AVX Overhead Mitigation: OS Support for Power-Limited Systems

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Mathias Gottschlag, M.Sc.

aus Bad Driburg

Tag der mündlichen Prüfung: 23. Juni 2023
Erster Referent: Prof. Dr. Frank Bellosa
Zweiter Referent: Prof. Dr. Timo Hönig

Acknowledgements

I am immensely grateful to my supervisor, Prof. Dr. Frank Bellosa, for giving me full freedom when choosing my direction of research, yet always giving invaluable advice whenever needed. His thorough feedback on paper and thesis drafts proved to be very helpful, as did his focus on making sure that I always had the tools required for my experiments. I would also like to extend my gratitude to Prof. Dr. Timo Hönig for his valuable commentary on drafts of this thesis and for his invitation to present my work at the 2022 PEACHES workshop.

My heartfelt thanks go to the other members of the operating systems group for their continued support throughout my work. Thorsten Gröninger, Marius Hillenbrand, Jens Kehne, Marc Rittinghaus, and Lukas Werling all had a large impact on this thesis, be it via valuable insights during discussions or via thorough feedback on drafts and presentations. In many cases, you taught me the finer points of the scientific crafts, for which I am very grateful.

I am also deeply thankful for the assistance from Maximilian Heß and my brother Bernd Gottschlag, who dedicated many hours to proofreading this thesis. It is not only this direct support that makes a thesis such as this one possible, though. Especially during the more stressful phases of a PhD, emotional support is often equally valuable. In this sense, I would like to extend my gratitude to all the friends who lifted my spirits over countless rounds of board games and computer games.

I would also like to thank my parents, without whose continued guidance throughout life I would have hardly been in a position where I would be able to perform such research work. Finally, thank you Itzel for always being there and supporting me whenever I needed support.

Abstract

In recent years, processors have become more and more power-limited. As a result, one of the main goals of CPU frequency selection of recent processors is to maximize performance under the given power constraints. On these processors, the choice of instructions has started to affect operating frequencies as different instructions require different amounts of energy. For example, current Intel server CPUs with support for the AVX2 and AVX-512 instruction set extensions reduce the frequency of individual cores that execute these power-intensive SIMD instructions. The resulting temporary frequency reduction reduces the throughput of the SIMD code itself, but also affect other, less power-intensive code that is, for example, executed on another hardware thread of the same physical CPU core at the same time.

This thesis focuses on the overhead that is caused by such situations where code consisting only of simple instructions with little energy consumption is executed at sub-optimal CPU frequencies. This *remote AVX overhead* has been reported to slow some workloads down by up to 30% and presents a major challenge in the use of AVX2 and AVX-512. As we show, remote AVX overhead is largely avoidable using OS-level techniques, though.

In this thesis, we perform a thorough analysis of the origins of remote AVX overhead and present a profiler that is able to accurately measure its performance impact. We then demonstrate that existing frequency selection policies implemented by the CPU provide little potential for improvement in most scenarios – instead, major performance gains are possible via techniques implemented in the operating system. We present two scheduler modifications to mitigate the impact of remote AVX overhead in different situations. First, we show how limiting the use of AVX-512 to few CPU cores and migrating individual tasks to appropriate cores can greatly reduce remote AVX overhead in heterogeneous workloads involving AVX-512 code. Second, we show how prioritizing tasks affected by remote AVX overhead can greatly improve performance isolation in the presence of remote AVX overhead caused by either AVX2 or AVX-512 code.

Our work demonstrates the need for increased involvement of the operating system with CPU frequency selection in current – and, potentially, future – power-limited systems. In addition, we uncover a range of potential improvements to existing CPU designs that would allow for even more effective mitigation of effects such as remote AVX overhead.

Zusammenfassung

In den vergangenen Jahren haben sich Prozessoren dahingehend entwickelt, dass sie mehr und mehr durch ihre Leistungsabgabe begrenzt wurden. Aus diesem Grund ist eines der Hauptziele der Strategie zur Auswahl der Frequenz aktueller Prozessoren, die Performance innerhalb der gegebenen Leistungsgrenzen zu maximieren. Auf diesen Prozessoren beeinflusst deshalb die Wahl der Instruktionen mittlerweile oft die Betriebsfrequenz, da unterschiedliche Instruktionen unterschiedlich viel Energie benötigen. Ein Beispiel dafür findet sich in aktuellen Intel-Server-Prozessoren mit Unterstützung für AVX2- und AVX-512-Instruktionen. Diese Prozessoren reduzieren die Frequenz von CPU-Kernen, die leistungsintensiven Code mit solchen SIMD-Instruktionen ausführen. Die dadurch verursachte Frequenzreduktion reduziert den Durchsatz dieses SIMD-Codes, beeinflusst aber auch anderen weniger leistungsintensiven Code, der zum Beispiel gleichzeitig auf einem anderen Hardware-Thread des gleichen physischen CPU-Kerns ausgeführt wird.

Diese Arbeit konzentriert sich auf die Kosten, die in solchen Situationen dadurch entstehen, dass einfacher, wenig leistungsintensiver Code mit einer suboptimalen CPU-Frequenz ausgeführt wird. Diese Kosten – *Remote AVX Overhead* genannt – verlangsamen laut früherer Arbeiten manche Workloads um bis zu 30% und stellen eine große Herausforderung bei der Verwendung von AVX2 und AVX-512 dar. Remote AVX Overhead kann jedoch, wie wir zeigen, größtenteils durch im Betriebssystem umgesetzte Techniken vermieden werden.

In dieser Arbeit führen wir eine umfangreiche Analyse der Gründe für Remote AVX Overhead durch und beschreiben einen Profiler, der in der Lage ist, Remote AVX Overhead mit hoher Genauigkeit zu quantifizieren. Zudem zeigen wir, dass die Frequenzauswahlstrategie existierender Prozessoren in den meisten Fällen wenig Potential für Performance-Verbesserungen bietet – stattdessen sind wesentliche Verbesserungen durch Betriebssystemmodifikationen möglich. Wir beschreiben zwei Modifikationen des Schedulers, die in unterschiedlichen Situationen dem Einfluss von Remote AVX Overhead entgegenwirken. Zuerst zeigen wir, wie dadurch, dass AVX-512-Nutzung auf wenige Prozessorkerne beschränkt wird und Tasks auf passende Kerne migriert werden, Remote AVX Overhead stark reduziert wird, wenn Teile der ausgeführten Software AVX-512 verwenden. Danach zeigen wir, wie eine Priorisierung von Tasks, die durch Remote AVX Overhead verlangsamt werden, die Performance-Isolierung zwischen Tasks in Situationen verbessert, wenn das System teilweise entweder AVX2 oder AVX-512 ausführt.

Unsere Arbeit demonstriert, dass das Betriebssystem wesentlich stärker in der Wahl von Prozessorfrequenzen in aktuellen und zukünftigen leistungslimitierten Systemen involviert sein muss. Zudem zeigen wir eine Reihe von möglichen Verbesserungen existierender Prozessorarchitekturen auf, die eine noch effektivere Reduktion des Einflusses von Effekten wie Remote AVX Overhead ermöglichen würden.

Contents

Abstract . . . iii

Zusammenfassung . . . v

1 Introduction . . . 1
1.1 Scope . . . 3
1.2 Contributions . . . 4
1.3 Student Theses and Publications . . . 4
1.4 Document Structure . . . 6

2 Background: Power-Limited Computing . . . 9
2.1 Transistor Scaling and Processor Design . . . 9
2.1.1 Pipelining and Out-Of-Order Processors . . . 10
2.1.2 SIMD and Multi-Core Processors . . . 15
2.2 CMOS Power Consumption . . . 16
2.2.1 Dynamic Voltage and Frequency Scaling . . . 17
2.2.2 Voltage Guard Band . . . 18
2.2.3 DVFS Policies . . . 21
2.3 Dennard Scaling and Leakage-Limited Scaling . . . 23
2.3.1 Power-Limited DVFS . . . 25
2.3.2 Energy-Efficient Accelerators . . . 27
2.3.3 Heterogeneous Systems . . . 29

3 Performance Implications of AVX2 and AVX-512 . . . 33
3.1 Local Impact on Performance and Power . . . 34
3.2 Frequency Management for AVX2 and AVX-512 . . . 36
3.2.1 AVX2 and AVX-512 Frequency Levels . . . 37
3.2.2 Frequency Reduction . . . 38
3.2.3 Delayed Frequency Increase . . . 42
3.3 AVX Overhead . . . 44
3.3.1 Local AVX Overhead . . . 44
3.3.2 Remote AVX Overhead . . . 46
3.3.3 Implications of Frequency Change Delays . . . 47
3.3.4 Implications of Hyper-Threading . . . 48
3.3.5 Impact on Real-World Software . . . 48
3.3.6 Impact of Speculative Execution . . . 52
3.4 Information Available at Runtime . . . 52
3.5 Applicability to Other Microarchitectures . . . 55

4 Runtime Profiling of AVX2 and AVX-512 Overhead **57**
4.1 Existing Profilers 58
4.2 Detecting Unnecessary Frequency Reduction 60
4.2.1 Frequency Reduction Sampling 61
4.3 DVFS Performance Prediction Model 64
4.3.1 Existing Models 65
4.3.2 Stall Cycle Counting for Intel Skylake-SP 68
4.4 Source of Overhead 70
4.5 Evaluation 71
4.5.1 Profiler Accuracy 72
4.5.2 Impact of Instrumentation Perturbation 73
4.5.3 Profiler Overhead 74
4.5.4 Overhead Source Analysis 76
4.6 Discussion 77
4.6.1 Optimized Frequency Reduction Sampling 78
4.6.2 Proposed Hardware Changes 79

5 Viability of Improved DVFS Policies **83**
5.1 Parallels to Dynamic Power Management 84
5.2 Characterization of AVX Frequency Changes 87
5.2.1 Frequency Reduction Overhead 87
5.2.2 Frequency Boost Overhead 89
5.2.3 Break-Even Time of Frequency Changes 91
5.3 Simulating Improved Frequency Scaling 92
5.4 Discussion 96

6 Separating AVX-512 and Non-AVX-512 Code **99**
6.1 Existing Mitigation Techniques 100
6.2 Core Specialization for AVX-512 Applications 102
6.2.1 Scheduling Policy 104
6.2.2 Number of AVX-512 Cores 106
6.2.3 Detecting AVX-512 Code 107
6.2.4 Detecting Non-AVX-512 Code 110
6.3 Implementation 112
6.3.1 Choice of Scheduler 112
6.3.2 Tripled Run Queues 113
6.3.3 AVX-512 Feature Detection 114
6.4 Estimation of Effectiveness 115
6.5 Evaluation 117
6.5.1 Effectiveness 117
6.5.2 Overhead 121
6.5.3 Short Non-AVX-512 Phases 121
6.5.4 Estimation of Effectiveness 122
6.6 Discussion 123
6.6.1 Reducing Migration Overhead 123

6.6.2 NUMA Support 124
6.6.3 Imprecise Detection of Power-Intensive Code 124
6.6.4 Proposed Hardware Changes 125

7 Scheduling for Improved Performance Isolation 127
7.1 Fairness and Performance Isolation 128
7.2 Quantifying AVX2 and AVX-512 Performance Isolation Problems 130
7.3 Metrics for Performance Isolation 131
7.4 Modified CPU Time Accounting 134
7.4.1 Attribution of Frequency Changes 134
7.4.2 Estimating Remote AVX Overhead 135
7.4.3 Frequency Reduction Compensation 138
7.5 Thread Mobility 139
7.6 Evaluation 141
7.6.1 Setup 142
7.6.2 Performance Isolation 142
7.6.3 Overhead 144
7.6.4 Comparison with CFS 145
7.7 Discussion 146
7.7.1 Attribution of Remote AVX Overhead 146
7.7.2 Throttle Cycles 147
7.7.3 Compatibility with Profiling Tools 147
7.7.4 NUMA-Support 148

