

# Technical Report: Dim Silicon and the Case for Improved DVFS Policies

Mathias Gottschlag, Yussuf Khalil, Frank Bellosa

Operating Systems Group  
Karlsruhe Institute of Technology  
E-mail: `os@itec.kit.edu`

## Abstract

*Due to thermal and power supply limits, modern Intel CPUs reduce their frequency when AVX2 and AVX-512 instructions are executed. As the CPUs wait for 670  $\mu$ s before increasing the frequency again, the performance of some heterogeneous workloads is reduced. In this paper, we describe parallels between this situation and dynamic power management as well as between the policy implemented by these CPUs and fixed-timeout device shutdown policies. We show that the policy implemented by Intel CPUs is not optimal and describe potential better policies. In particular, we present a mechanism to classify applications based on their likeliness to cause frequency reduction. Our approach takes either the resulting classification information or information provided by the application and generates hints for the DVFS policy. We show that faster frequency changes based on these hints are able to improve performance for a web server using the OpenSSL library.*

## 1 Introduction

In recent years, performance became increasingly limited by power consumption as Dennard scaling has come to an end [33]. The effect where the available power budget allows for different maximum frequencies depending on the number of cores is called dim silicon [17]. The same effect also applies to different instruction mixes. As different operations cause different switching activity on the chip, they consume different amounts of energy, so complex instructions have to be executed at a lower frequency. Similarly, if unused parts of the chip are power-gated because they are not required by simpler operations, the resulting power savings can be used to increase the frequency.

The power budget is not only limited due to thermal constraints but also due to power supply limitations<sup>1</sup>, where even short-term transgressions could cause instability due to voltage drops.

As the large size of the SIMD registers used by recent SIMD instruction set extensions causes high power variation, recent CPUs have started to vary their frequency based on the workload to maximize performance under power budget constraints. For example, Intel CPUs reduce their clock speed as soon as code containing AVX2 and AVX-512 instructions is executed [5]. However, every frequency change causes some overhead [26], because the system has to wait for voltages to change<sup>2</sup> and clock signals to stabilize. Therefore, even if no AVX2 and AVX-512 instructions are executed anymore, these CPUs delay increasing the clock speed [6]. This mechanism ensures that if the code continues executing these vectorized instructions shortly after, no excessive numbers of frequency changes are performed.

For some workloads, the delay causes overhead, though, as parts of the software which could be executed at higher frequency are needlessly slowed down. For example, a simple benchmark using the nginx web server is slowed down by 10% if the SSL library used by the web server is compiled with support for AVX-512, as the CPU frequency is reduced during AVX-512-heavy encryption and decryption, but the frequency change also affects the non-vectorized parts of the web server [20].

A policy similar to this constant-delay policy is employed in the area of dynamic power management. In this area, a similar trade-off is found, as disabling

<sup>1</sup>In our tests, recent Intel CPUs have reported maximum current as the most common reason for frequency changes in AVX-512-heavy workloads.

<sup>2</sup>The frequency can only be increased when sufficient voltage is available, leading to frequency change delays and a resulting “underclocking loss” [26].

devices saves energy but incurs overhead both during shutdown and reactivation. The widely-used *fixed timeout* policy shuts down devices after a fixed delay [7], where the delay is usually equal to the *break-even time* in order to improve worst-case power consumption [18]. In the area of power management, research has brought up a plethora of other shutdown strategies promising higher energy savings [7] and has shown that input from the application can be used to further improve power efficiency [34]. It is likely that similar approaches can be used to reduce DVFS overhead for partially power-intensive workloads. In this work, we show that, in particular, input from the application can be used to predict whether immediate relocking makes sense. Our contributions are as follows:

- We describe the parallels between DVFS in dim silicon scenarios and dynamic power management. The duality allows to apply research from the area of dynamic power management to the former.
- We determine the frequency change cost on a current server system and calculate the break-even time for frequency changes. We use this result to show how the delay specified by Intel does not provide optimal worst-case behavior.
- We show that application knowledge about execution phases or the instruction types used by individual processes can be used to improve performance by passing hints about future instruction set usage to the DVFS policy. We validate this finding through simulation of different DVFS policies on a web server workload.
- We describe a mechanism to determine at runtime whether individual processes will trigger frequency reductions due to their usage of power-intensive instructions. Unlike existing approaches, our design can reliably distinguish between all three frequency levels provided by current Intel CPUs. This information can be used as input for an improved DVFS policy to trigger frequency changes during context switches.

## 2 Effects of AVX2 and AVX-512

Starting with the Haswell microarchitecture which introduced the AVX2 instruction set, Intel introduced a separate maximum frequency for AVX2-intensive code segments [15]. The Skylake microarchitecture added AVX-512 instructions and a third AVX-512 frequency level [29]. Table 1 shows the maximum turbo frequency

for the Intel Xeon Gold 6130 server processor. The maximum frequency depends both on the number of active cores – with larger numbers of active cores requiring larger frequency reduction – as well as on the type of instructions executed. AVX-512 causes a particularly large frequency reduction due to the complexity of operations on 512-bit vectors. As described above, the reduced frequency is maintained longer than necessary to prevent excessive relocking overhead.

There are two situations where this delay can cause the frequency reduction to negatively affect unrelated non-AVX code and cause a significant performance reduction. First, on a system with simultaneous multi-threading (SMT), if one of the hardware threads causes the frequency of the physical core to be reduced, the other hardware threads on the same core also execute at lower frequency even if their code is not as energy-intensive [22]. Second, in heterogeneous applications consisting of power-intensive and less power-intensive parts – or if the OS frequently switches between power-intensive and less power-intensive tasks – the delay before increasing the frequency causes reduced performance for the less power-intensive code [13].

As an example for the latter, previous work describes overhead caused by AVX-512 in a web server workload, where the nginx web server provides up to 10% lower performance when the SSL library uses cryptography primitives implemented with AVX-512 instructions, because unrelated web server code is slowed down following calls into the SSL library [20]. We replicated this experiment, the result is shown in Figure 1 alongside other experiments with workloads consisting of multiple different processes to show that the performance impact is also present in such scenarios. For these other experiments, we execute different non-AVX workloads while concurrently executing the x265 video encoder configured to use AVX, AVX2, or AVX-512 instructions. The experiments are conducted on a system with an Intel Xeon Gold 6130 processor.

Our first multi-process experiment determines the impact on an interactive web server workload: We executed the nginx web server alongside the x265 video encoder and configured the wrk2 client to generate a fixed number of requests to the web server. This setup imitates the scenario where a web server is not fully utilized and the remaining CPU time is used for background batch tasks. Figure 1 shows the normalized CPU time required by the nginx web server to serve a unencrypted static file (“nginx+x265”). The results show a 6.6% performance impact when the background process uses AVX2 instructions and a 21.8% performance impact for AVX-512. As the web server is not operating at 100% utilization, the background process

Active cores	1-2 cores	3-4 cores	5-8 cores	9-12 cores	13-16 cores
Normal	3.7 GHz	3.5 GHz	3.4 GHz	3.1 GHz	2.8 GHz
AVX2	3.6 GHz	3.4 GHz	3.1 GHz	2.6 GHz	2.4 GHz
AVX-512	3.5 GHz	3.1 GHz	2.4 GHz	2.1 GHz	1.9 GHz

Table 1: Maximum turbo frequency of the Intel Xeon Gold 6130 processor [3]. The frequency reduction caused by AVX2 and AVX-512 instructions increases when more cores are active.

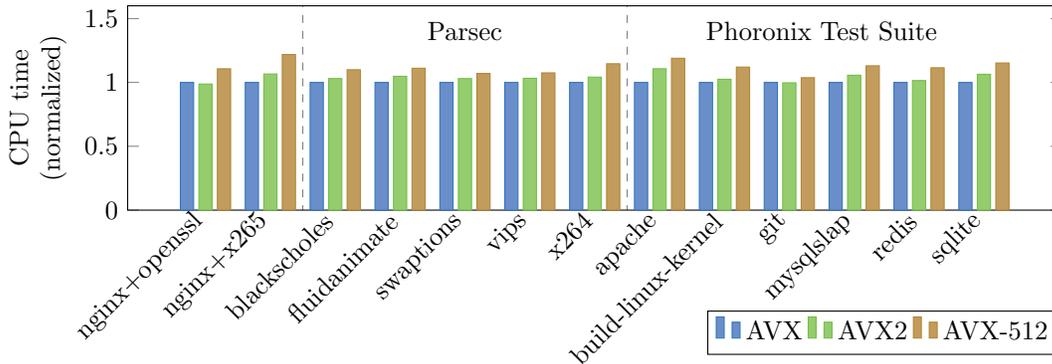


Figure 1: CPU time required to run various benchmarks under the influence of different instruction sets to measure the impact of AVX frequency reduction.

is often executed inbetween two consecutive requests or is executed in parallel on the other hardware thread of the same core, causing a particularly large performance impact.

