Reducing AVX-Induced Frequency Variation With Core Specialization

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

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

ABSTRACT

Recent Intel server processors temporarily reduce their frequency when many AVX2 or AVX-512 SIMD instructions are executed. The frequency change is only reverted two milliseconds after the system has stopped executing such instructions. Before this time, any non-vectorized (and potentially unrelated) code which could execute at higher frequencies is slowed down. The effect on overall performance depends on the specific workload and is hard to predict. We describe a scenario where vectorizing one component with AVX-512 instructions improves performance by 10% for one workload and reduces performance by 10% for another workload.

If only some of the cores of a system execute such vectorized code, the frequency effect is limited to those cores. We propose a scheduling algorithm as well as a mechanism to intercept problematic code sections so that threads executing vectorized code are transparently migrated to a small subset of the cores. While our work is still in progress, we describe a partial implementation which is able to reduce the negative performance impact of AVX2 and AVX-512 instructions by over 70%.

1 INTRODUCTION

The end of Dennard Scaling – i.e., constant power density scaling of transistors [4] – has lead to the current situation where the performance of CPUs is mainly limited by their power consumption. As a result, the maximum attainable frequency depends on the amount of silicon area which is actively used [18]. Different instructions cause different amounts of switching activity and therefore consume different amounts of energy [15], so the maximum frequency depends on the executed instruction mix. Selecting a single conservative operating frequency that results in acceptable power consumption for all different instruction mixes results in wasted performance for instruction mixes with low power consumption.

Whereas historically this effect did not have any significant impact on CPU performance, the introduction of wider and more complex SIMD instructions has resulted in larger variation of power and current requirements. Therefore, recent Intel CPUs optimize their operating frequency depending on the use of AVX2 and AVX-512 vector instructions [3]. For example, the Intel Xeon Silver 4116 CPU provides a base frequency of 1.1 GHz for a dense mix of AVX-512 floating point or fused-multiply-add instructions and a base frequency of 2.1 GHz for scalar (non-vectorized) code. As these lower frequencies are required not only due to thermal limits but also for system stability in the face of increased current requirements [5], the frequency is reduced instantly when such an instruction mix is detected.

SFMA'19, March 25, Dresden, Germany
2019.

Frequency changes cause overhead [14], though, which limits the granularity of instruction-mix-based frequency selection. Once a high rate of AVX instructions has forced a frequency change, the core waits at least approximately two milliseconds before reverting to the previous frequency [1], most likely to limit the overhead of frequent changes. During these two milliseconds, the CPU executes code which could have been executed at a higher frequency. Therefore, infrequent phases of vectorized code can reduce overall performance, even if the vectorization locally improves performance.

This negative effect can be observed for real-world applications. For example, OpenSSL's implementation of the ChaCha20-Poly1305 encryption algorithm provides encryption speeds of up to 2.89 GB/s, whereas BoringSSL only achieves 1.6 GB/s [9]. One reason for this performance difference is that the former uses AVX-512 instructions with 512-bit wide vector operations whereas the latter only uses AVX2 instructions with 256-bit wide operations. In a scenario where an NGINX web server serves dynamic web sites with on-the-fly compression via HTTPS using these two libraries, the performance advantage of AVX-512 turns into an overall disadvantage, though [9]. The system using OpenSSL performs 10% slower than the system using BoringSSL.

This example shows that the impact of vectorization depends on the ratio between time spent in AVX-heavy code and time spent in other code, as the former benefits from vectorization whereas the latter suffers from reduced frequencies. We repeated the web server experiments and indeed found that with just slight configuration changes the 10% disadvantage of AVX-512 turns into a 10% advantage, which demonstrates a high dependency on the specific workload. Sections of AVX-heavy code therefore can have – but do not always have – significant negative impact on completely unrelated code, which is problematic for a number of reasons:

First, these effects break the assumption that local optimizations only have local effects, which is especially problematic during the development of software libraries. As library developers do not know all situations in which the library is going to be used, they usually resort to microbenchmarks. Potential performance problems only show up at the end user's system, where library updates can potentially violate real-time requirements and reduce service quality, even if the updates promise performance improvements. In particular, the frequency effects make it difficult to develop one version of the library which works equally well for all potential workloads.