8 Conclusion 149
8.1 Future Work 150

Bibliography 151

List of Figures 169

List of Tables 171

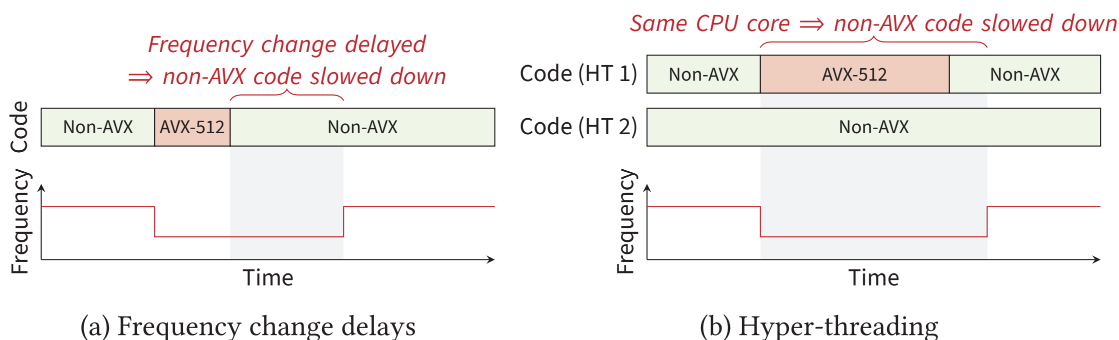
Listings 173

1 Introduction

For a long time, technical progress in the area of semiconductor manufacturing meant that the power density of processors at full load remained constant despite rising CPU frequencies and exponentially increasing transistor density [64]. In recent years, this trend stopped as CPU manufacturers were unable to further reduce operating voltages – such voltage reduction had been central to the required continuous improvements to power efficiency [243]. Increasing power densities soon reached the limits of economically viable cooling solutions, requiring CPU manufacturers to reduce operating frequencies or to leave parts of the chip inactive. In this scenario, the desire of CPU manufacturers to maximize performance has led to the introduction of a range of techniques to fully exploit the available power budget. For example, overall power dissipation is reduced when CPU cores are disabled. Techniques such as Intel Turbo Boost [117] use the resulting thermal headroom to increase CPU frequencies whenever some CPU cores are inactive.

Similarly, power dissipation depends on the energy consumption of individual instructions. To maximize performance for all types of code, a processor can execute code consisting of only simple instructions at a higher frequency than code involving energy-intensive complex instructions. The first widely available processors showing such behavior were those using the Intel Haswell, Broadwell, and Skylake microarchitecture which reduced their operating frequencies when executing code using AVX2 and AVX-512 instructions [76, 77, 213]. AVX2 and AVX-512 are single-instruction multiple-data (SIMD) instruction set extensions operating on large 256-bit and 512-bit vector registers supporting, for example, up to 16 32-bit multiplications in a single instruction. Due to the complexity of these instructions, the Skylake-SP CPUs targeted by this thesis provide three discrete sets of operating frequencies – a low "AVX-512" frequency level, an intermediate "AVX2" frequency level, and a high non-AVX frequency level [113, p. 2-14]. Upon execution of 512-bit or 256-bit SIMD instructions, individual CPU cores switch to the appropriate level and revert the change after a period where no such power-intensive instructions are executed.

This frequency management scheme results in increased performance for many workloads, in particular those only consisting of simple instructions with low energy consumption. Such workloads would otherwise have to be executed at unnecessarily low frequency levels. In other workloads, the frequency reduction triggered by AVX2 and AVX-512 – or, in future CPUs, other similarly power-intensive instructions – has a substantial performance impact. This performance impact is most pronounced when, as shown in Figure 1.1, power-intensive code causes other, less power-intensive code to be executed at lower frequencies, which we term *remote frequency reduction*. This remote frequency reduction causes what we, in the case of AVX2 and AVX-512, call *remote AVX overhead* – code which uses power-intensive instructions slows other code down. Remote AVX overhead has been observed by related work in web server scenarios where AVX-512 in cryptography routines caused the whole web server to be slowed down by 10% [141]. Remote AVX



(a) Frequency change delays

(b) Hyper-threading

Figure 1.1: Power-intensive instructions such as AVX2 and AVX-512 cause the CPU frequency to be temporarily reduced. This frequency reduction also affects less power-intensive code that directly follows on the same hyper-thread (a) or that is executed in parallel on the sibling hyper-thread (b) The resulting slowdown is called remote AVX overhead.

overhead was also demonstrated in a multi-application scenario where a latency-critical was slowed down by up to 30% by a concurrently running machine learning workload using AVX-512 [55]. Apart from an obvious impact on overall system throughput, the multi-application scenario also shows how remote AVX overhead impacts performance isolation. Concurrently executed processes could easily cause the latency-critical application to fail to meet real-time requirements.

In this thesis, we provide a detailed analysis of remote AVX overhead. We show how it effects a wide range of workloads and why it is hard to predict as it is caused by the interaction of multiple software components. Most importantly, though, we show that the impact of remote AVX overhead can often be mitigated. To this end, we describe a set of tools to quantify and prevent remote AVX overhead. First, we describe a profiler that creates accurate estimates of the amount of remote AVX overhead by periodically pausing individual CPU cores and analyzing the resulting frequency changes. This profiler is able to measure remote AVX overhead with an average error of only 2.2 percentage points and is able to determine the reason for remote AVX overhead. We also show how information from the OS such as predictions of the time of the next AVX2 or AVX-512 phase can help the CPU to deploy better frequency scaling policies. While such improved frequency scaling policies are very effective for systems without hyper-threading as they remove frequency change delays, all server CPUs with support for AVX-512 also support hyper-threading. For these systems, we therefore show how the scheduler can be modified to reduce the co-scheduling of power-intensive and less power-intensive code that leads to remote AVX overhead. Limiting AVX-512 code to a subset of the available CPU cores results in an average reduction of remote AVX overhead by 90.4% over a wide range of workloads. Due to limitations of existing CPUs, our prototype is ineffective for workloads with AVX2 code, though. To mitigate the remaining performance isolation problems – in particular those in workloads with AVX2 – caused by remote AVX overhead, we finally describe how existing fair schedulers can be modified to prioritize tasks affected by remote AVX overhead to counteract the slowdown. This approach is, for example, able to reduce the average slowdown of non-AVX applications caused by AVX2 applications by 70.8%.

Our work leads to mainly two conclusions: First, the performance and performance isolation improvements achieved by our prototypes show how important it is for the OS to be designed with remote AVX overhead or similar effects in power-limited systems in mind. We expect techniques such as those described in this thesis to become even more important in the future as CPU manufacturers will likely integrate more accelerators similar to AVX-512 into CPUs in an effort to improve performance and power efficiency [234]. Second, our prototypes demonstrate the requirement for improved interfaces between the OS and the CPU. For example, while our prototypes are functional, they often suffer from limited effectiveness due to a lack of information provided by the CPU. We propose that future CPUs should provide more fine-grained information about power requirements of currently executed code and should give the OS more control about how and when frequency changes occur.

1.1 Scope

This thesis focuses on the effects of AVX2 and AVX-512 instructions on the performance of applications running on Skylake-SP server CPUs. Power-intensive instructions cause similar effects on many other CPUs. Generally, these CPUs lie outside of the scope of this thesis. For example, the predecessors to Skylake CPUs – Haswell and Broadwell CPUs – also feature frequency reduction during execution of AVX2 code [76]. Also, later server CPUs using the Cascade Lake or Ice Lake microarchitecture feature similar frequency management as Skylake-SP CPUs [188], as do many recent desktop CPUs [58]. We expect most of the techniques described in this thesis to be applicable to microarchitectures similar to Skylake-SP.

Our findings are, in contrast, largely not applicable to other types of power-intensive accelerators that behave differently to the function units used for AVX2 and AVX-512. These function units are tightly coupled accelerators which are placed within individual CPU cores [51]. Other accelerators such as graphics processing units (GPUs) are placed further away from CPU cores and constitute loosely coupled accelerators. Despite their different behavior – for example, these accelerators generally operate asynchronously to the CPU – they sometimes show similar behavior. For example, on many mobile systems, CPU cores and GPUs are placed on the same system-on-chip (SoC) and therefore share their power budget. While the GPU is used, these SoCs often allocate less power to the CPU cores causing a temporary frequency reduction [34]. This frequency management is similar to that on current Intel server CPUs in that the CPU frequency is reduced when a power-intensive accelerator is used. Therefore, similar effects become visible – for example, performance isolation is impacted if a GPU application causes a CPU frequency reduction that affects another application. Consequently SoCs with loosely coupled accelerators may benefit from similar approaches as those presented in this thesis. However, the differences between tightly and loosely coupled accelerators mean that our concrete designs are not directly applicable. We therefore leave the development of similar techniques for power-limited systems with loosely coupled accelerators as future work.

1.2 Contributions

Our work makes the following contributions:

- We provide the most extensive analysis of effects of AVX2 and AVX-512 instructions on performance to date. We introduce the distinction between local and remote frequency reduction as well as between local and remote AVX overhead and provide direct measurements of remote AVX overhead in a wide range of workloads. In addition, we provide the most extensive analysis of the frequency transitions caused by AVX2 and AVX-512 instructions do date.

- We describe a profiler to determine, for a given workload, magnitude and reasons for the overhead caused by remote frequency reduction on existing hardware. Such a profiler facilitates software development as it uncovers the negative performance impact of AVX-512 code sections and enables software developers to make better design decisions. The profiler also facilitates system administration as it provides information on whether counter-measures against remote AVX overhead should be applied.

- We show that counter-measures against remote AVX overhead largely have to be implemented in software. Improved frequency scaling policies implemented in the CPU itself show little potential for performance improvement on systems with hyper-threading.

- We describe how core specialization can be used as a technique to reduce remote AVX overhead by reducing the co-scheduling of power-intensive and less power-intensive code. In addition, we describe a mechanism to identify power-intensive code based on register accesses and heuristics. This mechanism is architecture-agnostic and can be used on any CPU that is missing improved interfaces to identify power-intensive code.

- We describe simple modifications to existing time-based fair schedulers which improve performance isolation in situations where one task causes a frequency reduction during execution of another task. This approach complements other approaches to mitigate the impact of remote AVX overhead which often are not 100% effective.

- We sketch a range of improvements to future power-limited CPUs which enable the OS to apply more effective and efficient counter-measures against remote frequency reduction.

1.3 Student Theses and Publications

Parts of this thesis are directly or indirectly based on previously published information. In particular, we supervised a number of student theses which, to the extent described in the following, influenced this thesis:

- In his bachelor's thesis *Analysis and Optimization of Dynamic Voltage and Frequency Scaling for AVX Workloads Using a Software-Based Reimplementation* [131], Yussuf Khalil performed a detailed analysis of AVX-512 frequency management with a focus on transitions between the frequency levels. While our experiments performed in Section 3.2 to determine the steps involved in these frequency transitions use a different setup, they are heavily influenced by his work. In addition, Yussuf Khalil tested the viability of improved DVFS policies with eager frequency changes. We used parts of the resulting source code for the experiments described in Chapter 5.

- In his bachelor's thesis *Stage-Aware Scheduling in a Library OS* [216], Christian Schwarz showed the viability to partition applications and to execute parts on separate CPU cores. Despite chasing a radically different goal, our work on core specialization in Chapter 6 greatly benefited from the resulting insights on cache behavior and thread migration efficiency.

- In his master's thesis *Constructing a Library for Mitigating AVX-Induced Performance Degradation*, Ioannis Papamanoglou described how to implement core specialization for AVX-512 workloads completely in user-space [187]. Ultimately, his work uncovered the limitations of such user-level designs. Consequently, our work on core specialization in Chapter 6 is not based on his prototype but rather uses an approach that operates on unmodified applications.

- In his master's thesis *Core Specialization for AVX-512 Using Fault-and-Migrate* [35], Peter Brantsch described improvements to core scheduling that allow the OS to automatically detect AVX-512 code regions based on register use and additional heuristics. While the mechanism to detect the beginning of AVX-512 phases had been suggested in our earlier work [83, 84], Peter Brantsch came up with the innovative method to detect the end of AVX-512 phases based on system calls performed by the application. Parts of our implementation described in Chapter 6 are based on his code.

