Automatic Core Specialization for AVX-512 Applications

Mathias Gottschlag
Karlsruhe Institute of Technology
Karlsruhe, Germany
os@itec.kit.edu

Peter Brantsch
Karlsruhe Institute of Technology
Karlsruhe, Germany
os@itec.kit.edu

Frank Bellosa
Karlsruhe Institute of Technology
Karlsruhe, Germany
os@itec.kit.edu

ABSTRACT

Advanced Vector Extension (AVX) instructions operate on wide SIMD vectors. Due to the resulting high power consumption, recent Intel processors reduce their frequency when executing complex AVX2 and AVX-512 instructions. Following non-AVX code is slowed down by this frequency reduction in two situations: When it executes on the sibling hyperthread of the same core in parallel or – as restoring the non-AVX frequency is delayed – when it directly follows the AVX2/AVX-512 code. As a result, heterogeneous workloads consisting of AVX-512 and non-AVX code are frequently slowed down by 10% on average.

In this work, we describe a method to mitigate the frequency reduction slowdown for workloads involving AVX-512 instructions in both situations. Our approach employs core specialization and partitions the CPU cores into AVX-512 cores and non-AVX-512 cores, and only the former execute AVX-512 instructions so that the impact of potential frequency reductions is limited to those cores. To migrate threads to AVX-512 cores, we configure the non-AVX-512 cores to raise an exception when executing AVX-512 instructions. We use a heuristic to determine when to migrate threads back to non-AVX-512 cores. Our approach is able to reduce the frequency reduction overhead by 70% for an assortment of common benchmarks.

CCS CONCEPTS

• **Computer systems organization** → *Single instruction, multiple data*; *Multicore architectures*; • **Hardware** → *Platform power issues.*

KEYWORDS

AVX-512, core specialization, dim silicon

ACM Reference Format:
Mathias Gottschlag, Peter Brantsch, and Frank Bellosa. 2020. Automatic Core Specialization for AVX-512 Applications. In *The 13th ACM International Systems and Storage Conference (SYSTOR '20), June 2–4, 2020, Haifa, Israel.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3383669.3398282

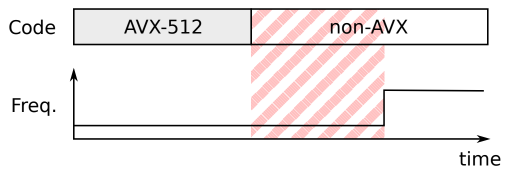
1 INTRODUCTION

The end of Dennard scaling, i.e. keeping the power density constant by scaling the threshold voltage, has lead to increasing power density and to a situation where the performance of processors is mainly limited by their power consumption [26]. The limited power budget, both in terms of the maximum heat conducted by the cooling system to maintain acceptable temperatures as well as in terms of the maximum current provided by the power supply, leads to situations where either parts of the chip have to be deactivated (*dark silicon* [26]) or the chip has to operate at a reduced frequency (*dim silicon* [13]).

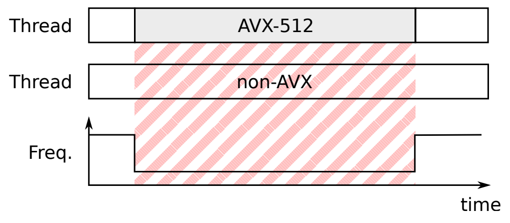
An example for the latter can be found in the variable turbo frequencies of current processors depending on the number of active cores – when fewer cores are active, the resulting power headroom can be utilized to increase the frequency and improve performance [22]. However, even within a single core, power consumption varies based on the type of instructions executed as complex instructions cause more switching activity on the chip. As a result, different maximum frequencies are possible for different instruction mixes.

The AVX2 and AVX-512 instructions found in current Intel CPUs operate on wide 256-bit and 512-bit vectors and therefore consume particularly large amounts of energy per operation. As a result, recent CPUs have started to reduce their operating frequency when executing such instructions [4]. This frequency reduction, while necessary for such power-intensive code, can negatively affect other code as well, for two reasons:

(1) *Restoring the previous higher frequency is delayed.* Each frequency change causes some overhead [21] as the



(a) AVX-512 and Non-AVX Execution Phases



(b) Hardware Multithreading

Figure 1: The frequency reduction caused by AVX-512 instructions can affect unrelated non-AVX code in two different ways: Restoring the non-AVX frequency is delayed (a) and frequency reductions affect other hyperthreads of the same core (b). The hatched region indicates non-AVX code slowed down by the AVX-512 frequency reduction. Some AVX2 instructions cause similar effects, albeit with lesser impact.

system has to wait for voltages to change and for the frequency to stabilize. Therefore, in order to prevent excessive numbers of frequency changes, current Intel CPUs delay restoring the clock speed even if no more AVX2 or AVX-512 instructions are executed [5]. As a result, in heterogeneous applications which frequently switch between power-intensive AVX2/AVX-512 execution phases and less power-intensive non-AVX phases, the latter can be temporarily slowed down as the frequency is still reduced due to the preceding power-intensive code (see Figure 1a). This can be observed, for example, in the nginx web server [15]: When the web server is used with an SSL library using AVX-512, the whole workload is slowed down by 10%. The frequent short sections of AVX-512-enabled code for cryptography cause reduced CPU frequencies for the rest of the web server.

(2) *The frequency reduction affects the sibling hyperthread.* If one hyperthread in an SMT system executes AVX-512 or AVX2 code, the core has to reduce its frequency. Other hyperthreads of the same core are equally affected by the frequency reduction, even if they are not executing AVX2 or AVX-512 instructions (see Figure 1b). For instance, an AVX-512-enabled deep-learning application has been shown to slow down other tasks on the same system by 10% [9].

Core scheduling – i.e., restricting the type of threads executing in parallel on the hyperthreads of the same core – has been suggested as a solution for the second scenario [19]. However, it causes significant performance degradation due to reduced CPU utilization in some cases [9] and does not solve the problem present in the first scenario.