To show that the problem affects both interactive and batch workloads, we also execute various benchmarks from the Parsec [8] benchmark suite and the Phoronix Test Suite (PTS) [1] benchmarks in parallel to the x265 video encoder. As shown in Figure 1, all these benchmarks are also affected by the frequency changes caused by x265. The Parsec benchmarks experience an average performance reduction by 10.0% for AVX-512. Similarly, the PTS benchmarks are slowed down by 12.4%.

As described above, one major mechanism for slowdown that is targeted by other approaches [22] is that software on one hardware thread slows down other hardware threads of the same core. To show that some of the slowdown is also experienced on systems without hyperthreading, we repeat all the benchmarks on a system with hyperthreading disabled. The results of this experiment are shown in Figure 2 and show that CPU-intensive non-interactive workloads are not significantly slowed down anymore once hyperthreading is disabled as the system does not switch between the processes often enough for frequency change delays to have a significant effect. For example, on a system

with the default Linux CFS scheduler, we observe only one context switch every 10 to 20 ms for the blackscholes workload whereas frequency increases are only delayed by less than one millisecond. Although disabling hyperthreading reduces the performance of the system and is therefore not a viable technique against the overhead caused by AVX-heavy code in these scenarios, other techniques such as core specialization [13] and core scheduling [22] can make sure that whenever possible either both hyperthreads are executing AVX-intensive code or none of them is.

Overhead caused by hyperthreading is out of the scope of this paper, though. Instead, the goal of our approach is to reduce the overhead in applications which periodically execute short sections of AVX-512 or AVX2 code as well as in workloads which frequently switch between AVX-512 or AVX2 and non-AVX applications on a single core. From the benchmarks shown in Figure 2, an example for the former is the nginx/OpenSSL benchmark, which executes AVX-512 instructions only when OpenSSL functions are called. The nginx/x265 benchmark as well as the Apache, MySQL and SQLite benchmarks from PTS, instead, trigger frequent context switches between the AVX-512-enabled background task and the benchmarked application and are therefore examples for the latter behaviour. These types of benchmarks are the bench-

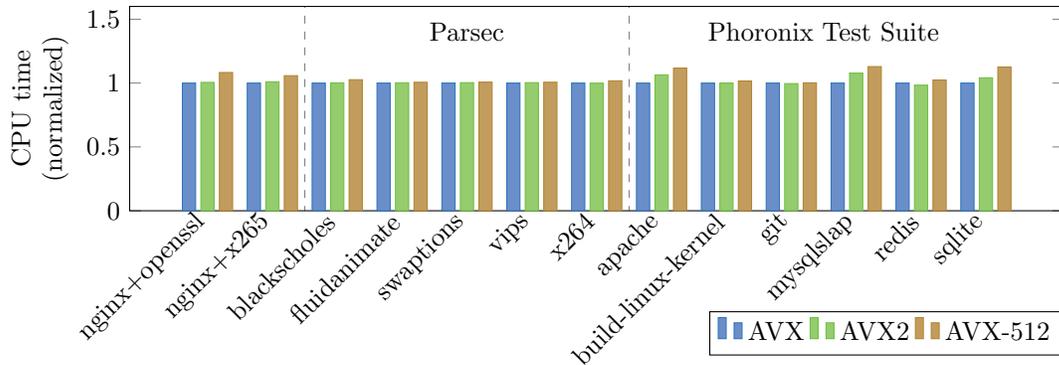


Figure 2: CPU time required for the experiment shown in Figure 1 on a system with hyperthreading disabled. Note that benchmarks with few context switches do not suffer from frequency changes anymore once hyperthreading is disabled, whereas benchmarks with frequent switches between AVX and non-AVX code (i.e., heterogeneous programs with short AVX-heavy phases as well as workloads consisting of an interactive service and an AVX-heavy background task) still suffer from the frequency change delay. Note that such workloads are often latency-critical and therefore particularly suffer from degraded performance.

marks which show overhead even when hyperthreading is disabled: For AVX-512, the nginx benchmarks are slowed down by 7.0% on average, whereas the three PTS benchmarks are slowed down by 12.4% on average.

Due to the frequent switches between AVX-512/AVX2 and non-AVX code during these workloads, the upclocking delay implemented by the CPU’s existing hardware DVFS policy is the main source for the overhead caused by AVX instructions. To isolate this overhead source and to demonstrate that improved DVFS policies are able to mitigate its effects, we conduct all further experiments in this paper with hyperthreading disabled. The assumption of CPUs without hyperthreading significantly simplifies the design of some parts of our approach. This does not mean that improved DVFS policies are inherently ineffective on systems with hyperthreading, although more research has to be conducted to identify appropriate heuristics for improved DVFS decisions.

### 3 Parallels to Dynamic Power Management

As described above, the complex frequency behavior of modern CPUs stems from the fact that it is not economically viable to cool modern CPUs when they are executing power-intensive code at their maximum frequency [17]. Instead, available thermal headroom is used to temporarily use higher frequencies (a form of computational sprinting [27]). In this scenario, the more the energy consumption per instruction varies,

the higher is the thermal headroom for code executing simple instructions. Therefore, modern Intel CPUs use different turbo frequencies for different types of code, with AVX2 and AVX-512 instructions triggering a transition to significantly lower frequency levels [29]. As shown by the registers provided by these CPUs to determine the reason for frequency changes, not only thermal headroom plays a factor for these frequency reductions, though: The power dissipation of the chip correlates with the current required from the power supply, and frequency changes are also required to prevent voltage drops due to increased current draw.

The frequency changes required to use the available headroom come at a cost. For example, Mazouz et al. have measured the cost of a single frequency change to be approximately 10  $\mu$ s on an Intel Ivy Bridge system [24] and our own experiments presented in Section 4.1 arrive at a similar cost (between 9  $\mu$ s and 19  $\mu$ s) on more recent Skylake server CPUs. Therefore, increasing the frequency to use thermal headroom is only viable if the higher frequency can be applied long enough that the performance improvement makes up for the frequency change overhead. This trade-off is similar to the problem of *dynamic power management* where devices are temporarily switched off or transitioned to a low-power state in order to save energy [7]. Here, the energy cost for the state transition means that switching devices off for only short periods of time is frequently unviable. As the operating system, however, does not know how long a device is going to stay unused, it is in general not possible to determine in advance whether shutting a device off is going to result

in a net improvement.

In the area of dynamic power management, significant effort has gone into developing heuristic approaches to guess when to shutdown devices [7]. One metric to measure the quality of heuristic approaches is their *competitiveness* in a worst-case scenario. The competitiveness is the worst-case ratio between the energy required by the approach compared to the energy required by an oracle policy that can determine in advance whether shutting off a device is viable. Karlin et al. [18] showed at most 2-competitiveness (meaning that the approach uses at most twice as much energy) is possible for deterministic algorithms. In dynamic power management, 2-competitiveness can be achieved by switching a device off after a fixed timeout. When that timeout equals the *break-even time* (i.e., the time of inactivity during with the low-power state would have made up for the transition costs), the device uses at most twice as much energy if it wakes up directly after being sent to a low-power state. Intel CPUs show a very similar behavior as they delay increasing the frequency by a fixed timeout after the CPU has stopped executing any AVX instructions [29]. However, the fixed delay is not optimal in terms of competitiveness because, as we show in Section 4, DVFS has wildly varying break-even times in different scenarios. Neither is the DVFS policy implemented by current Intel CPUs optimal for real-world workloads as we show in Section 6.2.

There are approaches that can, depending on the situation, perform better than simple heuristic approaches. For example, applications can give hints about expected future behavior to let the OS perform better informed decisions [23] or the OS can use the deadlines of I/O requests to change the device usage pattern to save more energy [35] Both these approaches can be applied to DVFS policies in dim silicon scenarios. In this paper, we show an example for the former approach. As software developers often know whether the application is going to execute no power-intensive code – i.e., no AVX2 and AVX-512 – in the near future, that information can be used by the CPU to forego the frequency change delay and immediately change frequencies for improved performance.

## 4 Behavior of Intel CPUs

According to the optimization manual, recent Intel CPUs implement a fixed-timeout policy where the CPU waits approximately 2 ms after the last section of AVX-intensive code before increasing the frequency again [6, p. 2-13]. In addition, before lowering the frequency, the core requests a power license from the package control

unit (PCU) which takes up to 500  $\mu$ s before granting the license. However, as shown by Schne et al. [29], the behavior of the hardware does not match the documentation. Instead, the processor waits for a significantly shorter timeout (approx. 670  $\mu$ s as measured in our experiments) before upclocking. We were able to confirm the observed behavior on a system with an Intel Core i9-7940X, where we measured the delay for frequency changes when executing sections of code consisting of scalar, AVX2, or AVX512 instructions. Note that frequency reduction is triggered almost immediately when AVX2 or AVX-512 instructions are executed, as required to prevent excessive power consumption.