- In his bachelor's thesis *Faires Scheduling unter Beachtung von AVX-512-Frequenzeffekten* [164], Philipp Machauer constructed a prototype for fair scheduling of workloads where some tasks use AVX-512. His work based on the Linux Completely Fair Scheduler showed the importance of fast load balancing. Our work in Chapter 7 does not use his prototype, but greatly benefits from his analysis concerning the attribution of frequency changes.

Furthermore, parts of this thesis have previously been published as conference papers, workshop papers, or technical reports:

- In our workshop paper *AVX Overhead Profiling: How Much Does Your Fast Code Slow You Down?* [87], we described a profiler that determines the amount of remote AVX-512 overhead in heterogeneous workloads. The prototype described in this paper forms the basis for our work in Chapter 4. Improvements described in this thesis include the use of a DVFS performance prediction model and additional steps to determine the reason for remote AVX overhead.

- In our technical report *Dim Silicon and the Case for Improved DVFS Policies* [86], we presented a simulation of improved DVFS policies based on information provided by the operating system. This report constitutes a preliminary version of our work presented in Chapter 5 and is partially based on the aforementioned bachelor's thesis by Yussuf Khalil. While the chapter of this thesis generally follows the same methodology, we improve upon our earlier work by simulating systems with hyper-threading with a wider range of workloads.

- In our conference paper *Automatic Core Specialization for AVX-512 Applications* [85], we described how core specialization can mitigate remote AVX overhead by reducing co-scheduling between AVX-512 and non-AVX-512 applications. The design presented in the paper forms the basis for the design presented in Chapter 6 of this thesis. The main difference between the designs is that the variant described in this thesis contains a technique to dynamically determine a suitable number of cores reserved for AVX-512 tasks. We also presented earlier versions of the design as part of a workshop paper [84] and a technical report [83]. These earlier versions formed the basis for the master thesis by Peter Brantsch listed above.

- In our conference paper *Fair Scheduling for AVX2 and AVX-512 Workloads* [88], we described a scheduler that improved fairness in workloads where the frequency reduction caused by some AVX-512 or AVX2 tasks affects other non-AVX tasks. The prototype of this scheduler forms the basis for the scheduler presented in Chapter 7, although this thesis presents a slightly different CPU time accounting policy that vastly improves performance isolation for tasks slowed down by remote AVX overhead.

1.4 Document Structure

The remainder of this document is structured as follows. First, in Chapter 2, we describe the technical background of our work. We describe the microarchitectural design of modern CPUs as well as the developments that led to the current era of power-limited computing before giving an overview over existing techniques to minimize energy consumption and to maximize performance in this power-limited design regime. Then, in Chapter 3, we describe the frequency scaling policy found in recent Intel CPUs with support for AVX-512 and analyze how it affects the performance of heterogeneous workloads consisting of both power-intensive and low-power code. In particular, we quantify the amount of remote AVX overhead for a wide range of workloads. As the technique used to measure remote AVX overhead is labor-intensive, the following Chapter 4 presents an easy-to-use profiler that is able to measure remote AVX overhead during execution of unmodified applications with little overhead. In the remaining chapters we then present techniques to mitigate the effects of remote AVX overhead. First, in Chapter 5, we show that the behavior of the CPU, while not optimal, does not leave much room for improvement. We show that even DVFS policies that use information by the OS about future workload behavior achieve hardly any performance gains in systems with hyper-threading. In Chapter 6, in contrast, we present core specialization as a technique to reduce co-scheduling between power-intensive and

low-power code and to reduce remote AVX overhead. As this approach is not able to completely eliminate remote AVX overhead, either, Chapter 7 then presents a scheduling approach to improve performance isolation and to minimize the performance impact on low-power code even if that code is executed at sub-optimal frequencies. Finally, we conclude our work in Chapter 8.

2 Background: Power-Limited Computing

Modern CPUs are the result of many decades of engineering and often feature very complex behavior stemming from extensive hardware optimization. This behavior often makes careful design of the operating system and of system software necessary to prevent negative impact on system performance. For instance, modern CPUs autonomously perform complex frequency management which can substantially reduce the performance of some workloads if not taken into account by the software. Such frequency management is implemented because it improves the performance of most other workloads. Systems have become more and more power-limited due to restricted transistor scaling in the last two decades, so modern CPUs commonly try to select frequencies that maximize performance without violating power limits.

This thesis presents operating system techniques to circumvent some of the negative side effects of this CPU-managed frequency scaling in power-limited systems. For better understanding of the following chapters, this background section therefore briefly summarizes the technological development which has led to this situation and provides a survey of existing CPU power management techniques. Specifically, we first give a brief overview over how transistor scaling according to Moore's law has resulted in increasingly complex processors (Section 2.1). We then summarize how dynamic voltage and frequency scaling (DVFS) can be used to reduce the energy consumption of these processors (Section 2.2). The focus of the techniques presented in the remainder of this thesis, however, lies not on energy but on power, as in recent years the breakdown of Dennard scaling has resulted in a rapid increase of power density. Therefore, we also describe this development as well as existing power management techniques for power-limited systems (Section 2.3). Readers who are well versed in processor architecture and power management are invited to directly skip ahead to this last section.

2.1 Transistor Scaling and Processor Design

While early semiconductor research dates back to the late 19th century [36] and the first transistors were developed in the late fourties of the 20th century [16, 219], the main invention that paved the way for modern computers was that of the integrated circuit a decade later [132]. The integrated circuit allowed the integration of many components on one semiconductor device, which resulted in the commodification of computing. Whereas early computers filled multiple cabinets and were only sold in low numbers [181, pp. 71 ff.], miniaturization of electronics substantially reduced size and costs and made computers an ubiquitous commodity.

Early during this process of miniaturization, Gordon Moore et al. made the observation that the number of components per integrated circuit increased exponentially with time

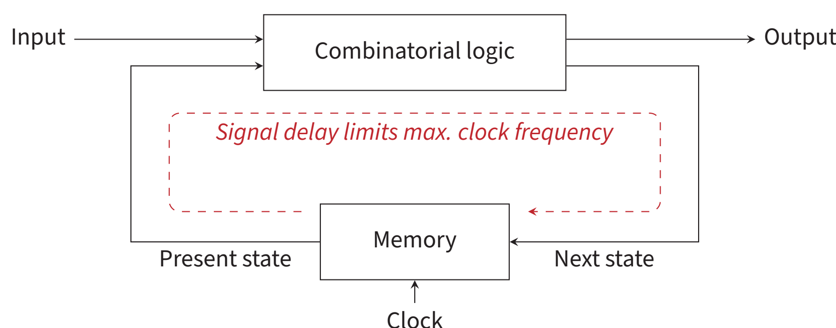


Figure 2.1: Synchronous sequential logic consists of memory elements connected to inputs and outputs of combinatorial logic [241, p. 147]. At the end of each clock cycle, the memory elements capture the state of the combinatorial logic; the data is then fed back into the combinatorial logic during the next clock cycle. The longest signal delay through the combinatorial logic limits the maximum operating frequency [241, p. 159].

and predicted continuous future exponential growth [175]. For most of the last decades, this prediction – commonly known as *Moore's law* – remained true, as the economics of silicon manufacturing resulted in continuous transistor size reductions. Soon, large numbers of transistors could be placed on a single chip, which allowed for the development of single-chip microprocessors [181, p. 113].

Increasing transistor counts not only enabled increasingly complex circuit designs, though. Reduced transistor sizes also resulted in increased energy efficiency as well as reduced signal delays [64]. Reduced signal delays, in turn, increased the operating frequencies of microprocessors. Modern microprocessors are mainly designed as *synchronous sequential logic* which, as shown in Figure 2.1, consists of combinatorial logic as well as memory elements which are connected to some of its outputs to capture its state [241, p. 147]. The memory elements then feed this state back into the combinatorial logic at the next clock cycle. Whereas combinatorial logic is asynchronous, meaning that it changes its output whenever the input signals change, synchronous sequential logic performs operations synchronously to a clock. The maximum operating frequency of such synchronous logic is mainly defined by the longest signal delay caused by the combinatorial logic as its output signals have to be stable by the end of each clock cycle [241, p. 159].

2.1.1 Pipelining and Out-Of-Order Processors

The maximum throughput achieved by a processor depends not only on its maximum operating frequency but also on the average number of instructions executed per clock cycle. When Moore's law made large numbers of transistors available, processor designers therefore used these transistors to implement techniques which exploit *instruction-level parallelism* (ILP) to let the processor execute multiple instructions from a single instruction stream during each clock cycle.

Pipelining is the most basic technique to exploit ILP, where the processor is divided into a number of *pipeline stages* [103, p. C-3] connected via memory elements commonly called

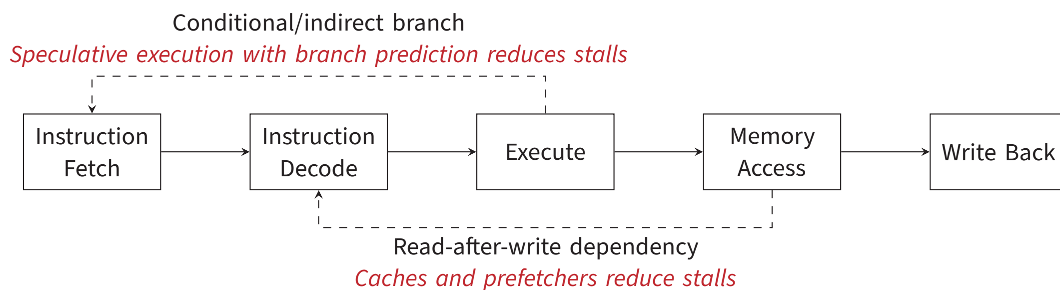


Figure 2.2: The two main reasons for pipeline stalls are conditional and indirect branches as well as memory accesses.[1]Both cause dependencies between the stages of pipelined processors such as the depicted five-stage RISC pipeline [103, pp. C–5 ff.]. CPUs commonly utilize a combination of predictive techniques and caching to improve performance.

pipeline registers [103, pp. C–8 f.]. Each pipeline stage corresponds to a different step during the execution of an instruction, and all instructions travel through all pipeline stages in order. This design allows multiple instructions to be processed by different pipeline stages in parallel as long as there are no dependencies between the instructions preventing such parallel execution. The parallel execution alone does not increase throughput as only one instruction is admitted to the pipeline per cycle. Instead, pipelining increases throughput by allowing substantially higher processor frequencies, as in general the signal delay of the combinatorial logic within individual pipeline stages – i.e., between pipeline registers – is much lower than that of the whole logic of a processor without pipelining. If the pipeline stages are perfectly balanced – i.e., they cause identical signal delay – a processor with n pipeline stages provides almost n times higher throughput provided that the processor is able to admit one instruction per cycle to the pipeline.

In practice, however, the throughput of simple pipelined processors is much lower than one instruction per cycle. As soon as one instruction depends on an earlier instruction, the pipeline is stalled until the earlier instruction has been completed [103, p. C-11]. For example, such dependencies occur if the former instruction reads the value written by the latter, or the latter instruction is a branch instruction which specifies the address of the former as shown in Figure 2.2. Pipeline stalls due to interdependence of pipeline stages limit the amount of ILP that can be extracted by the processor and can substantially impact performance.

The first point at which a pipeline can be stalled is when fetching instructions. In particular, branches constitute a major source for pipeline stalls as the address of the next instruction is only made available at some point during execution of the preceding branch instruction. Unconditional direct branches only stall the pipeline briefly until instruction decoding makes the branch target address available [103, pp. C–21 ff.]. Conditional or indirect branches can potentially cause longer stalls, though, as the conditional branches require the condition to be evaluated whereas the indirect branches require the branch

[1] We ignore register dependencies – i.e., one instruction reads from a register that is written by an earlier instruction – for non-memory operations as the resulting stalls are often comparably short. Many such stalls can be mitigated via simple techniques such as *forwarding* [103, pp. C–16 ff.].

target to be calculated. Modern processors therefore commonly try to predict branch targets and whether conditional branches will be taken, even before the branch instructions are decoded [103, pp. 203 f.]. Depending on the outcome of this prediction, the processors then perform *speculative execution* of either the branch target or the instructions directly following the branch [103, p. 183]. In the case of false predictions, the speculatively executed instructions are discarded and execution resumes at the correct location. Modern branch predictors are highly effective – for example, the Intel Xeon E5-2650L v3 CPU has an average branch miss rate of only 2.198% when executing SPEC CPU2017 [159].

