Fair Scheduling for AVX2 and AVX-512 Workloads

Mathias Gottschlag, Philipp Machauer, Yussuf Khalil, and Frank Bellosa,
Karlsruhe Institute of Technology

https://www.usenix.org/conference/atc21/presentation/gottschlag

This paper is included in the Proceedings of the
2021 USENIX Annual Technical Conference.

July 14–16, 2021

978-1-939133-23-6

Fair Scheduling for AVX2 and AVX-512 Workloads

Mathias Gottschlag
Karlsruhe Institute of Technology

Philipp Machauer
Karlsruhe Institute of Technology

Yussuf Khalil
Karlsruhe Institute of Technology

Frank Bellosa
Karlsruhe Institute of Technology

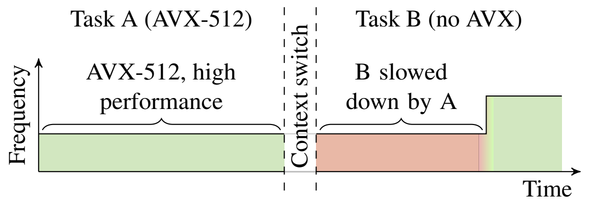
Abstract

CPU schedulers such as the Linux Completely Fair Scheduler try to allocate equal shares of the CPU performance to tasks of equal priority by allocating equal CPU time as a technique to improve quality of service for individual tasks. Recently, CPUs have, however, become power-limited to the point where different subsets of the instruction set allow for different operating frequencies depending on the complexity of the instructions. In particular, Intel CPUs with support for AVX2 and AVX-512 instructions often reduce their frequency when these 256-bit and 512-bit SIMD instructions are used in order to prevent excessive power consumption. This frequency reduction often impacts other less power-intensive processes, in which case equal allocation of CPU time results in unequal performance and a substantial lack of performance isolation.

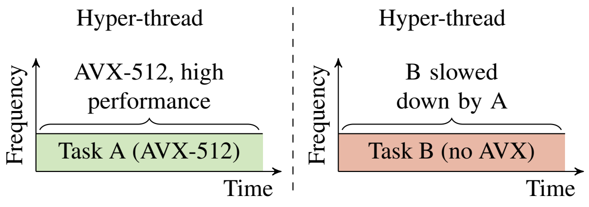
We describe a modification to existing schedulers to restore fairness for workloads involving tasks which execute complex power-intensive instructions. In particular, we present a technique to identify AVX2/AVX-512 tasks responsible for frequency reduction, and we modify CPU time accounting to increase the priority of other tasks slowed down by these AVX2/AVX-512 tasks. Whereas previously non-AVX applications running in parallel to AVX-512 applications were slowed down by 24.9% on average, our prototype reduces the performance difference between non-AVX tasks and AVX-512 tasks in such scenarios to 5.4% on average, with a similar improvement for workloads involving AVX2 applications.

1 Introduction

One common requirement for schedulers is to provide an acceptable quality of service to the tasks in the system [7]. Fair schedulers evenly share CPU performance among the tasks or groups of tasks in the system to achieve good quality of service for all tasks [15]. In the past, it was commonly assumed that even shares of the CPU time result in even shares of CPU performance. The Linux Completely Fair Scheduler [20], for



(a) Unfairness after context switches



(b) Unfairness due to hyper-threading

Figure 1: With AVX2 and AVX-512, equal CPU time shares do not translate into fair shares of CPU performance. AVX2 and AVX-512 cause the CPU core to reduce its frequency which can affect other non-AVX tasks. In these situations, our prototype would increase the priority of task B to achieve improved fairness. In this figure, the two frequencies represent the non-AVX and AVX-512 turbo levels.

example, always executes the task with the lowest CPU time used (scaled by priority) to allocate equal shares of CPU time to individual tasks.

In systems with dynamic voltage and frequency scaling (DVFS), the assumption that equal CPU time results in equal performance does not hold. Reduced frequencies affect the performance of applications differently depending on how memory-limited the applications are, and techniques have been developed to include this effect when distributing CPU time [14]. Such techniques have not found their way into common operating systems, though, most likely because fairness mostly matters when system utilization is high, in which case operating systems rarely reduce the CPU frequency.

Recent CPUs, however, have started to use autonomous DVFS outside of the control of the operating system to prevent excessive power consumption. In the last years, transistor scaling has increased power density, leading to a situation where the performance of recent CPUs is largely limited by their power consumption [28]. One common technique to improve CPU performance in this power-limited design regime is to use additional transistors to implement specialized accelerators which remain inactive most of the time. CPU cores can operate at high frequencies while the accelerators are inactive, whereas use of the accelerators requires a temporary frequency reduction to prevent excessive power draw. This adaptation of the CPU frequency to meet thermal and power supply constraints increases individual cores' frequencies to always utilize their whole power budget but also decreases frequencies if the cores risk overheating or instability, similar to techniques such as Turbo Boost [1] which increases the frequency of the whole processor when some cores are idle.

Recent Intel CPUs provide an example for such behavior where low-power code is executed at increased frequencies. For some workloads, these CPUs provide a substantial performance advantage over previous CPU generations due to the introduction of the AVX2 and AVX-512 SIMD instruction sets [6] which support operations on up to 512-bit vector registers. The large difference in power consumption between AVX2/AVX-512 instructions and other instructions, however, requires individual CPU cores to reduce their frequency while they execute AVX2 and AVX-512 code, whereas other less power-intensive code is executed at higher frequencies [23]. As an example, the Intel Xeon Gold 6130 server CPU executes non-AVX code at an all-core turbo frequency of 2.8 GHz, whereas AVX2 and AVX-512 code is executed at 2.4 GHz and 1.9 GHz, respectively [4].

If only AVX2 or AVX-512 code would be selectively slowed down, no problem for fair schedulers would arise. Only tasks *choosing* to execute AVX2 or AVX-512 code would be slowed down and the increased throughput of these instructions would more than compensate the frequency reduction. However, the impact of the lower frequency is not limited to the AVX2 and AVX-512 code responsible for the reduction [10]. In two scenarios, the frequency reduction also affects other potentially unrelated tasks:

1. Context switches: When the CPU switches from an AVX2 or AVX-512 task to a non-AVX task, the CPU waits for a fixed timeout before restoring the standard frequency [5]. Such a delay is sensible as it limits worst-case overhead due to frequent clock changes. The following task is consequently slowed down as well.

2. Hyper-threading: When one hyper-thread executes either AVX2 or AVX-512 code, the whole physical core needs to reduce its frequency, so any code running on the other hyper-thread is affected as well even if it does not execute AVX2 or AVX-512 instructions [10]. This effect has shown to be particularly prominent, with workloads often being slowed down by more than 15% when executed alongside an AVX-512 application on Linux [10].

As shown in Figure 1, when contemporary fair schedulers allocate the same share of CPU time to AVX2/AVX-512 tasks which cause a frequency reduction as to other tasks which are also affected by the reduced frequencies, the latter would receive a substantially reduced share of the system's performance. This unfairness and the resulting performance reduction is particularly problematic for tasks with soft real-time requirements and for multi-tenant systems where individual users commonly pay to receive a specific share of the CPU time. While these effects are currently only observed on systems with recent Intel CPUs, we expect future power-limited systems to show similar behavior as outlined above.

In this paper, we show that for these systems it is necessary to rethink our notions of scheduling fairness. In particular, we show that equal CPU time, minimal effects on quality of service, and fair distribution of CPU performance are mutually exclusive goals. We do so by describing a scheduler that is commonly able to achieve the latter when some tasks – called *victim tasks* in the following – are negatively affected by a frequency reduction caused by other tasks.