The upclocking delay is constant independent from the number of cores in use. As described in the last section, maximum competitiveness in worst-case scenarios is reached when the timeout equals the break-even time, but the break-even time depends not only on the cost for the frequency transition but also on the performance advantage at a higher frequency. In this case, the frequency change is higher if more cores are active [5], so the performance overhead for downclocking is higher and the break-even time is shorter when more cores are active. Therefore, the policy implemented by Intel does not provide maximum competitiveness. To show the potential for improved timeout-based policies, the following sections describe experiments to determine both frequency transition overhead as well as performance impact for different situations to determine the corresponding break-even times.

### 4.1 Cost of Frequency Changes

One factor required to determine the break-even time is the frequency change overhead: If the cost of individual frequency changes increases, more time between consecutive changes is required in order to make up for the overhead. For the Intel Ivy Bridge architecture, Mazouz et al. determined that a CPU is stopped for approximately 10  $\mu$ s during a frequency change [24]. This pause is required to allow the new frequency to stabilize [26]. However, in particular in the case of frequency changes caused by AVX instructions, additional factors increase the overall overhead. Therefore, and because our systems use a newer CPU architecture than the one considered by Mazouz et al., we measure the overhead of frequency changes on a system with an Intel Xeon Gold 6130 CPU.

To measure the overhead due to frequency reduction caused by AVX2 and AVX-512 instructions, we execute the same amount of such instructions twice, once when the system is already at the appropriate frequency, and once when it executes at a higher frequency and the

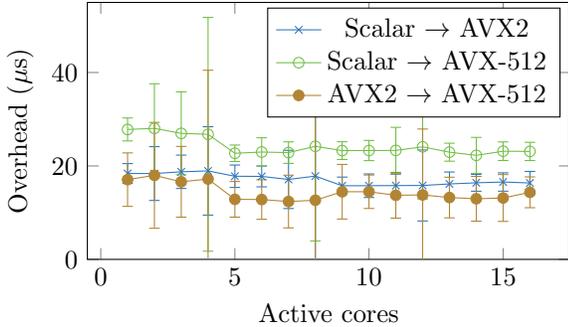


Figure 3: Overhead when the frequency is reduced, measured as the mean of 1000 runs. The error bars indicate the standard deviation. The overhead seems to vary slightly based on the number of active cores and on the resulting frequencies. Note that a transition from scalar to AVX-512 frequencies incurs two separate frequency transitions.

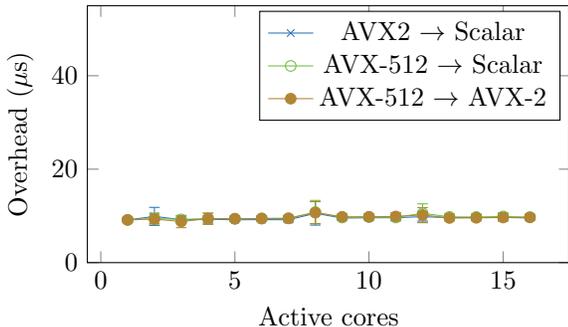


Figure 4: Overhead when the frequency is increased. No variation based on the number of active cores can be observed.

code triggers a frequency change. The overhead of the frequency change can be calculated as the difference of the two runtimes. The results of this experiment for all combinations of scalar, AVX2, and AVX-512 instructions are shown in Figure 3, which shows significantly higher overhead than measured by Mazouz et al. [24]. For example, a transition from the maximum frequency to the AVX2 frequency level takes  $17\mu\text{s}$  on average, whereas a transition to the AVX-512 frequency level takes  $24\mu\text{s}$ . The reason for this increased overhead is likely the reduced IPC due to additional throttling before the frequency switch is complete [11]. As AVX2 and AVX-512 instructions would draw excessive power at the previous higher frequency, the system temporarily employs throttling to reduce power consumption [9]. Note that the overhead appears to vary slightly for the

different frequencies and frequency differences caused by different numbers of active cores.

Measuring the overhead of frequency increases is slightly more complex due to the large – and, in our experiment, somewhat variable – delay before the system restores the non-AVX frequency level. In this case, we employ the technique employed by Mazouz et al. to determine frequency change costs [24] as we start at a system running at either AVX2 or AVX-512 frequencies and repeatedly execute a short code section which consists of instructions allowing a higher frequency. We measure the runtime of the code section each time, so that frequency changes are shown as spikes in the measured runtime. As other sources such as the activation of additional cores can trigger additional reduction of the maximum frequency, we simply assume that the first frequency change is the one triggered by the lack of AVX2 and AVX-512 instructions and discard any further runtime spikes. The size of the spike is assumed to be the overhead of the frequency change, which is plotted in Figure 4. The results closely match those of Mazouz et al. [24] and show no variation based on the absolute frequency of the core or the magnitude of the frequency change, both of which vary with the number of active cores. Note, however, that this experiment does not consider the performance loss due to the system temporarily executing at a lower frequency while the voltage is ramped up to the level required for the frequency change [26]. For many dynamic power management approaches, state changes can be predicted in advance, so voltage changes can likely be conducted speculatively, removing the need for such additional delays. For example, for fixed-timeout policies, the timeout can be slightly reduced accordingly.

## 4.2 Performance Versus Frequency

The break-even time for frequency changes depends not only on the overhead for frequency transitions but also on the relative performance advantage due to the higher frequency. Whereas the performance of CPU-bound tasks is nearly proportional to the CPU frequency, the same is not true for memory-heavy workloads as the memory latency is independent from the CPU frequency. In this work, to simplify the prototype, we assume the former.

The result of this simplification is that the break-even time is underestimated for memory-heavy applications. To quantify this error for the workloads used in this paper, we executed most of the individual applications described in Section 2 – nginx, x265, the Parsec benchmarks and the PTS benchmarks with the exception of mysql due to the long execution time of

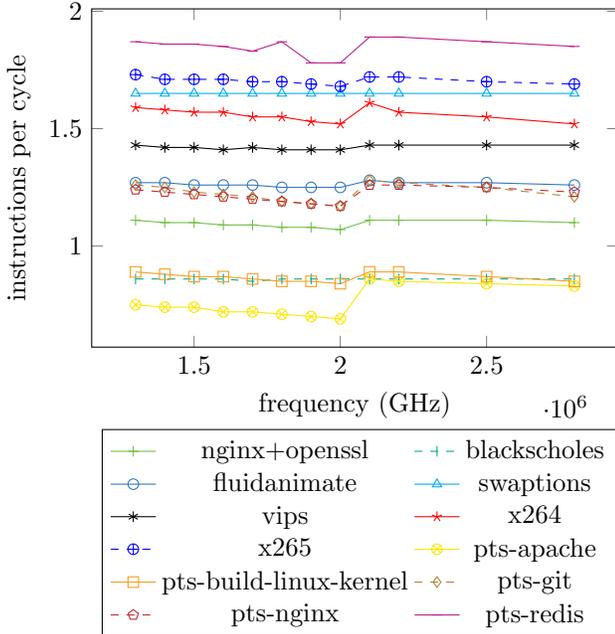


Figure 5: IPC of various parsec and PTS benchmarks as well as the nginx/OpenSSL workload described in Section 2 when executed at different frequencies. In the monotone region between 2.1 GHz and 2.8 GHz the benchmarks show little IPC variation. The step between 2.0 GHz and 2.1 GHz is likely because by either memory or bus frequency scaling in relation to the frequency of the cores.

the corresponding benchmark and sqlite due to its particularly I/O-heavy nature – at different frequencies and measured the instructions per cycle (IPC). We executed the applications at frequencies between 2.8 GHz and 1.3 GHz on a system with a 16-core Intel Xeon Gold 6300 processor. We configured the application to use all cores of the system except for the nginx server benchmark where we allocated three cores to the HTTP request generator<sup>3</sup>. Maximizing the number of active cores should maximize the working set of the application and should therefore maximize the impact of memory accesses on performance.

Figure 5 shows the results of this experiment. Counterintuitively, IPC consistently improves when the frequency is increased from 2.0 GHz to 2.1 GHz – we assume this is due to the chip adapting either memory or bus frequency to the core frequency. For all other frequency ranges, higher frequency correlates with lower IPC. When comparing the IPC at 2.1 GHz

<sup>3</sup>x265 failed to fully saturate all cores due to inter-thread dependencies.

and 2.8 GHz, the biggest difference was found for x264 which had 5.9% higher IPC at 2.1 GHz. This IPC difference would translate into a error of 5.9% during break-even time calculation, which is likely low enough for the simplified model to be viable for this workload.

The reason for the low IPC changes is found in the low cache miss rates for all these applications: The workloads trigger at most 2.03 last-level cache misses per 1000 instructions (in the case of PTS build-linux-kernel).

Note that our simulation to show the viability of improved DVFS policies in Section 6.2 also uses the simplified performance model. However, as our experiment shows, the resulting error is negligible and does not influence our conclusions. The simulation uses the nginx web server with the configuration marked as “nginx+openssl” in Figure 5. In this configuration, the nginx web server showed less than 1% IPC difference between 2.1 GHz and 2.8 GHz.