Second, the frequency changes also affect other processes running on the same core. Therefore, short sections of AVX-heavy code are not only a threat to scheduling fairness, as one process can reduce peak CPU performance for another process with little effort. They also allow the construction of a covert channel to transmit information between otherwise isolated processes, as one process can detect whether another has executed AVX instructions by monitoring the operating frequency.

Previous work has hinted that placing AVX-heavy code on separate cores (*core specialization*) might be a viable method to reduce its negative impact on overall system performance [19][12]. In this paper, we describe an implementation of such core specialization. Our contributions are as follows:

- We describe an implementation of core specialization which can reduce the frequency impact of AVX code (Sections 3 and 3.1). Our design restricts code regions accessing large vector registers to a subset of the cores. Threads are automatically migrated to those cores by a fault-and-migrate mechanism whenever they execute wide vector instructions. We describe a method to make those instructions fault on cores which are not supposed to execute vectorized code.
- We propose a scheduling policy for core specialization and we show how an existing Linux scheduler can be modified to implement the policy (Section 3.2).
- We evaluate whether core specialization can be used to reduce the performance variability caused by AVX (Section 4). Although we have not yet implemented the whole design, we are able to show that the scheduler implementation has sufficiently low overhead to be a viable mechanism for core specialization. Our prototype is able to reduce frequency variation as well as performance variation of a web server setup similar to the one above by over 70%.
- Based on our evaluation, we describe how improved software-hardware interfaces can help the operating system to reduce the performance impact of instruction mixes with high power consumption.

2 FREQUENCY IMPACT OF AVX INSTRUCTIONS

Recent Intel server processors provide three sets of turbo and base frequencies (named *frequency level 0-2* or *non-AVX, AVX2 and AVX-512 frequencies*) depending on the executed instruction mix [1]. The lowest set of frequencies (frequency level 2) is used for code which frequently executes AVX-512 floating point instructions or AVX-512 integer multiplications. The intermediate set of frequencies is used for code which predominantly executes AVX-2 floating point instructions, AVX-2 integer multiplications, or other types of AVX-512 instructions. The highest set of frequencies is used for all other code which does not make heavy use of these instructions.

On recent Intel processors, each core can select an individual frequency level [1]. Once a core recognizes an instruction mix which requires reduced frequencies, it immediately reduces performance to ensure system stability. The core then requests a new voltage and frequency level from the central *power control unit* (PCU), which requires up to 0.5 ms to determine the cores' maximum frequencies based on the number cores at the same frequency level. Once the request has been processed, all affected cores reduce their frequency accordingly.

The reduced frequency itself would not be particularly harmful if the core would immediately revert to the previous frequency when execution of AVX instructions ends. This is not the case, though, as every frequency change causes overhead [14]. The core applies an additional 2 ms delay before it switches back to a higher frequency level [1], most likely to prevent excessive numbers of frequency changes. During this time, all code executes at the reduced frequency even if higher frequencies would be possible.

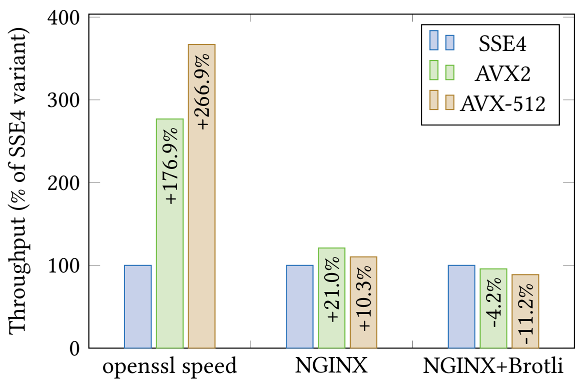


Figure 1: Different workloads show different sensitivity to the frequency reduction caused by wide SIMD instruction sets (normalized to SSE4 performance).

