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

Julia Lawall
Inria
Paris, France

Himadri Chhaya-Shailesh
Inria
Paris, France

Jean-Pierre Lozi
Oracle Labs
Zurich, Switzerland

Baptiste Lepers
University of Sydney
Sydney, Australia

Willy Zwaenepoel
University of Sydney
Sydney, Australia

Gilles Muller
Inria
Paris, France

Abstract

To best support highly parallel applications, Linux's CFS scheduler tends to spread tasks across the machine on task creation and wakeup. It has been observed, however, that in a server environment, such a strategy leads to tasks being unnecessarily placed on long-idle cores that are running at lower frequencies, reducing performance, and to tasks being unnecessarily distributed across sockets, consuming more energy. In this paper, we propose to exploit the principle of core reuse, by constructing a nest of cores to be used in priority for task scheduling, 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 range of applications using fewer tasks than cores, Nest improves performance 10%–2× and can reduce energy usage.

CCS Concepts: • **Computer systems organization**;

Keywords: Scheduling, Linux kernel

ACM Reference Format:
Julia Lawall, Himadri Chhaya-Shailesh, Jean-Pierre Lozi, Baptiste Lepers, Willy Zwaenepoel, and Gilles Muller. 2022. OS Scheduling with Nest: Keeping Tasks Close Together on Warm Cores. In *Seventeenth European Conference on Computer Systems (EuroSys '22), April 5–8, 2022, RENNES, France.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3492321.3519585

1 Introduction

The primary goal of an operating system (OS) task scheduler is to allocate tasks to cores in a way that maximizes application performance. A well-known desirable property is *work conservation*, *i.e.*, if a task 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 task, it is also important to consider whether the chosen core will allow the task to access needed (hardware) resources efficiently. The performance that a task can achieve is determined in part by the *frequency* of the chosen core [7]. 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 tasks on cores in a way that causes higher frequencies to be used can improve performance.

We consider scheduling on large multicore servers. Such servers are today becoming more accessible and affordable. They can be used for traditional high-performance computing, where applications are often designed to decompose to the number of cores available, so that tasks can be 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 task scheduler for task placement. The number of cores required may vary from few to many, and back, across the course of the application. To get the best performance, the OS task scheduler must 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 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 number of cores on the same socket and their current degree of activity. To obtain the highest possible frequencies, it is necessary to minimize the number of cores used ("keeping tasks close together") and ensure a sustained activity ("keeping cores warm").

In this paper, we propose the task scheduler Nest, designed according to the principles *reuse cores* and *keep cores warm*. To increase core reuse, Nest tries to place tasks within

EuroSys '22, April 5–8, 2022, RENNES, France

ACM ISBN 978-1-4503-9162-7/22/04…$15.00
https://doi.org/10.1145/3492321.3519585

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 task must be placed outside the nest because another task is using its previous core, Nest remembers this previous core, and attempts to return the task to the same core in the nest the next time a core is chosen for the task. On purely sequential applications and applications with as many or more tasks than cores, Nest performs similarly to CFS. Nest is particularly beneficial for applications with a moderate number of effective concurrent tasks, and where tasks fork, block, and terminate frequently, resulting in many task placements. We implement Nest within the Linux kernel, by modifying the CFS scheduler.

Our contributions are as follows:

- We motivate two new principles for task schedulers: reuse cores and keep cores warm, and implement these principles in the Linux kernel v5.9.
- 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 energy usage by up to 20%, on our software configuration benchmark.
- Nest often achieves comparable or better performance than the Linux kernel v5.9's *performance* power governor, which requests use of at least the nominal frequency, while using the *schedutil* power governor, which allows 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. Section 3 presents Nest and Section 4 presents its implementation 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 tasks* for the set of tasks that are running at a given point in time. This does not include tasks that are sleeping or waiting to run. We evaluate Nest on Intel servers, where two hardware threads share a single physical core (simultaneous multithreading). For simplicity, we refer to the number of cores on a machine as the number of hardware threads. We say that one core is a *hyperthread* of another core if both share the same physical core. A key concept in Linux's Completely Fair scheduler is the set of cores that share a last-level cache. We refer to such cores as being on the same *die*.

2 Background

We present Linux's Completely Fair (CFS) scheduler (as of 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 tasks on cores on task fork, task exec, task 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. On our multi-socket Intel servers, the domains are NUMA (all cores on the machine), SMP (cores on the same die), 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 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 various criteria, including the number of idle cores, the recent load on the cores, the expected time to wake from idle states, and the new task'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.

Search for a core across the entire machine favors work conservation. On the other hand, searching in a fixed order means that recently used cores may be overlooked, thus hindering reuse, if other idle cores come earlier in the order. Taking recent load 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 task's previous core or the core performing the wakeup. Heuristics are used to choose between them, combining information about core idleness, the type of wakeup, and the recent load on the core. CFS then searches for an idle core on the target's die. First, it searches for a core where 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 whether the hyperthread of the target is idle. If all previous searches fail, CFS selects the target.

Wakeup is not work conserving, as it only considers a single die, and it only makes a limited effort to find an idle core on that die. It traverses the cores in a fixed order, and thus it may overlook recently used idle cores. On the other hand, it does not consider recent load 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.* [7] identified the problem of *frequency inversion*, in the common case where a task T_{parent} forks or wakes another task 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 this problem causes 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 if other tasks are waiting, such a strategy may cause T_{child} to incur high latency. Thus, 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 T_{child} is not scheduled on T_{parent}'s core within a brief delay. When the timer expires, however, S_{move} does nothing to ensure that T_{child} 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 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. *Performance* gives tasks high performance, but misses the potential energy savings that can be obtained by running non-demanding tasks at lower frequencies.

Schedutil takes into account information from the scheduler about recent task 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 tasks 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 task. 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 nest according to applications' current needs. A nest that is too large will result in task dispersal, replicating the dispersal problem of CFS. A nest that is too small will result in tasks competing for the same cores, inducing overloads.

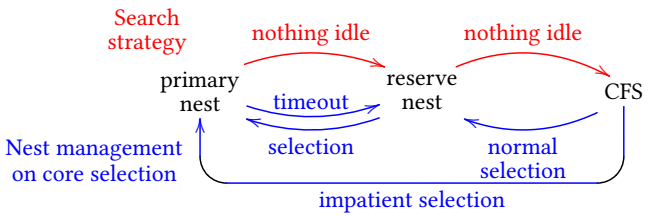

Figure 1. Core-search path through the nests (top) and core movement between nests (bottom).

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 task placement. Cores in the *primary nest* are currently in use or 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 they have recently been selected by CFS and have not yet proved their necessity for the current set of tasks.

The top of Figure 1 (red arrows) describes the core-search heuristic. For a forking or waking task, Nest first searches for an idle core in the primary nest, then if none is found it searches for an idle core in the reserve nest. If that also fails, then it falls back on CFS. The search in the primary nest starts at the task's previous core (or the parent's core, for 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 on which the system call that started Nest was executed, to reduce task dispersal. In both cases, the search first considers cores on the same die as the task's previous core (or the parent's core, for a fork), before considering cores on the other dies. This heuristic reduces the number of used dies, thus increasing the chance of leaving some dies completely idle and saving energy. Unlike CFS, Nest selects any core that is found to be currently idle, independent of recent load, in order to favor core reuse. 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 strategy of selecting cores where the hyperthread is idle.