We present a concrete implementation of this design for Intel systems and AVX2 and AVX-512 instructions. Our approach mitigates the performance impact of frequency reduction via modified CPU time accounting where the CPU time of victim tasks is scaled according to the decrease in CPU frequency (Sections 3). We show how victim tasks can often be recognized by the lack of characteristic register accesses (Section 4). Variable turbo frequencies complicate calculating the actually experienced frequency reduction, so we show how on Intel systems the average frequency reduction during a scheduling time slice can be calculated using only two configurable performance counters and a cycle counter (Section 5). As the load balancing mechanism of the Linux CFS scheduler prevents achieving fairness, we describe an implementation of our design based on the Linux MuQSS scheduler modified to use the CFS scheduling algorithm (Section 6). Our evaluation based on a range of real-world applications shows that our prototype has close to zero runtime overhead, yet is able to improve the fairness in workloads with AVX2 and AVX-512, reducing the performance difference between two applications from 24.9% to 5.4% if one of the applications uses AVX-512 (Section 7). Finally, we discuss different fairness criteria, showing how our approach can be modified to achieve even stronger performance isolation (Section 8), as well as the limitations of our approach (Section 9).

2 AVX Frequency Reduction

To mitigate the unfairness caused by power-intensive instructions, we first need to know which instructions cause the

CPU to reduce its frequency and by how much the frequency is reduced. Our prototype targets the AVX2 and AVX-512 instruction sets. Both are categorized by Intel into "heavy" and "light" instructions, where the former consist of all floating point instructions as well as multiplication instructions, whereas the latter consist of all other instructions [5]. In the default CPU configuration at the maximum non-AVX frequency, all light and heavy instructions have the potential to draw excessive current, which causes voltage drops and system instability. During execution of all these instructions, the CPU therefore increases its voltage to allow for increased currents [8].

Not all 256-bit and 512-bit instructions, however, also cause a frequency change. CPUs with support for AVX-512 provide three different frequency ranges named "non-AVX frequencies", "AVX2 frequencies", and "AVX-512 frequencies" [5]. Unlike the name suggests, "AVX-512 frequencies" are not triggered by arbitrary 512-bit instructions but rather only by 512-bit heavy instructions. Similarly, "AVX2 frequencies" are triggered by 512-bit light instructions and 256-bit heavy instructions. Short stretches of AVX2/AVX-512 instructions or long code sections with infrequent AVX2/AVX-512 instructions may cause less frequency reduction or may not cause any change in frequency at all [18]. After the last AVX2/AVX-512 instruction requiring the frequency reduction, the CPU waits for 670 µs before reverting to a higher frequency [11].

3 Frequency Reduction Compensation

As described in Section 1, there are two situations in which tasks executing AVX2 and AVX-512 code can slow other tasks down [10]. First, the delay before the CPU reverts to a higher frequency means that after a context switch away from an AVX2/AVX-512 task the next task continues to execute at the lower frequency. Second, in a system with hyper-threading both hyper-threads of a physical core share the same frequency, thus a AVX2/AVX-512 task executing on one hyper-thread slows down the task executing on the other hyper-thread.

In a situation where the frequency reduction caused by power-intensive instructions is not limited to tasks executing such instructions, CPU time is not proportional to performance anymore and existing schedulers which allocate equal CPU time to tasks of equal priority fail to provide a fair distribution of CPU performance. In this paper, we take relative application performance compared to isolated execution as the main metric for fairness and consider a distribution of CPU performance "fair" if each task receives the same fraction of the CPU performance that it would receive if it was executing on the CPU alone. In a system without frequency changes and with two tasks, for example, each task would obtain 50% of the performance of the task executed in isolation. If one of the tasks executes power-intensive instructions and the other one does not, however, only the latter task (i.e., the victim task) is affected by the resulting *remote frequency reduction overhead* [11] and receives a lower share of the CPU performance. Note that AVX2/AVX-512 tasks themselves, for example, can be assumed to obtain full CPU performance despite executing at reduced frequencies. They profit from the increased throughput of AVX2 and AVX-512 – applications have little reason to use complex power-intensive instructions if their speedup does not outweigh the frequency reduction.

Current schedulers allocate equal CPU time to individual tasks even though only some are affected by remote AVX overhead, resulting in compromised performance isolation. Some existing techniques using consumed energy as the basis for scheduling [22, 26, 31] may be applicable to solve this problem (see Section 8 for a discussion of these and other scheduling criteria). Such approaches are, however, not viable on current hardware as the CPUs lack interfaces required for sufficiently accurate energy models.

In this paper, we instead propose *frequency reduction compensation* as a simple technique to modify existing fair schedulers to reduce the impact of frequency reduction on the performance of victim tasks. To increase the performance share allocated to these tasks, we propose modifying CPU time accounting to take the performance reduction into account. If less CPU time is credited to a task, the scheduler will automatically schedule the task more often to make up for the perceived CPU time inequality. For victim tasks, we scale the CPU time credited to the tasks by the ratio between the actual average CPU frequency and the frequency at which the tasks could be executed if they were executed in isolation, so that this virtual CPU time matches the CPU throughput experienced by the tasks. The scheduler will then automatically mitigate the performance impact experienced by victim tasks to the point where all tasks receive equal shares of CPU performance. Such frequency reduction can be applied to all scheduling algorithms based on CPU time. In the following sections, we first describe how to identify victim tasks (Section 4) and then describe how to calculate the performance impact due to the frequency reduction (Section 5) before we describe important details in our implementation of the techniques (Section 6).

4 Attribution of Frequency Changes

Any scaling of the virtual CPU time must be limited to victim tasks, as only those tasks are supposed to be executed more frequently. First, we therefore need to identify whether a task is a victim task, i.e., whether it is affected by excessive frequency reduction caused by other tasks.

On recent Intel CPUs, it is trivial to determine whether the CPU frequency was reduced, as the CPUs provide performance events to count the cycles spent at AVX2 and AVX-512 frequency levels [5]. The CPUs do not provide a mechanism for the operating system to detect whether a code region triggers frequency reduction or not, though [11]. While the performance events can be used to detect frequency level transitions, they cannot be used to identify the hyper-thread which caused a physical core to reduce its frequency. Also, during the time after a context switch the counters do not provide information about whether the current task could be executed at a higher frequency.

We therefore detect power-intensive code based on register accesses – many complex instruction set extensions introduce additional architectural registers, so accesses to these registers signal such power-intensive code. In our prototype, we use an approach found in previous work that uses a trap-based mechanism to identify AVX-512 code [10]: When the OS clears the `ZMM_Hi256` and `Hi16_ZMM` bits in the `XCR0` register, context switching for 512-bit vector registers is disabled [2, p. 13-1ff] and AVX-512 instructions trigger undefined instruction exceptions [2, p. 14-34].

Whereas previous work uses this mechanism to permanently disable AVX-512 on select CPU cores [10], we extend the mechanism to AVX2, yet we only trap the first 256-bit or 512-bit register access within a time slice to detect whether the task uses AVX2/AVX-512. At each context switch, we test whether the next task has valid 256-bit or 512-bit register content and flag the task as an AVX2/AVX-512 task if it does. If there is no valid 512-bit register state, we prevent further 512-bit register accesses as described above. If there is no valid 256-bit register state, either, we clear the `AVX` bit in the `XCR0` register as well to make future AVX2 instructions trigger exceptions. Then, if 256-bit registers are not enabled when the undefined instruction exception handler is called, we simply re-enable those registers, flag the current task as an AVX2 task and continue execution. Similarly, if 256-bit registers are already enabled but 512-bit registers are not, we enable 512-bit registers and flag the current task as an AVX-512 task. During the next scheduler invocation, we can test whether the previous task was marked as an AVX2 or AVX-512 task to determine whether to apply frequency reduction compensation.

Note that this implementation triggers two exceptions for the first AVX-512 instruction after a context switch. A more optimized implementation can reduce overhead by checking the register size of the trapped instruction and, if it accesses 512-bit registers, enabling both AVX2 and AVX-512 at once.