Whether this frequency reduction has significant effects on performance depends on the workload. Figure 1 shows the impact on different workloads using the ChaCha20-Poly1305 encryption functionality from the OpenSSL library. For each workload, the OpenSSL library was compiled with support for AVX2, for AVX-512, and without support for either. A simple OpenSSL microbenchmark shows large speed increases by both instruction sets, which is expected as the increased parallelism makes up for the reduced frequency.

Workloads which spend the majority of time in non-AVX code, however, execute larger amounts of non-AVX code at reduced frequencies. For example, an NGINX web server serving uncompressed static files via TLS using OpenSSL receives a 20% performance boost from AVX2 OpenSSL, whereas AVX-512 encryption only yields 10% faster throughput compared to non-AVX OpenSSL. If the web server additionally compresses the files on-the-fly with the Brotli compression algorithm, both AVX2 and AVX-512 instead reduce performance as even more non-AVX code is affected by the frequency reduction.

3 CORE SPECIALIZATION

As each core individually determines its own frequency level, the performance reduction caused by AVX-heavy code only affects a single core. For the AVX-512-enabled version of the x265 video encoder, Tiwari et al. [19] therefore suggest restructuring the application to concentrate AVX-512 code on a limited subset of the cores, whereas all other cores only execute non-AVX-heavy code (*core specialization*). If the code sections with high frequency impact are restricted to a subset of the system's cores, the negative performance impact is limited to code running on these cores.

Restructuring the application, however, requires a significant engineering effort and might cause overhead in cases where no problematic frequency reduction is expected. Therefore, we describe an application-agnostic approach which does not require modifications to the software. Our design takes unmodified applications and uses migration of their threads as a mechanism to limit execution of AVX-heavy code to some of the cores. We use fault-and-migrate [13] to detect AVX-heavy code and we modify the operating system's scheduler to implement core specialization.

3.1 Intercepting AVX Instructions

Fault-and-migrate is a technique to automatically migrate threads to a suitable core in a heterogeneous multiprocessor system [13]. Whenever a thread executes an instruction that is not supported by its current core, an undefined instruction exception is raised. The operating system handles this exception by resuming the thread on a core which supports the instruction. Although this mechanism allows the operating system to recognize when special instructions are required, no information is given about how long the instructions are required. Therefore, after a fixed time period the operating system simply migrates the thread back to the original core, assuming that no instructions are required anymore that are not supported by that core.

To simulate a functionally heterogeneous multiprocessor on an existing symmetric x86 multiprocessor, the floating point unit can be completely disabled [13], in which case all floating point and vector instructions cause exceptions. We modify this mechanism to identify and trap only AVX2 and AVX-512 instructions. To disable AVX-512 instructions, we clear the corresponding bits in the XCR0 register (`Opmask`, `Hi16_ZMM` and `ZMM_Hi256`) which govern access to the registers specific to 512-bit AVX instructions [2]. Similarly, 256-bit register access can be disabled by clearing the `AVX` bit of XCR0. The corresponding instructions will then cause an undefined instruction exception when executed on cores on which these bits have been cleared.

3.2 Scheduling Algorithm

Whenever an AVX instruction causes an exception, the thread is temporarily flagged as an *AVX task*. The system's cores are divided into *AVX cores* and *non-AVX cores*, and AVX tasks are only scheduled on AVX cores. To keep system utilization high, the AVX cores can execute arbitrary tasks, although AVX tasks are strictly prioritized over non-AVX tasks.

We implement this policy as an extension of the MuQSS out-of-tree Linux scheduler [8]. We choose this scheduler over the in-tree CFS scheduler due to its significantly lower code complexity. As our implementation mostly replicates scheduler data structures and does not require any major conceptual changes, we expect an implementation in a different scheduler to be viable as well.