Workloads with higher cache miss ratios than the benchmarks shown in Figure 5 can show lower correlation between performance and frequency [16]. While we show that improved DVFS policies in general have the potential to improve performance for workloads involving AVX2 and AVX-512 code, our simplified linear model might not be sufficient for these workloads in practice. Concrete DVFS policy implementations for such workloads might require a better prediction of the performance at different frequencies to make decisions on whether to change the CPU frequency or not. Such predictions can be made, for example, by using performance counters to determine the impact of frequency changes on the number of stall cycles [19]. Further research has to be conducted to show whether DVFS policies based on such approaches are viable and provide a significant performance advantage for a wider range of workloads.

### 4.3 Break-Even Time

The break-even time  $t_{BE}$  – i.e., the time after which the performance increase due to increased frequencies offsets the cost to increase and decrease the frequency – can be calculated according to the following formula:

$$p_{low}t_{BE} = p_{high}(t_{BE} - t_o)$$

In this formula,  $p_{low}$  and  $p_{high}$  are the performance at the lower and higher frequency, respectively, and  $t_o = t_{o,d} + t_{o,u}$  is the total overhead for reducing ( $t_{o,d}$ ) and increasing ( $t_{o,u}$ ) the frequency, measured as the equivalent CPU time as in Section 4.1.

If we insert the results from the last sections and calculate  $t_{BE}$ , we arrive at the times shown in Figure 6.

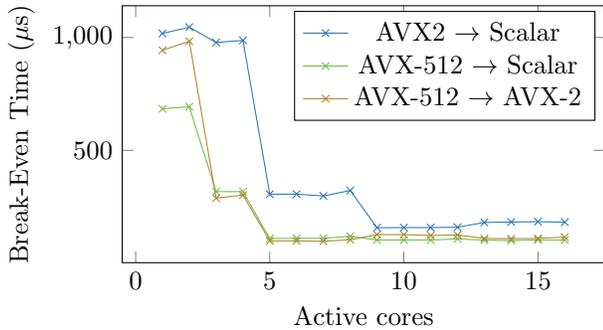


Figure 6: Break-even time for frequency changes calculated from Figure 3 and 4, assuming performance to be proportional to frequency. The break-even times vary with the number of active cores due to the different magnitude of the frequency change. The results show that a single fixed timeout as implemented by Intel CPUs can not be optimal in terms of worst-case competitiveness.

As the performance is dominated by the frequency whereas the overhead is fairly constant, the break-even time is significantly affected by the number of active cores. For example, for a transition between AVX2 and non-AVX frequencies, the break-even time in situations with less than four active cores is approximately  $1000 \mu\text{s}$  due to the low frequency swing of only 100 MHz (see Table 1), whereas for more than eight cores frequency changes between 400 and 500 MHz cause break-even times between 150 and  $190 \mu\text{s}$ .

As Karlin et al. [18] show, a fixed-timeout policy achieves optimal competitiveness – in our case, minimal overhead when the system has to switch back to a lower frequency at the least opportunistic time – when the timeout equals the break-even time. In this case, the timeout before the CPU increases its frequency should therefore be based on the frequency difference to achieve good competitiveness in all cases. Intel CPUs, however, only implement one fixed timeout for all core counts and instruction sets. As shown in Section 2, some applications are negatively affected by the overhead of frequency changes, which shows that an improved DVFS policy with variable timeout based on the frequency difference can likely have positive impact on these applications.

## 5 Exploiting Application Behavior

While the 2-competitive fixed-timeout policy is optimal in the worst case for unpredictable workloads, it is not when the behavior of the workload is predictable,

in which case earlier decisions to increase the CPU frequency can result in higher performance. In this work, we focus on two types of predictions about whether the system is going to use AVX-512 in the near future. First, the application developer has knowledge about the structure of the application and can tell the operating system when AVX-intensive parts begin and end, which can aid workloads where one process switches between AVX-intensive code and code without power-intensive instructions. Second, the operating system can statistically determine whether a process is likely to require a reduced frequency and can change the CPU frequency during context switches in order to immediately let non-power-intensive processes profit from higher frequencies.

### 5.1 Heterogeneous Applications

If an application consists of vectorized and non-vectorized parts and those are executed alternately – such as the web server example in Section 1 – the non-vectorized part is slowed down due to the frequency change caused by the vectorized part. Often, software developers know which part of the application is vectorized and how long execution of each part takes. In that case, assuming that a suitable hardware-software interface exists, they can notify the CPU after each vectorized code portion if the next scalar portion is likely *long enough* to warrant for an early frequency increase. The CPU could use that hint to immediately switch to a higher frequency. Such a hint could therefore improve performance, as the existing DVFS policy of the CPU would instead needlessly keep the frequency reduced for some time.

### 5.2 Classification of Tasks

Even if each individual application is sufficiently uniform, it is still possible that context switches between different applications cause overhead as an application is slowed down by the preceding AVX-enabled application as described in Section 2. For most workloads, this overhead is avoidable, as scheduler time slices are usually longer than the break-even time. During a switch from an AVX-enabled application to a non-AVX application, the scheduler should usually immediately select a higher frequency.

To trigger such frequency changes, the scheduler needs a categorization of the individual processes based on their instruction set usage and their expected frequency reduction. To this end, we introduce the notion of a *power score* which serves as a measure of the expected power consumption of the instruction mix exe-

cuted by a process. A high power score signals that the process will likely trigger significant frequency reductions. More specifically, a power score of 1 means that the process is assumed to execute at AVX2 frequencies, whereas a power score of 2 means that the process likely causes a reduction down to AVX-512 frequency levels.

This power score could potentially be determined either via a static analysis of the application binary or via a dynamic analysis of the frequency changes at runtime. A static analysis can detect whether an executable contains any AVX2 or AVX-512 instructions that could trigger a frequency reduction. However, applications might contain such instructions even if they do not execute them frequently enough to significantly reduce the average frequency. Also, functions like `memset` make use of AVX-512 instructions, but only for inputs of certain size which is hard to detect via static analysis. Overall, a static analysis is therefore bound to be unreliable.

We expect dynamic analyses to yield a better estimate of the instruction set usage of individual processes as they are able to observe the effects of the actual execution patterns within the process. Simply mapping the frequency level to the active process is, however, not accurate in situations with frequent context switches, because the delays mean that some of the time spent at lower frequencies is attributed to the wrong processes. Counting the AVX2 and AVX-512 instructions executed by the active process might be sufficient to draw conclusions about the resulting frequency requirements in most cases, but recent Intel CPUs only provide performance counters for specific types of such instructions [4, p. 19-20f]. In any case, though, more accurate statistics would be possible if the processor provided the operating system with information about whether the conditions for each frequency level were fulfilled at each point in time, for example via appropriate performance counters. Current hardware does not provide such performance counters, either.

As a method to collect reliable information about frequency requirements and to determine the processes responsible for frequency reductions, we therefore suggest distinguishing between two cases based on the time between subsequent scheduler invocations.

If the time between subsequent scheduler invocations is significantly longer than the frequency increase delay of 670  $\mu$ s, the scheduler can sample the CPU frequency level and can directly attribute the frequency to the last process, as any influence of its predecessor on the CPU frequency has ended. To determine the CPU frequency level, we configure the performance counters to track the cycles spent at *power license levels* 0, 1, and 2 which correspond to the frequency levels for non-

AVX, AVX2, and AVX-512 code, respectively [6].

If the time between subsequent scheduler invocations is shorter than the frequency increase delay, such an approach would risk misattributing frequency changes. In this case, our main observation is that if the frequency is reduced during the execution of a process, then that process is most likely responsible for the change. For short periods of execution of a process, we therefore only attribute the resulting frequency to the process in case of a frequency change during the period. In some rare cases, however, frequency changes can occur during the execution of a process that did not trigger the change – most likely due to delays during frequency selection as documented by Intel [6]. Therefore, the power score is calculated as the moving average over all CPU frequency samples attributed to a process to reduce the impact of occasional misattribution. The following steps are conducted to calculate the power score of the processes:

1. Initially, the power score of new processes is set to 0, i.e., the system assumes that new processes will not use AVX-512 or AVX2.
2. At each scheduler invocation, we detect the current power license level by sampling all power license level performance counters twice in a row. The counter that is incremented during the short time inbetween indicates the current frequency level.
3. We compare the level during two consecutive context switches. If the levels match, the power license did not change. In this case, for short CPU bursts, the current process might not have had enough time to have an impact on the power license, so the power score is not updated.
4. If context switches are more than 1 ms apart – longer than the frequency delay as reasoned above – or if the power license decreases below or increases above the current power score, however, the power score of the process is updated as the exponential moving average of such power license changes. Assuming  $S_{t-1}$  is the old power score and  $L_t$  is the new power license, the new power score is  $S_t = 0.2L_t + 0.8S_{t-1}$ .

