above 5% for Nest-schedutil, and this is the case for 22% of the tests on the E7-8870 v4.

Table 4. Overview of the Phoronix multicore results

		slower by		same	faster by	
CPU	scheduler	> 20%	(5,20]%		(5,20]%	> 20%
2 socket 6130	CFS-perf.	0 (0%)	1 (0%)	208 (92%)	9 (4%)	6 (2%)
	Nest-sched.	1 (0%)	17 (7%)	190 (85%)	12 (5%)	3 (1%)
	Nest-perf.	1 (0%)	13 (5%)	188 (85%)	14 (6%)	5 (2%)
4 socket 6130	CFS-perf.	1 (0%)	7 (3%)	191 (87%)	9 (4%)	10 (4%)
	Nest-sched.	2 (0%)	20 (9%)	158 (73%)	18 (8%)	18 (8%)
	Nest-perf.	2 (0%)	15 (7%)	152 (73%)	18 (8%)	19 (9%)
2 socket 5218	CFS-perf.	0 (0%)	1 (0%)	203 (92%)	10 (4%)	6 (2%)
	Nest-sched.	1 (0%)	8 (3%)	189 (86%)	17 (7%)	4 (1%)
	Nest-perf.	1 (0%)	6 (2%)	192 (87%)	14 (6%)	6 (2%)
4 socket E7-8870 v4	CFS-perf.	0 (0%)	5 (2%)	119 (61%)	18 (9%)	52 (26%)
	Nest-sched.	1 (0%)	11 (5%)	132 (70%)	28 (15%)	14 (7%)
	Nest-perf.	1 (0%)	5 (2%)	106 (57%)	26 (14%)	47 (25%)

Figure 9 shows detailed results for those tests where either CFS-performance or Nest-schedutil shows a speedup or degradation of at least 20% on at least one machine. We have observed that the speedup of Nest-performance is typically comparable to the maximum of these two values. Thus, we omit this scheduler for readability. The tests are numbered according to the order in which they appear on Phoronix web site [15].

We highlight some of the results that illustrate specific patterns. We emphasize that these represent exceptional cases, and are typically highly dependent on the number of concurrent application threads and the architecture.

Zstd compression 7 and 10 – high speedup with CFS-performance and Nest-schedutil. With Zstd compression 7 and 10, CFS-performance and Nest-schedutil both give significant speedups on the 6130 and 5218 machines. As in Figure 2, CFS spreads the threads out over all of the cores, and the threads run for very short times, and thus obtain a low frequency. CFS-performance restricts the hardware to higher frequencies, giving speedups of 36-76% on all machines. Nest-schedutil reuses cores, placing all threads on a small set of cores on a single socket, giving speedups of 25-98% on the 6130 and 5218 machines. On the E7-8870 v4, Nest-schedutil also concentrates the threads on a few cores of a single socket, but the degree of activity is still too low, and the cores remain at a very low frequency.

Rodinia 5 – opposite behavior with CFS-performance and Nest-schedutil. On the 6130 and 5218 machines, Rodinia 5 gives a speedup with Nest, but no speedup with CFS-performance. On the other hand, on the E7-8870 v4, Rodinia 5 gives a speedup with CFS-performance, but a small degradation (8%) with Nest. Rodinia uses 36 cores for most of its computation. With CFS-schedutil, they remain on one socket, while with Nest they scatter across the machine. On the 6130 and 5218, with CFS-schedutil, intensively computing threads share physical cores, reducing their performance as compared to Nest-schedutil, where threads mostly run on a core that has an idle hyperthread. On the E7-8870 v4,