The MuQSS scheduler is built around a queue of runnable tasks ordered by virtual deadlines calculated based on the tasks' priority. We configure MuQSS to maintain one such queue per physical processor core. Additionally, we modify the scheduler to replicate the runqueues three times, with each runqueue holding a different class of tasks. The first runqueue holds tasks which are classified as AVX tasks. Whenever an AVX task is determined not to require AVX instructions anymore, it is moved to the second runqueue. AVX cores never schedule tasks from the second runqueue if any other runqueue contains runnable tasks, whereas non-AVX cores never schedule tasks from the first runqueue at all. Finally, the third runqueue holds all tasks which have never executed AVX instructions. Tasks from this runqueue are taken into consideration by both AVX cores and non-AVX cores. This policy excludes tasks from core specialization if they never use AVX. The third runqueue, in particular, includes all system tasks which might be pinned to individual cores, which prevents such system tasks from being starved by AVX tasks. Other approaches integrate a more complex priority system to prevent such starvation [16] which makes those algorithms more difficult to integrate into the prioritization schemes of existing schedulers.

As the MuQSS scheduler implicitly performs load balancing at every scheduler invocation by (locklessly) checking the runqueues of all other cores, tasks which cannot be scheduled on their current core are quickly picked up by a different core. We add additional inter-processor interrupts where required when tasks are reclassified in order to enforce a scheduler invocation on a suitable core.

4 EVALUATION

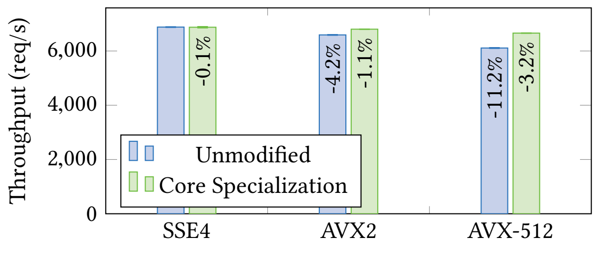
While our work is still in an early stage, we have constructed a partial prototype to show that the approach can reduce the performance impact of AVX-heavy code sections. In particular, with our evaluation we want to show that thread migration is a viable mechanism to specialize cores and that extensive (and expensive) restructuring of existing applications is not required.

To evaluate the prototype, we mainly use a web server scenario derived from the setup described in [9]. We configure the NGINX web server to serve a static file with on-the-fly compression over an HTTPS connection using the ChaCha20-Poly1305 encryption algorithm. Encryption is provided by OpenSSL and we compare performance for versions of OpenSSL which use either AVX-512 or AVX2 instructions or no AVX instructions at all. All benchmarks are executed on a system with a 16-core Intel Xeon Gold 6130 processor and 24 GiB of DDR4 RAM. Four cores execute the wrk2 benchmark client, whereas the other 12 cores execute the web server.

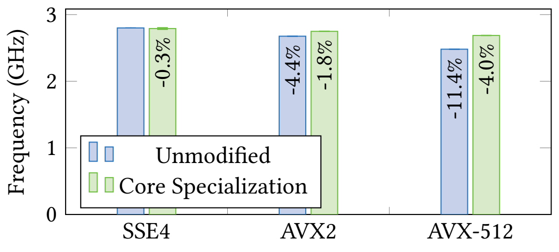
Our prototype does not yet include the automatic detection of AVX instructions described in Section 3.1. Instead, for our evaluation, we manually annotate the web server to indicate the parts which are only allowed to execute on AVX cores by placing a system call before and after these parts. For the NGINX web server, we mark the call into OpenSSL to encrypt or decrypt data as AVX code. Note that this manual annotation is very pessimistic, as the call into OpenSSL not only encrypts or decrypts data but also wraps Linux socket system calls. Therefore, AVX cores not only execute AVX-heavy code but also large amounts of networking code from the Linux kernel. Our prototype is therefore not able to show the full potential of the technique. Automatic detection of AVX code as described in Section 3.1 most likely arrives at a better estimate of the AVX code contained in the application and is likely able to achieve higher average processor frequencies.

