Remote AVX Overhead: Detection and Mitigation

Mathias Gottschlag
Karlsruhe Institute of Technology
mathias.gottschlag@kit.edu

ABSTRACT

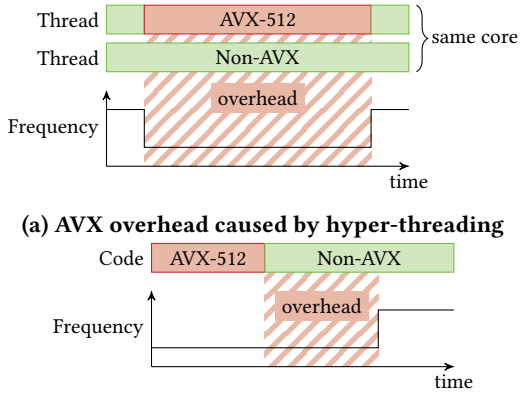
Due to power constraints, recent Intel CPUs reduce their frequency when executing AVX2 and AVX-512 instructions. Often, this frequency reduction affects other applications as well, which reduces overall performance and prevents contemporary operating systems from fairly distributing system resources. In our work, we show that these problems are fundamental problems of power-limited computing. We analyze the problems and show a method to quantify the underlying *AVX overhead*. Based on our analysis, we then describe a set of operating system techniques to improve performance and scheduler fairness. Our results show the importance of active management of hardware-controlled frequency scaling by the OS. Based on this observation, we sketch improved hardware-software interfaces which could further reduce AVX overhead and improve the efficacy of our approach.

KEYWORDS

power management, operating systems, profiling, scheduling, DVFS, AVX-512, AVX2

1 INTRODUCTION

With shrinking transistor sizes, the power density of integrated circuits has increased to the point where the performance of modern CPUs is mainly limited by their power consumption [16]. To extract maximum performance, modern CPUs therefore need to make full use of their power budget. These CPUs increase their frequency when only a subset of the cores is active [2, 5] and when cool heatsinks allow temporary excursions beyond regular thermal limits [14]. In addition, the use of specialized accelerators has been proposed to achieve further performance improvements [16]. For example, the AVX2 and AVX-512 single-instruction multiple-data (SIMD) instruction set extensions found in recent Intel CPUs provide both improved energy efficiency and performance [4].



(a) AVX overhead caused by hyper-threading

(b) AVX overhead in heterogeneous tasks

Figure 1: In many scenarios, AVX2/AVX-512 code slows other code down. Figure taken from [8].

Despite their increased energy efficiency, such complex instructions can substantially increase power consumption. To provide maximum performance for all types of code, the CPU therefore needs to execute code using such energy-intensive instructions at reduced frequencies while executing other code at higher frequencies. The underlying power limits are not only of thermal nature – the CPU also needs to react quickly to power changes to prevent instability due to voltage droops [12].

When Intel introduced the AVX2 and AVX-512 instruction set extensions, the company introduced such a technique to execute different types of instructions at different CPU frequencies [1]. If no complex and energy-intensive 256-bit (AVX2) and 512-bit (AVX-512) vector operations are used, these CPUs increase their frequency to a high non-AVX frequency level. Conversely, the execution of AVX2/AVX-512 instructions quickly triggers a frequency reduction to prevent system instability due to excessive current draw [6]. The scale of the frequency reduction depends on the instructions used – only particularly energy-intensive AVX-512 instructions trigger a transition to the lowest "AVX-512" frequency level, whereas all other instructions cause the CPU to enter an intermediate "AVX2" level [1].

FGBS '21, March 11–12, 2021, Wiesbaden, Germany

https://doi.org/10.18420/fgbs2021f-04

Although such a frequency scaling technique in general provides improved performance due to improved utilization of power budgets, there are two situations where low-power code is not executed at its optimal frequency [1]. First, as shown in Figure 1a, any frequency reduction also affects the code executing on the sibling hyper-thread [8]. Second, as shown in Figure 1b, the affected Intel CPUs do not immediately restore a higher frequency at the end of a power-intensive code region as a rate-limiting technique to limit worst-case overhead, which causes any code following a section of AVX2/AVX-512 code to be slowed down. Both situations contribute to what we call *remote AVX overhead*: AVX2/AVX-512 code slows other potentially unrelated code down.