The resulting power score indicates the potential frequency reduction caused by the process. The dynamic analysis of frequency changes can be combined with the results of a static analysis of the executable – e.g., by overriding the score to be 0 if the executable does not contain AVX2 nor AVX-512 instructions – and with manual instrumentation as described in Section 5.1, in

which case hints from the developer override the automatically determined power score.

Note that with hyperthreading the frequency is determined by two programs. Thus, this technique only works on systems with deactivated hyperthreading and on systems which always schedule the same program on both cores, as recently suggested for the Linux kernel [10]. On other systems, the hardware has to be modified to provide a more reliable source of information about the energy consumption of the instructions executed by individual processes.

### 5.3 Using Hints For DVFS

Once predictions about the instruction set use are available, the system can use this information to improve performance. When the code running on a core – i.e., all hardware threads in the case of a system with hardware multithreading – indicates that no power-intensive instructions are going to be executed in the near future, for example, via the mechanisms presented in Sections 5.1 or 5.2, the system can eagerly increase the frequency when it is not already at the highest level possible for the expected instructions.

Ideally, the DVFS policy should be implemented in the CPU to be able to provide quick reactions to changing instruction usage and to prevent power budget violations, so any hint about future instruction usage needs to be communicated to the CPU using an appropriate software-hardware interface. For example, the operating system or the application software could temporarily configure a different frequency change timeout depending on the type of executed code, to force earlier frequency changes or to prevent any changes.

#### 5.3.1 Viability on Current CPUs

Current hardware does not provide any such interface. It does, however, provide a mechanism to manually set the CPU frequency, which can be used to implement a wide range of DVFS policies in software. For the dim silicon scenario described in this paper, the limitations of the hardware prevent both practical software-based implementations of DVFS policies as well as limited implementations to estimate the performance of hardware-based implementations.

Any practical implementation of a DVFS solution for AVX-512 or AVX2 code is prevented both by the inability to detect problematic AVX-512 or AVX2 code as well as by the delay of manual frequency changes. First, conservative detection of problematic code is necessary so that the OS knows when frequency reductions are required. Our approach in Section 5.2 is not usable

as it only results in approximate long-term classification of applications. In contrast, conservative short-term estimation based on register set usage can detect any access to 512-bit and 256-bit vector registers but will often select lower frequencies than necessary as we show in our evaluation in Section 6.1, leading to reduced performance. Second, software-based DVFS policy implementations require the ability to change the frequency at a precise point in time, yet current CPUs delay frequency changes significantly. As described by Hackenberg et al. [15], the frequency selection logic of Intel CPUs starting with the Haswell microarchitecture only allows frequency changes once every 500  $\mu$ s, so any frequency change request is delayed until the end of the next such 500  $\mu$ s window. The immediate throttling of AVX-512 instructions [11], however, shows that immediate power reduction is necessary for stability, so such delays are unacceptable.

These limitations not only prevent practical software solutions but unfortunately also prevent the construction of a prototype based on existing hardware to evaluate the performance of hardware implementations. Such a prototype would not necessarily have to be able to ensure system stability, but would have to trigger frequency changes in a way that results in equal performance compared to a complete implementation. As from the point of view of the OS the frequency change delay often appears to be random with an even distribution, a naïve approach might assume that the average delay of frequency increases cancels out the average delay of frequency reductions. However, for short sections of AVX- or non-AVX code, both frequency increase and decrease might occur within the same 500  $\mu$ s window, in which our experiments showed that no frequency change occurs.

In this paper, we suggest improved DVFS policies as a method to reduce the overhead caused by AVX2 and AVX-512. As we cannot use existing hardware to conduct a performance evaluation, we are limited to demonstrating the performance impact through simulations and microbenchmarks as shown in Section 6.2.

## 6 Evaluation

As described in the last section, this paper proposes using hints from the application or the operating system to provide improved frequency scaling. Our approach consists of two main pieces, namely the classification of the processes – or, alternatively, hints from the application developer – and a modified DVFS algorithm that takes those hints into account. For existing processors, it is impossible to build a complete implementation of this design, as the exist-

ing DVFS policy implemented by the CPU cannot be extended as required. Deactivating all AVX-induced frequency changes and completely reimplementing the policy in software is impossible due to the latency of software-triggered frequency changes which can be as long as  $500\ \mu\text{s}$ . Our evaluation is therefore limited to qualitatively showing that the individual components are functional and that application-directed DVFS can have an advantage over the existing policy.

## 6.1 Categorization of Processes

The main goal of the process classification mechanism described in Section 5.2 is to be able to detect the required power license of individual processes even if they are running in a heterogeneous multi-process workload where the effects of one process on the CPU frequency might shadow the effects of another process. To show that the mechanism fulfills this goal, we constructed a prototype based on Linux 5.2. We modified the kernel’s completely fair scheduler (CFS) and inserted the power license detection code in the main scheduler function `__schedule()`. Our implementation uses the Linux perf framework to read the power license performance counters.

We let our prototype estimate the power score of the x265 video encoder using different instruction sets running in isolation. To show that our prototype is able to correctly distinguish between different processes executing on the same system and is able to attribute frequency changes to the correct process, we also executed the Apache benchmark from the Phoronix Test Suite as well as the swaptions benchmark from Parsec in parallel with x265. The two applications were configured to share the same set of cores without any restrictions to scheduling. Note that we specifically selected an interactive benchmark as well as a batch workload, to show that the classification works with both. Table 2 shows both the expected power score for the applications – we expected our prototype to classify x265 according to the instruction set used, and neither of the other two benchmarks used significant amounts of vector instructions – as well as the estimated power score from our prototype, averaged over the runtime of the application.

The first three rows show that x265 was correctly classified in all cases, except for some uncertainty if neither AVX2 nor AVX-512 instructions were used. The next two table rows then show the results for the mixed scenarios. In both cases, our prototype can correctly identify x265 as the process responsible for the frequency reduction. For x265 executed alone, we compared the performance of our prototype to a stock

Linux kernel and were not able to measure any statistically significant performance overhead.

We compare our approach to the state-of-the-art technique available in the Linux kernel. Linux provides the time elapsed since the last use of AVX-512 as part of the `arch_status` file in the proc file system [2]. The time since the last use of AVX-512 is calculated by checking the state of the FPU registers at each context switch. Like our approach, this mechanism is able to detect AVX-512 usage in the benchmarks described above as shown in the upper half of Table 2.

The approach found in the Linux kernel has a significant drawback, though, as the use of specific FPU registers is only loosely connected to the resulting frequency change. For example, a dense sequence of multiplication instructions on 512-bit vector registers causes the CPU to transition to the lowest frequency, whereas other instructions only trigger the intermediate “AVX2” frequency. Therefore, in a workload consisting of processes showing the former behavior as well as processes of the latter type, the time since the last 512-bit register usage cannot be used to identify the processes responsible for a frequency reduction. We demonstrate this effect by executing a sequence of 512-bit and 256-bit multiplications and additions both with our approach and on an unmodified Linux 5.5 kernel. The results shown in the lower half of Table 2 show that our prototype is correctly able to detect the three different frequency levels caused by different types of instructions, whereas the stock Linux kernel is only able to detect whether 512-bit registers are used.

To show that the problem also affects real-world workloads, we execute a web server benchmark using nginx and OpenSSL similar to the one described in Section 2 and measure the average time since the last AVX-512 usage as determined by the Linux 5.5 kernel on a system running Fedora 30. We let the nginx web server serve a static file with compression at runtime and use OpenSSL compiled with either AVX2 and AVX-512 instruction support for TLS encryption. As shown above, the web server provides significantly higher performance when using AVX2 instructions due to the resulting higher frequencies. Even in this case the system uses 512-bit registers, though, as the C library provides AVX-512 variants of `memset()`, `memcpy()`, and `memmove()`. Therefore, the stock Linux kernel detects AVX-512 usage in both cases, with similar reported average time since the last usage of 512-bit registers. Note that the implementation tests whether registers are in use only during context switches. Different scheduling causes large variation in the resulting values, making a quantitative comparison for such experiments difficult.

Scenario	Predominant Freq. Level	AVX Score	Average AVX512_elapsed_ms
x265 (AVX)	0	0.356	N/A
x265 (AVX2)	1	1.005	N/A
x265 (AVX-512)	2	1.899	103.4 ms
x265 (AVX-512)	2	1.827	181.3 ms
+ pts-apache	0	0.428	N/A
x265 (AVX-512)	2	1.679	86.0 ms
+ parsec-swaptions	0	0.099	N/A
512-bit FMA	2	1.821	0.10 ms
512-bit add	1	0.917	0.10 ms
256-bit FMA	1	0.934	N/A
256-bit add	0	0	N/A

Table 2: Estimated AVX scores for different scenarios and a comparison to the mechanism found in the Linux kernel to track AVX-512 usage. The first three rows show the score for an isolated instance of x265 using different instruction sets. The next two rows show the scores in scenarios with two different applications running concurrently on the same set of cores, to show that the score is estimated correctly on a per-process basis even if one process affects the frequency of another. The remaining rows show how our approach is able to distinguish between the three frequency levels of the CPU, whereas the stock Linux kernel is only able to track AVX-512 register usage.