In this paper we present a different kind of core specialization which is able to improve performance of workloads using AVX-512 instructions in both scenarios through improved scheduling of AVX-512-heavy code. Our prototype detects when a thread starts using AVX-512 instructions and then restricts the thread to a subset of the cores to limit frequency changes to those cores only. These cores prioritize AVX-512 code in order to reduce the performance impact on non-AVX-512 code. When the thread stops using AVX-512 instructions, the scheduling restrictions have to be removed again. To detect the end of AVX-512 phases, we employ a heuristic: As AVX-512 is commonly used for CPU-heavy computation phases with little I/O, we assume that the AVX-512 phase has finished by the time of the next system call, at which point we allow execution on non-AVX-512 cores again.

We describe an implementation of our design based on the Linux kernel and the MuQSS scheduler [14] and show that the prototype can significantly improve performance for different types of workloads. In a single-process web server workload, our prototype was able to reduce the slowdown caused by AVX-512 by 71%, and in multi-process batch workloads the prototype reduced the impact of a parallel AVX-512-heavy background process by 70%.

2 AVX-INDUCED FREQUENCY CHANGES

Intel CPUs reduce their frequency when executing AVX2 and AVX-512 instructions. To demonstrate the impact of the frequency changes on different types of workloads, we executed a number of benchmarks involving AVX2 or AVX-512 instructions and measured the CPU time required on a system with an Intel Xeon Gold 6130 processor using the Linux 4.18.19 kernel. Figure 2 shows the results of this experiment. Our workloads cover both scenarios described in Section 1. As an example for a single-process workload, we executed a web server scenario similar to the one described in [15]. We configured nginx to serve a static website via TLS using the ChaCha20-Poly1305 encryption scheme and configured the OpenSSL library to use either AVX-512, AVX2, or neither for encryption and decryption. To simulate the effect on non-AVX web application code, we let the web server compress the files on-the-fly with the Brotli compression algorithm.

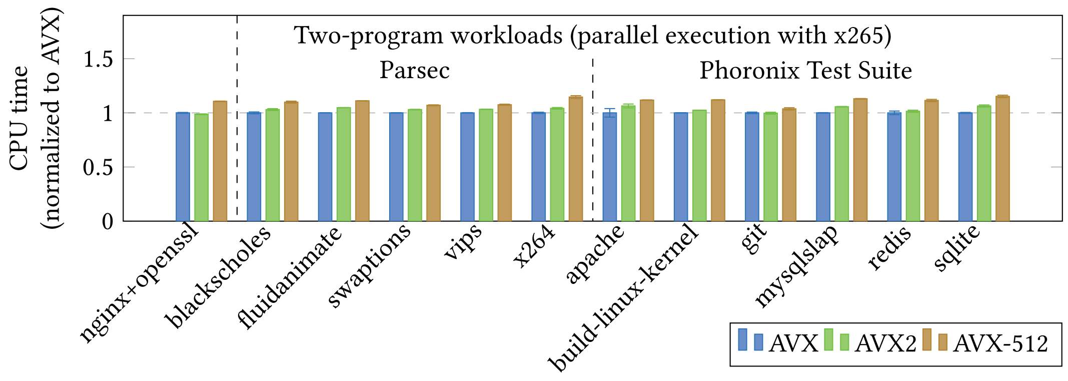


Figure 2: CPU time required to run several benchmarks either alone with support for different instruction sets (nginx+openssl) or in parallel with an instance of the x265 video encoder configured to use AVX, AVX2, or AVX-512. The CPU used for the experiments provides a maximum all-core turbo frequency of 1.9 GHz for AVX-512, 2.4 GHz for AVX2, and 2.8 GHz for other code (including AVX). In all experiments, the frequency reduction caused by complex SIMD instructions (AVX2 and AVX-512) slows down the rest of the workload.

Such heterogeneous single-process workloads are affected by the delay when restoring higher frequencies. We measured the CPU time required per HTTP request. As the processor reduced its frequency when executing the AVX-512 variant of OpenSSL’s cryptography routines, the benchmark was slowed down by 10.7%, even though less than 1% of the CPU time was spent in OpenSSL. As an example for multi-process workloads where a power-intensive workload on one hyperthread reduces performance for code running on the sibling hyperthread, we measured the performance of various benchmarks from the Parsec 3.0 [7] benchmark suite and Phoronix Test Suite v9.0.1 [2]. We executed these in parallel with an instance of the x265 video encoder configured to use either AVX-512 or AVX2 instructions or neither. x265 is an example for an application that profits from the usage of AVX-512 instructions [27]. As Linux frequently scheduled x265 on the same core as the benchmarks, the benchmarks were slowed down by 4.1% on average for AVX2 and by 11.3% for AVX-512. Note that our analysis in Section 4.3 rules out increased resource contention among sibling hyperthreads as the main reason for this slowdown, leaving only the reduced average frequency as an explanation.

Although Intel refers to the frequencies triggered by AVX2/AVX-512 instructions as “AVX 2.0 and AVX-512 frequencies” [4] – the latter being as much as 30% lower as the corresponding non-AVX frequency when all cores are active – the criteria for frequency selection are more complex than implied by this naming. In particular, instructions from the AVX-512 instruction set extension can operate on 256-bit registers with lower impact on the frequency. Intel [5] classifies 512-bit and 256-bit operations further as either *heavy instructions* (floating point operations and multiplications) and *light instructions*. The *AVX2 frequency* is selected when more than one heavy 256-bit instructions is executed per two cycles [18] or when light 512-bit instructions are executed. The *AVX-512 frequency* is selected when more than one heavy 512-bit instructions is executed per two cycles. Furthermore, combinations of light 512-bit instructions and sufficient numbers of heavy 256-bit instructions can also trigger a switch to the AVX-512 frequency. Finally, executing any SIMD or floating-point instruction causes some frequency reduction if the upper 256-bit half of the 512-bit registers contains valid contents [10].

This shows that register usage alone does not determine the resulting frequency when executing a specific piece of code. An example for AVX-512 code that does not cause a transition to the AVX-512 frequency is provided by older versions of the glibc library in the form of `memset()` and similar functions. These functions use 512-bit instructions for sufficiently large buffers but do not use any heavy AVX-512 instructions, so the processor only selects the AVX2 frequency. Nevertheless, the usage of 512-bit instructions is a good indicator for the expected frequency reduction for most applications. Our approach as described in the next sections therefore uses the detection of any AVX-512 instruction to guide its scheduling decisions.