This slowdown can cause a reduction in performance of more than 30% for the affected code [10]. The performance reduction also reduces fairness – the scheduler commonly allocates equal CPU time to individual processes which results in substantially different shares of CPU performance. These two problems are to a large degree caused by the fact that applications are implicitly in control of the dynamic voltage and frequency scaling (DVFS) policy as instructions executed at the user level directly affect CPU frequencies. Remote AVX overhead, however, is caused by the interaction of multiple software components or tasks, and applications commonly lack the required information to make informed decisions on whether to use AVX2 or AVX-512.

The operating system, instead, has a more holistic view of the system. In our work, we describe operating system techniques to measure the amount of remote AVX overhead present the system and describe operating system techniques to improve both performance and fairness. Our results show that the operating system should take a more active stance managing hardware-controlled DVFS. As our work is frequently hindered by insufficient hardware-software interfaces provided by the CPU, we also sketch improved interfaces to the CPU which would improve the efficacy of our approaches and would allow for improved DVFS policies to be implemented by the CPU.

2 PROFILING

Before deciding on countermeasures against remote AVX overhead, software developers and system administrators need to know to which degree their workloads are affected. The difficulty in quantifying remote AVX overhead lies in differentiating it from *local AVX overhead* where the frequency reduction affects the AVX2/AVX-512 code itself. While local AVX overhead is unavoidable, the increased parallelism of AVX2/AVX-512 instructions makes up for it.

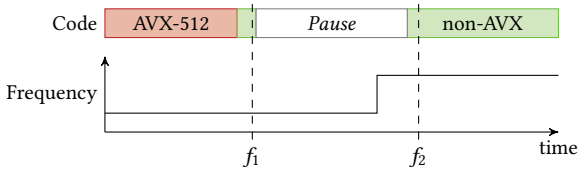


Figure 2: To measure remote AVX overhead, we pause a physical CPU core and then unpause one hyper-thread. If the frequency f_1 before the pause is low and the frequency f_2 after the pause remains high, the hyper-thread was affected by remote AVX overhead – in such a situation, the code was executed at a sub-optimal frequency at the time of the pause. Figure taken from [11].

Figure 3: We employ core specialization to reduce remote AVX overhead [8]. If only some cores are allowed to execute AVX-512 instructions, tasks on the other cores will never execute at AVX-512 frequencies.

To determine whether a frequency reduction is actually required for the currently executed code, we propose a profiler which periodically samples the frequency of a CPU core, then pauses the whole core by spinning for 700 µs – longer than the delay seen in Figure 1b – and finally resumes execution on one of the two hyper-threads [11]. If, as shown in Figure 2, the previous frequency is not immediately restored, the current task was affected by remote AVX overhead. We use this technique to construct a profiler which can be used at runtime to quantify the AVX overhead and to identify the affected tasks.

We determine the accuracy of the profiler via comparisons to direct reference measurements taken using non-AVX/AVX2/AVX-512 variants of the same workload [1]. Our experiments show an average error of only 1.2 percentage points for a range of scenarios involving AVX2 or AVX-512 applications.

3 CORE SPECIALIZATION

Once significant remote AVX overhead has been identified, countermeasures can be employed to improve performance.

[1] In a practical environment, such variants are often not available. If they were, the need to restart the application would often make direct measurements highly impractical.

For example, core scheduling can be employed to restrict co-scheduling of AVX-512 and non-AVX-512 tasks on the same physical core [13]. If one hyper-thread executes an AVX-512 task, the other hyper-thread is not allowed to execute a non-AVX-512 task as such a task would be substantially slowed down. As a result, however, hyper-threads can be left idle which can have a potentially large impact on performance.

We therefore propose core specialization as a technique against remote AVX overhead [8]. We designate a subset of the CPU cores as *AVX cores*, and only these cores are allowed to execute AVX-512 instructions. As Figure 3, all other cores only execute non-AVX-512 code and are therefore not affected by AVX-512 frequency reduction. As a result, the impact of AVX-512 code on non-AVX-512 code is greatly reduced.

To detect attempts to execute AVX-512 instructions on non-AVX-512 cores, we restrict access to 512-bit registers so that all corresponding instructions trigger exceptions and thread migration to a AVX-512 core [8]. We employ heuristics to detect the end of AVX-512 regions. Our prototype is able to reduce remote AVX overhead by 70% for a range of workloads with AVX-512 code, despite imprecise detection of power-intensive instructions [8] and overhead introduced by thread migration [7].