## 6.2 Potential of Eager Frequency Changes

Once it is known which parts of the system use power-intensive instructions – either using manual annotation as described in Section 5.1 or via automatic detection as described in the previous section – this information can be used to optimize performance. Whereas other approaches perform core specialization to separate AVX-512 code from non-AVX code [13,22], we, as described in Section 5, suggest that improved DVFS policies can also significantly reduce the overhead caused by AVX-512 instructions and similarly power-intensive instruction sets. In particular, we suggest that the delay for frequency increases as implemented by recent Intel CPUs is unnecessary if the system can predict that the software executed in the near future does not require power-intensive instructions.

### 6.2.1 Methodology

The most direct method to show the potential of improved DVFS policies would be to compare the performance of a benchmarked application when using a fixed-timeout policy such as the one implemented by the processor to the same benchmark instrumented to change the processor frequency at points in the program selected by the developer. However, recent Intel CPUs delay frequency change requests by up to 500  $\mu$ s [15], making it impossible to precisely specify the points in the program at which frequency changes

occur. Therefore, our evaluation relies on simulation of different DVFS policies based on a trace generated while running a web server benchmark (Section 6.2.2) and uses a microbenchmark to demonstrate the potential performance impact of a single eager frequency change (Section 6.3) and to check the accuracy of the simulation.

The following experiments were conducted on a system with an Intel Core i9-7940X processor, with the simulation configured to match this system. This processor was selected because, as it is designed for over-clocking, it allows configuration of the *AVX offsets* which specify the frequency reduction caused by AVX2 and AVX-512 instructions. For tests to determine the baseline performance of the system, we configured the offsets to match the frequencies reported in news articles [32], where the base frequency of the processor is reported to be 3.1 GHz and the frequencies for AVX2 and AVX-512 code are 2.7 GHz and 2.4 GHz, respectively, providing similar frequency ratios compared to server processors. Note that no authoritative information about AVX2 and AVX-512 frequencies is found in official Intel documentation and that mainboards such as ours frequently provide non-default AVX offsets.

With minimal AVX offsets, the AVX2 and AVX-512 frequencies are both 3.0 GHz, so the AVX frequency reduction cannot be disabled completely. We discuss the effect of this minimum frequency change where applicable below. All experiments were executed with Turbo Boost disabled and with C-states limited to C1

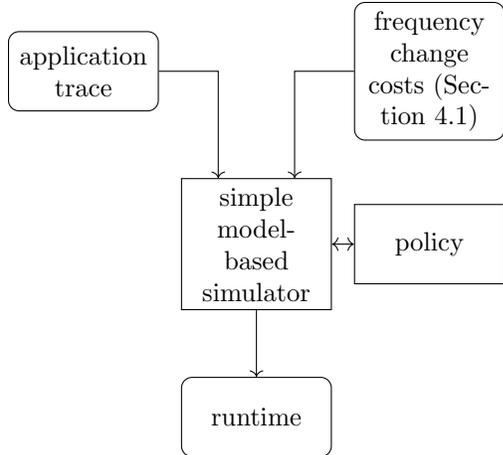


Figure 7: Experimental setup to estimate the performance resulting from different DVFS policies. Our simulator implements a very simple performance model assuming fixed frequency change costs and performance proportional to the CPU frequency. We instrumented the nginx web server and used the resulting trace of AVX- and non-AVX periods to determine the runtime with different DVFS policies.

in order to reduce variance in the measurement results.

### 6.2.2 Web Server Simulation

For our simulation experiments, the workload used is the nginx web server example from Section 2. We configure the web server to serve a single static file using gzip compression and we encrypt HTTP requests and replies using the OpenSSL library. The library is configured to vectorize encryption and decryption using AVX-512 instructions, which in other experiments has resulted in a 10% slowdown. We instrument the web server to record the times when the OpenSSL functions for encryption and decryption are called and when they return. When generating the log of the OpenSSL function calls, we execute the benchmark with minimal AVX offsets. Although the resulting frequencies would not be stable and would result in frequent system crashes with all cores utilized, this setup yields more representative timing input for the simulator, as the simulator itself is supposed to slow down the AVX-512 portions of the simulated workload. To ensure system stability and to simplify simulation, the web server is only executed on a single core. We do not expect individual web server threads to behave significantly different when additional web server threads are placed on the other cores of the system.

The resulting application trace contains a list of periods where the system is assumed to execute only AVX-512 code (the function calls into OpenSSL) alternating with periods where the system is assumed not to execute any AVX-512 or AVX2 instructions. We feed this trace into a simple model-based simulator as shown in Figure 7 to estimate the application runtime resulting from different DVFS policies. The simulator applies a DVFS policy to the trace and dilates the time during periods where the CPU would be executing at a lower frequency. During the simulation, to get results more representative for a server scenario, we assume that most of the cores are active and assume a corresponding large frequency reduction whenever AVX-512 code is executed.

We implement fixed-timeout policies with the timeout used by Intel processors as well as with a timeout of 180  $\mu$ s which was shown to be more competitive in Section 4.3. As an example for a policy based on developer input, we also implement a policy which only increases the frequency when the last packet of an HTTP request was received and decrypted, which we identify by the return value of the corresponding OpenSSL function call<sup>4</sup>. After this call, the web server processes the request and takes a significant amount of time before any further AVX-512 code is executed when the HTTP reply is sent, so at this point eager frequency changes are most likely to be beneficial for application performance. For all the policies, the simulator assumes a performance impact of 16  $\mu$ s per frequency change, similar to the values determined experimentally in Section 4.1.

The simulation result shows that a lower timeout than what is used by Intel CPUs results in a 2.9% higher performance in the simulated scenario. With a lower timeout, the policy can exploit shorter non-AVX program phases and wastes less time at lower frequencies throughout the program. The resulting performance improvement outweighs the (simulated) overhead of the larger number of frequency changes. Even though the difference is small, the result shows that the timeout does have a measurable impact on application performance.

The developer-directed DVFS policy performed even better, with a 3.9% performance improvement compared to the policy implemented by Intel CPUs, as the policy was able to completely mitigate overhead due to low CPU frequencies during the longest non-AVX phases of the program. While this improvement might seem minor, it covers most of the 5.7% overhead caused

<sup>4</sup>A more generic and robust implementation would be to instrument the HTTP request parsing logic to increase the frequency whenever the end of a HTTP request is detected. Our implementation suffices to show that the approach is generally possible.

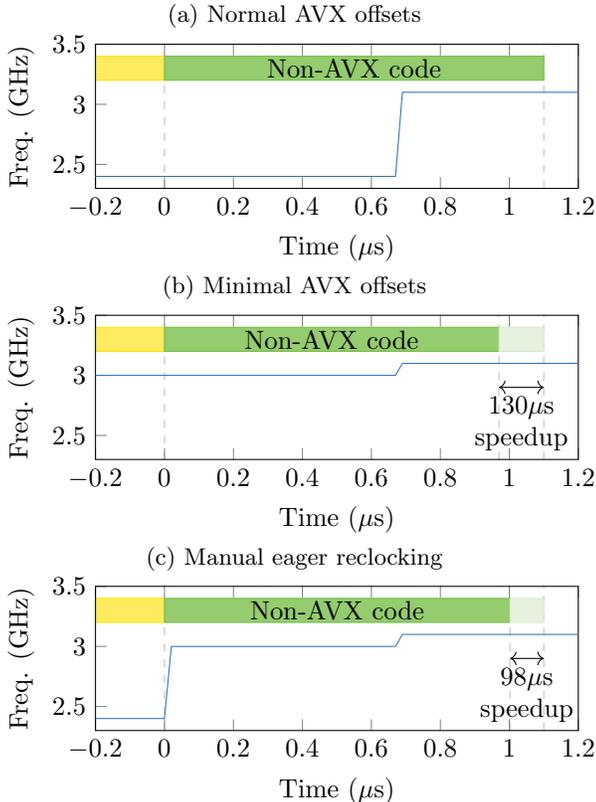


Figure 8: To determine the potential performance improvement for a single frequency change, we execute a fixed amount of non-AVX code directly following some AVX-512 code. We compare the time required at default AVX offsets (a) to the time required at minimal AVX offsets (b) as well as with eager frequency changes simulated by a manually inserted frequency change at the beginning of the non-AVX code (c).

by AVX-512 for this workload as shown in Figure 2. Workloads with more frequent AVX-512 phases might benefit more from improved policies. In addition, the policy did not increase the frequency during some other non-AVX phases where a frequency change would have been beneficial, showing that a carefully optimized prototype might achieve higher performance.