For some programs which use AVX-512 registers but do not trigger transitions to the lowest frequency level, the approach described in the next sections might not yield satisfactory results, though. We discuss this issue and potential countermeasures in Section 5.

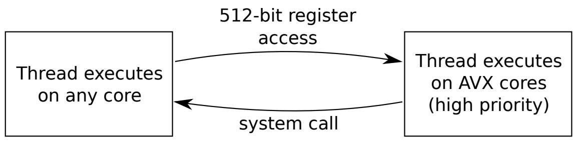

512-bit register access
Thread executes on any core
Thread executes on AVX cores (high priority)
system call

Figure 3: Our design detects usage of AVX-512 instructions and restricts the corresponding thread to a subset of the system's cores.

3 APPROACH

We present a design that is able to reduce the slowdown caused by AVX-512 instructions both in single-process as well as multi-process scenarios. Figure 3 shows the basic principle of our design, which divides cores into *AVX-512 cores* and *non-AVX-512 cores*. We employ *core specialization* to run AVX-512 code on the AVX-512 cores only, which limits frequency reductions to those cores. As the AVX-512 cores then mostly execute AVX-512 code, little non-AVX-512 code is affected by the frequency changes, which reduces the performance overhead caused by the AVX-512 instructions. Our approach consists of three main parts:

Initially, all threads are allowed to execute on all cores of the system. As soon as a thread executes AVX-512 instructions, though, the thread is marked as an *AVX-512 thread* and its execution is restricted to the AVX-512 cores. We describe the mechanism to detect execution of AVX-512 instructions before they are able to reduce the frequency of a non-AVX-512 core in Section 3.1.

The core restriction will eventually lead the scheduler to migrate the thread to an AVX-512 core. In Section 3.2, we describe the scheduler modifications necessary to ensure quick migration to the appropriate core as well as good utilization of all cores. In particular, the latter means that AVX-512 cores have to be allowed to execute non-AVX-512 code when they would otherwise be idle, but AVX-512 threads have to be prioritized to achieve good overall throughput.

For heterogeneous programs consisting of AVX-512 and non-AVX-512 execution phases, the system has to determine when the thread is unlikely to execute further AVX-512 instructions in the near future. As described in Section 3.3, we solve this problem with a heuristic and assume that the AVX-512 phase has finished when the application executes the next system call. In that situation, our design marks the thread as a *non-AVX-512 thread* again and removes the scheduling restrictions so that the thread can be executed on non-AVX-512 cores. As suggested by related work [20] we additionally implement a fixed-timeout policy for scenarios where this assumption is wrong.

3.1 Making AVX-512 Instructions Fault

To determine when to place a thread on an AVX-512 core, it is necessary to know whether and when the thread executes AVX-512 instructions. Current CPUs do not provide a method to receive a notification when the CPU is about to reduce its frequency[1]. Previous work on core scheduling for AVX-512 applications [19] instead uses the `arch_status` interface provided by Linux. This interface provides an estimate of the time since the last usage of 512-bit registers which is used for long-term categorization of threads as either AVX-512 threads or non-AVX-512 threads. This technique has a number of downsides, though. First, Linux only samples register usage during context switches, causing short sections of AVX-512 code to frequently go unnoticed if the application executes the `vzeroupper` instruction afterwards to clear the registers. Second, and more importantly, information about register usage is only available after AVX-512 instructions have been executed, at which point the frequency has already been reduced. For these reasons, the technique is not usable for workloads which frequently switch between AVX-512 and non-AVX-512 phases such as the web server example described in Section 1. In such scenarios, the operating system must be able to prevent execution of an instruction if it would otherwise cause a frequency reduction on a core that is supposed to execute at higher frequencies.

Our design adapts the trap-and-migrate technique described by Li et al. [20]. It configures the non-AVX-512 cores in a way that they raise an exception when executing AVX-512 instructions. The exception handler then marks the thread as an AVX-512 thread and migrates it to an AVX-512 core. To raise an exception for AVX-512 instructions, we leverage the fact that the CPU only allows execution of those instructions if the operating system provides context switch code for the corresponding registers. Modern x86 CPUs provide support for the `XSAVE` instruction which allows fine-grained control over CPU context saving. This feature can be configured with the `XCR0` register which describes which registers shall be saved to memory [3, ch. 13]. AVX-512 instructions may only be executed when the `opmask`, `ZMM_Hi256`, and `Hi16_ZMM` bits of that register are set, as those bits control context switching for the registers specific to AVX-512. We clear these bits on non-AVX-512 cores but let them remain set on AVX-512 cores, as only those cores are supposed to execute AVX-512 instructions without exceptions.

3.2 Scheduler Changes for Core Specialization

Once threads have been marked as AVX-512 threads, the threads must only be executed on AVX-512 cores. AVX-512

[1]See Section 5 for a discussion of such interfaces.

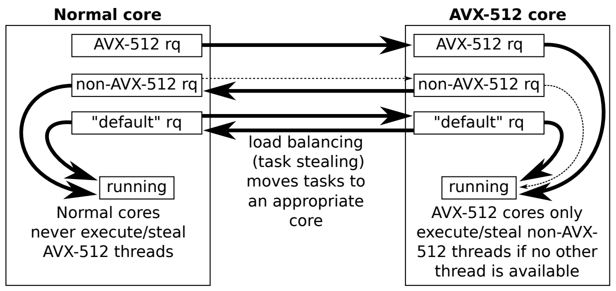

Normal core
AVX-512 core
AVX-512 rq
non-AVX-512 rq
"default" rq
running
Normal cores never execute/steal AVX-512 threads
load balancing (task stealing) moves tasks to an appropriate core
AVX-512 rq
non-AVX-512 rq
"default" rq
running
AVX-512 cores only execute/steal non-AVX-512 threads if no other thread is available

Figure 4: In our prototype, separate runqueues achieve static prioritization and efficient task stealing. The "default" thread type is introduced to prevent starvation of non-AVX-512 OS tasks pinned to AVX-512 cores.

cores should execute normal threads only if no AVX-512 threads are runnable to minimize the non-AVX-512 code affected by the frequency reduction of AVX-512 code. A central part of our approach therefore is a modification of the scheduler to implement this policy. In particular, we modified the MuQSS scheduler [14] for the Linux kernel to add support for different core types and hard prioritization of AVX-512 thread on top of the existing scheduling algorithm.