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 tasks. If a core is selected from the reserve nest, then the core is promoted to the primary nest. If a core is selected when Nest falls back to CFS, then it is normally placed in the reserve nest. The size of the reserve nest is, however, limited to R_{max} cores; if this size is exceeded, then a core selected using CFS is not added to any nest. On the other hand, if a core in the primary nest has not been used for some time (at most P_{remove} ticks), it becomes eligible for *nest compaction*, and is demoted to the reserve nest, or discarded if the reserve nest is full, as soon as a task tries to use it. Nest compaction is not applied to the reserve nest, due to its bounded size. Finally, if a task 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 tasks are trying to share the primary nest. In this case, a task may bounce between cores, if it often wakes up to find that the core on which it was running previously is occupied by another task. If a task finds that its previous core is not idle $R_{impatient}$ times in a row, then the task is labeled as *impatient*. To place such a task, 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, to increase its size, and the task's impatience counter is reset.

3.2 Keeping the cores in the nest warm

Our goal of keeping cores at a high frequency is hampered by the fact that tasks 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 soon be used again. Accordingly, we modify the idle process to spin for up to a small number of ticks (S_{max}) when a task blocks. The spin continually checks whether there is a task on the core's hyperthread; if one appears the spinning stops immediately, leaving it to the task running on the hyperthread to keep the core warm.

3.3 Returning to the nest

It is desirable for a waking task to return to its previous core; moving the task to another core may cause another task to find its previous core occupied, triggering a cascade of migrations. At the same time, to keep the nest small and improve core reuse, we do not want to permanently place a task on a core outside the primary nest just because the task happens to wake up at the same time as *e.g.*, a brief daemon task. Accordingly, in addition to recording the core used on the previous execution of the task, 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, *i.e.*, if the core has once returned to its previous core, the task is considered to be *attached* to that core. The first choice of a task is always the core to which it is attached, if that core is in the primary nest and is idle. A task can even reclaim a core that is in the primary nest and is eligible for nest compaction, as long as no other task 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 tasks that are trying to run concurrently, which means that work conservation (no task waiting on a busy core when some core is idle) is an important property. To reduce wakeup latency, CFS is not work-conserving on task wakeups: it only considers the previous die of the waking task and the die 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 tasks more quickly to cores where they can run with minimal interference from other tasks, Nest enhances the degree of work conservation of CFS. Specifically, Nest extends the CFS core selection on task wakeup to examine all of the dies, if CFS does not find an idle core on its chosen die.

The Linux kernel places a forked or waking task on a core in two steps: first selecting a core, and then adding the task to that core's run queue. The absence of a global lock in the first step means that cores that are forking or waking tasks at the same time can choose the same core for their tasks, 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 collisions, Nest associates a flag with each run queue, indicating whether a task has been placed on the corresponding core. This flag is checked using a compare-and-swap instruction, ensuring that Nest places at most one task 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.[1] 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.

Table 1 shows the values chosen for the thresholds mentioned in Section 3. Notably, P_{remove}, the delay before a core becomes eligible for nest compaction, 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 idle cores prevents active cores on the same

[1] 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 (= 8ms)
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

socket from reaching the highest turbo frequencies. We evaluate the impact of these values in Section 5.

5 Evaluation

Evaluating a scheduler requires testing its impact on applications that exhibit a high diversity of task behaviors. To comprehensively evaluate Nest, we compare its behavior to that of CFS on a wide range of benchmarks, including applications that involve a small, fixed number of tasks, that use one task per core, and that have more variable task behaviors. Nest particularly targets applications in which the number and set of concurrent tasks 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: a 2-socket 64-core Intel 6130, a 4-socket 64-core Intel 6130, a 2-socket 64-core Intel 5218, and a 4-socket 160-core Intel E7-8870 v4. These reflect three generations of Intel Xeon architectures, released between 2016 and 2019. On these machines, a die coincides with a socket, *i.e.*, all of the cores on a given socket share the same last level cache. 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 first perform two warmup runs and then average over 10 runs. As we observe low standard deviations in these performance results, frequency traces are collected from one run. Power measurements are over 30 runs, to reduce their standard deviation. Speedups are obtained by dividing the average running time with the given scheduler by the average running time with CFS-schedutil. 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 comparisons are normalized such that 0 represents identical performance, percentages greater than 0 represent improvements, and percentages below 0 represent degradations. Speedup and energy savings graphs include a horizontal dotted line at ±5%, to highlight the larger improvements.

We measure CPU energy consumption using `turbostat`, with a 0.5-second measurement interval. We report the CPU energy consumption for the entire machine.

5.2 Software configuration test suite evaluation

Typical software configuration code is based on shell scripts, and forks off hundreds or even thousands of tasks, 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 tasks 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 tasks and their frequencies with CFS on the Intel 5218.[2] Starting from the task in the lower left, tasks are forked and are placed on cores with increasing core numbers. This even occurs when cores closer to the starting task are idle, due to the consideration of the load average; CFS prefers a fully idle core to one that has recently been used. The tasks end up dispersed across 8 cores. Even though only one or two tasks run at a time, the processor does not react quickly enough to the change of core activity, and the cores stay in the lower turbo range. Eventually, the recent load's influence times out, and CFS starts using the cores near the initial one again. The pattern repeats. This pattern makes up around half of the execution; the remainder involves longer non-concurrent tasks. In contrast (Figure 2(b)), Nest places the tasks on only two cores, which mostly stay at the highest frequencies.

To quantify this situation, we define the concept of *underload*. Underload in a given time interval is the difference between the number of cores used at any point in the interval and the maximum number of tasks that are simultaneously runnable in the interval. A positive underload indicates insufficient core reuse, as a long-idle core has been chosen rather

[2] In all execution traces, cores have been renumbered such that 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 the 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	–

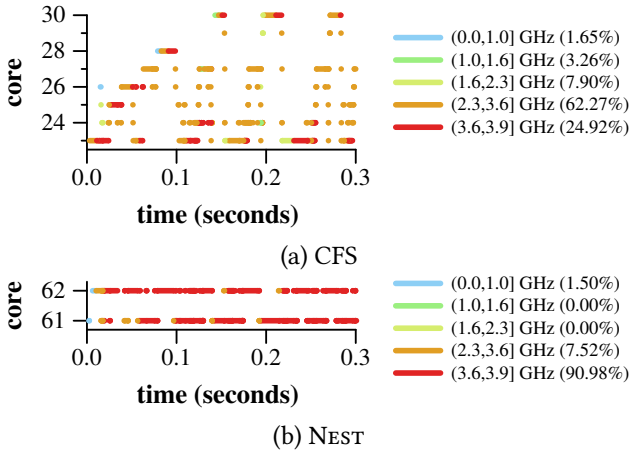

Figure 2. Core frequency trace for LLVM configuration for build with Ninja, when using the schedutil governor, on a 64-core, 2-socket, Intel Xeon Gold 5218.

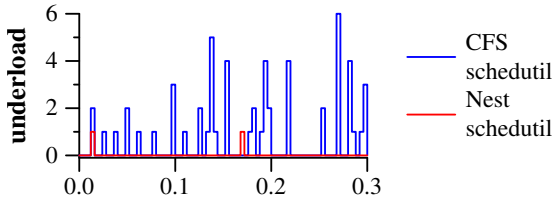

Figure 3. Underload for LLVM configuration for build with Ninja, when using the schedutil governor, on a 64-core, 2-socket, Intel Xeon Gold 5218.

than a core that was already used in the current interval. This long-idle core may initially run at a lower frequency than a core that could have been reused, thus decreasing the performance. Figure 3 shows the underload occurring in the first 0.3 seconds of the execution of LLVM configuration for build with Ninja, with CFS- and Nest-schedutil. We use an interval of 4ms (one tick). While CFS-schedutil gives substantial underload, with Nest-schedutil it has almost disappeared.

To facilitate comparisons, we compute the *underload per second*, *i.e.*, the average amount of underload occurring within the execution of an application over 1 second. Nest is designed to reduce this value without introducing *overload*, *i.e.*, multiple tasks trying to run on a single core.