As Gottschlag et al. discuss, such a mechanism is not a precise indicator of whether code requires a frequency reduction [10]. As described in Section 2, the conditions for a frequency change are much more complex, so the mechanism can cause false positives as not all 256-bit and 512-bit register accesses cause a transition to AVX2/AVX-512 frequencies. Victim tasks using light AVX2/AVX-512 instructions may therefore not benefit from frequency reduction compensation even if the tasks themselves do not require reduced frequencies. In addition, AVX supports 256-bit registers as well, so our approach cannot cleanly distinguish AVX and AVX2. We discuss the impact of this limitation in Section 9.1.

4.1 CPU Feature Detection

One problem of disabling AVX2 and AVX-512 instructions via changes in the `XCR0` register is that this change breaks CPU feature detection in most applications. Intel states that before using any AVX instructions programs should first read the `XCR0` register via the `XGETBV` instruction and test whether AVX2 and AVX-512 are supported by the OS and then execute the `CPUID` instruction to test whether the required instructions are available [2, p. 14-15]. Related work suggests that this problem can be solved by virtualizing the `CPUID` instruction [10]. Recent Intel CPUs indeed provide an MSR that can be used to disable CPUID. However, we found it impossible to trap the `XGETBV` instruction without also disabling all vector instructions including widely used instruction set extensions such as SSE.

Instead, we propose patching all executables to replace the `XGETBV` instructions with an invalid instruction. In the invalid instruction exception handler, the kernel can then emulate `XGETBV` before returning to the application. We have verified this technique to be functional, and the additional exception causes minimal overhead given that most applications only determine the available CPU features once at startup.

In our evaluation, we wanted to compare our prototype to a stock Linux kernel, where we would not be able to execute such modified applications as the kernel lacks support for virtualization of `XGETBV`. As described in Section 7, we therefore did not integrate the technique into our prototype and instead manually modified the applications to assume that AVX2 and AVX-512 were available.

5 Calculation of the Performance Impact

Whenever a victim task has been identified, CPU time accounting for the task has to be scaled to increase the share of actual CPU time allocated to that task. As described above, the scaling has to occur in proportion to the performance impact. If we assume that the workload is CPU-bound, the performance impact p during a single scheduler time slice is defined by the ratio between the average CPU frequency experienced by the task and the average ideal CPU frequency at which the task could have been executed in isolation:

$$p = \frac{f_{\text{measured}}}{f_{\text{ideal}}} \tag{1}$$

In the case of Intel CPUs, the ideal frequency is the non-AVX frequency for non-AVX tasks and the AVX2 frequency for tasks which accessed 256-bit registers during the time slice. Note that memory-bound workloads suffer less from frequency reduction. We discuss the impact of this limitation in Section 9.2.

Whereas the average CPU frequency during a time slice is easily measurable, the ideal CPU frequency is not. Both non-AVX frequency and AVX2 frequency depend on the turbo level which is selected by the CPU depending on the number of active cores. The number of active cores, however, can change at any point during the time slice, so counting the active cores during scheduler invocations is not sufficient to get a high-quality estimate of the average ideal frequency. Instead, we need to determine the average turbo level throughout the whole time slice. To determine the turbo level, we compare the measured frequency against each frequency expected had the chip operated at one particular turbo level during the time slice. If the measured frequency matches one of the expected frequencies, we can assume the corresponding turbo level. Else, the calculation of the ideal frequency has to be made via linear interpolation between the closest turbo levels.

As a first step, we calculate the expected frequency at a given turbo level for the measured amount of cycles spent at AVX2 and AVX-512 frequency levels. Assuming that during a time slice of length t_{total} the system spends t_i time at frequency f_i, the average frequency during the time slice can be calculated as follows:

$$f = \frac{1}{t_{\text{total}}} \sum t_i f_i \tag{2}$$

In the following, we assume that f_0 is the non-AVX frequency at the given turbo level, whereas f_1 and f_2 are the AVX2 and AVX-512 frequencies, respectively. These frequencies are published by Intel for their server CPUs [4]. As we count the cycles instead of the time spent at the three frequency levels, we substitute $t_i = c_i / f_i$ where c_i are the cycles at frequency f_i. We arrive at the following equation:

$$f(c_0, c_1, c_2) = \frac{f_0 f_1 f_2 (c_0 + c_1 + c_2)}{f_1 f_2 c_0 + f_0 f_2 c_1 + f_0 f_1 c_2} \tag{3}$$

$c_{\text{total}} = c_0 + c_1 + c_2$ is the total CPU cycle count and can be measured via a fixed-function performance counter, so only two programmable performance counters are required. If we further substitute $c_0 = c_{\text{total}} - c_1 - c_2$ as well as $r_1 = c_1 / c_{\text{total}}$ and $r_2 = c_2 / c_{\text{total}}$ as the ratio between the cycles spent at AVX2/AVX-512 frequencies and the total cycle count, we arrive at the final formula for the expected frequency at a specific turbo level:

$$f(r_1, r_2) = \frac{f_0 f_1 f_2}{(f_0 - f_1) f_2 r_1 + (f_0 - f_2) f_1 r_2 + f_1 f_2} \tag{4}$$

While calculating this formula requires a division, all frequencies lie within the same order of magnitude, so the formula can be calculated using fixed-point arithmetic and no floating-point arithmetic is required. The two-dimensional case of this formula for a Xeon Gold 6130 CPU under the assumption that $r_1 = 0$ – the system did not spend any time at the AVX2 frequency level – is shown in Figure 2.

At the end of each time slice, the scheduler measures the cycles spent at AVX2 and AVX-512 frequency levels as well as the average actual frequency. The formula above is then applied to the cycles to calculate the expected frequencies for the given number of AVX2/AVX-512 cycles at the different turbo levels. For each turbo level, the scheduler also calculates $f(0,0)$ as the ideal frequency for non-AVX tasks and $f(r_1, 0)$ as the ideal frequency for AVX2 tasks.[1]

The expected frequencies at the different turbo levels can then be compared to the measured actual frequency to determine the turbo level of the CPU, and that turbo level is used to determine the ideal frequency for the current task. If, as mentioned above, the measured frequency matches neither of the expected frequencies, linear interpolation is applied to determine the ideal frequency. The resulting ideal frequency can then be inserted into equation (1) to obtain the scaling factor for the task's CPU time.

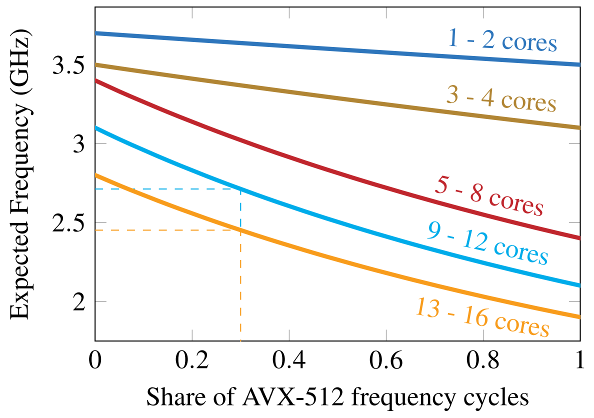


Figure 2: For a given amount of AVX2 and AVX-512 cycles during a time slice, the expected frequencies at the different turbo levels can be compared with the actual measured CPU frequency to determine the turbo level of the CPU. For example, if during a time slice 30% of all cycles were spent at the AVX-512 frequency level while an average frequency of 2.5 GHz was measured, the system likely spent most of the time at the lowest turbo level and some time at the second lowest as indicated by the dashed lines.

6 Implementation

We base the implementation of our design on the MuQSS scheduler [16], modified to use the scheduling policy of the CFS scheduler for enhanced fairness. We initially intended to use the CFS scheduler [20] as it provides particularly strict fairness for non-AVX workloads. While we show that the main scheduling policy of CFS is well-suited to our design, the implementation of CFS is not.