4.1 CPU Frequency and Throughput

Figure 2a shows the throughput achieved by NGINX with and without our prototype and with the different versions of OpenSSL. With an unmodified system, enabling AVX2 instructions in OpenSSL reduces performance by 4.2% whereas enabling AVX-512 instructions



(a) Throughput



(b) Average frequency of all cores

Figure 2: Benchmark results for NGINX with OpenSSL compiled for different instruction sets: The blue bars show the results for the unmodified web server, whereas the green bars show results when SSL code is restricted to a subset of the cores.

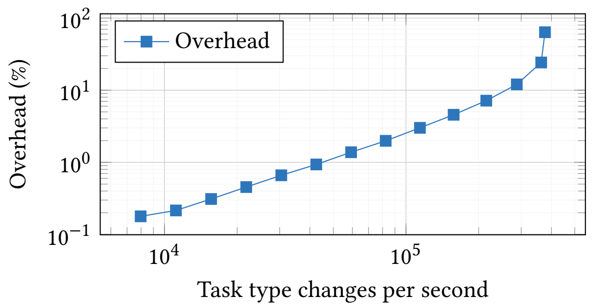


Figure 3: Overhead of core specialization in a CPU-intensive microbenchmark. The horizontal axis shows the task type changes (AVX vs. non-AVX) per second. For reference, the web server benchmark described above migrates tasks between AVX and non-AVX cores up to 55000 times per second.

reduces performance by 11.2%. Our prototype reduces this performance loss to 1.1% and 3.2%, respectively, which shows that core specialization is a viable technique to reduce the negative impact of AVX code on performance and performance predictability.

As our version of core specialization mainly affects processor frequency, Figure 2b contains the average processor frequencies recorded in the experiment. Whereas the performance of the unmodified web server closely matches the corresponding average frequency, our prototype shows a slightly better performance than expected from the frequency results, even though increased migration of threads between cores should cause additional overhead.

To determine the reason for this anomaly, we perform a performance counter analysis for both web server setups with non-AVX OpenSSL. As expected, the analysis shows a slightly increased amount of executed instructions (+0.7%), but it also shows an increased amount of instructions per cycle (+0.7%), which makes up for the overhead. While core specialization causes some additional stall cycles due to memory accesses, it more importantly reduces the stall cycles due to mispredicted branches, which is most likely the cause for the IPC improvement. Other approaches using core specialization or cohort scheduling have shown similar impact on cache effectiveness [6][7][11].

4.2 Overhead

As described in the last section, the overhead of frequent thread migration and of the scheduler modifications required for core specialization is below one percent for the tested benchmark. The overhead, however, depends on how often tasks are reclassified as AVX tasks or non-AVX tasks. We measure the overhead over a wider range of such task change rates with a benchmark which executes a varying number of instructions between two task type changes in order to generate different task change rates. The program is structured so that 5% of the instructions are annotated as if they were AVX code so that they are only executed on AVX cores.

Figure 3 shows the resulting overhead for this program if it is executed in parallel on 12 cores. The graph shows that the overhead is approximately proportional to the rate of task type changes, with task type changes costing approximately 400-500 ns over a wide range of rates[1]. Although for frequent task type changes the overhead can be larger than any advantage due to core specialization, even 100,000 task type changes per second (corresponding to 50,000 invocations of AVX code) only cause 3% overhead, which is less than the 6% improvement shown in Section 4.1.

Note that this analysis explicitly excludes cache effects, as those effects highly depend on the application. While possible that applications suffer from cache line bouncing, many applications can profit from the use of private caches from multiple cores [7].