4 FAIR SCHEDULING

In many cases, techniques such as core specialization [8] or core scheduling [13] can improve overall throughput. Not all remote AVX overhead can be prevented though. Even if remote AVX overhead is present, it is highly desirable that the system limits the impact that AVX2/AVX-512 tasks can have on other tasks.

Commonly, such performance impact between different tasks is prevent by a fair scheduler such as CFS that allocates equal CPU time to different tasks as a technique to share CPU performance equally between them. As Figure 4a shows, the frequency reduction caused by AVX2/AVX-512 means that equal CPU time does not result in equal CPU performance anymore – whereas AVX2/AVX-512 tasks profit from the increased parallelism of these SIMD instructions, other tasks do not, yet still execute at reduced frequencies.

We therefore propose fair scheduling that takes remote AVX overhead into account by scaling CPU time accounting in proportion to remote AVX overhead. As shown in Figure 4b, the resulting prioritization of tasks whose performance is affected negatively by other AVX2/AVX-512 tasks restores fairness. In such a situation where overall CPU performance is impacted by reduced frequencies, fairness is not equivalent to performance isolation, though. If, as shown in Figure 4c, we additionally reduce the priority of tasks causing remote AVX overhead, we get a scheduling algorithm

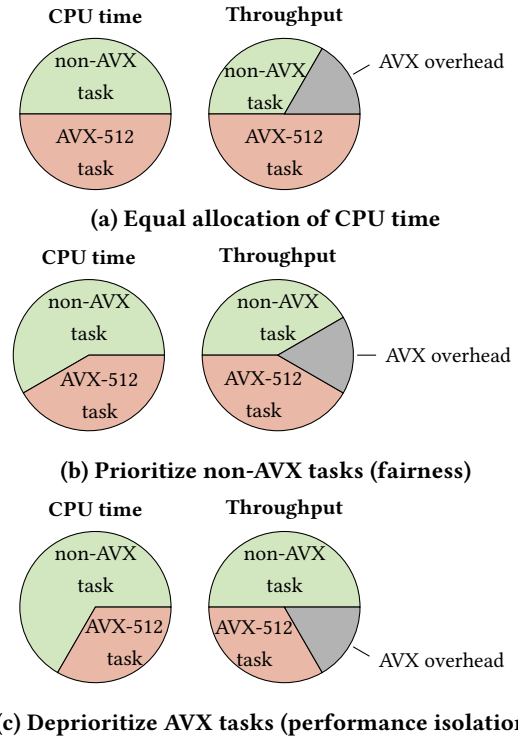


Figure 4: As remote AVX overhead slows non-AVX – and sometimes AVX2 – tasks down, equal CPU time allocation leads to unequal throughput. Our scheduler prioritizes affected tasks to restore a fair allocation of CPU performance [10] (b) and optionally slows down the tasks causing overhead to minimize the performance impact on non-AVX tasks (c).

that minimizes performance impact on, for example, non-AVX tasks at the expense of AVX2 and AVX-512 tasks. Such performance isolation may be desirable in situations where a customer pays for fixed CPU performance.

While implementing these scheduling algorithms, we noticed that quick thread migration between CPU cores is necessary to give the scheduler sufficient leeway for prioritization between different types of tasks. A prototype using a custom fair scheduler based on the MuQSS scheduler for the Linux kernel was able to reduce both unfairness as well as inter-task performance impact by 70%.

5 OS MANAGEMENT OF HARDWARE-CONTROLLED DVFS

The results gained with the prototypes described in the last two sections show the large potential of active management of AVX frequencies by the operating system. Although our prototypes target recent Intel CPUs, the concepts are not

limited to these CPUs. As described in the introduction, frequency scaling based on the energy consumption of different instructions is an efficient approach to solve a fundamental problem of power-limited computing and, albeit currently patented by Intel [15], is likely to be used in other future CPUs as well. We expect active management of hardware-controlled DVFS by the OS to be beneficial on these platforms as well.