Performance analysis. As there is no software configuration benchmark, we use the configuration scripts for the software in the Phoronix Timed Code Compilation Test Suite [16]. We consider only software where the configuration scripts are generated using `autotools` or `cmake`, which clearly separate configuration from the rest of the build. Figure 4 shows the underload per second for these scripts, with CFS and Nest, using the *schedutil* and *performance* governors, based on one run. Figure 5 shows the speedups achieved with Nest. Figure 6 shows the frequencies observed on the cores that are executing the configuration script. Figure 4 shows that Nest reduces the number of cores used, almost eliminating the underload. As a result, the cores running tasks become highly utilized and, as shown by Figure 6, the highest frequencies are almost always used. Speedups compared to CFS-schedutil exceed 5% except on NodeJS, which is trivial. The greatest speedup, on the E7-8870 v4, is 37%.

We next compare the speedups shown in Figure 5 by governor. On the recent 6130 and 5218 machines, Nest gives about the same speedup with both *schedutil* and *performance*, and gives much greater speedup than CFS-performance. CFS-performance gives little speedup (*i.e.*, never more than 5%) 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 sometimes gives slightly 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, preventing the cores from reaching the highest turbo frequencies.

Next, we compare the speedups (Figure 5) across machines. The speedups are comparable for the two 6130 machines. As the computation fits into one socket, the number of sockets is irrelevant. The 5218, which is faster than the 6130, shows

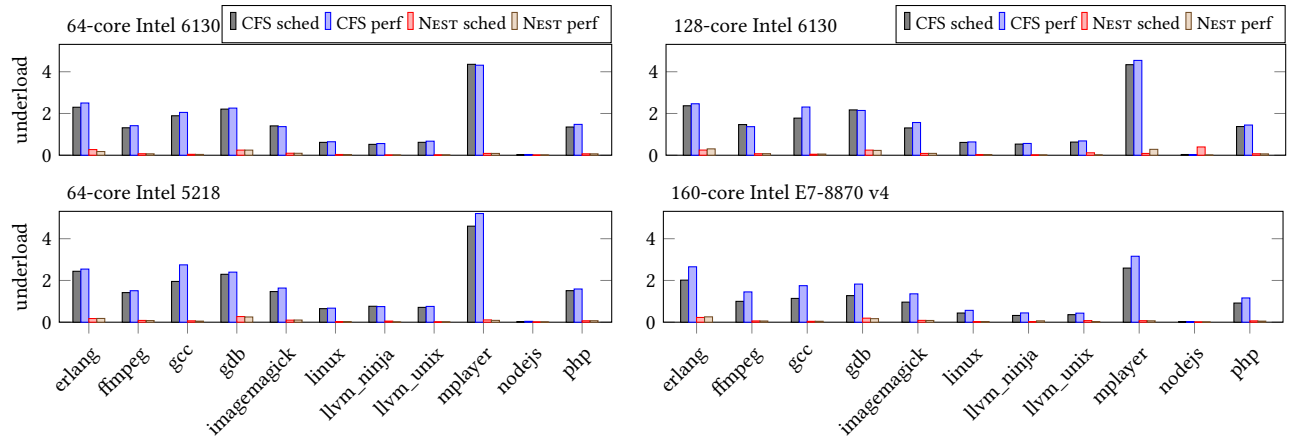

Figure 4. Underload per second with CFS and Nest. "sched" abbreviates "schedutil" and "perf" abbreviates "performance".

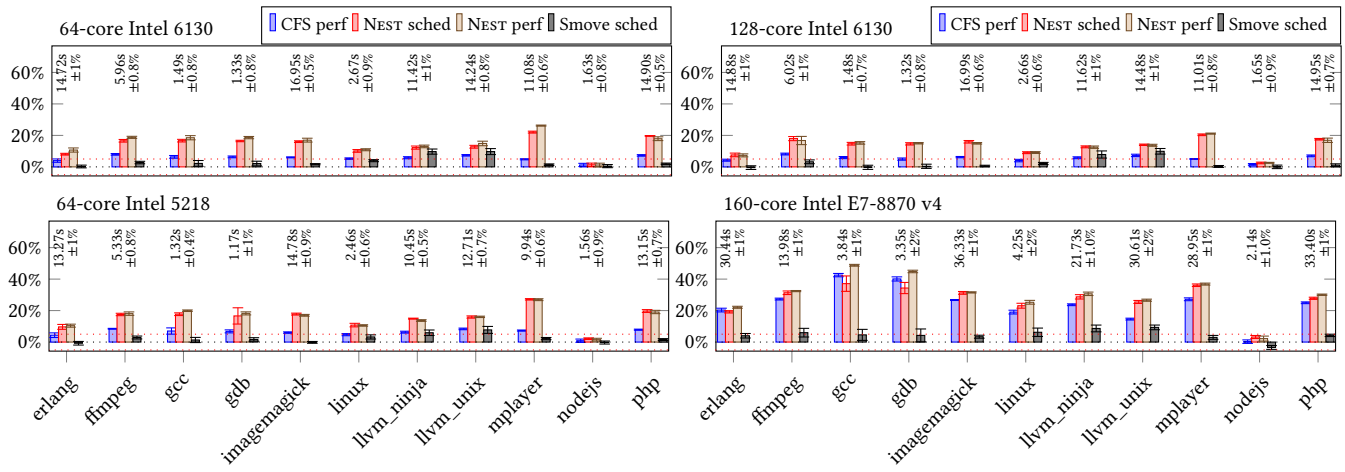

Figure 5. Configuration tests, speedup as compared to CFS schedutil. The average execution time and the percentage standard deviation with CFS-schedutil are shown at the top of each graph.

both better performance with Nest and better speedup than observed on the 6130 machines. Finally, the older E7-8870 v4 is the slowest, as indicated by the running times for CFS-schedutil listed at the top of the graph, which are always around 2× that of the other machines. Still, it achieves the greatest speedup (up to 37% for Nest-schedutil).

At high frequencies, cores consume more energy than at lower frequencies, and thus Nest could increase energy consumption. Figure 7 shows the CPU energy consumption. As Nest reduces the execution time, it reduces the machine's CPU energy consumption by up to 19% between the start and end of its execution. On these tests, 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 energy 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 energy can be saved by concentrating all of the tasks on a single socket, the greatest CPU energy 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 0.5, 2, or 10. On these variants, we test llvm_ninja and mplayer configuration, using schedutil. The only change is if we remove the reserve nest, degrading performance by around 5% on the 6130 and 5218 machines, and by up to 16% on the E7-8870 v4. The reserve nest allows the primary nest to remain small, but allows some extra cores to be chosen just because they are idle, without considering recent load as done by CFS.