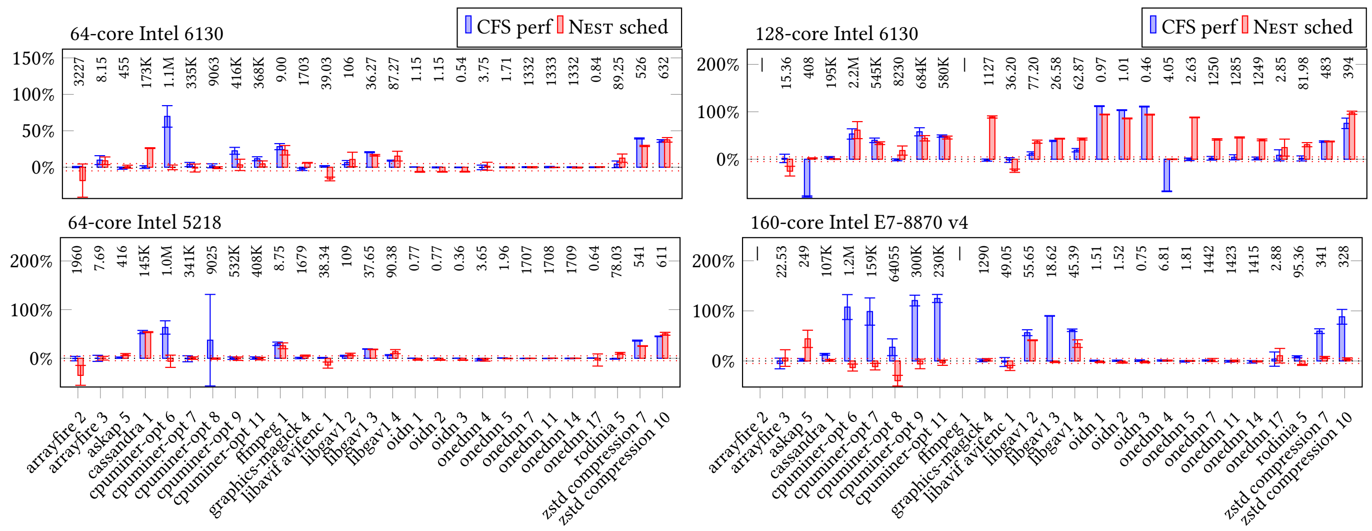


Figure 9. Phoronix multicore suite, speedup as compared to CFS schedutil

the scattering of running threads done by Nest implies that the hardware sees less activity and thus uses lower frequencies, reducing performance. CFS-performance ensures high frequencies, solving the problem.

libavif avifenc 1 – degradation with Nest-schedutil on all machines. The greatest degradation (24%) is on the four-socket 6130. In this case, with CFS-schedutil, most of the threads start on one socket, but some migrate to other sockets over time, resulting in a number of concurrent threads (up to 12) on these other sockets that allows them to reach the middle turbo frequencies. Nest-schedutil keeps the threads on the initial socket, where they run at the lowest turbo frequency, reducing performance.

6 Related Work

Recent work [4, 6, 10, 11] has shown that for modern multicore servers a critical component of a scheduler is its strategy for selecting a core for each thread. Lozi *et al.* [11] found that CFS could overlook idle cores on some sockets while overloading cores on others, violating *work conservation* and thus degrading performance. Subsequently, Lepers *et al.* [10] proposed a strategy for formally proving work conservation and showed performance benefits. Gouicem *et al.* [6], however, found that work conservation is not enough – CFS can induce suboptimal performance even when every thread is placed on an idle core, when the chosen cores are running at a low frequency.

Even when Nest concentrates all of the computation on a single socket, the hardware does not put the other sockets into the deepest sleep states, to facilitate any accesses to memory on those sockets. Nitu et al. [12] suggested separating the power supplies of sockets and memory, in order to suspend servers in a cloud environment, but leave their memory accessible, and thus reduce energy consumption. Keeton [9] proposed a distributed system memory-centric architecture that would provide a similar capability.

Solaris assigns each thread a home node, and the scheduler tries to keep a thread on this node [13]. The goal is to improve memory locality, rather than favoring a core with a higher frequency. Indeed, on modern servers, core frequencies vary independently, and thus attaching a thread to a particular node is not sufficient to obtain the best performance.

Lowering the core frequency (DVFS) to save energy has a long history. Some work has considered DVFS for servers in order to offer more energy efficient cloud computing [18, 19].

Our work targets the problem of threads that run slowly because they are placed on idle cores that have dropped to a low frequency. Cores may also run slowly to maintain thermal properties in the case of AVX vector instructions. Gottschlag *et al.* [5] adjust the CPU time accounting to maintain fairness in this situation. We focus on the choice of cores, to avoid placing threads on slow cores when possible.

Core scheduling [20] adjusts core selection to avoid placing untrusted threads on a core's hyperthread, to prevent side-channel attacks. It is orthogonal to Nest.

7 Conclusion

In this paper, we have presented the Nest scheduling policy that favors concentrating threads on a reduced set of cores. We have evaluated Nest on a variety of recent Intel multicore architectures, comprising moderate-size and large multicore servers. Nest shows substantial performance improvements on applications where the number and set of running threads changes frequently, and maintains the performance of CFS on applications that involve only one or a handful of threads as well as applications that fully use all of the cores of the machine. Our results show that processor frequencies can have a large impact on application performance. It may thus be worth considering redesigning hardware to allow a greater number of cores to run at the higher turbo frequencies.

Availability. The implementation of Nest is available at https://sysartifacts.github.io/eurosys2022/.