The second major source for pipeline stalls is when an operand to an operation is provided by a preceding instruction, as then execution can only proceed once the operand has been made available. Due to the high latency of external memory, memory accesses are especially likely to stall subsequent instructions. Therefore, modern processors commonly feature multiple levels of *caches* [103, pp. 72 f.] to reduce the average latency of memory accesses and the number of resulting memory stall cycles. In addition, many processors employ prefetchers to anticipate future memory accesses and to hide their latency [103, p. 91]. As a result, while cache efficacy depends on working set size and access patterns, modern caches often feature high overall hit rates. For example, on a system with an Intel Core i7-8700K, 97% of all data accesses performed by the SPEC CPU2017 benchmarks can be served from either L1, L2, or L3 cache [102]. Similarly, *translation lookaside buffers* (TLBs) cache most virtual address translations to reduce the number of memory accesses caused by page table walks [190, p. 502].

With such effective techniques to reduce the likeliness of pipeline stalls, the main limiting factor remaining for the performance of simple pipelined processors is that they can execute at most one instruction per cycle. To increase the amount of ILP that can be exploited, modern processors therefore commonly implement *multiple-issue* designs [103, pp. 193 f.]. These processors provide multiple *functional units*, i.e., different, potentially specialized execution pipelines. Whenever dependencies between the instructions permit, the processors can execute multiple instructions per cycle using these functional units. Multiple-issue processors can be categorized as either statically scheduled or dynamically scheduled processors depending on how the execution order of instructions is defined.

Static scheduling lets the compiler determine the execution order [103, p. 193]. Statically scheduled processors can be further divided into statically scheduled superscalar processors and very long instruction word (VLIW) processors.[2] For *superscalar* processors, the compiler simply emits individual instructions without specifying whether instructions are to be executed in parallel, so the hardware itself determines which instructions are suitable for parallel execution [211]. Each cycle, statically scheduled superscalar processors check for dependencies between instructions in hardware and submit as many consecutive instructions to suitable functional units as possible [103, pp. 193 f.]. These processors are commonly called *in-order* superscalar processors as the execution order of instructions always equals their program order. The approach is commonly found in processor designs such as the ARM Cortex-A8 for low-power applications or embedded devices due to its

[2] For the sake of simplicity, we ignore Intel's Explicitly Parallel Instruction Computing (EPIC) architecture which combines many features of VLIW and statically scheduled superscalar processors [211]. Today, this design is largely irrelevant due to its lack of economical success.

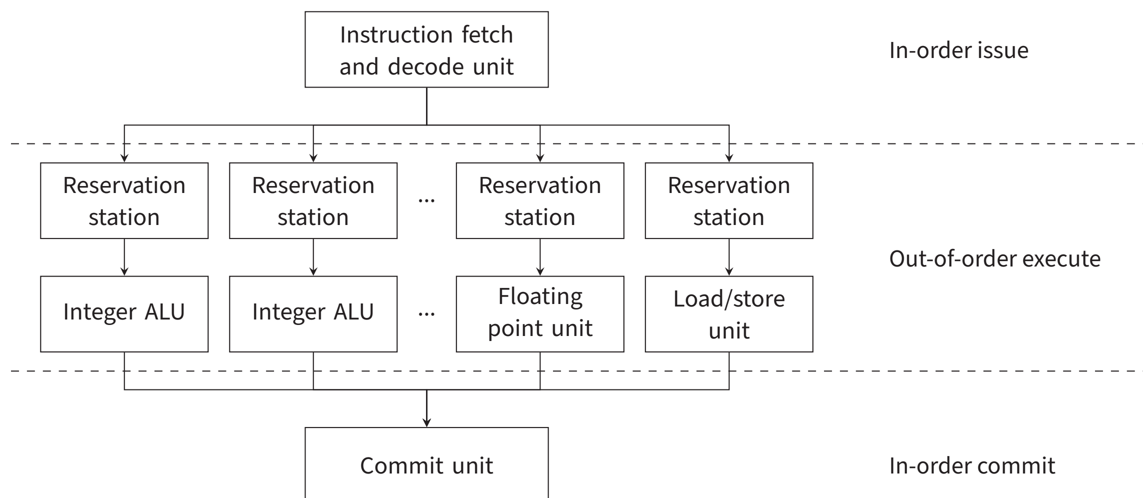


Figure 2.3: Simplified structure of an out-of-order processor – instructions are fetched and decoded in program order and are then processed by one of several functional units whenever all required operands are available [190].

low complexity when compared to dynamic scheduling. VLIW processors achieve even lower hardware complexity by implementing the dependency checks in software. These processors execute only one instruction word per cycle, so parallelism is achieved by the compiler explicitly packing multiple instructions into such an instruction word. This approach has mainly found success in digital signal processors (DSPs) such as the Texas Instruments TMS320 C6000 series.

The main limitation of both types of static scheduling is that instructions are always issued in program order [103, pp. 168 ff.]. As long as a dependency blocks execution of a single instruction, none of the following instruction can be executed, which greatly limits the amount of ILP that can be exploited. Therefore, most modern server and desktop CPUs instead use dynamic *out-of-order scheduling*. These CPUs dynamically reorder the instruction stream when reordering enables the parallel execution of instructions. Figure 2.3 shows the simplified structure of an out-of-order superscalar processor. Whereas instructions are still fetched and decoded in program order, they are then distributed to *reservation stations* corresponding to the individual functional units [190, p. 399]. The reservation stations then admit instructions to the functional units whenever all operands are available. The order in which operations are performed by the functional units does not necessarily match the instruction order in the program, so the results are not sent directly to the register file but rather to a *commit unit* holding a reorder buffer. The commit unit makes results visible in the register file only when it is safe to do so, i.e., when all preceding instructions – in program order – have been executed. In this design, a full reorder buffer or a full reservation station queue presents a potential bottleneck as no further instructions can be processed, so processor designers have continuously increased the size of these buffers. The Skylake microarchitecture found in the processors used throughout this thesis provides a reorder buffer with 224 entries as well as a unified reservation station with a queue of up to 97 pending instructions [223] and is, under ideal conditions, able to execute five instructions per cycle [222].

Similar to other pipelined processors, out-of-order processors heavily depend on techniques such as caching or speculative execution for good performance [223]. In particular, speculative execution is highly beneficial because the frontend – i.e., instruction fetch and decode – is operating in program order, which means that stalls caused by control flow instructions affect the whole processor [190, p. 399]. Memory instructions or other instructions with potentially high latency, instead, do not always cause the whole processor to stall – due to its out-of-order nature, the processor can often simply select other instructions for execution. If memory accesses cause particularly long stalls, there is often not enough ILP left to hide the stalls, though. Out-of-order processors therefore generally rely on techniques such as caching to increase available ILP.

For many applications even this combination of speculative execution and caching is not enough to ensure good utilization of modern processor cores which can issue multiple instructions per cycle [103, p. 223]. Therefore, most modern server or desktop processor cores try to additionally exploit *thread-level parallelism* (TLP) via *hardware multi-threading* to increase utilization. Hardware multi-threading allows multiple threads of execution to share one processor core by presenting multiple logical processors to the operating system. If one thread of execution stalls due to a memory access, another can use the idle functional units.

Different methods exist to implement hardware multi-threading. For simple statically scheduled processors, in particular, hardware multi-threading can be implemented via fine-grained and coarse-grained multi-threading [103, p. 224]. Fine-grained multi-threading switches to a different thread on each clock cycle, skipping threads that are stalled by, for example, memory accesses. Coarse-grained multi-threading only switches to a different thread on the occurrence of an expensive pipeline stall. Coarse-grained multi-threading is unable to hide the cost of short stalls because each context switch to a different thread incurs substantial cost as the processor pipeline has to be refilled. Most modern out-of-order processors, instead, use a variant of fine-grained multi-threading called *simultaneous multi-threading* (SMT) where during each clock cycle the processor frontend potentially fetches and decodes instructions from multiple hardware threads and submits them to the reservation stations [103, pp. 224 f.]. In contrast to coarse-grained multi-threading and similar to fine-grained multi-threading, SMT can hide the cost of even short stalls as no pipeline refills are required. Unlike fine-grained multi-threading, SMT can issue instructions from multiple threads each cycle, thereby achieving better processor utilization during cycles where single threads alone would not be able to provide the maximum amount of instructions to the functional units. The Intel processors used in the remainder of this thesis implement SMT, which Intel calls hyper-threading. In most cases, hyper-threading substantially improves performance. For example, Hebbar and Milenković [102] analyzed the scalability of most benchmarks from the SPEC CPU2017 benchmark suite on a system with an Intel Core i7-8700K CPU. When executing two threads per physical CPU core, they measured an average speedup of 14% compared to a setup with one thread per physical CPU core.

One downside of simultaneous multi-threading is that a hardware thread is able to affect the performance of other hardware threads within the same physical processor core. In the past, shared functional units and shared L1 caches have shown to pose a security risk as they can be used to leak information via side channels and covert channels [247]. In addition, all

hardware threads of a physical processor core are executed at the same processor frequency, so frequency changes caused by one thread can be observed by the other threads which allows for the creation of covert channels [125]. Such security considerations are out of scope for this thesis. Instead, we cover the performance implications when – as described below – a whole physical processor core has to reduce its operating frequency because one hardware thread executes AVX2 or AVX-512 instructions.

2.1.2 SIMD and Multi-Core Processors

The techniques described above were made possible by the continuously increasing number of transistors made available by miniaturization according to Moore's law. Over time, however, it became harder to improve the performance of individual processor cores. Increasing numbers of transistors were required to extract more ILP – or, in the case of hardware multi-threading, TLP – which resulted in an increasingly inefficient use of silicon and power [103, p. 344]. This development is reflected by *Pollack's Rule* which states that the performance increase through microarchitectural improvements follows the square root of the processor complexity [78]. As a result, processor designers additionally started to exploit data-level parallelism and allowed for more thread-level parallelism via multi-core designs.

Data-level parallelism (DLP) is present in situations when the same operation is applied to many data items [103, p. 9]. The two main techniques to utilize data-level parallelism in modern systems are external accelerators such as graphics processing units (GPUs) as well as *single instruction multiple data* (SIMD) instruction set extensions [103, p. 262]. Given the CPU-centric nature of this thesis, this section will focus on the latter. SIMD instruction set extensions such as the x86 extensions MMX, SSE, AVX, AVX2, or AVX-512 provide large vector registers as well as instructions operating on these registers [103, pp. 282 ff.]. The instructions commonly divide the registers into identically-sized segments and then perform the same operation on each of the segments. For example, a single AVX-512 `PADDQ` add instruction can perform up to eight 64-bit integer additions at the same time [115, pp. 4–204 f.]. These operations are commonly performed in parallel by multiple ALUs in order to increase performance [190, p. 649]. For problems with sufficient data-level parallelism, such parallel execution provides substantial throughput increases at modest power [41, 93, 112] and silicon area [223] increases. See Section 3.1 for a more detailed discussion of the impact of AVX2 and AVX-512 in particular.