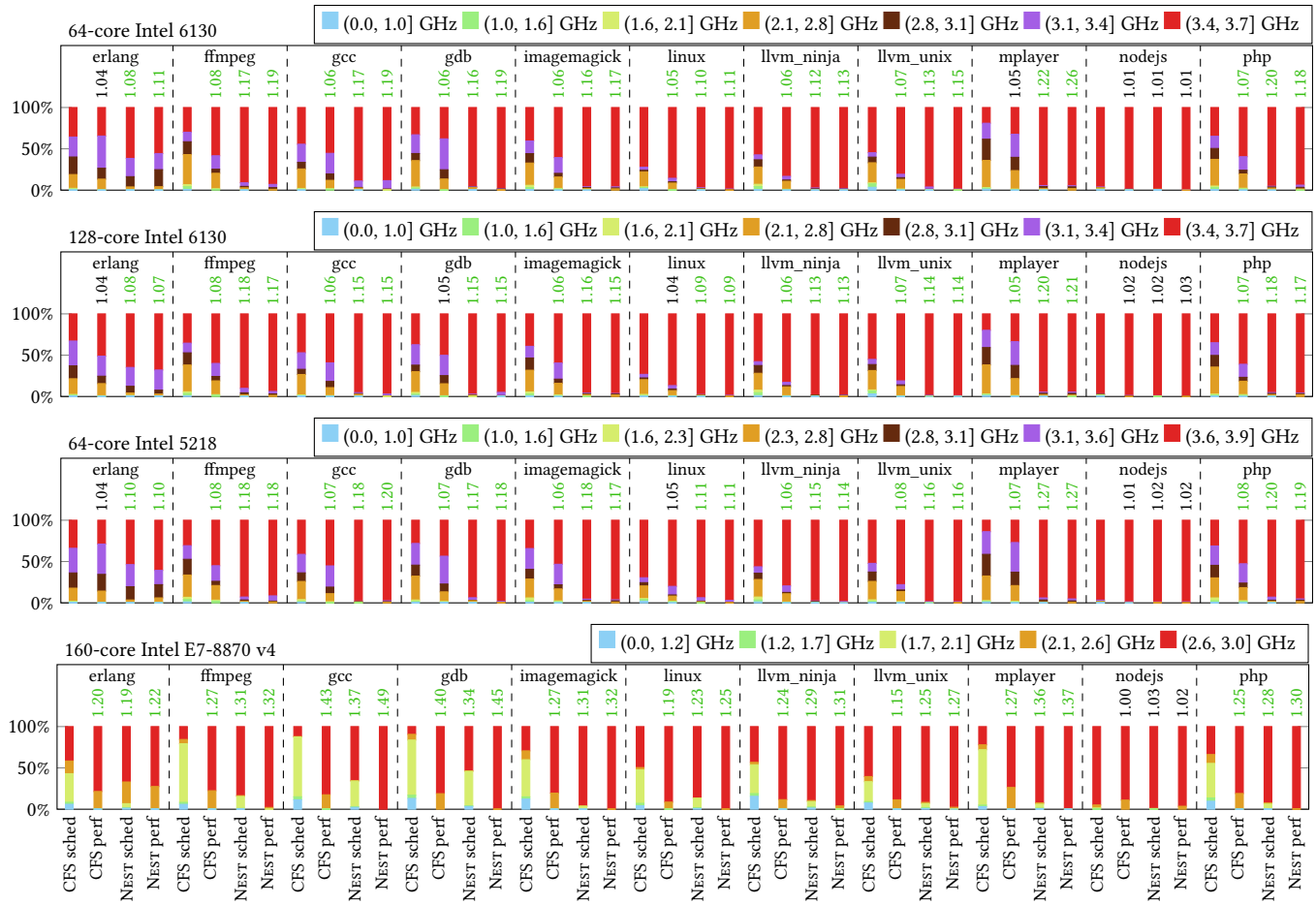

Figure 6. Configuration tests, frequency distribution. The numbers above the bars indicate the speedup as compared to CFS schedutil. Green numbers indicate a speedup of more than 5%.

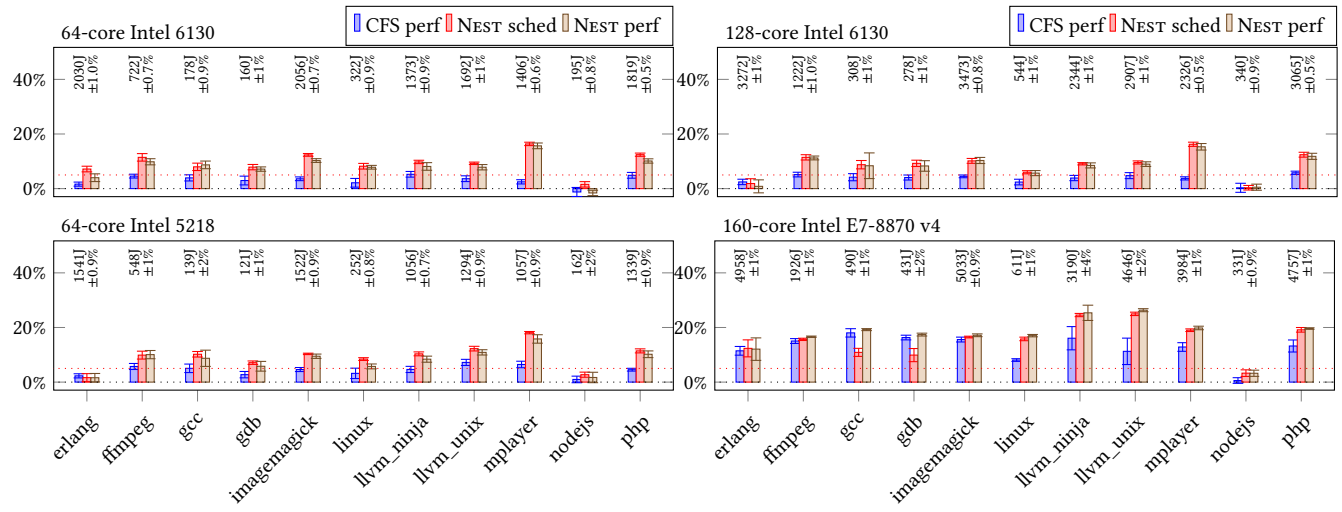

Figure 7. Configuration tests, reduction in CPU energy consumption as compared to CFS-schedutil. The average energy consumption and the percentage standard deviation with CFS-schedutil are shown at the top of each graph.

Comparison with S_{move}. Figure 5 also compares S_{move}-schedutil to CFS-schedutil. While Nest gives speedups of 10% to over 20% for almost all configuration scripts on the 6130's and the 5218, and even more speedup 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 task on the core of the parent only when the frequency observed at the last clock tick on the core chosen by CFS is low. On the 6130's and the 5218, when a core 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 a high frequency, and thus that S_{move}'s placement heuristic is unnecessary. S_{move} gives slightly higher speedups on the E7-8870 v4, but they remain far from those of Nest.

Software configuration is a best case for S_{move}, because there is mostly only one executing task, with one task 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.[3] We give the latter applications names ending in "-eval". We omit the benchmarks that do not run on our Debian OpenJDK 11.0.2.[4] To avoid interference of the JIT compiler and the garbage collector, the DaCapo benchmark developers propose to consider only the last of N runs. 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 applications with a small fixed number of tasks and highly multicore applications with varied execution patterns. The results with Nest (Figure 10) range from a 6% degradation with fop, which only involves one application task (in addition to JIT compilation and garbage collection), on the E7-8870 v4 (the only degradation 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 have a high underload per second (indicated as "u:X" in Figure 10). They achieve higher frequencies with Nest (Figure 11). These applications also run significantly faster on the two-socket machines than on the four-socket 6130. We focus on h2.

[3]Jar files dacapo-9.12-MR1-bach.jar and dacapo-evaluation-git+309e1fa.jar.
[4]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.

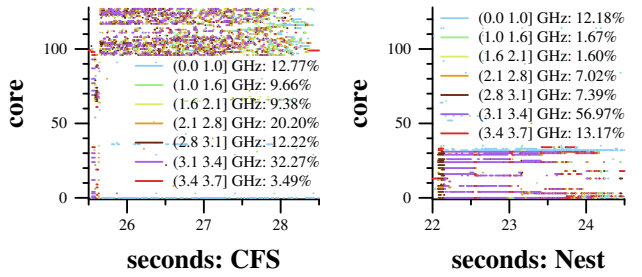

Figure 8. One run of h2 on CFS (left) and on Nest (right).