Acknowledgement. Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr). This work was partially funded by an ERO gift from Oracle.

A Artifact Appendix

A.1 Abstract

The Nest artifact includes the Nest Linux kernel patch, the configure, DaCapo, and NAS benchmark suites used in the paper, tools for running these benchmarks, and instructions for running the Phoronix benchmarks.

A.2 Description & Requirements

A.2.1 How to access. The artifact is available at https://gitlab.inria.fr/nest-public/nest-artifact. This repository includes more detailed instructions for setting up and running the benchmarks. In this appendix, all mentioned files are from this repository.

A.2.2 Hardware dependencies. We have only tested the artifact on the machines described in the paper. The experiments are intended to be run on the hardware, not in a virtual machine.

A.2.3 Software dependencies. The artifact has many software dependencies. It relies on Debian 11. The required packages are listed in `image_creation/debian11_packages`. The file `image_creation/debian11_list` gives information about version numbers. The file `image_creation/README.md` (also available in pdf) explains how to install these packages. This file also explains how to install some software that is not available from Debian.

A.2.4 Benchmarks. The software tested for the configure, DaCapo, and NAS experiments (Figures 3-8) are found in `configure-dacapo-nas/software`. These experiments rely on tools found in `image_creation/ocaml-scripts` and configuration files found in `configure-dacapo-nas/scripts`.

The procedure for obtaining the Phoronix benchmarks is located at the end of `README.md`.

A.3 Set-up

Please see `image_creation/README.md` for detailed environment set up instructions and `README.md` for detailed instructions on running the experiments.

A.4 Evaluation workflow

A.4.1 Major Claims. We observe that when there are fewer concurrently running thread than cores, CFS is prone to scatter the threads across the machine, causing threads to often wake up on recently idle cores running at a low frequency. We claim that in this case performance can be improved, often with no additional energy consumption, by restricting the threads to a smaller number of cores. Such cores are considered by the OS and by the hardware to be more highly utilized and thus run at a higher frequency. We further claim that this strategy does not harm the performance of applications that use all cores intensively, and thus do not benefit from it.

These claims are mainly illustrated by Figures 3-8 and Figure 9 in the paper, and the accompanying discussion in Section 6. We aim to make these figures reproducible in this artifact.

A.4.2 Experiments. Detailed instructions on how to reproduce the environment used by the experiments are found in `image_creation/README.md` and how to run the experiments in `README.md`. We give only an overview here. The experiment times are very approximate.

Experiment (E1): [configure] [4 hours]: This experiment produces Figures 3-5. It produces graphs describing running times, core frequencies, and CPU energy consumption. It compares CFS with the schedutil and performance governors, Nest with the schedutil and performance governors, and a previous approach S_{move} with the schedutil governor. This experiment is described in Section 5.2 of the paper.

[Preparation] In `image_creation/ocaml-scripts`, run `make` and `sudo make install`. This step is a prerequisite for all of the experiments. Set the environment variable `LC_MYNAME` to your username.

These experiments should be performed using the 5.9.0freq, 5.9.0Nest, and 5.9.0smoveoriginal kernels. The document `image_creation/README.md` explains how to build and install these kernels. The associated kernel patches are found in `image_creation/{freq,nest,smoveoriginal}.patch`.

[Execution] In the directory `configure-dacapo-nas/scripts`, edit the files `configure.config` and `smove_configure.config` to add a block for your test machine. The `name` field should be the result of running `hostname` on your test machine. It should be sufficient to copy the other fields from another entry as is. Then, as root, run `run_everything configure.config smove_configure.config`. This command will place the results (.dat files, .json files, and .turbo files, for each test) in the directory `configuretraces` in your home directory.

[Results] In the `configuretraces` subdirectory of your home directory, graphs can be made using `read_csvs configure.tex 5.9.0freq_schedutil *json`. The resulting file `configure_acm.pdf`

resembles the graphs in the paper, although with a less fine-tuned layout. The information found in the other generated pdf files is described in `README.md`. Our versions of these files are found in `configure-dacapo-nas/results`.

Experiment (E2): [DaCapo] [36 hours]: This experiment produces Figure 6 and 7 in the paper. It produces graphs describing running times and frequencies, comparing CFS with the schedutil and performance governors with Nest with the schedutil and performance governors. This experiment is described in Section 5.3 of the paper.

The experiment is done in essentially the same way as the configure experiment. The main steps are:

- Add an entry for the local hostname to `dacapo.config`.
- `run_everything dacapo.config`
- `read_csvs dacapo.tex 5.9.0freq_schedutil *json`

Experiment (E3): [NAS] [2 hours]: This experiment produces Figure 8 in the paper. It produces graphs describing running times, comparing CFS with the schedutil and performance governors with Nest with the schedutil and performance governors. This experiment is described in Section 5.4 of the paper.