### 6.3 Maximum Potential per Frequency Change

When looking at a single developer-directed eager frequency change, the simulation resulted in a CPU time saving of 195  $\mu\text{s}$  for sufficiently long stretches of non-AVX code compared to a fixed timeout of 670  $\mu\text{s}$  as implemented by current Intel CPUs, as the CPU was operating 30% faster during this time. To show that

the assumptions made in our simulator yield realistic results, we validate this value against measurements based on a simple microbenchmark. The microbenchmark first executes a series of AVX-512 instructions and then executes a fixed amount of non-AVX instructions. The number of instructions is chosen so that they take longer than the frequency change timeout implemented by the CPU. We measure the time required for the code section to determine the impact of the frequency change caused by the preceding AVX-512 code in different configurations. All experiments are repeated 1000 times.

First, to measure the overall impact of such frequency changes on the CPU, we compare the average time at default frequencies (Figure 8a) with the average time with minimal AVX offsets (Figure 8b). Our experiment shows that with minimal frequency changes the code executes 130  $\mu\text{s}$  faster. In this configuration the AVX-512 code still reduces the CPU frequency by 100 MHz as described in Section 6.2.1 and the measured runtime still includes the overhead of the corresponding frequency change which needs to be taken into account when comparing the values with the model used for our simulation.

Second, we manually insert frequency changes into our prototype so that the frequency is reduced when the AVX-512 code starts and is immediately increased when the non-AVX code starts. Note that, as described in Section 6.2.1, frequency changes are applied with a random delay of up to 500  $\mu\text{s}$ . Therefore, for this experiment, we do not take the average time but instead take the 5th percentile as this value represents the situation when an optimized DVFS policy implementation almost immediately triggers frequency changes. In this experiment, we measure a runtime for the non-AVX code which is 32  $\mu\text{s}$  slower than the result with minimum frequency changes, but 98  $\mu\text{s}$  faster than regular frequency changes (Figure 8c). The performance is slightly lower than in the experiment with minimal AVX offsets because the benchmark triggers not one but two frequency changes – one by the hardware due to the 100 MHz reduction described above, and one manual frequency change to simulate the DVFS policy. Apart from this overhead and minor overhead due to the additional system calls, the runtime mostly matches the optimal case, which supports our model that eager frequency changes can mitigate most of the overhead caused by AVX instructions.

However, the absolute runtime differences are lower than determined by the simulation. As described above, two potential reasons for the deviation are the larger number of frequency changes as well as some remaining frequency reduction. As shown in Fig-

ure 4, the additional frequency change costs approximately 10  $\mu$ s, and an expected 3% performance overhead due to the 100 MHz frequency difference costs another 20  $\mu$ s. While the measured results mostly match our model when taking these effects into account, further analysis of the CPU behavior has to be conducted to provide a better quantitative model of the performance in similar situations.

## 7 Discussion

In this paper, we showed that the fixed timeout policy implemented by recent Intel CPUs for AVX frequencies yields less-than-optimal average processor frequencies for heterogeneous workloads. We also argue that better timeouts and developer-directed frequency changes can improve performance. Even though our evaluation lacks experiments to directly demonstrate the effects on real-world workloads, the estimate generated by our simulation shows that it is highly likely that such a performance improvement is to be expected. This basic result opens up a number of further research questions which we will discuss in the following sections.

### 7.1 Hardware Interfaces

In Section 6.2.1, we show why the frequency change delays on current Intel CPUs prevent constructing a full prototype demonstrating our approach. Even if frequency changes were triggered instantly, though, a software-only DVFS policy implementation would not be viable for two reasons: First, the CPU would still need to be able to autonomously reduce power consumption when executing AVX-512 instructions to ensure system stability, for example, by reducing the frequency or applying other forms of throttling. Second, not all applications in the system would be modified to make use of developer-directed frequency scaling, making a hardware fallback necessary.

If the DVFS policy is implemented in hardware, a software-hardware interface is required to influence policy decisions. We propose the combination of two such interfaces:

1. **Configurable frequency change delay:** As we show, the problem of AVX-induced frequency changes is similar to the dynamic power management problem, and the main decision is whether to immediately increase the frequency when possible or whether to wait or not increase the frequency at all. While it would be possible to tell

the CPU to immediately increase the frequency after the next section of AVX code, we expect such an interface not to be viable in many situations, because the boundaries of AVX-intensive program execution phases are not well defined and variations in the program’s control flow might cause unnecessary frequency changes. Instead, we suggest an interface to manually set a different frequency change timeout for individual parts of the program – i.e., until the application manually reverts the change or sets a different timeout – to allow applications to enable eager frequency changes in certain situations.

2. **Forced immediate frequency change:** In addition, the CPU should provide an interface to immediately increase the frequency to the maximum frequency for use by the operating system to increase the frequency during context switches when it is known that the next task is unlikely to use AVX-512 or AVX2.

Further work has to be conducted to test whether these interfaces are sufficiently flexible to implement a wide range of DVFS policies in software.

### 7.2 Hardware Multithreading

One significant limitation of our work is that all our experiments were conducted with a system in mind that does not use hardware multithreading. On a system with hardware multithreading, the CPU frequency has to be reduced when either of the threads executes AVX instructions, thereby limiting the potential performance advantage of developer-directed approaches as it is hard to predict when another completely unrelated hyperthread will affect the frequency. Also, as shown in Section 2 on systems with hyperthreading many additional types of workloads experience slowdown due to frequency reductions. Despite the differences, improved DVFS policies might be viable and their effectiveness might even be amplified as more code is affected by frequency reductions. More research should be conducted to create a statistical model of the CPU frequency selection in systems with hardware multithreading and to develop suitable DVFS policies.

#### 7.2.1 AVX Overhead Profiling

In our controlled experiment, we used a benchmark that had a clearly defined performance metric. In general it is, however, not always clear whether the overhead caused by AVX-512 is large enough to warrant

the usage of techniques to reduce it and it is not always clear whether these techniques are successful. In particular when techniques have the potential to cause additional overhead – for example, due to increased numbers of frequency changes – it would be beneficial to be able to profile a system to estimate the impact of AVX-512 on performance. The result of such a profiler could also be used to implement close-loop policies. For example, the system could repeatedly try out different DVFS policies depending on the resulting performance change.

The performance counters on current CPUs, however, cannot be used to construct such a profiler, as they can only be used to count cycles spent at reduced frequencies but do not provide sufficient information about how long the reduced frequencies are actually required. In particular, the performance monitoring units of these CPUs can not be used to detect any executed AVX2 and AVX-512 instructions as they can only count floating point instructions. Instead, we envision an approach which periodically samples the frequency of the system, pauses the system to let the CPU switch back to the highest possible frequency, and then checks whether the system will immediately switch back to a lower frequency when the workload is continued. The latter check determines whether a frequency reduction is required due to ongoing AVX code or whether the reduction represents avoidable overhead. Further experiments have to determine the accuracy of such an approach, and further work has to be conducted to show whether modified hardware-software interfaces can provide a more accurate profiling mechanism with lower CPU time overhead.

## 8 Related Work

This paper presents improved DVFS policies as a method to reduce the overhead of the frequency reduction caused by AVX and AVX-512 instructions on recent Intel CPUs. Other approaches to this and similar problems have used core specialization or have modified the application to reduce the impact of varying power consumption and of frequent frequency changes.

### 8.1 Core Specialization

Another method to limit the performance impact of AVX and AVX-512 code on unrelated non-AVX code is to place AVX and non-AVX parts of the workload on separate sets of cores. As performance problems occur when non-AVX code is executed on the same core following AVX code which reduced the frequency,

specialization of cores can prevent such overhead. Approaches for core specialization either targeted heterogeneous programs consisting of AVX and non-AVX code within one process [13] or targeted workloads consisting of AVX and non-AVX processes [22]. The former detects the usage of AVX instructions either by instrumentation inserted by the developer or by reconfiguring the CPU to trigger exceptions when executing AVX instructions [13, 14]. Based on this information, individual threads are migrated between cores to concentrate the AVX part of the program on as few cores as possible. The latter technique which is targeted at multi-process workloads instead relies on heuristics to identify processes using AVX-512 instructions and modifies the scheduler to prevent scheduling an AVX-512 and a non-AVX task on hardware threads of the same core at the same time [22]. This approach currently uses the Linux `arch_status` interface which only gives a rough estimate of AVX-512 usage. In this paper, we present a method to identify applications which cause frequency reductions with higher accuracy.

Note that all these approaches can cause significant performance overhead themselves. Task migrations can increase cache miss rates, and restricting scheduling of different processes on the same core at the same time can cause significant overhead with some workloads [10]. We present a technique which might provide advantages in situations where other approaches cause too much overhead.

The fact that co-scheduling applications on the hardware threads of a single core can cause varying overhead depending on the type of the applications has been observed by other works before and many scheduling techniques have been developed to improve the performance of SMT systems. For example, existing approaches use sampling-based techniques [31], cache conflict detection [30], or performance counters [12, 25] to determine whether two tasks are suited for parallel scheduling on the same physical core. We describe a similar approach which uses performance counters to identify tasks requiring execution at reduced frequency and which can likely be used for improved co-scheduling of AVX-512 applications as described above.