CFS measures the accumulated *virtual runtime* of each task, which is the actual CPU time multiplied by a priority factor [20]. The runqueue is sorted by the virtual runtime of

[1]These calculations can be performed once at startup.

the tasks, and the scheduler always schedules the task with the lowest virtual runtime to ensure that all tasks are given an equal share of the CPU time. Changes to the virtual runtime as described above therefore cause preferential scheduling of victim tasks which counteracts the unfairness caused by AVX frequency reduction.

Our design, however, is incompatible with the current implementation of CFS. CFS maintains one separate runqueue per logical CPU, so any frequency reduction compensation for victim tasks only changes their priority within the runqueue of their current logical CPU. If, for example, one hyper-thread only executes AVX2/AVX-512 tasks, whereas the other only executes non-AVX tasks, the latter tasks only compete against each other during scheduling and there is no preferential treatment compared to the AVX2/AVX-512 tasks.

Frequent load balancing might improve the fairness in such a situation if an idle or more lightly loaded CPU would fetch and execute victim tasks first. In CFS, however, different cores can have different virtual runtime ranges and virtual runtimes are normalized during load balancing, so the virtual runtime advantage gained by victim tasks is often lost if tasks are migrated to a different logical CPU.

Rewriting CFS so that runtimes of tasks from different logical CPUs are comparable and introducing fast load balancing based on these runtimes is likely possible without introducing much overhead. However, we deemed the task too complex for our limited resources, so we based our implementation on the simpler MuQSS scheduler [16] which is a scheduler for Linux based on virtual deadlines. MuQSS differs from CFS in that it performs load balancing as part of the main scheduler function so that logical CPUs very frequently try to fetch tasks from other more heavily loaded CPUs.

Our tests, however, showed that MuQSS is often less fair than CFS even when no AVX2/AVX-512 is involved. We therefore replaced the deadline-based scheduling policy of MuQSS with the CPU-time-based policy of CFS, resulting in a hybrid of CFS scheduling policy and MuQSS load balancing. We then extended the policy with frequency reduction compensation as described in the previous sections. Unlike CFS, we do not perform full renormalization of the virtual runtime of tasks during load balancing. Instead, when a CPU selects a task from a different CPU, we simply limit the virtual runtime advantage the incoming task can have compared to the CPU's current lowest virtual runtime to prevent starvation of other tasks.

7 Evaluation

We evaluate our prototype to show to which degree our design can improve fairness and to determine the limitations of the design. The evaluation is conducted on a system with an Intel Xeon Gold 6130 CPU, 24 GiB of 2666 MHz DDR4 RAM, the Fedora 31 operating system, and the Linux 5.9 kernel (with the CFS scheduler or with our modified scheduler based on MuQSS). As benchmarks for our evaluation, we use the nginx web server, most benchmarks from the Parsec 3.0 benchmark suite[2], as well as the Linux kernel build benchmark from the Phoronix Test Suite 9.0.1 (called "kernel-build" below). These benchmarks serve as potential victim tasks for frequency reduction compensation. In our experiments, the background application responsible for AVX frequency reduction is the x265 video encoder as it provides support for AVX-512 [29]. As described in Section 4.1, we did not implement our mechanism for CPU feature detection as part of our prototype as it would have made comparisons to an unmodified kernel much more difficult. Instead, we patch x265 to assume that AVX-512 is always available. All experiments are repeated ten times. MuQSS can be configured to share runqueues between multiple logical CPUs, and we selected runqueue sharing between hyper-thread siblings as we expected this setting to further help with the load-balancing issues described in the last section. As we show in Section 7.4, however, this setting does not seem to have much impact on the results.

7.1 Fairness

The main goal of our scheduler is to improve the fairness in a system where some tasks cause AVX frequency reduction and the performance of other tasks is affected. Fairness, in this case, means that equal CPU performance is available to the individual tasks. It is difficult to measure such fairness directly. Simply measuring the completion time of two applications when executing individually and then measuring the completion time of each application while both are executed at the same time does not yield the expected results. In particular, in a system with hyper-threading, two hyper-threads share CPU resources, and different applications may suffer differently from contention on these shared resources. An application with a large degree of instruction-level parallelism (ILP) may be able to utilize all available CPU resources when executed in isolation, but not when executed in parallel with a second application, whereas an application with little ILP may not be affected as much by other applications if the CPU resources are shared in a fair fashion. This and similar effects make it difficult to measure the unfairness caused by AVX frequency reduction.

We therefore choose a different, more indirect approach. Our experimental setup consists of two applications, a non-AVX foreground application of which we measure the completion time[3], and a background application (the x265 video encoder as described above) which can be configured to use

[2]We excluded raytrace as it failed to finish even on a system with CFS and x264 as it showed too much variation in all experiments to provide meaningful results.

[3]Note that to generate HTTP requests for nginx we use the wrk2 benchmark client which has a constant benchmark duration. For the nginx benchmark, we therefore substitute the completion time with the inverse of the web server throughput (i.e., the time required per HTTP request).

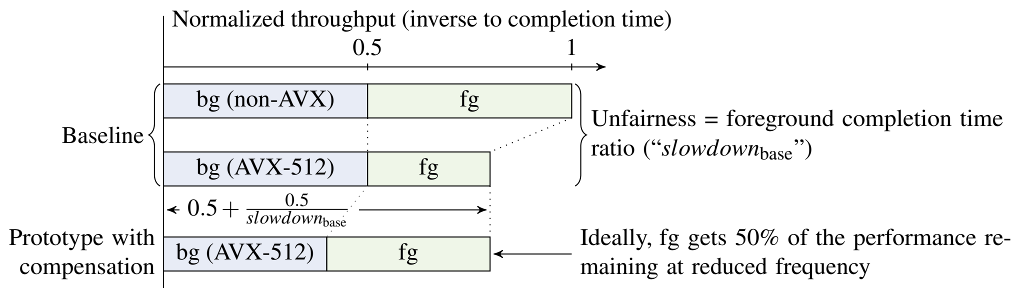

Normalized throughput (inverse to completion time)
0.5
1
Baseline
bg (non-AVX)
fg
bg (AVX-512)
fg
Unfairness = foreground completion time ratio (“$slowdown_{base}$”)
$0.5 + \frac{0.5}{slowdown_{base}}$
Prototype with compensation
bg (AVX-512)
fg
Ideally, fg gets 50% of the performance remaining at reduced frequency

Figure 3: To calculate the fairness achieved by our prototype, we have to take into account that less overall CPU performance is available due to AVX-induced frequency reduction. Each application is supposed to be allocated half of that reduced performance.

either AVX, AVX2, or AVX-512 instructions. We start four instances of x265 with 8 threads each as otherwise x265 would not be able to utilize all available logical CPUs. Note that the completion time of x265 does not change much for the different instruction sets and that AVX, AVX2, and AVX-512 instructions are executed on the same functional units of the CPU core. Therefore, we avoid the problems described above and completion time differences for the foreground application should be representative of the AVX frequency reduction.

In a first experiment, which we call the *baseline experiment*, we execute the applications using our modified scheduler but without all code for frequency reduction compensation. [4] We calculate the slowdown due to reduced frequencies as the completion time of the foreground application when the background application uses AVX2/AVX-512 divided by the completion time when the background application only uses AVX which does reduce the CPU frequency:

$$slowdown_{\text{base,AVX-512}} = \frac{t_{\text{base,AVX-512}}}{t_{\text{base,AVX}}} \tag{5}$$

$$slowdown_{\text{base,AVX2}} = \frac{t_{\text{base,AVX2}}}{t_{\text{base,AVX}}} \tag{6}$$

In this experiment, the slowdown provides a good metric for unfairness as no slowdown ($slowdown_{\text{base}} = 1$) means that the foreground application received the same share of CPU throughput irrespective of the choice of instructions:

$$unfairness_{\text{base}} = slowdown_{\text{base}} - 1 \tag{7}$$

We then repeat identical completion time measurements in a *prototype experiment* where we include frequency reduction compensation. As Figure 3 shows, in this case calculating the unfairness is slightly more complex. In a completely fair situation, both x265 and the foreground application would receive 50% of the CPU performance. However, overall CPU performance is reduced when x265 uses AVX2 or AVX-512, so 50% of this reduced performance are less than 50% of the CPU performance without any frequency reduction. Consequently, even with complete fairness, the foreground application still runs somewhat slower if x265 reduces the CPU frequency.