The MuQSS scheduler has two advantages over the default CFS scheduler: First, it provides slightly more efficient migration of threads between cores, which is relevant as core specialization can cause more thread migration compared to regular scheduling. The performance advantage per migration is only 10%, though, so it is likely that our design can be ported to other schedulers without significant performance implications. Second, MuQSS has lower complexity compared to CFS and makes modifications easier.

By default, MuQSS maintains one runqueue per physical core (i.e., per each pair of hyperthreads). To be able to efficiently implement different behavior for AVX-512 and normal (non-AVX-512) threads, we replicate the runqueues of the cores and introduce one runqueue per thread type and core as shown in Figure 4. The main scheduling routine is modified so that non-AVX-512 cores do not pick threads from the AVX-512 runqueue, whereas AVX-512 cores try to pick a thread from their local AVX-512 runqueue. If that runqueue is empty, AVX-512 cores try to pick a thread from another core's AVX-512 runqueue, potentially migrating AVX-512 threads from non-AVX-512 cores to the AVX-512 core in the process. Only if no AVX-512 thread is available, AVX-512 cores will pick a normal thread to maximize CPU utilization, even if that thread is then affected by reduced frequencies.

As AVX-512 threads are prioritized on AVX-512 cores, the execution of normal threads is preempted if an AVX-512 thread becomes runnable. If a thread becomes an AVX-512 thread while executing on a non-AVX-512 core, the scheduler sends an inter-processor interrupt (IPI) to a random AVX-512 core that is currently not executing AVX-512 threads to force a scheduler invocation and a subsequent migration of the thread. If a thread becomes an AVX-512 thread while all AVX-512 cores are busy executing AVX-512 threads, no IPI is required – instead, as soon as one of the cores runs out of local AVX-512 threads, it will inspect the runqueues of all non-AVX-512 cores as described above and will migrate one AVX-512 thread if possible.

To improve cache locality, threads are sometimes limited to specific cores. Such restriction of the CPU affinity of the threads can happen via manual invocation of the `taskset` tool or can be automatically configured by applications or the kernel and requires special treatment in our scheduler. We identified two main corner cases:

(1) *Threads which do not use AVX-512 can be restricted to AVX-512 cores.* If a non-AVX-512 thread is restricted to AVX-512 cores, it can be starved as it has a lower priority than any AVX-512 thread. We observed this condition mainly for operating system tasks and never for processes started by the user. Therefore, we introduce a mechanism to limit core specialization to processes outside of the operating system.
We introduce a third "default" thread type which is used for all threads which did not inherit a non-default thread type from their parent and have themselves not yet been classified. Classification happens either by manual specification of the thread type via a system call or via execution of AVX-512 instructions. We introduce a third runqueue on each core for these default threads. As shown in Figure 4, the threads are executed on both AVX-512 and non-AVX-512 cores, with a priority equal to normal threads on the former and AVX-512 threads on the latter to prevent starvation.
(2) *Conversely, threads which execute AVX-512 instructions can be restricted to non-AVX-512 cores.* If a thread is restricted to non-AVX-512 cores but executes an AVX-512 instruction, the policy described above starves the thread as the thread is not able to migrate to an AVX-512 core. We only observed this case if the user actively configured the CPU affinity of threads. Our prototype detects this special case by intersecting the CPU affinity mask with the mask of AVX-512 cores and in response temporarily allows execution of AVX-512 instructions on the non-AVX-512 core while the thread is running.

3.3 Detecting Non-AVX-512 Code

Scheduling of AVX-512 threads is restricted in our design. The threads of applications consisting of AVX-512 phases and non-AVX-512 phases will eventually stop executing AVX-512 code and enter a non-AVX-512 phase. In this case, the scheduling restrictions should be removed and the thread should be migrated to a non-AVX-512 core. It is, however, not as easy to detect the absence of AVX-512 instructions than it is to detect their attempted execution, as the CPU does not provide any method to receive a notification when no instruction of a certain type is executed. Therefore, our prototype has to depend on heuristics to estimate the length of AVX-512 phases instead.

The main observation leading to the heuristic used in our design is that AVX-512 is predominantly used to speed up compute-heavy execution phases. Those phases usually perform no I/O and few other system calls[2]. Therefore, system calls are a good indicator that such an AVX-512 phase is over. On each kernel entry on an AVX-512 core, our prototype marks the current thread as a normal thread, removes the scheduling restrictions, and invokes the scheduler to schedule any (prioritized) AVX-512 thread instead. Eventually, a non-AVX-512 core will then pick up the previous thread, which will thereby automatically be migrated away from the AVX-512 core.

As a fall-back for workloads that do not execute any system calls over long periods of time, we suggest periodically resetting the thread type to normal after a fixed timeout, similar to existing approaches suggested by related work [20]. If, however, the thread still executes AVX-512 instructions at the time of the timeout, changing the thread type will result in two migrations in quick succession of the thread to a non-AVX-512 core and back to an AVX-512 core. As thread migrations cause overhead, the timeout therefore needs to be long enough to provide a sensible upper bound on the rate of migrations and the resulting overhead. While we implemented a timeout of 2 ms in our prototype, we found that the system-call-based policy is sufficient for the benchmarks in our evaluation.

3.4 Valid 512-bit Register State

In our prototype, we slightly modify the policy described in the last section to handle one special case: As the context switch code of Linux does not preserve 512-bit register state if context switch support is disabled as described in Section 3.1, migrating a thread with valid 512-bit register content to a non-AVX-512 core would cause the content to be corrupted. It would be possible to modify Linux to preserve the state. However, to reduce implementation complexity, we instead leave the type of the thread unchanged and prevent migration if our prototype detects valid 512-bit register

[2]In a setup with 28 threads on a 28-core system, each thread of the x265 video encoder used in this paper executes one futex system call on average every 5 milliseconds.