### 8.2 Profile-Guided Software Modifications

The approach in this paper is designed either for applications which are only available in binary form or which can benefit from AVX2 and AVX-512 instructions. If a program only makes use of such instructions in very short execution phases, those parts could al-

ternatively be rewritten to use instructions with lower power consumption.

Kumar et al. [21] use such an approach to improve the efficiency of power-gating the processor’s SIMD unit. In this scenario, devectorizing parts of the program reduces the speedup caused by SIMD instructions, but reduces the power-gating overhead. The authors use a profiler to determine the SIMD instruction usage in individual parts of the program. As static recompilation based on this information is problematic as the profiling results are only accurate for specific input data, the authors integrate the profiler into a system which uses dynamic translation at runtime to devectorize those parts which only rarely use SIMD instructions. Such an approach could likely be applied to AVX-512 to improve average CPU frequencies, although hardware modifications would be required – current CPUs can only count floating-point AVX-512 instructions, but not integer operations [4, p. 19.20f]. Even with such hardware changes, it is not possible to use the approach with existing ahead-of-time compilers, though. In our work, we explore techniques usable within the existing software environment.

Roy et al. [28], instead, suggest a similar technique that uses information from dynamic profiling to insert static power management code into an application at compile time. Their approach inserts instructions for power gating of parts of the processor in order to save energy. A similar approach, however, could potentially be used to let the application guide frequency selection decisions of the processor.

## 9 Conclusion

Modern Intel CPUs reduce their frequency whenever power-intensive AVX2 or AVX-512 instructions are executed to prevent violating power limits. The frequency is only increased again after a fixed timeout has elapsed, in order to prevent excessive numbers of frequency changes. This behavior reduces the performance for heterogeneous workloads where code sections with and without such AVX instructions alternate, as parts of the latter are executed at a lower frequency than necessary.

We show the similarity between this behavior and mechanisms from dynamic power management. We show that the constant delay before increasing the frequency is not optimal in terms of worst-case competitiveness and show how the delay should depend on the magnitude of the frequency change. We also sketch how information from the OS or the developer can be used to inform the CPU about future system behavior so that the CPU can implement more efficient DVFS

policies. Although we do not have a complete implementation due to constraints of the hardware, we show that it is possible to reliably determine whether an application will cause frequency changes and we show that eager frequency changes based on such information about the workload can improve performance.

### 9.1 Future Work

Although we show that an oracle-style DVFS policy can improve performance, it remains to be seen whether other approaches from the area of dynamic power management can be applied as well. In particular, some shutdown strategies achieve lower power consumption compared to the simple fixed-timeout policy even without application-level knowledge.

In addition, due to hardware constraints, we do not present any complete implementation of our approach. We plan to construct a testbed for other DVFS policies and to use it to evaluate different hardware-software interfaces which would allow input from the operating system or from applications to affect hardware-controlled frequency scaling.

## References

- [1] Phoronix test suite. <https://phoronix-test-suite.com/>.
- [2] The `/proc` filesystem. Linux, Documentation/filesystems/proc.txt.
- [3] Wikichip: Xeon Gold 6130 – Intel. [https://en.wikichip.org/wiki/intel/xeon\\_gold/6130](https://en.wikichip.org/wiki/intel/xeon_gold/6130).
- [4] *Intel 64 and IA-32 Architectures Software Developer’s Manual - Volume 3 (3A, 3B, 3C & 3D): System Programming Guide*, May 2018.
- [5] *Intel Xeon Processor Scalable Family – Specification Update*. Intel Corporation, Feb. 2018.
- [6] *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Sept. 2019.
- [7] L. Benini, A. Bogliolo, and G. De Micheli. A survey of design techniques for system-level dynamic power management. *IEEE transactions on very large scale integration (VLSI) systems*, 8(3):299–316, 2000.
- [8] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

- [9] N. Bonen, R. Gabor, Z. Sperber, V. Svilan, D. N. Mackintosh, J. A. B. Paredes, N. Kumar, and S. Gupta. Performing local power gating in a processor, Sept. 26 2017. US Patent 9,772,674.
- [10] J. Corbet. Core scheduling, Feb. 28 2019. <https://lwn.net/Articles/780703/>.
- [11] T. Downs. Gathering intel on intel avx-512 transitions, Jan. 17 2020.
- [12] A. El-Moursy, R. Garg, D. H. Albonesi, and S. Dwarkadas. Compatible phase co-scheduling on a cmp of multi-threaded processors. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, pages 10–pp. IEEE, 2006.
- [13] M. Gottschlag and F. Bellosa. Reducing avx-induced frequency variation with core specialization. In *The 9th Workshop on Systems for Multi-core and Heterogeneous Architectures*, Dresden, Germany, Mar. 25 2019.
- [14] M. Gottschlag, P. Brantsch, and F. Bellosa. Automatic core specialization for avx-512 applications. In *Proceedings of the 13th ACM International Systems and Storage Conference (to appear)*. ACM, 2020.
- [15] D. Hackenberg, R. Schöne, T. Ilsche, D. Molka, J. Schuchart, and R. Geyer. An energy efficiency feature survey of the intel haswell processor. In *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 896–904. IEEE, 2015.
- [16] R. Hebbbar SR and A. 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.
- [17] W. Huang, K. Rajamani, M. R. Stan, and K. Skadron. Scaling with design constraints: Predicting the future of big chips. *IEEE Micro*, 31(4):16–29, 2011.
- [18] A. R. Karlin, M. S. Manasse, L. A. McGeoch, and S. Owicki. Competitive randomized algorithms for nonuniform problems. *Algorithmica*, 11(6):542–571, 1994.
- [19] G. Keramidas, V. Spiliopoulos, and S. Kaxiras. Interval-based models for run-time dvfs orchestration in superscalar processors. In *Proceedings of the 7th ACM international conference on Computing frontiers*, pages 287–296, 2010.
- [20] V. Krasnov. On the dangers of intel’s frequency scaling, Oct. 10, 2017. <https://blog.cloudflare.com/on-the-dangers-of-intels-frequency-scaling/>.
- [21] R. Kumar, A. Martinez, and A. Gonzalez. Efficient power gating of simd accelerators through dynamic selective devectorization in an hw/sw codesigned environment. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(3):25, 2014.
- [22] A. Li. Core scheduling: prevent fast instructions from slowing you down. Linux Plumbers Conference, Sept. 9 2019.
- [23] Y.-H. Lu, L. Benini, and G. De Micheli. Power-aware operating systems for interactive systems. *IEEE transactions on very large scale integration (VLSI) systems*, 10(2):119–134, 2002.
- [24] A. Mazouz, A. Laurent, B. Pradelle, and W. Jalby. Evaluation of cpu frequency transition latency. *Computer Science - Research and Development*, 29(3-4):187–195, 2014.
- [25] R. L. McGregor, C. D. Antonopoulos, and D. S. Nikolopoulos. Scheduling algorithms for effective thread pairing on hybrid multiprocessors. In *19th IEEE International Parallel and Distributed Processing Symposium*, pages 10–pp. IEEE, 2005.
- [26] S. Park, J. Park, D. Shin, Y. Wang, Q. Xie, M. Pedram, and N. Chang. Accurate modeling of the delay and energy overhead of dynamic voltage and frequency scaling in modern microprocessors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(5):695–708, 2013.
- [27] A. Raghavan, Y. Luo, A. Chandawalla, M. Papaefthymiou, K. P. Pipe, T. F. Wenisch, and M. M. Martin. Computational sprinting. In *IEEE international symposium on high-performance comp architecture*, pages 1–12. IEEE, 2012.
- [28] S. Roy, N. Ranganathan, and S. Katkoori. A framework for power-gating functional units in embedded microprocessors. *IEEE transactions on very large scale integration (VLSI) systems*, 17(11):1640–1649, 2009.
- [29] R. Schöne, T. Ilsche, M. Bielert, A. Gocht, and D. Hackenberg. Energy efficiency features of the intel skylake-sp processor and their impact on performance. *arXiv preprint arXiv:1905.12468*, 2019.

- [30] A. Settle, J. Kihm, A. Janiszewski, and D. Connors. Architectural support for enhanced smt job scheduling. In *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.*, pages 63–73. IEEE, 2004.
- [31] A. Snaveley and D. M. Tullsen. Symbiotic job-scheduling for a simultaneous multithreaded processor. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 234–244, 2000.
- [32] C. Spille. Skylake X: Das heitere AVX-Takteraten hat ein Ende, Sept. 8 2017. <https://www.pcgameshardware.de/Skylake-X-Codename-266252/News/Takt-Reduzierung-AVX2-AVX512-1238210/>.
- [33] M. 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.
- [34] V. Venkatachalam and M. Franz. Power reduction techniques for microprocessor systems. *ACM Computing Surveys (CSUR)*, 37(3):195–237, 2005.
- [35] A. Weissel, B. Beutel, and F. Bellosa. Cooperative i/o: A novel i/o semantics for energy-aware applications. *ACM SIGOPS Operating Systems Review*, 36(SI):117–129, 2002.