[4]We did not compare our prototype to CFS directly as we wanted to isolate the impact of frequency reduction compensation. Related work shows that CFS does not prevent AVX2/AVX-512 tasks from slowing other tasks down [10], either, which violates the fairness definition used by this paper.

As Figure 3 shows, a slowdown during the baseline experiment of $slowdown_{\text{base}}$ results in a remaining CPU performance of $perf_{\text{cpu}} = 0.5 + 0.5/slowdown_{\text{base}}$ of the original performance. We can use this information to calculate the unfairness in the prototype experiment using the slowdown in this experiment $slowdown_{\text{proto}}$ as well as $slowdown_{\text{base}}$. If we, as in the baseline experiment, define the unfairness as the ratio between the completion time of the background application and the foreground application and then substitute the time with the inverse of the throughput, we get the following formula:

$$unfairness_{\text{proto}} = \frac{tp_{\text{bg}}}{tp_{\text{fg}}} - 1 = \frac{perf_{\text{cpu}} - tp_{\text{fg}}}{tp_{\text{fg}}} - 1 \tag{8}$$

As shown in Figure 3 the throughput of the foreground application is $tp_{\text{fg}} = 0.5/slowdown_{\text{proto}}$. Inserting into equation (8) and simplifying yields the following formula which we use in the next sections to calculate the remaining unfairness in our prototype:

$$unfairness_{\text{proto}} = slowdown_{\text{proto}} + \frac{slowdown_{\text{proto}}}{slowdown_{\text{base}}} - 2 \tag{9}$$

7.1.1 Benchmark Results

Figure 4 shows the results of the experiments described above. It can immediately be seen that our prototype greatly reduces the unfairness between the two applications. Whereas the baseline system shows an average unfairness of 7.9% for AVX2 and 24.9% for AVX-512, these numbers are reduced to 2.5% and 5.4% by our prototype, respectively.

Some benchmarks show substantially worse results than others, though. In particular, the blackscholes and dedup benchmarks show a high degree of unfairness even with our prototype. An analysis of these benchmarks shows that the benchmarks are not able to scale to all logical CPUs of the system. The blackscholes benchmark, in particular, contains long

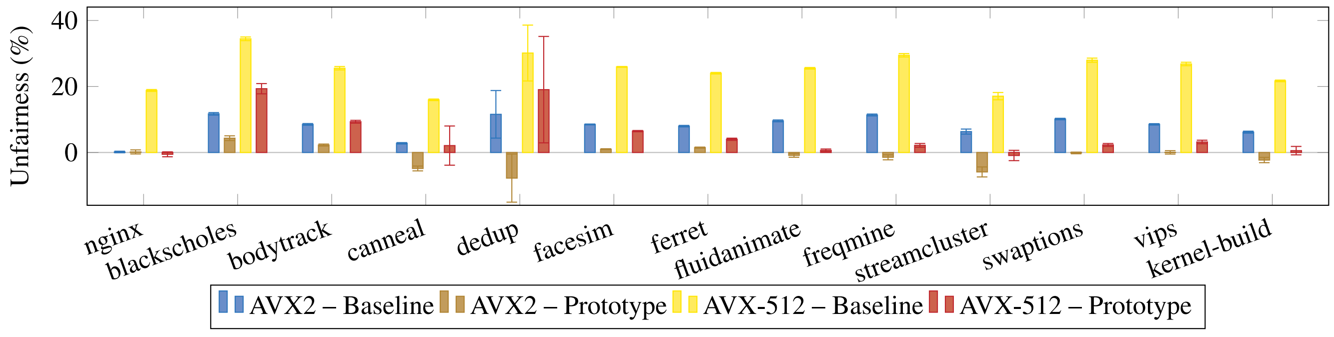


Figure 4: Unfairness for applications executed alongside x265 – for most workloads, our scheduler greatly reduces the unfairness between AVX2/AVX-512 applications and other applications executed in parallel. For AVX2, the unfairness was reduced from 7.9% (blue bars) to 2.5% (brown bars) on average, and for AVX-512 from 24.9% (yellow bars) to 5.4% (red bars).

serial phases where only one thread was active. In this case, the baseline setup already allocates almost one full logical CPU to the application. Therefore, our prototype is ineffective as it is not able to provide much more CPU time due to the lack of additional runnable threads.

This result also shows the main limitation of our approach. While overall our prototype is able to greatly increase fairness, the central mechanism – i.e., increasing the priority of victim tasks – is only effective if an increased priority translates into increased CPU time. This is not the case if an application already receives as much CPU time as it can utilize. Similarly, our approach also fails if victim application and AVX2/AVX-512 application are restricted to non-overlapping sets of logical CPUs, for example, via non-overlapping affinity masks, but effectively share physical cores via hyper-threading.

7.1.2 Synthetic Workload

Our methodology to measure the fairness for real-world applications relies on unusually complex indirect calculation of the target metric. Although the results of our experiments seem consistent, we therefore conduct an additional direct measurement of the fairness in a purely synthetic workload. For this experiment, our foreground application simply consists of 32 threads which together execute a fixed amount of 256-bit fused multiply-add (FMA) instructions requiring AVX2 frequencies, whereas the background application similarly executes a fixed amount of 512-bit FMA instructions at AVX-512 frequencies. As the two applications use very similar CPU resources, the resource contention effects described in Section 7.1 should affect both equally and should therefore not influence the observed fairness.

We try to choose the number of instructions executed by the two applications so that both require the same time if executed in isolation. We then perform a direct measurement of the fairness by comparing the completion time of the applications when executed in parallel at the same time. After one of the application finishes, we immediately start it a second time

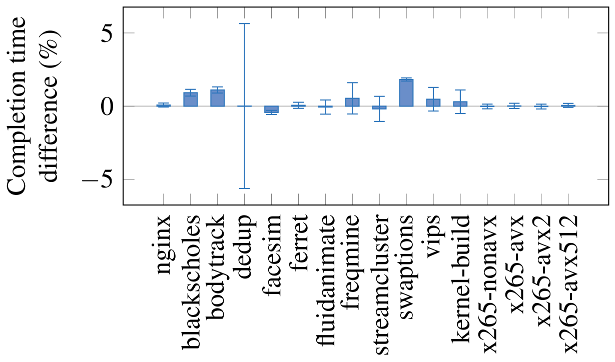


Figure 5: Our prototype causes almost no overhead when compared to a scheduler without frequency reduction compensation.

to prevent situations where only one application is running, as it then receives substantially more CPU resources. The increased throughput during that phase would mean that the measured completion time difference would not be representative for the unfairness during parallel execution.

In this setup, equal completion times are an indicator that both applications received equal shares of CPU performance. Our prototype provides almost perfect fairness. Whereas on a system without frequency reduction compensation the completion times differ by 19.2%, only a completion time difference of 0.5% remains in our prototype.

7.2 Overhead

While frequency reduction compensation can reduce the performance impact of AVX2/AVX-512 tasks on other tasks, the required modifications to the operating system can also cause overhead. In particular, the modified scheduler algorithm is executed many times per second on each core. The additional code not only increases the time spent in the scheduler itself

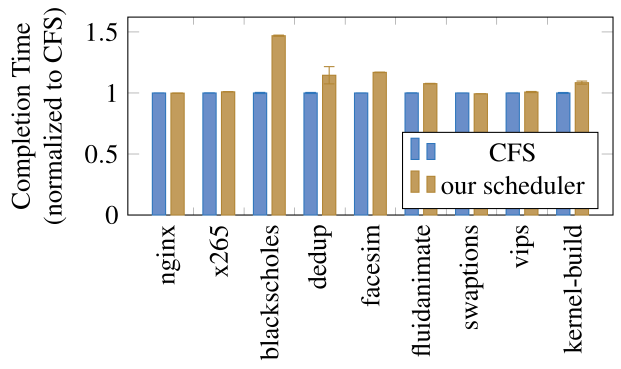


Figure 6: Our scheduler mostly provides competitive performance compared to CFS.

but also the cache footprint of the scheduler. Furthermore, the design triggers additional exceptions to detect 256-bit and 512-bit register accesses.