4.3 Discussion

These results show that core specialization is able to notably reduce the negative impact of AVX-512 and AVX2 on performance. Therefore, a system with core specialization has increased performance predictability. Our prototype does not yet fully implement our proposed design, which warrants a closer look at the prototype's limitations. Mainly, our prototype does not yet implement automatic fault-and-migrate for AVX instructions. Automatic faulting would most likely reduce the amount of code executed on the AVX cores in the scenario described above. As our manual annotation causes large amounts of non-AVX code to be executed on the AVX cores, a more accurate estimate of the problematic AVX code sections would likely result in better performance isolation of the AVX code. On our system, the additional exception required

[1]For more than 300,000 task type changes per second our prototype shows high lock contention in the scheduler which reduces CPU utilization. Improved data structures can likely produce significant performance improvements.

for fault-and-migrate only costs 300 ns each time a task is migrated to an AVX core. Therefore, we expect the improved performance isolation to compensate for the additional overhead.

The Need for Better Power Management Interfaces. In the case of AVX-512 and AVX2, fault-and-migrate is not the ideal solution. Current CPUs only allow the system to be configured to make all AVX-512 or AVX2 instructions fault – however, single or infrequent AVX instructions do not trigger frequency changes. This type of fault-and-migrate overestimates the impact of individual instructions and is only a workaround for an insufficient hardware-software interface. Ideally, the CPU would accurately notify the OS of impeding frequency changes. Upon notification, the OS could then migrate problematic tasks to a different core to avert the frequency change. A specialized interface for such notifications would provide higher accuracy and lower overhead than any mechanism conceivable on current CPUs.

We predict that the frequency variation and therefore the need for such an interface will increase in the future. Currently, only instructions from specific functional units in the core draw so much power that a frequency reduction is required. As an increased use of task-specific accelerators is widely accepted as one method to improve performance in a power-limited environment [18], future systems will have even higher power variation depending on the instruction mix and will therefore suffer from greater frequency variation. On these systems, efficient core specialization could have even higher benefit.

5 RELATED WORK

Frequency Effects of AVX2 and AVX-512. The introduction of lower *AVX frequencies* was first described for Haswell-EP processors [5]. Later, Skylake-SP processors introduced an even lower frequency level for AVX-512. In a blog post, Daniel Lemire [12] describes how this frequency level is, however, only used for limited types of instruction mixes, so during optimization software engineers have to balance the resulting frequency with the increased parallelism brought by wider vector instructions. As future work, Daniel Lemire mentions that a framework or scheduler which limits execution of problematic code to a subset of the cores can potentially reduce the negative performance impact of AVX-heavy code. We describe a concrete implementation of this approach and evaluate its effects on overall performance.

A similar conclusion was reached by engineers at Intel who implemented AVX-512 vectorization for the x265 video encoder [19]. Despite doubled vector sizes, x265 only became less than 10% faster, so the authors suggest that better performance could be achieved by restructuring x265 so that only some threads (on a subset of the cores) use AVX-512. Such restructuring, however, is costly and we show that migration of unmodified threads already provides notable performance improvements at low overhead.

OS Support for Heterogeneous Multiprocessors. Extensive research has been performed in the area of heterogeneous multiprocessors, where different cores provide different instruction sets [13] or different performance characteristics [10]. Our approach specializes cores to create software-based heterogeneity.

On heterogeneous multiprocessors with varying instruction set, the choice of cores for a task is limited by the available instructions. One possible mechanism to automatically move tasks to a suitable core is fault-and-migrate [13] where execution of an instruction not supported by the core triggers an exception and causes the task to be migrated to a different core. We suggest the use of fault-and-migrate on current server CPUs and describe a mechanism to selectively make AVX2 and AVX-512 instructions trigger exceptions.

Shen et al. [16] describe scheduling algorithms for heterogeneous multiprocessors. In particular, they describe how tasks should be scheduled on the core which most closely matches the required instruction set and how static priorities can be used to prevent tasks which require a larger instruction set from starving tasks with fewer required instructions. Our approach, in contrast, does not use any specific priority scheme and can therefore be integrated into a wider range of existing schedulers.