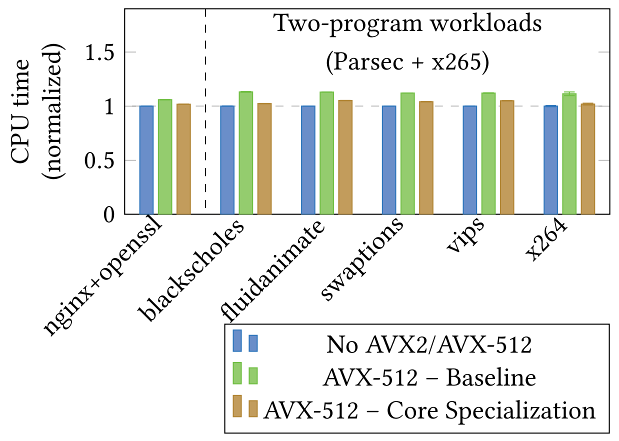


Figure 5: CPU time required to run various benchmarks. The Parsec benchmarks were executed in parallel with an instance of the x265 video encoder which was configured to use either AVX-512 instructions or neither AVX2 nor AVX-512. Our prototype noticeably reduces the influence of this background process on the benchmark.

contents. In practice, this limitation is of little relevance: Most software uses the `vzeroupper` instruction directly after each AVX-512 phase to clear the register state, as simply having valid 512-bit register state is enough to make many non-AVX-512 instructions trigger frequency changes [10].

4 EVALUATION

To demonstrate the potential of our prototype in terms of performance improvement, we evaluated the prototype on a system with an Intel Xeon Gold 6130 CPU and 24 GB DDR4-2666 RAM. We tested our prototype with the single-process scenario from Section 2 consisting of the nginx web server and OpenSSL as an example for an interactive single-process workload, and we tested the two-process scenarios with x265 and the Parsec benchmarks as examples for non-interactive two-process workloads. For these benchmarks, we measured the performance with our prototype and compared it to results using the unmodified MuQSS scheduler, both using the Linux 4.17 kernel. For our experiments, we disabled kernel page-table isolation as we expect future systems to implement more efficient hardware mitigations against the corresponding attacks.

As the main performance metric, we selected the overall CPU time required to execute the benchmark. Our prototype modifies the scheduling policy and might in some cases affect the number of cores used by the benchmark. As overall system throughput and per-core performance improvement is the main goal of our approach, the CPU time is a better indication for performance than the wall-clock time required. The results of our performance measurements are shown in Figure 5. We describe and discuss the results in Sections 4.1 and 4.2.

Besides raising the average CPU frequency, core specialization and the required thread migrations have a number of other impacts on performance. Migrations of threads between cores can introduce overhead, but core specialization can also improve cache effectiveness as has been shown in other research [17, 25]. Therefore, we also perform a performance analysis based on performance counters to show that the performance improvement is mainly caused by the improvement of the average CPU frequency. This experiment is described in Section 4.3.

4.1 Single-Program Workload

As an example for a heterogeneous single-program workload with AVX-512-heavy execution phases and non-AVX phases, we replicate the nginx web server example used in related work to demonstrate the problem of AVX-512 frequency reduction [15]. We let the nginx web server serve a static file with on-the-fly compression using the brotli compression algorithm to simulate some page generation overhead, and we let the web server serve the file over HTTPS provided by the OpenSSL library. We let the web server use the ChaCha20-Poly1305 encryption algorithm and select either the AVX-512 implementation of the algorithm or disable both AVX-512 and AVX2. In our prototype, three cores were designated as AVX-512 cores. As shown in Figure 5, with the unmodified MuQSS scheduler AVX-512 reduces the throughput by the web server by 6.0%. In our prototype, however, performance with AVX-512 OpenSSL matches the non-AVX-512 implementation more closely, with only 1.7% slowdown. The impact of AVX-512 on the benchmark's performance was therefore reduced by 71%.

4.2 Two-Program Workloads

As examples for heterogeneous two-program workloads consisting of an AVX-512-enabled program and a non-AVX-512 program, we execute several benchmarks from the Parsec 3.0 benchmark suite [7] alongside an instance of the x265 video encoder configured to use either AVX-512 or AVX. In all cases, x265 was configured to use 32 threads. Such a large number of threads allows the CPU-intensive background task to exploit phases where I/O bound tasks leave cores idle. As x265 by default reduces its own priority, we modified the application to run all threads at default priority. Our prototype was configured to use half of the system's cores as AVX-512 cores. As Figure 5 shows, the Parsec benchmarks are slowed down if x265 uses AVX-512. If core specialization is used, the slowdown shrinks from 12.3% down to 3.7%, meaning that the impact of the frequency reduction was reduced by 70%. Note that the overhead determined in this experiment is different to the values shown in Section 2 due to the different scheduler (MuQSS vs. CFS) and the resulting differences in co-scheduling.

The results show that in all cases some slowdown remains. We expect that the reason for this slowdown is that x265 often cannot not fully utilize its cores, as data dependencies limit the exploitable parallelism. In this case, the benchmark might be scheduled in parallel on the same core as x265. Performance counter analysis of Parsec vips showed that our prototype reduces the number of cycles spent at "AVX-512 frequencies" by the Parsec vips benchmark by 70%, but 30% remain despite core specialization, which supports the hypothesis that co-scheduling of the two processes still takes place. While this effect seems like a significant disadvantage compared to "hard" core scheduling which never schedules AVX-512 tasks alongside non-AVX-512 tasks on the same core, it actually is not: Any approach which never co-schedules AVX-512 threads and non-AVX-512 threads on the same cores will inevitably cause hyperthreads to idle, whereas our approach fully utilizes all available hyperthreads and therefore improves the overall throughput of the processor.