Most of the overhead, however, is only generated under specific circumstances. For example, the frequency impact of AVX2/AVX-512 is only calculated when the code detects a reduced frequency level at some point during the last time slice. The exceptions to detect AVX2 and AVX-512 tasks only affect such tasks and are triggered at most once per scheduler time slice.

To determine the overhead caused by our prototype, we measure the completion time of a range of benchmarks when executed in our prototype. We compare this time to the time when the benchmarks are executed with the same scheduler but without the code for frequency reduction compensation. The benchmarks include the nginx, Parsec, and PTS benchmarks used in Section 7.1 as well as the x265 video encoder as an example of an application using AVX2 or AVX-512 instructions. Due to the large variance of the dedup benchmark, we execute it 100 times to reduce the chance of misleading results.

Figure 5 shows the results of this experiment. The highest average overhead is measured for swaptions and amounted to 1.82%. All other benchmarks show less overhead, with an average overhead across all benchmarks of only 0.3%. We expect this overhead to be acceptable in most scenarios. In particular, the experiment shows that the additional exceptions to detect AVX2/AVX-512 code do not cause substantial overhead, as can be seen by the low overhead for the AVX benchmarks.

7.3 Comparison With CFS

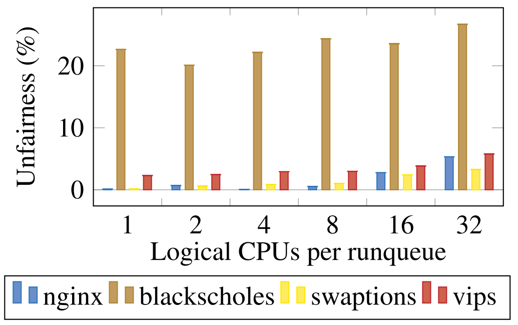
In all experiments described above, the baseline for our measurements is our own scheduler, with the code for frequency reduction compensation removed at compile time. Even though our scheduler implements the same basic algorithm as CFS, it provides a radically different implementation based on MuQSS. While MuQSS has been shown to provide similar performance as CFS [17], the implementation differences would have likely had an impact on the behavior of the benchmarks. Because we did not directly compare our MuQSS-based prototype to CFS in the experiments above, we show that the underlying scheduler – MuQSS modified to use the CFS policy, but without frequency reduction compensation – provides performance competitive to CFS. We execute a subset of the benchmarks with these two schedulers comparing the completion times.

Figure 6 shows the completion times of the benchmarks when executed with our scheduler normalized to the completion times with CFS. For most benchmarks, our scheduler causes at most 17% overhead when compared with CFS. This range matches the performance reported for unmodified MuQSS [17]. We used perf to measure the instructions per cycle (IPC) as an effort to determine the reason for the overhead. Overall, the IPC differences were much smaller than the overhead, with no clear correlation between the two, which shows that ineffective caching due to the fast load balancing required by our approach is not the main reason for the overhead. Implementing a suitable load balancing mechanism in other schedulers such as CFS is therefore unlikely to have substantial negative impact.

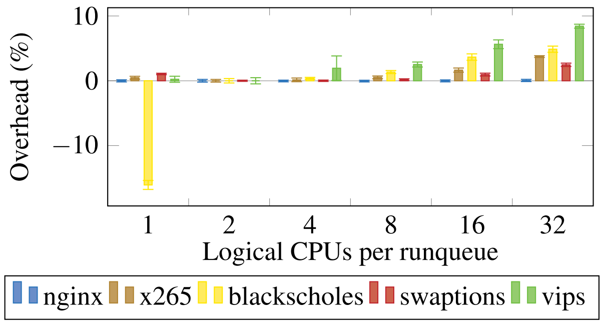
7.4 Runqueue Sharing

As described in Section 6, our implementation relies either on runqueues shared between multiple logical CPUs or on quick load balancing as otherwise the scheduler would often not have a choice between enough tasks to be able to effectively implement prioritization of victim tasks. The MuQSS scheduler used as the basis for our implementation provides both flexible runqueue sharing options as well as quick load balancing. As we want to determine whether shared runqueues were required for our approach to be effective, we measure the fairness for a subset of the benchmarks and with runqueues shared between different numbers of logical CPUs. The results of this experiment are visualized in Figure 7a, where we show the unfairness when AVX frequency compensation is enabled. Note that sharing runqueues between cores increases lock contention and potentially has a negative impact on throughput, so we also measure the overhead compared to a setup with runqueues shared between two logical CPUs. The results of this experiment are shown in Figure 7b.

Counterintuitively, sharing runqueues beyond sibling hyperthreads does not appear to have any beneficial impact on our prototype, which shows that the load balancing mechanism of our scheduler is able to provide all CPUs with enough choices so that they are able to prioritize victim tasks. The runqueue sharing options do result in slightly different throughput, however, with most benchmarks performing best if runqueues are shared between two logical CPUs. We assume blackscholes



(a) Remaining unfairness due to AVX-512 frequency reduction in our prototype (parallel execution with x265)



(b) Overhead for different runqueue sharing options compared to sharing among two logical CPUs (without frequency reduction compensation)

Figure 7: Our scheduler can share runqueues between a flexible number of CPU cores. While sharing runqueues between sibling hyper-threads – i.e., between two logical CPUs – provided the highest performance for most workloads, other configurations caused additional overhead and did not provide improved fairness.

benefits from improved cache efficiency when executed without shared runqueues. Note that despite the potential for such results the default option for MuQSS is to share one runqueue among all multicore siblings (i.e., 32 logical CPUs on our system).[5]

8 Fairness Criteria

Our evaluation shows that equal distribution of CPU time as implemented by contemporary schedulers fails to achieve its goal, which is to allocate equal CPU performance to individual tasks to improve performance isolation. We therefore propose AVX frequency compensation as a technique for fairer distribution of performance. While equal performance in terms of equal slowdown due to remote AVX overhead is an intuitive fairness definition, it is by far not the only one. We have identified two main dimensions spanning the design space of such definitions.

First, while current schedulers build upon CPU time as the main metric for fairness and we base our approach on a notion of CPU throughput, other metrics may be viable. One alternative would be to base scheduling on energy as a first-class operating system resource as suggested by related work [22, 26, 31]. Similar to our approach, such designs could penalize applications using power-intensive instructions such as AVX2 or AVX-512. While energy-based scheduling can therefore likely also solve the problem covered by this paper, it is not viable on current hardware. The CPU power model used by existing approaches either does not differentiate between executed instructions [22, 31] or uses the performance monitoring unit and requires a careful choice of performance events from which the energy can derived with sufficient accuracy [24]. Current CPUs, however, do not provide any sufficiently specific performance events for the relevant set of AVX2 and AVX-512 instructions [3, p. 19-3ff].[6] If future hardware provides a reliable method to estimate energy usage of individual logical CPUs, energy-based scheduling may prove to be a more effective solution than our approach.

Second, the effects of CPU power management can be compensated to different degrees, ranging from no compensation at all as implemented by most current operating systems to full compensation where only power-intensive tasks are affected by frequency reduction overhead. The latter provides a degree of performance isolation that is often desired in multi-tenant systems where customers pay for specific CPU performance. However, it also actively penalizes use of accelerators or similar specialized hardware, even though such code is often more energy-efficient due to the resulting speedup [6]. Our approach implements a compromise between performance isolation and incentives to use energy-efficient accelerators: Our definition of fair distribution of CPU performance means that the frequency reduction overhead is equally shared by low-power tasks as well as the high-power tasks which caused the overhead. In the future, we envision metadata such as the CPU quota information provided by Linux cgroups to be used to choose the appropriate type of frequency reduction compensation on a task-by-task basis.