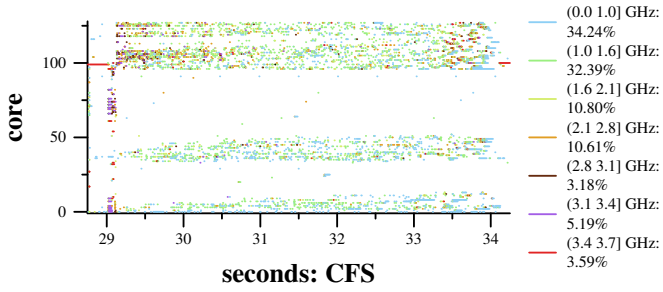

Figure 9. A slow run of h2 on CFS.

We first consider the performance on the four-socket 6130. Figure 8 shows a typical run of h2 on this machine, with CFS-schedutil and with Nest-schedutil. CFS-schedutil disperses the tasks over most of the cores of one socket, causing tasks to spend almost 2/3 of their time at low turbo (at most 3.1 GHz) or subturbo frequencies. In contrast, Nest-schedutil keeps the tasks on at most around 10 cores, causing the tasks to spend more than 2/3 of their time at high turbo frequencies (above 3.1 GHz), giving a speedup of around 20%. Some CFS-schedutil runs disperse the tasks over multiple sockets (Figure 9). In this case, there is little utilization of any given core, causing the cores to drop to the lower frequencies for more than 2/3 of the execution time. Nest always keeps the h2 tasks on a single socket, and thus gives a speedup of more than 2× as compared to Figure 9.

The *performance* governor forces the frequencies of active cores to be at least the nominal frequency. Thus, on the four-socket 6130, both CFS-performance and Nest-performance mostly use the turbo frequencies. Still, with Nest, the higher frequencies are used more often, because Nest reduces the number of cores used. The results on the two-socket 6130 and 5218 are similar to the four-socket case, but with only two sockets, the tasks are less dispersed, giving better performance for CFS and thus less speedup with Nest.

The task 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 tasks 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 tasks that frequently sleep for brief periods,

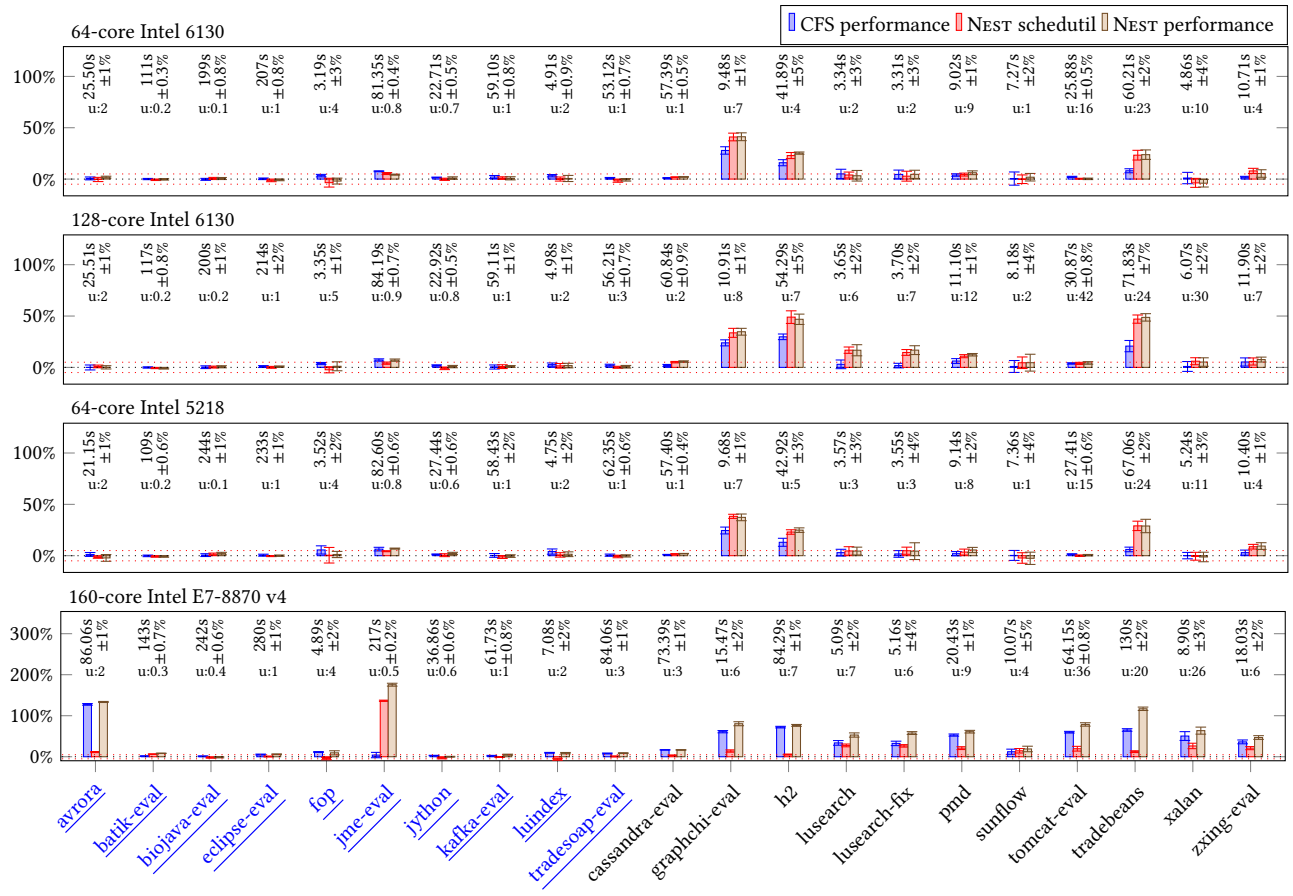

Figure 10. DaCapo tests, speedup as compared to CFS schedutil. Blue applications (underlined) involve only one or a few tasks. The average execution time, the percentage standard deviation, and the underload per second (u:X) with CFS-schedutil are shown at the top of each graph.

which causes the E7-8870 v4 to use the lowest frequencies. Even Nest's spinning is not sufficient to defeat this tendency. As for the 6130 and 5218, the *performance* governor causes the E7-8870 v4 to use turbo frequencies. For h2, as Nest-performance also reduces the number of cores used, it gives a larger speedup than CFS-performance.

Impact of Nest features. We study the performance of h2, graphchi-eval, and tradebeans if we remove spinning, nest compaction, or the reserve mask. Spinning has the greatest impact; removing it gives a degradation of 10-21% on the two-socket 6130 and 5218, and of 17-26% on the four-socket 6130. Without spinning, there is not enough core utilization to motivate the use of the higher frequencies. A too short spin (up to 1 tick) gives a small degradation of 2-7%. A too long spin (up to 20 ticks) also reduces frequencies, due to the large number of active cores. Eliminating nest compaction allows h2 and graphchi to spread out across too many cores, reducing their performance by around 5%. On the other hand, several Nest features are detrimental to tradebeans. For example, on the 2-socket machines, tradebeans' performance is improved by around 20% by reducing the nest compaction delay to 1 tick. With this change, the threads stay within a single socket, improving performance. Finally, unlike for the configure benchmark, the reserve mask has little impact on h2, graphchi-eval, and tradebeans; the primary mask builds up a pool of cores that the tasks use, without falling back to the reserve mask or CFS.