While SIMD sometimes provides large performance increases, many problems do not provide the required data-level parallelism and do not benefit from SIMD. Modern processors therefore also try to exploit thread-level-parallelism by providing multiple processor cores, as using transistors for additional cores ideally results in an almost linear performance improvement, in contrast to the diminishing returns from microarchitectural improvements mentioned above [78]. In practice, the performance gain of multi-core scaling is limited by a number of factors. First, the cores share memory, yet the view of memory has to be coherent among the cores, i.e., each write to memory must be made visible to all cores [177, p. 11]. This requirement adds additional circuitry and increases the rate of cache misses [103, p. 372] as well as the latency of memory accesses [103, p. 350]. Second and perhaps more importantly, the degree of parallelism in any given workload is

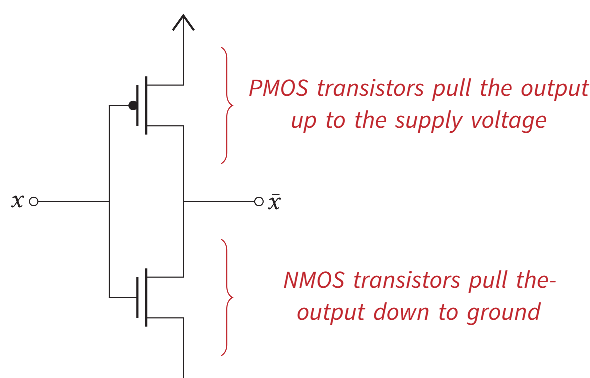


Figure 2.4: CMOS logic consists of a net of PMOS transistors and a net of NMOS transistors. Ideally, only one of the two nets is conductive at any point in time leading to very low static power consumption.

limited by *Amdahl's law* which states that the fraction of sequential code in the workload – either due to non-parallelizable parts of the problem or due to communication – limits the achievable speedup through parallelization [103, p. 46]. The availability of workloads with sufficient thread-level parallelism made processors with large core counts particularly common in server systems, where the size of the datasets and the natural request-level parallelism of the executed applications ensures good utilization of all cores [103, p. 344]. For example, recent Intel server CPUs provide up to 40 cores [75]. Multi-core architectures have also become common in desktop and mobile systems, mainly as they also increase power efficiency: As described in the next sections, a single core running at a frequency f often requires substantially more energy than two cores running at $f/2$.

2.2 CMOS Power Consumption

While performance improvements required increasing CPU complexity, this complexity had a substantial negative effect on power consumption. Modern processors are commonly designed using *complementary metal-oxide semiconductor* (CMOS) circuits with power consumption consisting of three components, namely leakage, short-circuit, and switching power [186, p. 143]:[3]

$$P = P_{leakage} + P_{short-circuit} + P_{switching} \tag{2.1}$$

These three components are caused by different effects during the operation of CMOS circuits. CMOS logic consists of a net of PMOS transistors as well as a net of NMOS transistors – at any time, either the PMOS transistors are supposed to pull the output up to the supply voltage or the NMOS transistors are supposed to pull the output down to 0 V [174]. As an example, Figure 2.4 shows a simple inverter where the two nets consist of only one transistor each. Ideally, at any time and in any part of the circuit, only one of the two nets conducts current, leading to very low static power consumption. In

[3] Sometimes, glitching power is named as a fourth component [186], representing the switching and short-circuit power of unwanted temporary voltage level transitions caused by the delay of individual signals. For the sake of clarity, we do not distinguish between necessary and unwanted transitions.

practice, there are two exceptions [186, p. 143]: First, short-circuit power occurs during signal transitions when both PMOS and NMOS nets conduct, creating a short-circuit path between the power supply and 0V. Generally, short-circuit power has been much lower than switching power [253]. Second, transistors allow small amounts of leakage current even when switched off, which results in leakage power even in the absence of any transistor switching activity. Leakage currents depend on the transistor threshold voltage, with lower threshold voltages resulting in an exponential leakage power increase [221, p. 1]. Section 2.3 discusses how this effect started to pose a substantial challenge in recent years.

Throughout the history of CMOS circuits, chips were designed to provide low static power – instead, switching power was responsible for a majority of the overall power consumption.[4] Whenever a logic gate – i.e., a set of transistors that drive a single output – switches its output value, the capacitance connected to the output of the gate needs to be either charged or discharged [186, pp. 147 ff.].[5] The switching power consumption caused by the resulting current depends on the total capacitance C, the operating voltage V, the circuit's operating frequency f and the average switching activity A [221, p. 11], with A being the average chance of each individual transistor to switch its state during a single clock cycle:

$$P_{switching} = CV^2Af \tag{2.2}$$

These factors are not independent – in particular, voltage reductions result in an increase of the signal delay caused by the circuits [44]. Increased delays, in turn, translate into reduced frequencies because synchronous sequential logic requires signals to be stable at the end of each clock period [241, p. 159] as described in Section 2.1. The following equation describes the approximate relationship between voltage V and maximum operating frequency f, where t_{delay} is the signal delay of the combinatorial logic, V_{th} is the threshold voltage, and k is a factor derived from other transistor properties [107]:

$$\frac{1}{f} > t_{delay} = k\frac{CV}{(V - V_{th})^2} \tag{2.3}$$

2.2.1 Dynamic Voltage and Frequency Scaling

Even before today's era of power-limited computing, many use cases required energy consumption to be efficient. In data centers, efficient energy usage results in reduced power bills, while portable devices profit from increased battery life [221, p. 2]. If we take the equation above and assume that the time required for a task is proportional to the frequency, the energy consumed due to transistor switching activity is as follows, which shows that reducing the operating voltage is a very effective method to reduce energy consumption:

$$E_{switching} = tP_{switching} = CV^2A \tag{2.4}$$

[4] Before the year 2000, large transistors resulted in very low leakage currents [30]. Afterwards, it was assumed that ideally "the leakage power should be only 30% of the dynamic power" [61].

[5] Commonly, this capacitance consists of the output capacitance of the gate's transistors, the gate capacitance of any transistors connected to the gate, as well as the capacitance provided by the wiring between the transistors [2, p. 523].

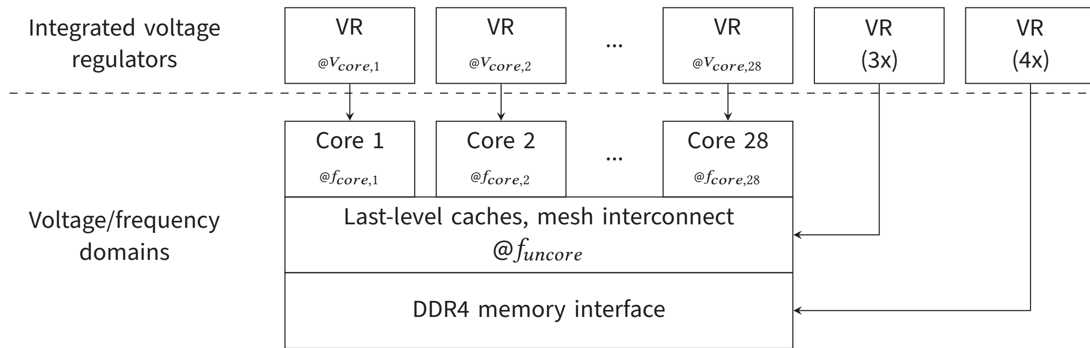


Figure 2.5: Frequency and voltage domains of a 28-core Skylake-SP server CPU (simplified) [231]; the individual CPU cores can operate at different frequencies and are provided with different voltages by integrated voltage regulators.

Even though a voltage reduction increases signal delays, it does not necessarily have to occur at the expense of performance. In particular, while the combinatorial logic has a *critical path* whose delay defines the maximum operating frequency, other paths are of lower complexity and provide *slack*. Slack – the difference between the maximum time and the actually required time – can be used to selectively reduce the supply voltage for the gates constituting these paths without any impact on the frequency [240].

To our knowledge, such fine-grained techniques are rarely used in recent processors, though, likely due to the additional complexity of the power supply circuitry. Instead, these processors employ coarse-grained dynamic voltage and frequency scaling, where either the whole chip or individual processor cores can be configured to operate at reduced voltages and frequencies. For example, on recent x86 CPUs such as the ones targeted by the remainder of this thesis the operating system can select a different frequency for each core at runtime [231, 220]. These processors provide a separate voltage regulator for each processor core as shown in Figure 2.5 to ensure efficient operation. Circuitry shared between multiple cores such as last-level caches, shared interconnects, or input/output circuitry is commonly operated at a separate, independent frequency.

2.2.2 Voltage Guard Band

Most commonly, the operating system requests a specific frequency or – on recent processors with autonomous frequency management – a specific performance level from individual cores [116, pp. 14–5 f.]. Voltage selection is then commonly performed by the processor which automatically selects a suitable voltage whenever the frequency changes. On some systems the operating system can manually configure the voltage when requesting a specific frequency instead. In this case the manufacturer provides a table with frequency/voltage pairs for the operating system to choose from. More fine-grained frequency selection may be possible by interpolating between these pairs [166].

In either case, the voltage applied at the CPU's transistors should be as low as possible to minimize energy consumption, but high enough that the signal delay of the critical path is lower than the clock period. Specifically, the voltage has to be high enough to fulfil the condition from Equation 2.3 during any voltage droops, as only then stable operation

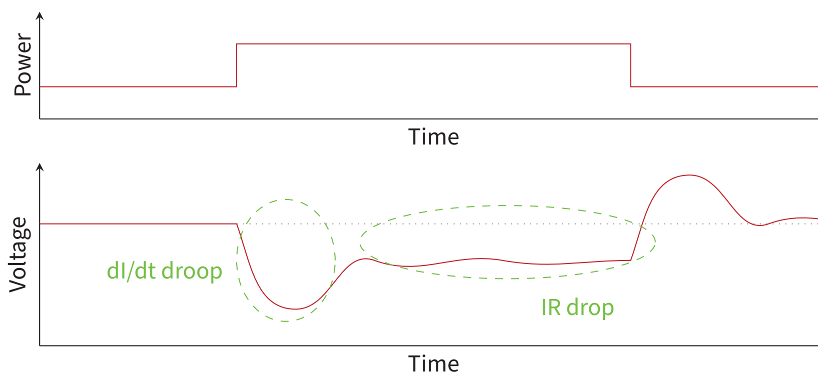


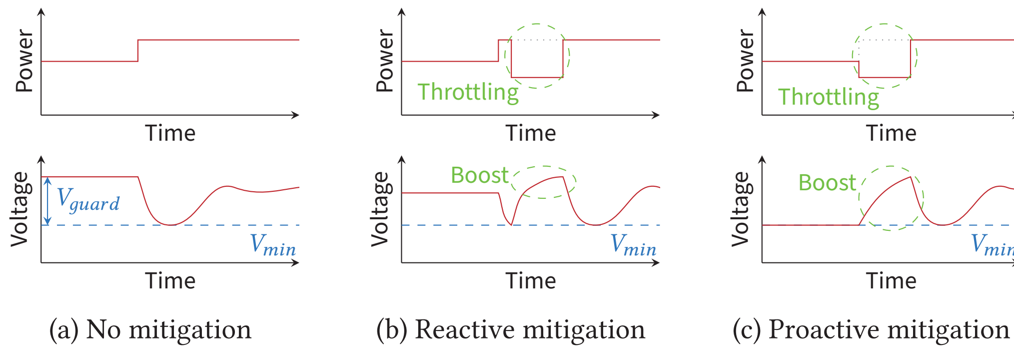
Figure 2.6: The voltage near logic circuits depends on power consumption. While the resistance of supply lines causes the voltage to sag as long as current draw is high (IR drop), the supply line's inductivity and delayed reaction of the voltage regulator to load changes causes temporary voltage droops whenever power consumption increases (dI/dt droop).

is guaranteed. Determining a good voltage for a desired CPU frequency is made difficult by two main challenges, though:

First, the voltage required by the transistors to achieve a specific operating frequency is not constant. Instead, it depends on factors such as temperature or age, where older chips or transistors operating at high temperatures require a higher voltage [46]. Traditionally, CPU designers added a substantial *voltage guardband*[6] on top of the voltage required by new, cool chips to arrive at a conservative selection of operating voltages that meet even worst-case requirements. However, as shown by the equations above, any such voltage increase has a negative impact on energy efficiency. Therefore, techniques have been developed that make the voltage guardband as thin as possible by measuring the impact of factors such as temperature and aging and letting the state of the the chip influence voltage selection [46, 91]. These techniques have proven to be effective without any negative impact on performance and are therefore of no relevance to the remainder of this thesis.