The experiment is done in essentially the same way as the configure experiment. The main steps are:

- Add an entry for the local hostname to `nas.config`.
- `run_everything nas.config`
- `read_csvs nas.tex 5.9.0freq_schedutil *json`

Experiment (E4): [Phoronix] [24 hours]: This experiment produces Figure 9. It presents results for benchmarks from the phoronix multicore suite, comparing Nest schedutil power governor with CFS schedutil and performance power governors. The key observations are described in Section 5.5 of the paper.

Following are the main steps of this experiment:

- Install phoronix-test-suite version 10.4 from GitHub.
- Install the benchmarks with test-profiles provided in nest-artifact.
- Boot with desired kernel and power governor using the scripts provided in nest-artifact.
- Perform two warm-up runs followed by 10 runs of the benchmark.
- Give resultfile a descriptive name.
- Repeat the same steps for Nest schedutil, CFS schedutil and CFS performance.
- Use read_csvs tool provided in nest-artifact to analyze the results.

References

[1] AMD. Amd turbo core technology. https://www.amd.com/en/technologies/turbo-core.

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

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

[4] 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 *USENIX Annual Technical Conference* (2018), pp. 85–96.

[5] Gottschlag, M., Machauer, P., Khalil, Y., and Bellosa, F. Fair scheduling for AVX2 and AVX-512 workloads. In *USENIX Annual Technical Conference* (2021), pp. 745–758.

[6] Gouicem, R., Carver, D., Lozi, J.-P., Sopena, J., Lepers, B., Zwaenepoel, W., Palix, N., Lawall, J., and Muller, G. Fewer cores, more hertz: Leveraging high-frequency cores in the OS scheduler for improved application performance. In *USENIX Annual Technical Conference* (2020), pp. 435–448.

[7] Humphries, J. T., Natu, N., Chaugule, A., Weisse, O., Rhoden, B., Don, J., Rizzo, L., Rombakh, O., Turner, P., and Kozyrakis, C. ghOSt: Fast & flexible user-space delegation of Linux scheduling. In *SOSP* (2021).

[8] Intel. Intel turbo boost technology 2.0: Higher performance when you need it most. https://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html.

[9] Keeton, K. The machine: An architecture for memory-centric computing. In *Workshop on Runtime and Operating Systems for Supercomputers (ROSS)* (June 2015).

[10] Lepers, B., Gouicem, R., Carver, D., Lozi, J.-P., Palix, N., Aponte, M.-V., Zwaenepoel, W., Sopena, J., Lawall, J., and Muller, G. Provable multicore schedulers with Ipanema: application to work conservation. In *EuroSys* (2020), pp. 3:1–3:16.

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

[12] Nitu, V., Teabe, B., Tchana, A., Isci, C., and Hagimont, D. Welcome to zombieland: practical and energy-efficient memory disaggregation in a datacenter. In *EuroSys* (2018), pp. 16:1–16:12.

[13] Nordén, M., Löf, H., Rantakokko, J., and Holmgren, S. Geographical locality and dynamic data migration for OpenMP implementations of adaptive pde solvers. In *IWOMP 2005: OpenMP Shared Memory Parallel Programming* (2005), pp. 382–393.

[14] Patel, Y., Yang, L., Arulraj, L. P., Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., and Swift, M. M. Avoiding scheduler subversion using scheduler-cooperative locks. In *EuroSys* (2020), pp. 9:1–9:17.

[15] Phoronix Test Suite. Multi-core. https://openbenchmarking.org/suite/pts/multicore.

[16] Phoronix Test Suite. Timed code compilation. https://openbenchmarking.org/suite/pts/compilation.

[17] Torvalds, L., 2007. https://lore.kernel.org/lkml/Pine.LNX.4.64.0703082226530.10832@woody.linux-foundation.org/.

[18] von Laszewski, G., Wang, L., Younge, A. J., and He, X. Power-aware scheduling of virtual machines in DVFS-enabled clusters. In *IEEE International Conference on Cluster Computing and Workshops* (2009), pp. 1–10.

[19] Wu, C.-M., Chang, R.-S., and Chan, H.-Y. A green energy-efficient scheduling algorithm using the DVFS technique for cloud datacenters. *Future Generation Computer Systems 37* (July 2014), 141–147.

[20] Ziljstra, P. sched: Core scheduling, 2019. https://lwn.net/ml/linux-

kernel/20190218165620.383905466@infradead.org/.