We repeated the two-program experiments with most of the PTS benchmarks used in Section 2[3]. PTS build-linux-kernel showed a reduction of the AVX-512-induced slowdown from 9.4% down to 1.6%, i.e., the impact of the frequency reduction was reduced by 83%. For the other PTS benchmarks, our setup – equal numbers of AVX-512 and non-AVX-512 cores – produced excessive speedup as the system started to increase its frequencies above the all-core turbo frequency, which eclipsed any performance improvement due to reduced AVX-512 overhead. The reason for this behavior was that the benchmarks were not able to fully utilize the non-AVX-512 cores either due to their I/O-heavy nature or their low number of threads. See Section 5 for a discussion of the ideal number of AVX-512 cores. We plan to rerun the benchmarks with optimized allocation of AVX-512 cores as part of future work.

4.3 Microarchitectural Analysis

The goal of our prototype is to show that core specialization can reduce the impact of AVX-512 frequency reduction on the performance of non-AVX-512 code. However, while our experiments show significant performance improvements, they do not prove that increased average CPU frequency is the reason for the results. In particular, in the past core specialization has been shown to affect cache effectiveness as each core only has to hold parts of the working set and the contents of

[3]We excluded mysqlslap due to its excessive runtime.

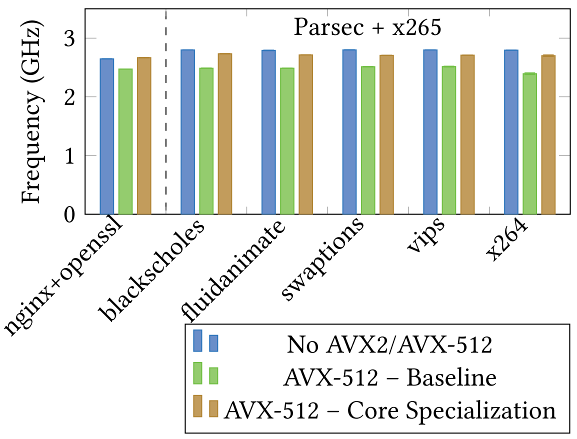


Figure 6: CPU frequency for some of the benchmarks shown in Figure 5. Core specialization noticeably reduces the frequency impact of the AVX-512 code.

different core's caches complement each other [17, 25]. To determine the reason for the performance changes caused by our prototype, we therefore first measure the average CPU frequency experienced by the benchmark and then perform a performance counter analysis.

The results of the frequency measurements are shown in Figure 6. Whereas AVX-512 reduces the average frequency for the benchmarks down to 2.47 GHz from 2.77 GHz, our approach causes a significant frequency improvement again, yielding an average frequency of 2.70 GHz across the benchmarks. The frequencies are closely correlated to the respective performance. For example, the nginx benchmark shows a rather low frequency difference as the web server often triggers AVX2 frequencies even if OpenSSL is not configured to use AVX2 or AVX-512, which corresponds to the comparably low performance difference described in Section 4.1.

We conduct an analysis of the instructions per cycle (IPC) to determine the impact of other factors such as changed cache miss rates. For the AVX-512 nginx benchmark, the IPC difference is minor. Whereas the web server executes 1.043 instructions per cycle with an unmodified MuQSS scheduler, our prototype increases the IPC by 1.6%, resulting in a throughput of 1.06 instructions per cycle. This IPC change is comparably small which supports the assumption that the frequency change is the main reason for the overall performance improvement. For the parsec benchmarks, the IPC improves by 2.8% on average – more than for nginx, but still not enough to explain the significantly larger performance difference. Furthermore, the different Parsec benchmarks showed varying behavior and the IPC results showed little correlation to performance, with fluidanimate and vips even suffering from reduced IPC in our prototype.

To determine the reason for the nginx IPC difference, we conduct a top-down performance counter analysis [28] which provides a method to attribute the IPC to different parts of the microarchitecture. The analysis shows that nginx is more often backend-bound in our prototype but experiences fewer stall cycles due to bad speculation. The former is likely a sign of increased data access stalls due to cache line bouncing between different cores, whereas the latter is a sign of improved instruction cache effectiveness due to the code being partitioned into different private caches. As the small IPC difference shows, however, the two effects cancel each other out and the fraction of time spent retiring instructions is very similar.

For the parsec benchmarks, a top-down performance counter analysis is not viable: On a system with hyperthreading the IPC experienced by one hyperthread always depends on the code executed on the other hyperthread, so different scheduling of two processes significantly distorts the IPC measurements for one of the processes. Therefore, we use perf [1] to directly compare the branch misprediction and cache miss ratio of the Parsec benchmarks. The branch misprediction ratio correlates with the improved IPC, as all benchmarks show slightly fewer mispredicted branches. The cache miss ratio, instead, does not appear to correlate with IPC: Whereas blackscholes and fluidanimate show a slightly worse cache miss ratio, the other benchmarks show improved cache hit rates. In particular, vips experiences 37% fewer cache misses per cache reference despite having lower IPC in our prototype.

5 DISCUSSION

The AVX-512 instruction set extension often enables significant speedup – for example, the x265 video encoder used in this paper gains almost 20% throughput for some configurations [27]. However, the frequency reductions caused by the AVX-512 code often slow down other non-AVX-512 code. Our evaluation showed that our approach is able to mitigate most of that slowdown for a range of benchmarks. However, our prototype still has a number of limitations, some of which can be easily resolved and some of which are caused by limitations of the underlying hardware. We will discuss those limitations in the following and sketch out potential solutions.

Detection of AVX-512 support: At startup, most software using AVX-512 checks for available instruction sets via the `CPUID` instruction. The return values of this instruction, however, are not only affected by the type of processor but also by the operating system support for specific instruction sets. In our case, when our prototype disables support for AVX-512 instructions on the non-AVX-512 cores, the corresponding bits returned by CPUID are affected as well. Therefore, depending on the initial core of an application, the application might either detect support for AVX-512 or not and will subsequently use only the detected instruction sets. Due to this nondeterminism, we simply modified the workloads to always use AVX-512 if requested, independent from the result of the CPU feature detection logic of x265.