5.4 NAS Parallel Benchmarks evaluation

The NAS Parallel Benchmarks [2] are a collection of HPC kernels. We use version 3.4 with OpenMP.[5] For conciseness, we show only the results for the "C" datasets, the largest of the standard test problems.[6] These benchmarks each involve one task per core. In the optimal case, each task is immediately placed on its own core at the time of fork, and remains there throughout its execution. Overloading some cores, while leaving others empty, causes the tasks to become

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

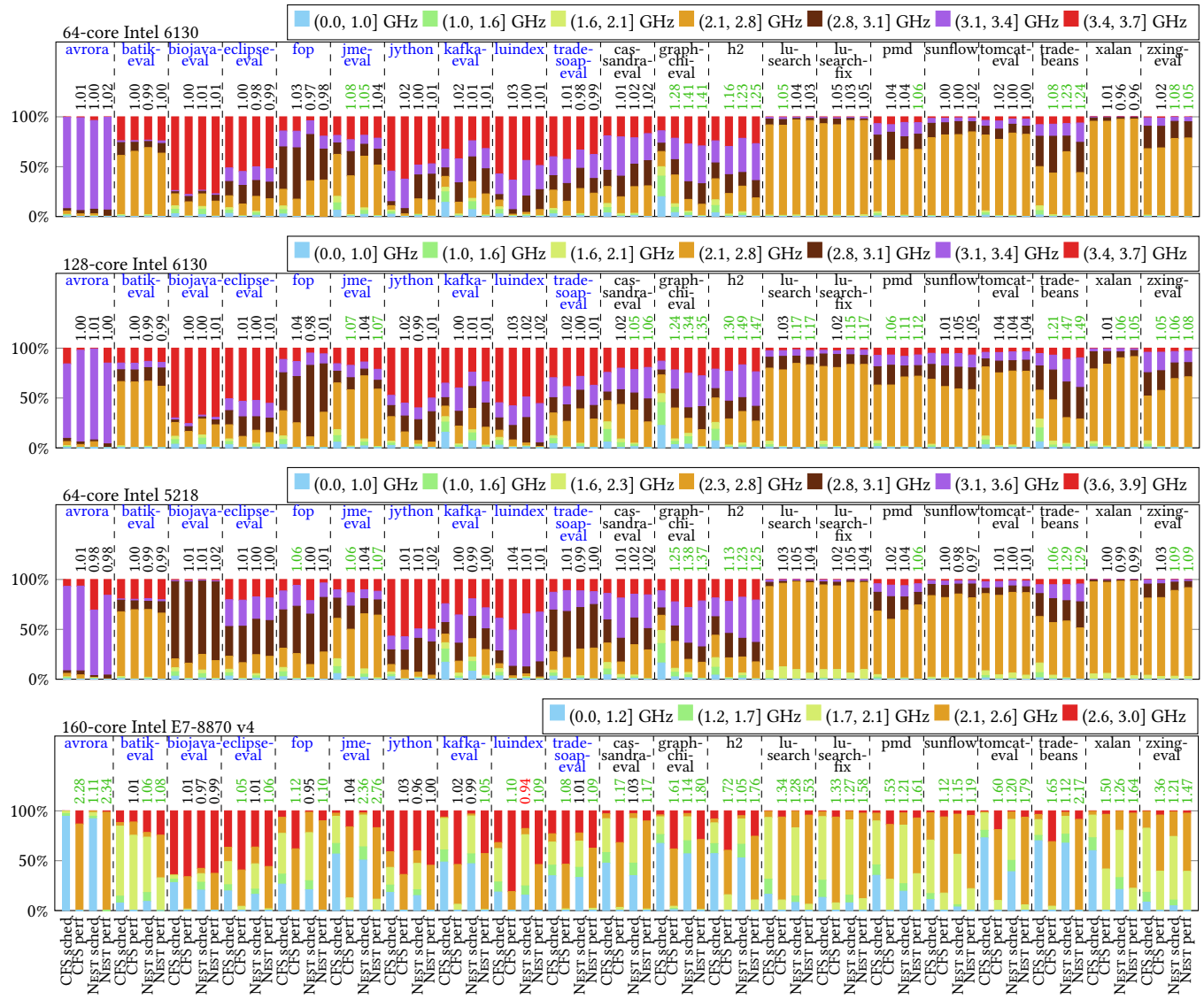

Figure 11. DaCapo tests, frequency distribution. The numbers above the bars indicate the speedup as compared to CFS-schedutil. Green numbers indicate a speedup of more than 5%. Red numbers indicate a degradation of more than 5%.

unsynchronized, requiring some to sleep to wait for others to complete their current work. The challenge for Nest is to achieve the optimal placement.

As shown in Figure 12, on the two-socket 6130 and 5218, CFS and Nest have essentially the same performance. Typically, both schedulers place each task on its own core as it is forked, the tasks 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 (31% and 28%, respectively), LU (166% and 160%, respectively), and UA (89% and 87%, 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 16% on BT (Nest-schedutil) 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 task 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 task wakeup. This feature allows the primary nest to quickly

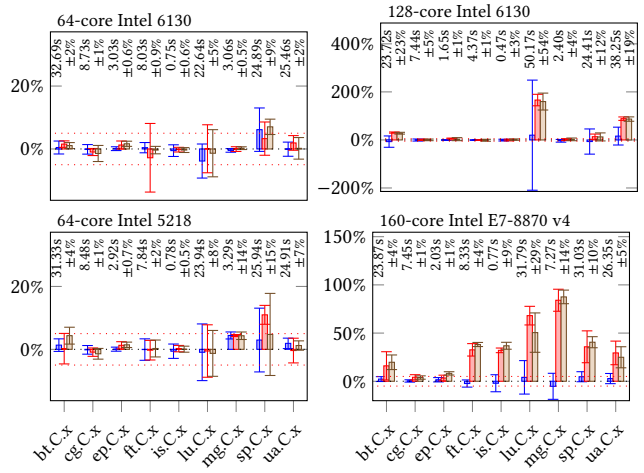

Figure 12. NAS tests, speedup as compared to CFS schedutil. The schedulers are color-coded as Figures 5, 7 and 10.

reach a size close to the number of concurrent tasks. 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 Nest feature that most affects NAS is the favoring of recently used cores, *i.e.*, the attached core or the previously used core. For example, on the 5218, MG has a 15% slowdown without these features. These features help tasks stay on their original cores. Features such as nest compaction, the reserve nest, and spinning have less impact on NAS, as they are rarely or never triggered.

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

5.5 Phoronix multicore suite evaluation

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 [15] comprises 90 benchmarks, each involving one or more tests, for a total of up to 222 tests that we were able to run (some tests were omitted on some machines due to installation issues). This is a much larger test set than used in recent scheduling papers [4, 6, 7, 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 7% of the tests have a speedup above 5% for Nest-schedutil, and this is the case for 21% of the tests on the E7-8870 v4.

Figure 13 shows 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. The tests are

Table 4. Overview of the Phoronix multicore results.

CPU	scheduler	slower by > 20%	slower by (5,20]%	same	faster by (5,20]%	faster by > 20%
2 socket 6130	CFS-perf.	0 (0%)	1 (0%)	206 (93%)	9 (4%)	6 (3%)
	Nest-sched.	1 (0%)	15 (7%)	191 (86%)	11 (5%)	4 (2%)
4 socket 6130	CFS-perf.	2 (1%)	7 (3%)	190 (87%)	9 (4%)	10 (5%)
	Nest-sched.	1 (0%)	19 (9%)	159 (73%)	21 (10%)	18 (8%)
2 socket 5218	CFS-perf.	0 (0%)	1 (0%)	200 (92%)	10 (5%)	6 (3%)
	Nest-sched.	0 (0%)	12 (6%)	189 (87%)	12 (6%)	4 (2%)
4 socket E7-8870 v4	CFS-perf.	0 (0%)	5 (3%)	94 (61%)	18 (12%)	37 (24%)
	Nest-sched.	1 (1%)	13 (8%)	107 (69%)	25 (16%)	8 (5%)

numbered according to their order on the Phoronix web site [15]. A key is provided in Table 5 in the appendix.