Second, the voltage present at the transistors often differs from the voltage provided by the voltage regulator. The impedance of the supply line between the voltage regulators and logic circuits causes the voltage applied to the logic circuits to fluctuate based on power consumption. Related work differentiates between two effects, *IR drop* and *dI/dt droop*, which are both depicted in Figure 2.6 [148, 153, 123]. IR drop is caused by the supply line's resistance – the voltage drop across this resistance is proportional to the current, so the voltage at the logic circuits is reduced whenever power consumption is high [148]. dI/dT droop is, instead, mainly caused by the supply line's inductivity. Both cause the voltage to temporarily sag when current draw is increased and – similar to effects described above – require the introduction of a voltage guard band to ensure stable operation even when the voltage is temporarily reduced. As both effects can be addressed with similar techniques, we use the term *voltage droops* in the following to refer to both. The guardband required

[6] Some sources alternatively use the phase *timing guardband* [148]. The underlying concept is identical, as any voltage margin results in higher possible frequencies and therefore in timing margin.



(a) No mitigation (b) Reactive mitigation (c) Proactive mitigation

Figure 2.7: In the absence of voltage droop mitigation techniques, the system has to provide a large voltage guardband to prevent instability if load changes cause voltage droops as shown in Figure (a). Figures (b) and (c) show systems which exemplarily achieve reduced guardbands via temporary voltage boosts during load changes. Until the voltage boost has been applied, the system is thottled to prevent excessive voltage droops. Reactive voltage droop mitigation (b) applies such techniques when a voltage droop is detected, whereas proactive voltage droop mitigation (c) predicts load changes and applies the techniques before the onset of a voltage droop. Due to its faster reaction time, proactive mitigation is more effective.

to ensure stability in the face of voltage droops depends mainly on their scale, so a wide range of techniques has been described to reduce voltage droops [46, 126, 148, 33].

Approaches to reduce voltage droops can be categorized as either reactive or proactive approaches. The former category contains approaches which detect voltage droops and then either temporarily boost the operating voltage [46, 148] or reduce the operating frequency [148, 33]. If these reactions occur before the voltage has reached its minimum, the scale of the voltage droop is limited, which allows operation with a reduced voltage guardband as shown in Figure 2.7b. Between the beginning of the voltage droop event and the onset of the mitigation techniques, some voltage droop remains, though. Especially on systems with particularly large power consumption variability, reactive approaches may be ineffective as the remaining voltage droop requires substantial voltage guardbands.

Therefore, some systems with support for very power-intensive vector operations employ proactive approaches against voltage droop, as on these systems vectorized code may require substantially more power than non-vectorized code. For example, the Qualcomm Hexagon DSP predicts future power consumption based on power-intensive microarchitectural events [126]. The pipelined nature of modern processors means that such events are often known multiple cycles in advance. When an impeding voltage droop is predicted, the system can temporarily throttle the processor by, for example, modulating the clock and skipping clock cycles to prevent the voltage droop. As throttling begins even before the voltage droop has started, the power-intensive events covered by the technique do not need to be taken into account at all when calculating the voltage guardband, resulting in lower guardbands and more energy-efficient operation as shown in Figure 2.7c.

Such an approach is also implemented by the Intel CPUs targeted by this thesis [68] which throttle individual processor cores whenever power-intensive AVX2 and AVX-

512 SIMD instructions are executed. This throttling ends whenever a processor core has increased its voltage guardband sufficiently for regular execution of these instructions. The impact of the throttling on application performance is comparably low and is dominated by other effects connected to the execution of AVX2 and AVX-512 instructions. In particular, as shown in Chapter 3, the thermally limited nature of the CPUs causes AVX2 and AVX-512 instructions to often trigger substantial frequency reductions to prevent excessive long-term heat dissipation.

2.2.3 DVFS Policies

As described in the previous section, voltage selection is rarely a task performed by software. The operating system's *DVFS policy* instead requests specific frequencies – or, as mentioned above, performance levels – from the CPU. The task of the DVFS policy is commonly to minimize energy consumption given the performance constraints present in the system. Here, the main opportunity stems from the fact that performance is not always proportional to frequency [221, p. 11]. Instead, DVFS policies can often exploit *slack* which reduces the impact of frequency reductions – while the CPU is not performing useful work, any frequency reduction has no impact on performance.[7] There are two types of slack utilized by different DVFS policies:

- *Scheduling slack* – commonly just called slack – occurs whenever the system finishes tasks earlier than required [221, pp. 22 ff.]. Interactive tasks in particular are usually associated with hard or soft deadlines, yet completing these tasks far ahead of their deadline may not yield any additional benefit. In such scenarios, energy can be saved by running the tasks at a lower frequency so that they exactly meet their deadline [73]. In real-time systems, the scheduler typically knows about the worst-case execution time as well as the deadline of all tasks and can therefore easily calculate slack and select suitable frequencies for individual tasks [182]. In contrast, such information about tasks is not available in general-purpose operating systems. Instead, these operating systems can only observe the idle time caused by slack. They therefore commonly base their estimate of slack on recent processor utilization [250], potentially paired with additional information such as information on communication or CPU utilization patterns [73]. An example for a utilization-based DVFS policy can be found in the Linux kernel [37].
- *Memory slack* is caused by memory accesses that temporarily stall the processor [251]. If memory is operated at a constant frequency independent from the processor frequency, as usually is the case for external DRAM, the constant latency of memory accesses has less impact on performance at reduced processor frequencies. Therefore, whenever energy savings shall be maximized while the system has to provide a specific level of performance, memory-heavy code should be executed at a lower frequency than CPU-bound code. Process cruise control [251], for example, estimates

[7] This slack on the software level should not be confused with the definition of slack on the circuit level given in Section 2.2. While the two concepts are very similar, circuit-level timing and the corresponding slack are not visible to software.

the degree to which individual threads are memory-bound. Whenever a different thread is scheduled, the scheduler then configures the optimal processor frequency for that thread.

While such an approach is able to exploit the differences between threads, its effectiveness is limited for heterogeneous tasks with memory-heavy and CPU-heavy execution phases, as the approach fails to identify such phases if they are sufficiently short. In addition, these phases are rarely aligned to context switches during which the scheduler potentially changes the frequency. Therefore, techniques have been proposed where a profiler identifies frequency-insensitive execution phases and a compiler inserts code to trigger frequency changes at the beginning as well as the end of these code regions [108, 257, 255].

Any approach which bases frequency selection on the amount of memory slack requires a model to predict the performance impact of frequency changes. Process cruise control, for example, uses the average number of memory requests and instructions per clock cycle to characterize threads [251]. This model is purely empirical – it is based on the observation that the selected performance events correlate with the impact of DVFS on performance. In contrast to mechanistic models [72], it does not intend to accurately model any hardware properties. We present a more complete description of the available modelling approaches in Section 4.3.

The goal of all these approaches to DVFS is to conserve energy. In recent years, however, the main use of DVFS has shifted. The increase of leakage power as described in the next section sometimes has made *race-to-halt* policies viable which try to minimize energy consumption by maximizing idle time [12]. At the same time, DVFS has become more and more important to limit power dissipation during phases of high load as described in the following section. This thesis completely focuses on this power-centric use of DVFS in recent CPUs. We therefore ignore the DVFS policies listed above and commonly assume that the system is fully utilized and operates at the highest available frequency. Nevertheless, our work is influenced by the aforementioned work on DVFS policies in mainly two places:

First, similar to energy-centric DVFS policies based on instrumentation [108, 257, 255], power-centric DVFS potentially causes frequent clock changes as it needs to react quickly to power spikes [213]. In both cases, the maximum viable rate of frequency and voltage changes is limited by the overhead caused by the changes [108]. In particular, before or after each frequency change, the affected processor cores temporarily operate at suboptimal voltages while the voltage is gradually ramped up or down [189]. In addition, frequency changes usually require the processor core to be temporarily halted while the clock synthesizer – commonly a phase-locked loop (PLL) – is reconfigured to provide the new frequency. The resulting impact on energy efficiency substantially impacts the efficacy of fine-grained energy-centric DVFS to the point where DVFS policies fail to achieve any energy reduction if they cause too many frequency transitions. Power-centric DVFS approaches such as the policies covered by this thesis experience very similar effects when they boost the processor frequency whenever there is sufficient power headroom. We analyze these effects in more depth in Section 5 where we show that each frequency

Table 2.1: Dennard scaling and leakage-limited scaling employ different scaling factors. Dennard scaling provided constant power density despite smaller transistors and rising frequencies [64]. Excessive leakage power at lower threshold voltages has lead to leakage-limited scaling which suffers from increased power density [243].

Transistor property	Dennard scaling	Leakage-limited scaling
Size	$1/S$	$1/S$
Spatial density	S^2	S^2
Capacitance	$1/S$	$1/S$
Frequency	S	S
Voltage	$1/S$	1
Power per transistor	$1/S^2$	1
Power density	1	S^2

boost is associated with a specific *break-even time* – if the frequency does not remain high for long enough, the frequency boost will, overall, not result in increased performance.

Second, energy-centric DVFS policies and power-centric DVFS policies are equally affected by memory slack. In particular, temporary frequency boosts for low-power code – in our case, code without AVX2 and AVX-512 instructions – are less effective for programs that experience many memory stalls than for CPU-limited programs. Both our profiler described in Chapter 4 as well as our fair scheduler described in Chapter 7 need to estimate this impact of frequency changes on performance. Therefore, we present a suitable model in Section 4.3.

2.3 Dennard Scaling and Leakage-Limited Scaling

Initially, the main goal of DVFS was to reduce energy consumption – power, on the contrary, was of no particular interest. DVFS policies commonly assumed that they could select any frequency supported by the CPU. This situation changed with the breakdown of *Dennard scaling*, after which power limits became increasingly problematic. Eventually, power-centric DVFS became necessary to achieve good performance in the presence of strict power limits as described in the following.

Dennard scaling is a set of scaling rules proposed by Dennard et al. [64] to keep power density constant despite the rapidly increasing transistor density caused by miniaturization according to Moore's law. Table 2.1 lists these scaling rules. If, for example, the transistor size is reduced by the factor S, the spatial transistor density is increased by S^2. The reduced dimensions result in a capacitance reduction which in turn allows the frequency to be increased by S, resulting in a substantial performance boost. To keep power density constant despite the frequency increase and the larger transistor count, the operating voltage has to be reduced by S. As stated by Equation 2.3, the maximum frequency depends on the difference between operating and threshold voltage, so this operating voltage reduction requires a corresponding threshold voltage reduction.

In the early 2000s, further threshold voltage reduction became more and more difficult due to increasing leakage. The following equation describes the subthreshold leakage current, where K_1 and n are experimentally derived, V is the supply voltage, W is the gate width, V_θ is the thermal voltage – 25 mV at room temperature – and V_{th} is the threshold voltage [133]:

$$I_{sub} = K_1 W e^{-V_{th}/nV_\theta} \left(1 - e^{-V/V_\theta}\right) \tag{2.5}$$

As the equation shows, subthreshold leakage scales exponentially when the threshold voltage is reduced [133]. Whereas previous CMOS circuits featured negligible leakage power, in the early 2000s it reached the same order of magnitude as dynamic power. If this trend had continued, leakage power would have quickly accounted for the vast majority of overall power consumption. Instead, CPU designers started to balance dynamic power against leakage power to minimize overall power. For example, leakage power constituted 30% of the power required by Intel Xeon Tulsa server CPUs introduced in 2006 [156], a ratio which newer work states is ideal for optimized CPU designs [61]. The exponential nature of subthreshold leakage mostly prevented any further substantial threshold voltage reduction, leading to a modified set of scaling rules shown in Table 2.1. While transistors kept becoming smaller, power per transistor remained constant, resulting in an overall power density increase of S^2 [243].

Any such continuous power density increase is unsustainable given the limits to power consumption imposed by the cooling system. Even though some high-end processor cooling solutions support power densities of up to 350 W/cm^2 [42], for most systems such solutions are not economically viable. Instead, most processors are limited to 100 W/cm^2 [61]. This power density limit combined with the increasing power density of leakage-limited scaling causes what is commonly called *dark silicon* [71]: The transistor properties provided by modern manufacturing processes do not allow the whole chip to be active during operation at its maximum frequency. Assuming frequency and power density to be scaled by S and S^2, respectively, as suggested by the scaling rules listed above, maximum chip area utilization would be scaled by $1/S^2$ [234], i.e., the additional transistors made available via miniaturization would be hardly usable for general-purpose logic that is commonly active.