As part of such a more flexible approach, our scheduler can easily be modified to support different degrees of performance isolation. As a possible variant, we changed our scheduler to not only reduce the CPU time credited to the victim task according to the frequency reduction but to also add an identical amount to the CPU time credited to the last AVX2 or AVX-512 task executed on the physical core. Whereas the original scheduler only reduced the performance impact on non-AVX tasks by a third during the benchmarks described

[5] https://github.com/ckolivas/linux/blob/5.9-muqss/kernel/Kconfig.MuQSS#L3

[6] While the CPUs provide performance events for floating point AVX2/AVX-512 instructions, other instructions are not covered.

above, this modification foregoes fairness and allocates more CPU time to victim processes, thereby achieving an average reduction by 70%, with many benchmarks experiencing a far lower performance impact.

9 Limitations

Our evaluation demonstrates that our scheduler modifications greatly improve fairness for workloads consisting of AVX2/AVX-512 tasks and non-AVX tasks. For many of the scenarios we test, our prototype achieves 10 times better fairness than a scheduler without frequency reduction compensation. This improvement comes at near-zero cost: As we show, frequency reduction compensation has almost no overhead, and it only contributes 329 additional lines of code to our prototype. While our design therefore provides a practical solution as-is, with substantial improvements over existing schedulers, it has a number of limitations, some of which are caused by limitations of the underlying hardware. We discuss these limitations below and sketch improvements of the design where applicable.

9.1 Detection of AVX2 and AVX-512

As our design tries to increase the CPU time share of tasks which suffer from AVX frequency reduction even though they do not have to be executed at reduced frequencies, correct attribution of reduced frequencies to AVX2/AVX-512 tasks is a central part of our design. We use and extend an existing approach [10] where the CPU is configured to trigger an exception when 256-bit and 512-bit registers are accessed. This simple policy does not match the actual conditions for reduced frequencies which are substantially more complex [5, 18]. Often, usage of 256-bit and 512-bit registers is detected even though a task does not actually require the respective AVX2 and AVX-512 frequency levels. Yet, our prototype assumes that a lower frequency level is required whenever a task accesses the corresponding registers.

If, for example, a task executes light AVX-512 instructions, our prototype never applies any compensation even if the task could be executed at AVX2 frequencies as shown in Section 2. Similarly, for tasks with light AVX2 instructions which are able to execute at non-AVX frequencies, our prototype only compensates the difference between AVX2 and AVX-512 frequencies. To the best of our knowledge, there is no better hardware interface available to detect power-intensive AVX2/AVX-512 code sections. In our case, however, it is important to note that the mechanism does not cause any false negatives and therefore results in a conservative implementation where applications are never unfairly rewarded for a frequency reduction they caused themselves.

In the future, an improved software mechanism to detect AVX2/AVX-512 code may improve accuracy. For example, the operating system could scan executable pages for 256-bit and 512-bit operations and categorize them. Then, instead of the current register-based mechanism, the OS could selectively unmap all pages containing instructions which may cause a transition to AVX2/AVX-512 frequencies to detect AVX2/AVX-512 tasks via page faults.

9.2 Memory-Bound Processes

As described in Section 5, our prototype currently assumes that performance is proportional to CPU frequency when deriving the performance impact from the calculated frequency reduction. This is an assumption that is found in related work as well [12]. The assumption is incorrect for memory-bound tasks, however. Memory latency does not increase when the CPU frequency is reduced, so memory-heavy applications suffer less from reduced frequencies [13]. Unequal performance impact due to different sensitivity to frequency changes has been identified as a challenge for fair scheduling [14].

In our case, the result is that memory-bound tasks may be given more than their fair share of the system's performance if frequency reduction compensation is applied. Consequently, the negative impact on the AVX2/AVX-512 tasks causing the frequency reduction is increased, as they in turn are allocated less than their fair share of CPU performance. This negative impact, however, is bounded: The AVX2/AVX-512 tasks always receive at least as much CPU performance as they would if all other tasks were completely CPU-bound.

In the future, our design could be extended with a more accurate performance model which takes memory accesses into account, similar to DTS [14] which has shown that better DVFS performance models can greatly increase fairness. Many such models have been described in the literature [21]. It has been shown that often very few performance counters suffice to characterize the workload and to predict performance at other CPU frequencies [25, 27]. As described in Section 5, our current technique to determine the performance impact requires at least two of the four configurable performance counters of each logical CPU (one for AVX-512 and one for AVX2 frequency level cycles) as well as a fixed-function CPU cycle counter.[7] The two remaining configurable counters can be used to determine how memory-heavy the workload is and to implement a better performance model.

10 Related Work

To the best of our knowledge, our approach is the first fair scheduler for workloads involving AVX2 and AVX-512 tasks. However, there have been scheduling algorithms for software-controlled DVFS and there have been other techniques to mitigate the performance impact caused by AVX frequency

[7] Currently, the prototype uses three configurable performance counters. The third counter is used to determine the total number of elapsed CPU cycles and can be replaced with a fixed-function counter.

reduction. In the following section, we describe the corresponding related work and lay out the differences to our approach.

10.1 Fair Scheduling and DVFS

Existing work on fair schedulers for systems with DVFS has focused on software-controlled DVFS where the operating system chooses the CPU frequency. Two main issues affecting fairness arise from such DVFS:

1. CPU quotas are commonly expressed in terms of CPU time, and a fixed CPU time quota translates into less performance when the system is running at a reduced frequency. Hagimont et al. [12] discuss the interaction between load-based CPU frequency selection and fixed and flexible CPU time quotas. Whereas fixed CPU time quotas result in reduced fairness and insufficient CPU throughput, flexible CPU time quotas commonly prevent any frequency reduction.

   Hypervisors often implement CPU time quotas in the form of credit schedulers. To solve the problems stemming from DVFS, the time per credit can be scaled by the CPU frequency and frequency selection can be modified to take flexible CPU quotas into account [12]. In this work, we use a similar approach where we implement scaling within CPU time accounting. Our work differs, though, in that we focus on hardware-controlled DVFS and on unfairness problems where one task reduces the CPU frequency experienced by another task. This scenario requires different techniques to identify the tasks affected by unfair scheduling and to determine the scale of the resulting performance impact.

   In Section 8, we discussed that CPU time is not the only possible metric for scheduling. Credit schedulers in particular have been modified to scale the time per credit based on power consumption [30]. While such policies could improve fairness in workloads with AVX2 and AVX-512, the required fine-grained accurate power models cannot be constructed on current hardware for the reasons outlined above.

2. Programs suffer from the same frequency reduction to different degrees depending on how memory-heavy the programs are. Jia et al. [14] show how this affects fairness when the operating system reduces the CPU frequency. They suggest dynamic time-slice scaling (DTS) to achieve level performance, where the operating system determines the performance impact of DVFS on applications and scales time-slices accordingly. Our work solves a different problem – unfairness caused by hardware-controlled DVFS – yet uses a very similar mechanism. Instead of time-slice scaling, we modify CPU time accounting as it requires fewer modifications to the CFS scheduling algorithm. In general, we argue, though, that the approach of DTS is orthogonal to our design. As we describe in Section 9.2, we currently assume that all tasks are affected equally by frequency reduction, so estimates of the actual performance impact as conducted by DTS can further improve the fairness of our approach.