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 tasks 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 for configuration (*e.g.*, Figure 2), CFS-schedutil spreads the tasks out over all of the cores, and the tasks 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 tasks on a small set of cores on a single socket, giving speedups of 26-97% on the 6130 and 5218 machines. On the E7-8870 v4, Nest-schedutil also concentrates the tasks 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 the wakeup work conservation of Nest allows them to scatter across the machine. On the 6130 and 5218, with CFS-schedutil, intensively computing tasks share physical cores, reducing their performance as compared to Nest-schedutil, where tasks mostly run on a core that has an idle hyperthread. On the E7-8870 v4, the scattering of running tasks done by Nest's wakeup work conservation 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 (22%) is on the four-socket 6130. In this case, with CFS-schedutil, most of the tasks start on one socket, but some migrate to other sockets

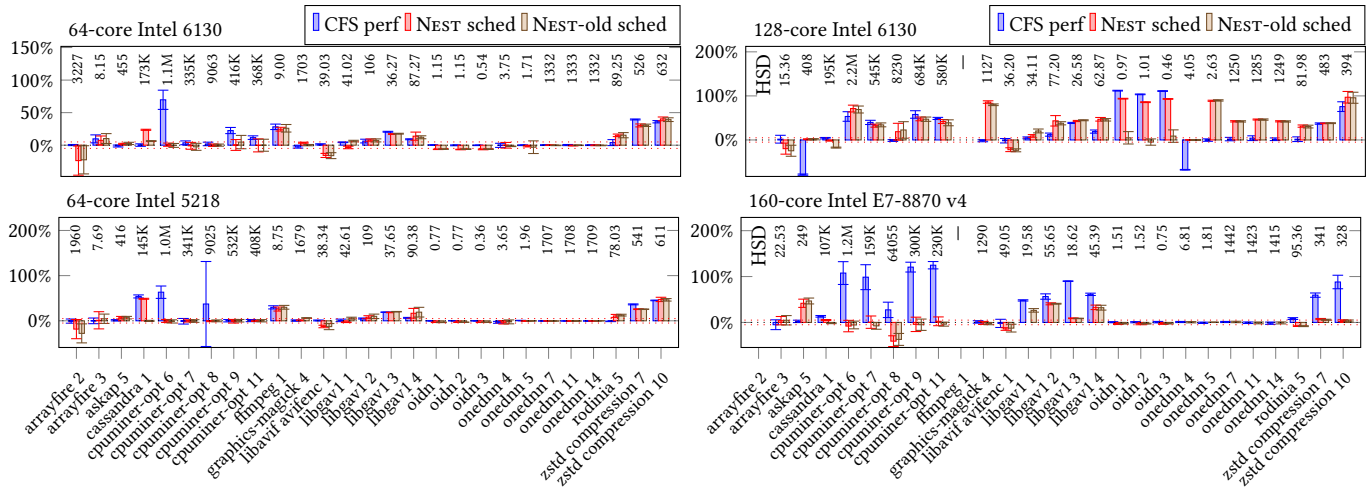

Figure 13. Phoronix multicore suite, speedup as compared to CFS schedutil. The numbers at the top of each graph show the metric value for CFS-schedutil. Cases where CFS-schedutil has a standard deviation of more than 15% are omitted.

over time, resulting in a number of concurrent tasks (up to 12) on these other sockets that allows them to reach the middle turbo frequencies. Nest-schedutil keeps the tasks on the initial socket, where they run at the lowest turbo frequency, reducing performance.

5.6 Other applications

We briefly consider some other benchmarks relevant to multicore scheduling and some other execution contexts.

***Hackbench and schbench* [5].** Hackbench involves pairs of processes or threads that constantly exchange messages. Its execution time is thus massively dominated by the time to schedule the threads (96% system time with CFS). We run `hackbench -g 100 -l 10000`, as used in the recent kernel commit 04b4b006139b. Nest gives a substantial slowdown. While the time on the 5218 with CFS-schedutil is 22.5 seconds, with Nest-schedutil it is sometimes 76 seconds and sometimes 380 or more seconds. The difference correlates with the number of instruction cache misses: 30M with CFS-schedutil, 49M with Nest-schedutil in the faster case, and 319M with Nest-schedutil in the slower case. Nest adds a lot of code to core selection, which could be optimized. Furthermore, the behavior of hackbench is atypical, as evidenced by our results on the very large Phoronix multicore suite.

Schbench is a scheduling benchmark that reports 99.9th percentile tail latency. We test schbench via the Phoronix infrastructure (2-32 message threads, and 2-32 workers per message). There is not a clear advantage for either CFS or Nest: sometimes the results are the same, sometimes CFS or Nest has a significantly longer tail latency. Like hackbench, schbench tests an extreme case.

Server tests. We evaluated CFS- and Nest-schedutil on the Intel two-socket 6130 with tests from the Phoronix server suite, including web-servers, databases and key value stores.

Nest is typically slower than CFS on the apache-siege benchmarks as the number of concurrent users/requests increases. On the other hand, for nginx, Nest has comparable performance to CFS, even as the number of requests increases. Nest also gives similar performance to CFS for node.js and php. Nest improves the performance of the key-value store leveldb by 25%, and improves the performance of the key-value store redis by 7%. Nest improves the performance of the perl benchmark by up to 16% but loses about 5% for a Random Read test of rocksdb.

Multiple concurrent applications. All of our previous tests involve only one application. We now briefly consider the impact of Nest when two applications run in parallel.

Figure 13 shows that Nest improves the performance of compress-zstd Compression Level: 3, 8 Long Mode (Zstd-compression-7, 10) and Libgav1 Chimera 1080p (libgav1-4) by up to 96% and 46%, individually. When we run them in parallel, we observed 4-48% improvement for Zstd-compression-7, 3-12% improvement for Zstd-compression-10, and 2-34% improvement for libgav1-4 compared to CFS in the multi-application scenario. Moreover, Some applications are faster with Nest in the multi-application scenario than with Nest in the single-application scenario, by 19% for Zstd-compression-7 and by 4-8% for libgav1-4.

Mono-socket machines. While Nest primarily targets large servers, we have also done some tests on two single socket machines, an Intel Xeon 5220 (Cascade Lake, 36 cores, maximum turbo frequency 3.9GHz) and an AMD Ryzen 5 PRO 4650G (12 cores, maximum turbo frequency 4.2GHz).

On the Intel 5220, for the configuration benchmarks, we obtain speedups similar to those of the 6130 and 5218. The small number of concurrent tasks again implies that the number of sockets has little impact on the results. For the DaCapo benchmarks, Nest only improves h2, graphchi-eval, and tradebeans. The speedups are lower than on the larger Intel machines. The thread dispersal of CFS does not result in threads being placed on multiple sockets, and thus there is less opportunity to improve performance with Nest. Finally, the performance of the NAS benchmarks with Nest (schedutil and performance) is identical to that of CFS.

On the AMD 4650g, for the configuration benchmarks, Nest-schedutil obtains high speedups of 20-80% in almost all cases. The speedups approach those obtained with CFS-performance. Nest-performance gives even higher speedups of 27-157%. For the DaCapo benchmarks, Nest-schedutil improves the performance of most benchmarks by 10-30%, but typically lags behind CFS-performance. Nest-performance meets and sometimes exceeds CFS-performance, with the speedups of 20-180% in most cases. The performance of the NAS benchmarks with Nest is again identical to that of CFS.

6 Related Work

Recent work [4, 7, 10, 11] has shown that for modern multicore servers a critical component of a scheduler is its core-selection strategy. 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.* [7], however, found that work conservation is not enough – CFS can induce suboptimal performance even when every task is placed on an idle core, when low frequency cores are chosen.