Forgoing any frequency increase when reducing transistor sizes in a bid to improve chip area utilization does not solve this problem, either. In this case, due to the reduced margin between operating voltage and threshold voltage, lower frequencies only result in minor operating voltage reduction [233]. Consequently, not all of the additional transistors made available via transistor size reduction can be used to implement, for example, additional processor cores. It is commonly assumed that even such constant-frequency scaling results in a maximum chip area utilization of only $1/S$ [234]. Nevertheless, stagnant processor frequencies show that this approach to scaling has been chosen for virtually all processors during the previous decade [61].

The increasing impact of dark silicon has started a new era of power-limited computing, where power consumption commonly is the main limiting factor for performance, with wide-reaching implications for system design. Specifically, this situation has led to mainly two developments. First, a wide range of techniques has been developed to exploit any available thermal headroom and to convert it into performance by temporarily increasing

either chip utilization or frequencies. Second, chip designers have increased their focus on energy-efficient computing, as increased energy efficiency translates into increased throughput. In the following, we give an overview over both to provide the reader with the necessary background information for the remainder of this thesis.

2.3.1 Power-Limited DVFS

When performance is limited by power consumption, maximum performance is achieved when the available power budget is fully utilized. Full utilization is achieved, as described above, when only a fraction of the chip is operated at maximum frequency, but also when larger parts of the chip are operated at substantially reduced frequencies, with the latter setup commonly being called *dim silicon* [110].

On modern multi-core processors, chip utilization is mainly defined by the number of active CPU cores – a multi-core system can either operate few active cores at high frequencies or many active cores at low frequencies. On such a system, dim silicon therefore presents a trade-off between single-core performance or multi-core throughput. To ensure good performance for both single-threaded and multi-threaded code, techniques such as EPI throttling [92], Intel Turbo Boost [117], AMD Turbo CORE [34], or AMD Precision Boost 2 [5] therefore automatically select suitable CPU frequencies based on the number of active cores. Recent Intel CPUs, for example, contain a central power control unit [23] which periodically checks the number of active cores and adapts core frequencies according to a predefined frequency table [118]. In contrast, AMD Precision Boost 2 dynamically measures, among others, temperature and power [5] and executes a closed-loop policy based on this input to increase core frequencies until the processor hits any power limits [59].

These techniques move responsibility for frequency changes from the operating system to the processor, as boost frequencies are commonly selected autonomously by the latter. As a result, automatic frequency boosting techniques are completely oblivious to the software executed on the system. In particular, the CPU does not have any information about task priorities or critical paths in multi-threaded applications. Wamhoff et al. [246] therefore propose giving applications user-space control over frequency selection to boost, for example, lock holders or high-priority threads. While allowing applications to influence frequency boosting can result in substantial performance improvements for some workloads, such approaches are often limited by the CPU and the available interfaces for frequency management. Whereas some AMD processors provide a suitable interface for manual control over boost frequencies, most others, including Intel CPUs, do not. In addition, as described in Section 2.2.3, high frequency transition latencies mean that frequency boosting is not viable when applied for short periods of time. Gouicem et al. [89] and Lawall et al. [145] showed that this is particularly problematic for workloads with limited parallelism that spawn many rather short-lived tasks. Existing schedulers such as the Linux Completely Fair Scheduler (CFS) often place such tasks on CPU cores that were previously idle and were therefore configured to use a reduced frequency by the CPU's autonomous power management or by the operating system's utilization-based DVFS policy. Lawall et al. [145] show that the scheduler should ideally limit the set of CPU cores used for workloads with limited parallelism, both because new tasks should be

placed on CPU cores operating at a high frequency and because limiting the set of active CPU cores often allows the CPU to select a higher turbo frequency.

In this thesis, we explore the viability of short-term frequency boosting and optimized placement of tasks to improve performance in Chapters 5 and 6, respectively. However, as described above we generally assume high system utilization and therefore ignore the impact of variable numbers of active cores on the CPU frequency. Instead, we study the two mechanisms in the context of variable power consumption of individual cores. As described in Section 2.2, the power dissipation of CMOS circuits depends both on the active chip area – affecting capacitance as well as leakage power – and on the switching activity within. Both factors cause the power consumption of individual processor cores to be highly variable. For example, parts of the processor can be *power-gated* – i.e., disconnected from the supply voltage – to reduce the active chip area when specific types of instructions are not in use [29], resulting in power variation depending on instruction set usage. Also, operations with different complexity trigger different switching activity. For example, multiplications often require more power than additions or data transfer instructions [172]. In some cases, even the values that are processed can have a substantial impact on switching activity and therefore power consumption [213].

In general, potential power variation is especially high when processor cores support both very simple instructions, such as bit manipulation on short integers, as well as very complex instructions, such as SIMD floating point instructions. Such processors should therefore take power consumption of individual CPU cores into account when selecting frequencies and should, for example, boost their operating frequencies while executing predominantly low-power code. Recent Intel CPUs with support for the very complex AVX2 and AVX-512 instruction set extensions show an example for such behavior and often execute AVX2 and AVX-512 code at much lower frequencies than less power-intensive code as part of a feature named AVX Turbo Boost [118]. As this behavior – executing power-intensive code at reduced frequencies – is the main target of our work, we dedicate Chapter 3 to an extensive description of the behavior of these Intel CPUs as well as its performance implications.

While AMD CPUs are outside the scope of this thesis, they also take the nature of the workload into account when selecting CPU frequencies [212], as the closed-loop DVFS policy of AMD Precision Boost 2 described above relies on measurements of actual CPU power [5]. Whereas the implementation in Intel CPUs is integrated with the voltage droop prevention techniques described in Section 2.2.2, AMD CPUs at the time of writing appear to lack similar proactive mechanisms to reduce the voltage guardband. While patents filed by Intel [29, 204, 201] currently cover Intel's implementation of proactive voltage guardband reduction and its integration with the DVFS policy, we expect that in the future the increasing need for energy efficiency will force other manufacturers to implement similar integrated techniques.

Finally, while all the techniques above commonly try to maximize performance while staying within the *thermal design power* (TDP) limit, it is often possible to temporarily or indefinitely exceed TDP. TDP specifies the required capabilities of the cooling solution and represents the "highest expected sustainable power while running known power intensive real applications" [70, p. 14]. In practice, TDP is a very conservative estimate of the power dissipation possible without violating chip temperature limits. In particular,

temporary power spikes above TDP do not result in excessive chip temperature if the initial chip temperature before the spike is low enough and if the thermal mass of the cooling solution is able to temporarily buffer the excessive thermal energy. This effect is utilized by *computational sprinting* [199] and Intel Turbo Boost 2.0 [206] which allow the system to temporarily exceed its sustainable power limits by activating additional CPU cores or by increasing CPU frequencies. The remainder of this thesis will largely ignore such techniques as we target server systems where we assume CPU load to remain fairly steady.

In addition, the definition of the TDP is conservative in that it assumes an arbitrary set of active CPU cores dissipating heat. If the active cores are evenly spread across the chip, equal overall power dissipation results in much lower maximum temperatures than if the cores are grouped together [185]. Similarly, a larger number of active CPU cores causes a more even spread of power dissipation across the chip and allows for higher power without violating temperature limits. This observation has led Pagani et al. to propose *thermal safe power* (TSP) [185] as a dynamic power limit that takes the number and placement of active cores into account. The practical relevance of TSP is shown by AMD Precision Boost 2, for example, which takes thermal interaction between different cores into account when selecting CPU frequencies [59]. TDP can also be exceeded if the system frequently cycles between different CPU cores as shown by Hao et al. [104], although we are unaware of any commercial implementation.

All aforementioned techniques to utilize available power budgets have in common that they increase performance variability from system to system. If a system fully utilizes its available power, performance inevitably depends on energy efficiency, yet variation during chip manufacturing means that no two systems provide identical efficiency [215]. Recently, even nominally identical CPUs have therefore often started to provide measurably different performance. While the differences are negligible for most software, they can severely impact software written for large-scale high performance computing installations consisting of thousands of processor cores or more. On such systems, a single slow CPU can cause all other cores to idle while waiting for an intermediate result, which impacts both energy efficiency and overall performance. In this thesis, we focus on single-socket setups and ignore software written for such distributed systems. However, some of the techniques presented in this thesis reduce performance variability and therefore may have a positive impact at scale.

2.3.2 Energy-Efficient Accelerators

When performance is limited by power consumption, techniques which increase energy efficiency usually also increase throughput, since the energy (E) per operation equals power (P) divided by throughput ($rate_{op} = n_{op}/t$):

$$\frac{E}{n_{op}} = \frac{tP}{n_{op}} = \frac{P}{rate_{op}} \tag{2.6}$$

As a result, a wide range of techniques to increase energy efficiency has been proposed in recent years, ranging from new computing paradigms such as approximate computing [98] to increased use of specialized accelerators and new system architectures such as

heterogeneous systems [234]. Whereas we are not aware of substantial use of approximate computing, the latter two approaches increasingly find their way into commercial systems.

Accelerators have already been used to speed up specific calculations a long time before the onset of the dark silicon era. For example, the performance necessary to render complex 3D graphics has long been provided by specialized graphics processing units (GPUs). As specialized accelerators can be optimized for specific problems, they have the potential to provide substantially higher performance than an equivalent software implementation on a general-purpose CPU. Besides the fixed-function 3D math and rasterization hardware provided by early GPUs [168], examples include cryptography accelerators provided by CPUs [105] or application-specific integrated circuits for machine learning [180]. Such accelerators not only increase performance but also energy efficiency, as they commonly either require fewer transistors, trigger less switching activity, or finish their task faster, thereby consuming less energy due to leakage power. This increased efficiency makes specialized accelerators a suitable addition to power-limited processors [234]. For example, Venkatesh et al. [243] propose *conservation cores* – accelerators[8] which do not focus on performance but instead mainly on efficiency – as a technique to substantially improve performance despite increasing amounts of dark silicon. Automated analysis of several applications allows the generation of a set of cores that is able to execute all these applications efficiently [244].

Commonly, the degree of specialization correlates with energy efficiency, with accelerators specialized for a specific workload providing the highest energy savings. However, even accelerators targeting a wider class of workloads or problems can cause substantial performance and energy efficiency improvements, albeit to a lesser degree. For example, SIMD units – as described in Section 2.1.2 – as well as general-purpose graphics processing units support a wide range of data-parallel problems at greater performance and commonly higher energy efficiency than CPUs [41, 109]. Even greater flexibility can be achieved by making the accelerators reconfigurable, allowing the users to place arbitrary data paths in reconfigurable fabric such as FPGAs [51].

For power-limited general-purpose systems, an advantage of specialized accelerators is that they are often inactive for most code [100]. Therefore, they can be placed in silicon that would otherwise have to remain unused due to power limits. Applications that do not use the accelerator are not negatively affected as power to the accelerator can be disconnected. Whenever accelerators are active, they potentially require the chip to operate at reduced frequencies, though. As mentioned above and as described in detail in the next chapter, recent Intel CPUs, for example, reduce their frequency whenever their SIMD function units execute AVX2 or AVX-512 code [113, pp. 2–13 f.].

Frequent power-gating of unused accelerators is associated with energy and performance overhead, so Kumar et al. [142] describe a design which detects short, inefficient phases of SIMD code and automatically devectorizes this code via recompilation at runtime. As this technique does not cover interactions between multiple tasks and because the reliance on

[8] Venkatesh at al. [243] explicitly differentiate between accelerators and conservation cores based on the primary goal to either improve performance or energy efficiency. For the sake of simplicity and as both architectures, in effect, improve system performance for specific tasks, we do not make this distinction.

a just-in-time compiler restricts the applicability to arbitrary workloads, we propose core specialization as a different solution to the same problem in Chapter 6.