10.2 Core Specialization

In this work, we describe a scheduler which compensates AVX frequency reduction to improve fairness. An alternative approach would be to prevent AVX frequency reduction whenever possible, as less AVX frequency reduction might also lead to improved fairness. One such technique to prevent AVX frequency reduction is to use core scheduling to limit co-scheduling of AVX-512 and non-AVX-512 tasks [19]. Similarly, core specialization [10] restricts scheduling of AVX-512 tasks to a subset of the system's CPU cores. While, in contrast to our approach, both techniques can potentially improve overall system throughput if they achieve higher average CPU frequencies, they have the potential to cause substantial overhead. Core scheduling leaves hyper-threads idle if necessary [19], which may cause reduced utilization of CPU resources. Similarly, an earlier version of core specialization was shown to cause substantial overhead when tasks frequently had to be migrated between cores [9]. We present an approach that is far less intrusive and has less potential to cause overhead. We therefore believe that our approach is applicable to a wider range of situations.

11 Conclusion

When power-intensive instructions are executed, power-limited CPUs may have to temporarily reduce their frequency to prevent excessive power consumption. Often, this frequency reduction also affects tasks that do not execute such power-intensive instructions. Therefore, as we demonstrate, these systems require new approaches to fair scheduling.

Recent Intel CPUs with support for AVX2 and AVX-512 show such behavior, and in this work we describe a technique to identify victim tasks whose performance is affected by other AVX2/AVX-512 tasks. We describe a modification to CPU time accounting where the CPU time credited to these victim tasks is scaled according to the frequency reduction experienced by the tasks. The change causes fair schedulers based on CPU time to prioritize the tasks to the degree where they receive their fair share of CPU performance again. Our prototype is able to reduce the unfairness from 7.9% to 2.5% on average for workloads involving AVX2 applications and from 24.9% to 5.4% for AVX-512.

References

[1] Intel® Turbo Boost Technology in Intel® Core™ Microarchitecture (Nehalem) Based Processors. Technical report, 11 2008.

[2] *Intel® 64 and IA-32 Architectures Software Developer's Manual - Volume 1: Basic Architecture*, May 2018.

[3] *Intel® 64 and IA-32 Architectures Software Developer's Manual - Volume 3 (3A, 3B, 3C & 3D): System Programming Guide*, May 2018.

[4] *Intel® Xeon® Processor Scalable Family – Specification Update*. Intel Corporation, February 2018.

[5] *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, September 2019.

[6] Juan M. Cebrian, Lasse Natvig, and Magnus Jahre. Scalability analysis of avx-512 extensions. *The Journal of Supercomputing*, pages 1–16, 2019.

[7] Edward G. Coffman, Jr. and Leonard Kleinrock. Computer scheduling methods and their countermeasures. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference (AFIPS '68 (Spring))*, pages 11–21, 1968.

[8] Travis Downs. Gathering intel on Intel AVX-512 transitions, January 17 2020. https://travisdowns.github.io/blog/2020/01/17/avxfreq1.html.

[9] Mathias Gottschlag and Frank Bellosa. Reducing AVX-induced frequency variation with core specialization. In *The 9th Workshop on Systems for Multi-core and Heterogeneous Architectures*, 2019.

[10] Mathias Gottschlag, Peter Brantsch, and Frank Bellosa. Automatic core specialization for AVX-512 applications. In *Proceedings of the 13th ACM International Systems and Storage Conference*, pages 25–35, 2020.

[11] Mathias Gottschlag, Tim Schmidt, and Frank Bellosa. AVX overhead profiling: How much does your fast code slow you down? In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 59–66, 2020.

[12] Daniel Hagimont, Christine Mayap Kamga, Laurent Broto, Alain Tchana, and Noel De Palma. DVFS aware CPU credit enforcement in a virtualized system. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 123–142. Springer, 2013.

[13] Ranjan Hebbar S R and Aleksandar Milenković. Impact of thread and frequency scaling on performance and energy efficiency: An evaluation of Core i7-8700K using SPEC CPU2017. In *2019 SoutheastCon*, pages 1–7. IEEE, 2019.

[14] Gangyong Jia, Xuhong Gao, Xi Li, Chao Wang, and Xuehai Zhou. DTS: Using dynamic time-slice scaling to address the OS problem incurred by DVFS. In *2012 IEEE International Conference on Cluster Computing Workshops*, pages 65–72. IEEE, 2012.

[15] Judy Kay and Piers Lauder. A fair share scheduler. *Communications of the ACM*, 31(1):44–55, 1988.

[16] Con Kolivas. MuQSS - the multiple queue skiplist scheduler. http://ck.kolivas.org/patches/muqss/sched-MuQSS.txt.

[17] Con Kolivas. First MuQSS throughput benchmarks, October 18 2016. http://ck-hack.blogspot.com/2016/10/first-muqss-throughput-benchmarks.html.

[18] Daniel Lemire. AVX-512 throttling: heavy instructions are maybe not so dangerous, August 25, 2018. https://lemire.me/blog/2018/08/25/avx-512-throttling-heavy-instructions-are-maybe-not-so-dangerous/.

[19] Aubrey Li. Core scheduling: prevent fast instructions from slowing you down. Linux Plumbers Conference, September 9 2019.

[20] Ingo Molnar. Modular scheduler core and completely fair scheduler [CFS]. *Linux-Kernel mailing list*, 2007.

[21] Barry Rountree, David K. Lowenthal, Martin Schulz, and Bronis R. de Supinski. Practical performance prediction under dynamic voltage frequency scaling. In *2011 International Green Computing Conference and Workshops*, pages 1–8. IEEE, 2011.

[22] Arjun Roy, Stephen M. Rumble, Ryan Stutsman, Philip Levis, David Mazieres, and Nickolai Zeldovich. Energy management in mobile devices with the Cinder operating system. In *Proceedings of the sixth conference on Computer systems*, pages 139–152, 2011.

[23] Robert Schöne, Thomas Ilsche, Mario Bielert, Andreas Gocht, and Daniel Hackenberg. Energy efficiency features of the Intel Skylake-SP processor and their impact on performance. In *2019 International Conference on High Performance Computing & Simulation (HPCS)*, pages 399–406. IEEE, 2019.

[24] Arsalan Shahid, Muhammad Fahad, Ravi Reddy Manumachu, and Alexey Lastovetsky. A comparative study of

techniques for energy predictive modeling using performance monitoring counters on modern multicore CPUs. *IEEE Access*, 8:143306–143332, 2020.

[25] Vasileios Spiliopoulos, Stefanos Kaxiras, and Georgios Keramidas. Green governors: A framework for continuously adaptive dvfs. In *2011 International Green Computing Conference and Workshops*, pages 1–8. IEEE, 2011.

[26] Jan Stoess, Christian Lang, and Frank Bellosa. Energy management for hypervisor-based virtual machines. In *2007 USENIX Annual Technical Conference (USENIX ATC'07)*, pages 1–14, 2007.

[27] Bo Su, Joseph L Greathouse, Junli Gu, Michael Boyer, Li Shen, and Zhiying Wang. Implementing a leading loads performance predictor on commodity processors. In *2014 USENIX Annual Technical Conference (USENIX ATC'14)*, 2014.

[28] Michael B. Taylor. Is dark silicon useful? harnessing the four horsemen of the coming dark silicon apocalypse. In *49th ACM/EDAC/IEEE Design Automation Conference*, pages 1131–1136. IEEE, 2012.

[29] Praveen Kumar Tiwari, Vignesh V Menon, Jayashri Murugan, Jayashree Chandrasekaran, Gopi Satykrishna Akisetty, Pradeep Ramachandran, Sravanthi Kota Venkata, Christopher A Bird, and Kevin Cone. Accelerating x265 with Intel® Advanced Vector Extensions 512. Technical report, Intel, 05 2018.

[30] Chengjian Wen, Jun He, Jiong Zhang, and Xiang Long. Pcfs: A power credit based fair scheduler under DVFS for multi-core virtualization platform. In *2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, pages 163–170. IEEE, 2010.

[31] Heng Zeng, Carla S. Ellis, Alvin R. Lebeck, and Amin Vahdat. ECOSystem: Managing energy as a first class operating system resource. *ACM SIGOPS operating systems review*, 36(5):123–132, 2002.