Our experiments, however, have also shown that current CPUs are not particularly suited for software control over their DVFS policies. Our prototypes provide large benefits, yet they are not able to completely mitigate the impact of remote AVX overhead, mainly due to insufficient hardware-software interfaces provided by existing CPUs. Often, the prototypes could benefit from more information about code currently executed on the CPU – such information would allow more precise low-overhead profiling and would allow fairer scheduling [11]. Similarly, if the operating system was informed in advance of impeding frequency changes with the option to intervene, more effective core specialization implementations might bring further performance improvements [8].

Besides these minor interface improvements, we identified the DVFS policy implemented by the CPU as the main area where hardware modifications could yield improved performance. Currently, as described in Section 1, the CPU waits for a fixed timeout before restoring higher frequencies which bears great similarity to fixed-timeout policies for dynamic power management [3]. We show that the delay and the resulting performance impact can be prevented by better policies. For example, if the software executed on the system notifies the CPU when no further AVX2/AVX-512 code is expected in the near future, the CPU can conduct an immediate frequency change [9]. Such notifications can, for example, be sent as part of the context switching code based on previously observed behaviour of the tasks. We simulated such an improved policy in a simple simulator and achieved a 4% performance improvement for a single-threaded workload. Although insufficient resources prevented implementing an optimized DVFS policy within a representative custom hardware platform, our results pave the way for future work by demonstrating the general potential of improved DVFS policies.

6 CONCLUSION

As modern CPUs are mainly limited by power consumption, Intel has introduced a frequency boost mechanism where code without complex AVX2 and AVX-512 instructions is executed at higher frequencies. The frequency reduction caused by AVX2/AVX-512 instructions often affects other potentially unrelated code that executes on the same physical CPU core, though, an effect which we call remote AVX overhead. We present technique to measure and to reduce this overhead and we propose improved scheduling algorithms for workloads consisting of AVX2/AVX-512 tasks and non-AVX tasks scheduled on the same CPU cores. Our experiments show that the approaches can reduce the impact of remote AVX overhead by 70%. This result shows the importance of active management of hardware-controlled DVFS by the operating system. As the underlying problem is caused by fundamental properties of power-limited computing and is likely to surface on more systems in the future, we sketch hardware changes to further empower the OS to take control over short-term power-related frequency changes.

REFERENCES

[1] 2019. *Intel® 64 and IA-32 Architectures Optimization Reference Manual.*

[2] Murali Annavaram, Edward Grochowski, and John Shen. 2005. Mitigating Amdahl's law through EPI throttling. In *32nd International Symposium on Computer Architecture (ISCA'05)*. IEEE, 298–309.

[3] Luca Benini, Alessandro Bogliolo, and Giovanni De Micheli. 2000. A survey of design techniques for system-level dynamic power management. *IEEE transactions on very large scale integration (VLSI) systems* 8, 3 (2000), 299–316.

[4] Juan M Cebrian, Lasse Natvig, and Magnus Jahre. 2019. Scalability analysis of AVX-512 extensions. *The Journal of Supercomputing* (2019), 1–16.

[5] James Charles, Preet Jassi, Narayan S Ananth, Abbas Sadat, and Alexandra Fedorova. 2009. Evaluation of the Intel® Core™ i7 turbo boost feature. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 188–197.

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

[7] Mathias Gottschlag and Frank Bellosa. 2018. Mechanism to Mitigate AVX-Induced Frequency Reduction. arXiv:1901.04982 [cs.DC]

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

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

[10] Mathias Gottschlag, Philipp Machauer, Yussuf Khalil, and Frank Bellosa. 2021. Fair Scheduling for AVX2 and AVX-512 Workloads. Submitted.

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

[12] Charles R Lefurgy, Alan J Drake, Michael S Floyd, Malcolm S Allen-Ware, Bishop Brock, Jose A Tierno, and John B Carter. 2011. Active management of timing guardband to save energy in POWER7. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–11.

[13] 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.

[14] Arun Raghavan, Yixin Luo, Anuj Chandawalla, Marios Papaefthymiou, Kevin P Pipe, Thomas F Wenisch, and Milo MK Martin. 2012. Computational sprinting. In *IEEE international symposium on high-performance comp architecture*. IEEE, 1–12.

[15] Daniel J Ragland, Pavithra Sampath, Kirk Pfaender, Kahraman D Akdemir, and Ariel Gur. 2017. Processors, methods, and systems to adjust maximum clock frequencies based on instruction type. US Patent App. 15/055,578.

[16] 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.