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 task a home node, and the scheduler tries to keep a task 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 task 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 exploits DVFS on servers in order to offer more energy efficient cloud computing [17, 18].

Our work targets the problem of tasks 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.* [6] adjust the CPU time accounting to maintain fairness in this situation. We focus on the choice of cores, to avoid placing tasks on slow cores when possible.

Core scheduling [19] adjusts core selection to avoid placing untrusted tasks 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 tasks 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 tasks changes frequently, and maintains the performance of CFS on applications that involve only one or a handful of tasks 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.

Most of the implementation of Nest amounts to a single block of code placed in front of the core selection function of CFS, making Nest easy to port to other Linux kernel versions. The effect of Nest is to concentrate tasks on a small set of cores, without introducing overload. We have seen that this strategy results in performance improvements as compared to Linux v5.9's scheduler on our test machines, in which this placement strategy results in the cores running at higher frequencies, and thus the applications terminating more quickly. The impact of Nest in the more general case will depend on the evolutions in the Linux kernel's power management strategy and the core frequency management strategy of the target hardware.

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

References

[1] AMD. 2022. AMD Turbo Core Technology. https://www.amd.com/en/technologies/turbo-core.

[2] 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. 1991. The NAS

parallel benchmarks summary and preliminary results. In *Supercomputing*. Seattle, WA, USA, 158–165.

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

[4] Justinien Bouron, Sebastien Chevalley, Baptiste Lepers, Willy Zwaenepoel, Redha Gouicem, Julia Lawall, Gilles Muller, and Julien Sopena. 2018. The Battle of the Schedulers: FreeBSD ULE vs. Linux CFS. In *USENIX Annual Technical Conference*. 85–96.

[5] Matt Fleming. 2017. A survey of scheduler benchmarks. https://lwn.net/Articles/725238/.

[6] Mathias Gottschlag, Philipp Machauer, Yussuf Khalil, and Frank Bellosa. 2021. Fair Scheduling for AVX2 and AVX-512 Workloads. In *USENIX Annual Technical Conference*. 745–758.

[7] Redha Gouicem, Damien Carver, Jean-Pierre Lozi, Julien Sopena, Baptiste Lepers, Willy Zwaenepoel, Nicolas Palix, Julia Lawall, and Gilles Muller. 2020. Fewer Cores, More Hertz: Leveraging High-Frequency Cores in the OS Scheduler for Improved Application Performance. In *USENIX Annual Technical Conference*. 435–448.

[8] Intel. 2022. 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] Kimberly Keeton. 2015. The Machine: An Architecture for Memory-centric Computing. In *Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*.

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

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

[12] Vlad Nitu, Boris Teabe, Alain Tchana, Canturk Isci, and Daniel Hagimont. 2018. Welcome to zombieland: practical and energy-efficient memory disaggregation in a datacenter. In *EuroSys*. 16:1–16:12.

[13] Markus Nordén, Henrik Löf, Jarmo Rantakokko, and Sverker Holmgren. 2005. Geographical Locality and Dynamic Data Migration for OpenMP Implementations of Adaptive PDE Solvers. In *IWOMP 2005: OpenMP Shared Memory Parallel Programming*. 382–393.

[14] Yuvraj Patel, Leon Yang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. 2020. Avoiding scheduler subversion using scheduler-cooperative locks. In *EuroSys*. 9:1–9:17.

[15] Phoronix Test Suite. 2021. Multi-Core. https://openbenchmarking.org/suite/pts/multicore.

[16] Phoronix Test Suite. 2021. Timed Code Compilation. https://openbenchmarking.org/suite/pts/compilation.

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

[18] Chia-Ming Wu, Ruay-Shiung Chang, and Hsin-Yu Chan. 2014. A green energy-efficient scheduling algorithm using the DVFS technique for cloud datacenters. *Future Generation Computer Systems* 37 (July 2014), 141–147.

[19] Peter Ziljstra. 2019. sched: Core scheduling. https://lwn.net/ml/linux-kernel/20190218165620.383905466@infradead.org/.

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 and at https://doi.org/10.5281/zenodo.6344960. 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. There are many software dependencies. The artifact 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 5-12) is 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. When there are fewer concurrently running task than cores, CFS is prone to scatter the tasks across the machine, causing tasks to often wake up on recently idle cores running at a low frequency. We claim that in this case performance can be improved, potentially with no additional energy consumption, by restricting the tasks 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 while applications that use all cores intensively do not benefit from this strategy, they also incur little or no performance degradation.

These claims are mainly illustrated by Figures 2-13, and the accompanying discussion in Section 5. 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 5-7. 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.

[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 10 and 11. 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.

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 12. 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.

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 13. 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.

The main steps of this experiment are as follows:

- 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 the result file a descriptive name.
- Repeat the same steps for Nest schedutil, CFS schedutil and CFS performance.
- Use `read_csvs`. provided in `nest-artifact`, to analyze the results.

The considered Phoronix benchmarks are listed in Table 5.

Table 5. Considered Phoronix benchmarks.

askap 5	ASKAP1.0 - Test: Hogbom Clean OpenMP
cassandra 1	Apache Cassandra4.0 - Test: Writes
arrayfire 2	ArrayFire3.7 - Test: BLAS CPU
arrayfire 3	ArrayFire3.7 - Test: Conjugate Gradient CPU
cpuminer-opt 6	Cpuminer-Opt3.15.5 - Algorithm: Blake-2 S
cpuminer-opt 8	Cpuminer-Opt3.15.5 - Algorithm: Myriad-Groestl
cpuminer-opt 11	Cpuminer-Opt3.15.5 - Algorithm: Quad SHA-256, Pyrite
cpuminer-opt 7	Cpuminer-Opt3.15.5 - Algorithm: Skeincoin
cpuminer-opt 9	Cpuminer-Opt3.15.5 - Algorithm: Triple SHA-256, Onecoin
ffmpeg 1	FFmpeg4.0.2 - H.264 HD To NTSC DV
graphics-magick 4	GraphicsMagick1.3.33 - Operation: Resizing
oidn 1	Intel Open Image Denoise1.4.0 - Run: RT.hdr_alb_nrm.3840x2160
oidn 2	Intel Open Image Denoise1.4.0 - Run: RT.ldr_alb_nrm.3840x2160
oidn 3	Intel Open Image Denoise1.4.0 - Run: RTLightmap.hdr.4096x4096
rodinia 5	Rodinia3.1 - Test: OpenMP Leukocyte
zstd compression 10	Zstd Compression1.5.0 - Compression Level: 3, Long Mode - Compression Speed
zstd compression 7	Zstd Compression1.5.0 - Compression Level: 8, Long Mode - Compression Speed
libavif avifenc 1	libavif avifenc0.9.0 - Encoder Speed: 6, Lossless
libgav1 4	libgav10.16.3 - Video Input: Chimera 1080p
libgav1 3	libgav10.16.3 - Video Input: Chimera 1080p 10-bit
libgav1 2	libgav10.16.3 - Video Input: Summer Nature 1080p
libgav1 1	libgav10.16.3 - Video Input: Summer Nature 4K
onednn 5	oneDNN2.1.2 - Harness: IP Shapes 1D - Data Type: f32 - Engine: CPU
onednn 4	oneDNN2.1.2 - Harness: IP Shapes 3D - Data Type: f32 - Engine: CPU
onednn 11	oneDNN2.1.2 - Harness: Recurrent Neural Network Training - Data Type: bf16bf16bf16 - Engine: CPU
onednn 7	oneDNN2.1.2 - Harness: Recurrent Neural Network Training - Data Type: f32 - Engine: CPU
onednn 14	oneDNN2.1.2 - Harness: Recurrent Neural Network Training - Data Type: u8s8f32 - Engine: CPU