A generic fix for this problem is available as well, however: In virtual machines, the CPUID instruction can be intercepted to present a specific CPU feature set to virtual machines. Techniques such as Dune [6] could be used to intercept CPUID instructions executed by regular processes as well and could be used to present a uniform CPU feature set independent from the core on which the application is running.

Imprecise detection of energy-intensive code: Our current prototype only isolates threads based on whether the thread executes AVX-512 instructions or not. Current Intel CPUs, however, provide three distinct frequency levels, so a solution with three types of cores would be required for optimal packing of code running at AVX-512, AVX2, or normal frequencies. Such a solution requires a different mechanism to categorize code, though, as the current mechanism only detects execution of AVX-512 instructions.

Not all AVX-512 instructions cause a transition to the same frequency level, though. For example, glibc provides implementations of memset() and similar functions using AVX-512 instructions that do not cause a transition to the lowest frequency level, and neither do very short sections of energy-intensive 512-bit FMA operations. Similarly, the mechanism is not able to detect AVX2 instructions which cause a frequency reduction. On current hardware, however, we were unable to identify a better mechanism to detect energy-intensive code and to intercept it before any frequency reduction takes place.

We suggest an improved hardware-software interface which allows the operating system to make decisions based on power consumption instead of instruction set usage. This interface would provide power consumption exceptions which are triggered when code requiring a specific operating frequency – e.g., a sufficiently dense sequence of 512-bit FMA instructions – is executed. This exception has to be triggered before the frequency is reduced, to allow the operating system to intervene and migrate the thread to a different core based on the frequency requirements. Similarly, the interface should provide a notification mechanism when less power-intensive code is executed. The mechanism should be configurable by the operating system. For example, the CPU could trigger an interrupt when only less power-intensive code was executed for a software-specified duration. This notification could replace the heuristics in Section 3.3 for more accurate decisions about when to migrate threads to cores with a higher frequency.

Number of AVX-512 cores: Our prototype in its current form requires the number of AVX-512 cores to be set manually. Ideally, the ratio between the number of AVX-512 and the number of non-AVX-512 cores matches the ratio between the AVX-512 and non-AVX-512 parts of the workload. If the number of AVX-512 cores is too low, some of the non-AVX-512 cores might remain idle. If the number is too high, isolation between AVX-512 and non-AVX-512 code is reduced as non-AVX-512 code is often executed by (otherwise idle) AVX-512 cores. In our experiments, we observed the former for some of the PTS benchmarks, where idle cores caused the system to increase its frequency above the all-core turbo frequency. For following prototypes we will dynamically determine the optimal number of AVX-512 cores by monitoring the CPU load caused by AVX-512 threads on the AVX-512 cores.

NUMA support: We did not evaluate our prototype on a NUMA system with multiple CPUs. While the underlying MuQSS scheduler tries to minimize migration of threads across NUMA domains, our prototype increases the overall thread migration rate and will therefore likely suffer from increased overhead on NUMA systems. We assume that on NUMA systems each NUMA node has to have its own set of AVX-512 cores and thread migration has to be restricted to migration within the NUMA node to achieve sufficient performance. Further research has to be conducted to measure the resulting overhead and the effect of such restrictions.

6 RELATED WORK

In this paper, we described a system which migrates threads to different sets of cores based on their usage of AVX-512 instructions to restrict the frequency impact of the instructions to a subset of the cores. In this section, we summarize the existing work on AVX-512 frequency effects and summarize the work on operating systems for asymmetric multiprocessors which builds the foundation for parts of our design. We also describe other existing solutions for similar problems and describe the differences.

6.1 Performance Impact of AVX-512

While the basic behavior when executing AVX-512 instructions is documented by Intel [5, p. 2-13], the documentation lacks detail and often deviates from the actual behavior observed during experiments on the hardware. Several reverse engineering efforts have tried to fill the gaps. For example, Schöne et al. [24] show that the delay when increasing the frequency is lower than suggested by the documentation and Travis Downs [11] provides a detailed analysis of the performance during frequency changes caused by AVX-512. The latter experiments also show that the CPU increases the voltage during the frequency transition when executing AVX-512 instructions, which shows that short-term voltage fluctuations caused by power-intensive AVX-512 instructions might be more problematic than violation of thermal limits caused by the increased power consumption.

For the first time, experiments at Cloudflare [15] showed that the frequency impact of AVX-512 instructions poses a problem for real-world workloads as AVX-512 was identified as the reason for a 10% slowdown for certain OpenSSL configurations. We use a single-process workload derived from this experiment in our evaluation.

Aubrey Li [19] subsequently first documented the effects on multi-process workloads and showed that an AVX-512-enabled deep-learning workload can significantly slow down the performance of a concurrently executing non-AVX-512 process. As a countermeasure, he proposed a variant of core scheduling – i.e., restricting co-scheduling of different processes on the same core in parallel [8] – to prevent threads using AVX-512 from being executed in parallel to non-AVX-512 threads. While we also modify the scheduler to reduce the impact of frequency changes due to AVX-512, our approach has a two main differences. First, core scheduling as suggested by Aubrey Li depends on long-term categorization of processes through analysis of their register set usage, whereas the trap-and-migrate mechanism used by our design can detect short-term AVX-512 usage and makes our approach usable for single-program workloads such as the web server scenario described above. Second, core scheduling prevents harmful co-scheduling by potentially causing hyperthreads to idle if no suitable thread is runnable. Idle hyperthreads, however, ultimately waste CPU performance. Our approach tries to separate AVX-512 and non-AVX-512 code whenever possible, but allows co-scheduling when necessary to fully utilize all available hyperthreads and to maximize overall throughput.

6.2 OS Support for Asymmetric Multiprocessors

One of the main techniques used by our approach is to trap and migrate threads when executing AVX-512 instructions. A very similar technique has been described by Li et al. [20] who disable the floating-point unit (FPU) on some cores of a multi-core system to simulate an asymmetric multiprocessor where only some cores have support for floating-point operations. On this system, executing floating-point instructions on a core with a disabled FPU causes the operating system to migrate the thread to a suitable core. We employ a similar technique to trigger exceptions when executing AVX-512 instructions, although we only disable parts of the register set so that other SIMD instructions can still be executed. As a heuristic to migrate threads back to a core without an FPU, Li et al. suggest a fixed timeout. We, instead, use system calls as a heuristic to detect execution phase changes.