Core Specialization. In the past, limiting parts of the system's workload to certain cores has been used with other objectives in mind. For example, FlexSC [17] executes applications and the operating system on different cores in order to split the working set. As a result, FlexSC is able to make more effective use of the cores' private caches. In our scenario, such cache effects are only responsible for a fraction of the measured performance improvement.

6 CONCLUSION

Current Intel CPUs temporarily reduce their frequencies when AVX-heavy instruction mixes threaten to violate power and current draw limits. As any frequency reduction lasts for at least two milliseconds, it can have a significant negative impact on overall performance for workloads which intermix short AVX-enabled sections with sections which could execute at higher frequencies.

We demonstrate that core specialization can be used to reduce this negative impact and to improve performance predictability. By limiting execution of AVX-heavy code to a subset of the system's cores, our approach limits the frequency reduction to those cores and improves performance isolation of the AVX-heavy parts of the workload. We also describe a method to implement fault-and-migrate for all AVX-512 instructions so that such AVX-512 code is automatically migrated to AVX cores.

We demonstrate the potential of such an approach with a partial prototype which shows that migration of unmodified threads provides significant performance advantages while causing only low overhead. Our prototype reduces the performance impact of AVX-induced frequency reduction by 70% in a web server scenario.

6.1 Future Work

We have shown that thread migration is a suitable mechanism for core specialization and have described a mechanism to selectively make AVX-512 instructions fault. In the future, we will integrate fault-and-migrate into our prototype in order provide fully automatic core specialization. In addition, we will develop metrics to estimate in advance whether core specialization results in a net benefit, as any practical solution needs to balance the overhead of thread migrations against the resulting frequency improvement.

REFERENCES

[1] *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, April 2018.

[2] *Intel® 64 and IA-32 Architectures Software Developer's Manual - Volume 2 (2A, 2B, 2C & 2D): Instruction Set Reference, A-Z*, May 2018.

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

[4] Robert H. Dennard, Fritz H. Gaensslen, V. Leo Rideout, Ernest Bassous, and Andre R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[5] Daniel Hackenberg, Robert Schöne, Thomas Ilsche, Daniel Molka, Joseph Schuchart, and Robin 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.

[6] Stavros Harizopoulos and Anastassia Ailamaki. Steps towards cache-resident transaction processing. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases*, volume 30, pages 660–671. VLDB Endowment, 2004.

[7] Prathmesh Kallurkar and Smruti R Sarangi. Schedtask: a hardware-assisted task scheduler. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 612–624. ACM, 2017.

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

[9] Vlad Krasnov. On the dangers of intel's frequency scaling, October 10, 2017. https://blog.cloudflare.com/on-the-dangers-of-intels-frequency-scaling/.

[10] Rakesh Kumar, Keith I Farkas, Norman P Jouppi, Parthasarathy Ranganathan, and Dean M Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 81. IEEE Computer Society, 2003.

[11] James R Larus and Michael Parkes. Using cohort-scheduling to enhance server performance. In *Proceedings of the USENIX 2002 Annual Technical Conference*, pages 103–114. USENIX Association, 2002.

[12] Daniel Lemire. Avx-512: when and how to use these new instructions, September 9, 2018. https://lemire.me/blog/2018/09/07/avx-512-when-and-how-to-use-these-new-instructions/.

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

[14] Abdelhafid Mazouz, Alexandre Laurent, Benoît Pradelle, and William Jalby. Evaluation of cpu frequency transition latency. *Computer Science - Research and Development*, 29(3-4):187–195, 2014.

[15] Daniel Molka, Daniel Hackenberg, Robert Schöne, and Matthias S Müller. Characterizing the energy consumption of data transfers and arithmetic operations on x86-64 processors. In *International Conference on Green Computing*, pages 123–133. IEEE, 2010.

[16] Hao Shen and Frédéric Pétrot. Novel task migration framework on configurable heterogeneous mpsoc platforms. In *Proceedings of the 2009 Asia and South Pacific Design Automation Conference*, pages 733–738. IEEE Press, 2009.

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

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

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

Online sources last accessed 16th January 2019.