6.3 Profiling-Based Approaches

Core specialization in its different forms is not the only conceivable method to solve the problem of frequency reductions caused by power-intensive instructions. Kumar et al. [16] describe a profiling-based approach for the single-process scenario where applications only use SIMD instructions in select parts of the program. The approach tries to improve the efficiency of power-gating by identifying the parts of the program where short stretches of SIMD instructions cause little overall performance improvement and uses the resulting information to "devectorize" the code to allow power-gating of the SIMD unit. Such an approach could be used to reduce the frequency reduction caused by AVX-512. However, the approach depends on a just-in-time compiler for devectorization. The approach described in this paper works within the existing software ecosystem and requires little changes to the operating system.

As an alternative, profiling information could be used to steer the processor's frequency selection policy to reduce the impact on non-AVX-512 code. For example, if it is known that the thread is not going to use AVX-512 instructions in the near future, the frequency can be immediately increased, thereby reducing the impact on the following non-AVX-512 code. Roy et al. [23] describe a technique which uses profiling information to let the software manually control power gating. Although changes would be required to the hardware, simulations showed that a similar approach could be employed to improve the efficiency of AVX-512 frequency management [12].

7 CONCLUSION

Intel CPUs reduce their frequency when executing AVX2 and AVX-512 instructions to limit power consumption. The frequency reduction, however, also affects other code when that code is executed on sibling hyperthreads in parallel or when the code directly follows the AVX2/AVX-512 code. We demonstrate that core specialization can mitigate most of the slowdown in both cases. We describe a core specialization approach which limits execution of AVX-512 instructions to a subset of the cores and makes the instructions raise exceptions when executed on other cores to trigger the appropriate thread migration. Our prototype is able to reduce the slowdown caused by AVX-512 by 70% on average for a range of benchmarks. While the design is therefore usable on current CPUs, we describe hardware changes which could improve its effectiveness.

REFERENCES

[1] [n.d.]. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page.

[2] [n.d.]. Phoronix Test Suite. https://phoronix-test-suite.com/.

[3] 2018. *Intel® 64 and IA-32 Architectures Software Developer's Manual - Volume 1: Basic Architecture.*

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

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

[6] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe User-level Access to Privileged CPU Features. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12).* 335–348.

[7] Christian Bienia. 2011. *Benchmarking Modern Multiprocessors.* Ph.D. Dissertation. Princeton University.

[8] Jonathan Corbet. 2019. Core scheduling. https://lwn.net/Articles/780703/

[9] Jonathan Corbet. 2019. Many uses for Core scheduling. https://lwn.net/Articles/799454/

[10] Travis Downs. 2018. Dirty upper 256 causes everything to run at AVX-512 frequencies. https://www.realworldtech.com/forum/?threadid=179700&curpostid=179700

[11] Travis Downs. 2020. Gathering Intel on Intel AVX-512 Transitions. https://travisdowns.github.io/blog/2020/01/17/avxfreq1.html

[12] Mathias Gottschlag, Yussuf Khalil, and Frank Bellosa. 2020. Dim Silicon and the Case for Improved DVFS Policies. *arXiv preprint arXiv:2005.01498* (2020).

[13] Wei Huang, Karthick Rajamani, Mircea R Stan, and Kevin Skadron. 2011. Scaling with design constraints: Predicting the future of big chips. *IEEE Micro* 31, 4 (2011), 16–29.

[14] Con Kolivas. 2016. MuQSS - The Multiple Queue Skiplist Scheduler v0.105. http://ck-hack.blogspot.com/2016/10/muqss-multiple-queue-skiplist-scheduler.html

[15] Vlad Krasnov. 2017. On the dangers of Intel's frequency scaling. https://blog.cloudflare.com/on-the-dangers-of-intels-frequency-scaling/.

[16] Rakesh Kumar, Alejandro Martinez, and Antonio Gonzalez. 2014. 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 (2014), 25.

[17] Min Lee and Karsten Schwan. 2012. Region scheduling: efficiently using the cache architectures via page-level affinity. In *ACM SIGARCH Computer Architecture News*, Vol. 40. ACM, 451–462.

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

[19] Aubrey Li. 2019. Core scheduling: prevent fast instructions from slowing you down. (Sept. 9 2019). https://linuxplumbersconf.org/event/4/contributions/430/ Linux Plumbers Conference.

[20] Tong Li, Paul Brett, Rob Knauerhase, David Koufaty, Dheeraj Reddy, and Scott Hahn. 2010. Operating system support for overlapping-ISA heterogeneous multi-core architectures. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on.* IEEE, 1–12.

[21] Sangyoung Park, Jaehyun Park, Donghwa Shin, Yanzhi Wang, Qing Xie, Massoud Pedram, and Naehyuck Chang. 2013. 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 (2013), 695–708.

[22] Efraim Rotem, Alon Naveh, Avinash Ananthakrishnan, Eliezer Weissmann, and Doron Rajwan. 2012. Power-management architecture of the intel microarchitecture code-named sandy bridge. *Ieee micro* 32, 2 (2012), 20–27.

[23] Soumyaroop Roy, Nagarajan Ranganathan, and Srinivas Katkoori. 2009. A framework for power-gating functional units in embedded microprocessors. *IEEE transactions on very large scale integration (VLSI) systems* 17, 11 (2009), 1640–1649.

[24] Robert Schöne, Thomas Ilsche, Mario Bielert, Andreas Gocht, and Daniel Hackenberg. 2019. Energy Efficiency Features of the Intel Skylake-SP Processor and Their Impact on Performance. *arXiv preprint arXiv:1905.12468* (2019).

[25] Livio Soares and Michael Stumm. 2010. FlexSC: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation.* USENIX Association, 33–46.

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

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

[28] Ahmad Yasin. 2014. A top-down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS).* IEEE, 35–44.