Accelerators can be integrated at different places in the system, differing by the type of connection to the processor cores. Based on work by Compton and Hauck on reconfigurable computing [51], we categorize accelerators as either *loosely* or *tightly coupled*. Loose coupling separates the accelerator from the processor via an I/O interface. For example, GPUs are commonly connected to the PCIe bus. The main disadvantage of loose coupling is that the interface used for data transfer between processor cores and the accelerator commonly introduces long latencies. The resulting communication overhead limits the use of such accelerators for fine-grained acceleration of short code sections. Tight coupling, where the accelerator is directly connected to a processor core as a coprocessor or a functional unit, is more suited to such fine-grained acceleration. Examples for tightly coupled accelerators include the SIMD units found in many current CPUs. This thesis focuses on the power management required for tightly coupled accelerators such as AVX2 and AVX-512. Even though loosely coupled accelerators such as GPUs may pose similar challenges compared to those covered in this thesis, they generally require different solutions and are therefore outside the scope of this thesis.

2.3.3 Heterogeneous Systems

In most current processors, all processor cores provide the same set of tightly coupled accelerators. For example, on current Intel server CPUs, all cores are identical and provide SIMD units for AVX2 and AVX-512 as well as cryptography accelerators [223]. However, as mentioned above only a fraction of the code executed on the system actually makes use of these accelerators. In this situation, *heterogeneous systems* – systems, where individual processor cores provide different feature sets or different performance characteristics – often make more efficient use of the available chip area as well as the power budget [143]. While this thesis focuses on systems with homogeneous processor cores, we utilize ideas from research on such heterogeneous systems in several places.

Heterogeneous systems are often categorized as either *single-ISA* or *multi-ISA* heterogeneous systems [143]. Single-ISA heterogeneous systems – often also called performance-asymmetric systems [13] – combine multiple cores with the same *instruction set architecture* (ISA), but with different performance, power, and area characteristics. Such a system could, for example, schedule code on large, complex cores with high single-core throughput if it demands low latencies [143] or if it cannot be parallelized [230, 9]. Highly parallel code or code without particular performance requirements could instead be executed on less complex, yet more energy efficient cores. Especially in mobile devices, single-ISA heterogeneous processors have seen large commercial success in recent years due to the increased energy efficiency and the high compatibility with existing code, as applications do not need to support multiple ISAs. For example, ARM big.LITTLE [90] processors combine high-performance out-of-order cores such as the Cortex-A15 core with more power-efficient in-order cores such as the Cortex-A7 core. More recently, ARM DynamiQ [11] has even allowed smartphones makers to combine more than two types of cores [134].

Applications are allowed to remain oblivious of differences between the cores of single-ISA heterogeneous processors. The system architecture provides a substantial challenge

to the operating system, though, as the scheduler has to create a mapping of tasks onto suitable processor cores. Proposed approaches include schedulers which use a priori information, for example executing system calls [171] or the virtual machine host [144] on a specific core type, as well as schedulers which characterize individual tasks based on the expected performance improvement on fast cores and use this information to select appropriate cores [209].

A special type of single-ISA heterogeneity is created not during system design but rather during chip production. Even if the design of all cores is completely identical, individual cores often provide different maximum frequencies at identical voltage [32] as well as different energy efficiency due to process variation. *Cherry picking* [200] exploits this heterogeneity by, for example, selecting the most energy-efficient cores for the execution of workloads which fail to scale to all available cores. Recent Intel CPUs implement a variant of cherry picking as part of Intel Turbo Boost Max Technology 3.0 which provides slightly increased frequencies for up to four CPU cores [238]. The operating system migrates demanding workloads to these CPU cores to improve performance.

In contrast to single-ISA heterogeneous systems, the cores of multi-ISA heterogeneous systems differ in their instruction sets. Researchers have proposed combining general-purpose cores with identical feature sets, but completely different ISAs such as ARM versus x86, harnessing the different properties of the instruction sets [242]. This approach is rarely used in practice, as the ISA, in contrast to its microarchitectural implementation, has shown to have little impact in the absence of any feature set difference [28], yet software development is complicated by multiple ISAs.

Commercially successful multi-ISA heterogeneous designs commonly combine cores with substantially different feature sets instead, thereby providing specialized cores – faster and more energy efficient – for different types of code. Examples include the Cell processor [106] which combines a general-purpose 64-bit Power processor with an array of eight less complex Synergistic Processor Elements (SPE) specialized for highly-parallel multimedia workloads [124]. Similarly, many processors with integrated GPUs [34], DSPs [49], or similar loosely coupled programmable accelerators constitute multi-ISA heterogeneous processors.

These approaches also suffer from increased software development complexity. Processor cores such as the Cell SPE or GPUs often provide a different memory model [124] or a different threading model [179] compared to general-purpose CPU cores. Therefore, code designed to run on general-purpose CPU cores commonly needs to be restructured to support other types of cores. In addition, the disjoint instruction set means that code, once compiled, is only runnable on one type of core, preventing migration of tasks between different types of cores at runtime. Popcorn Linux [15, 24] tries to solve these problems for sufficiently flexible processor cores such as those of the Xeon Phi [47] capable of running an operating system. Popcorn Linux applications are compiled for multiple architectures, with the compiler inserting code for migration between different types of cores at select points in the application. If different core types do not share a cache-coherent view of memory, Popcorn Linux provides such a coherent view via distributed shared memory.

Heterogeneous systems without cache coherence or with very dissimilar instruction sets are not of particular relevance to our work. The concept of *instruction-based asymmetry* [157] – also called shared-ISA or functional asymmetry [6] – is more relevant to our

work. Instruction-based asymmetry means that all cores share the same base instruction set but differ in specific extended parts of the instruction set. For example, individual cores could provide different closely coupled accelerators [94] or some cores could do without a floating-point unit to conserve energy when executing integer or fixed-point arithmetic [6]. In Chapter 6, we emulate such an architecture on top a regular homogeneous Intel server processor to increase performance for workloads where only parts of the workload utilize AVX-512 SIMD instructions.

On systems with instruction-based asymmetry, all cores can execute tasks which only use the common base instruction sets. Whenever tasks use instructions outside of this set, the operating system needs to migrate the tasks to the appropriate CPU core. Li et al. [157] propose *fault-and-migrate* as a technique to transparently trigger such migrations. Whenever a task tries to execute an instruction that is unsupported by its current core, the core generates an exception and the exception handler moves the task to a suitable core. Li et al. propose to migrate the task back to its original core once the task has not made use of any instructions only supported by its current core for a specific amount of time. We use a very similar approach in Chapter 6 when we restrict execution of AVX-512 tasks on a subset of the CPU cores, albeit with a different heuristic for migrating tasks back to their original (non-AVX-512) core.

3 Performance Implications of AVX2 and AVX-512

In the previous chapter, we showed that complex frequency management is required to extract maximum performance in modern systems with heavily power-limited processors. In particular, the varying power consumption of individual processor cores means that these cores should be operated at different frequencies depending on the program executed. The most prominent example for such behavior can be found in recent Intel processors and their frequency management for AVX2 and AVX-512 SIMD instructions. This chapter provides a detailed analysis of this frequency management as well as its performance implications to provide the required background information for the next chapters where we describe techniques to improve performance and fairness for workloads involving AVX2 and AVX-512.

We first demonstrate the substantial local speedup caused by these instructions as well as the resulting energy savings for problems with sufficient data-level parallelism in Section 3.1 to demonstrate the general usefulness of the instructions. We also show how the instructions increase power consumption. The resulting power consumption variability has motivated Intel to implement a frequency management scheme that executes different types of code at different frequencies to provide high performance for all types of code. We describe this scheme in detail in Section 3.2. While this frequency management policy greatly increases the performance for low-power non-AVX code, it results in AVX2 and AVX-512 code being executed at reduced frequencies. As we describe in Section 3.3, this frequency reduction affects the performance of many workloads. We classify the negative performance impact as either *local* or *remote AVX overhead*. In this classification, local AVX overhead is the impact of AVX2 and AVX-512 frequency reduction on code sections that use AVX2 or AVX-512. We show how this local AVX overhead can be quantified using existing tools and does not pose a particular challenge to software developers. This thesis therefore mainly concentrates on remote AVX overhead which occurs when the frequency reduction caused by AVX2 and AVX-512 affects other code that could have been executed at higher frequencies. We describe these remote effects in Section 3.3.2 and show how, in some cases, non-AVX code in real-world applications can be slowed down by more than 20%. Section 3.4 then contains a description of the information available to the OS about AVX-induced frequency reduction and about the resulting overhead as many of the techniques described in the following chapters require such information. In particular, we show that changes to applications often yield results that are hard to predict. Finally, as this chapter mainly focuses on the Skylake-SP microarchitecture, we discuss the applicability of our work to other microarchitectures in Section 3.5 and argue that an increasing number of future CPUs will show similar behavior due to increasingly stringent power limits.

3.1 Local Impact on Performance and Power

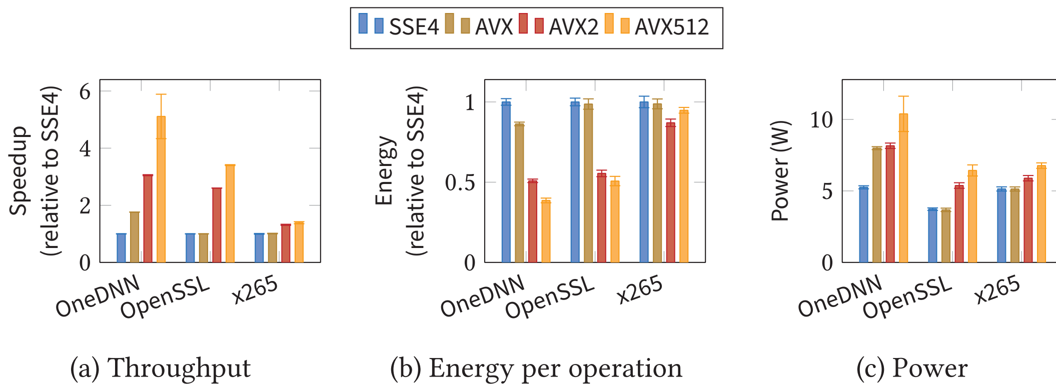
The AVX2 and AVX-512 instruction sets targeted by this thesis are recent SIMD instruction set extensions defined by Intel and implemented by recent x86 CPUs [57]. AVX2 was first supported in 2013 by the Intel Haswell microarchitecture and provided many vector operations on 256 bit vector registers [97]. On top of AVX2, AVX-512 introduced more flexible operations and increased the maximum vector size to 512 bit and was first supported in 2016 by the Intel Knights Landing many-core processor [226] and, more importantly, in 2017 by Skylake-SP server processors [231].

As described in the previous chapter, such SIMD instructions provide increased performance by exploiting data-level parallelism. In the case of AVX-512, a single instruction can execute up to 16 fused multiply-add (FMA) operations on 32-bit floating point numbers [115, p. 5-128]. While not all applications provide the necessary data-level parallelism, AVX2 and AVX-512 have been shown to provide substantial performance improvements for problems from many different domains. For example, compared to an implementation without SIMD instructions, Lemmetti et al. [152] showed that optimization performed on the Kvazaar HEVC video encoder using AVX2 doubled its performance, while Tiwari et al. [236] showed that AVX-512 enabled a 16% performance increase for the x265 video encoder compared to an implementation using AVX2. Similarly, Miralles and Iwamoto [170] optimized routines for software radios and found that using AVX2 instead of the earlier SSE2 SIMD instruction set was responsible for most of a 50% performance improvement for many routines, with AVX-512 often further improving performance. For the ChaCha stream cipher and the Poly1305 message authentication code, Goll and Gueron [82, 81] showed that AVX2 and AVX-512 each have the potential to double performance for sufficiently large messages. Also, Dreseler et al. [69] demonstrated the large impact of optimization using AVX-512 on database query processing.

These performance improvements demonstrate the usefulness of wide vector instructions. However, SIMD instructions such as AVX2 and AVX-512 not only increase performance. Often, energy efficiency is improved as well [41, 112], as reduced CPU time means that most transistors switch less often and that less energy is consumed due to static power dissipation. The increased energy efficiency comes at the cost of increased power dissipation, though. As described in the previous chapter, the increasing complexity of individual SIMD operations means that average transistor switching activity per instruction increases. Assuming equal frequencies, this increased switching activity increases switching power.[1] For the older SSE and AVX instruction sets, this effect has been demonstrated by Hiroshi Inoue [112], who showed that merge sort requires up to 15% more power when implemented with SSE, with similar power required by the AVX implementation.[2] However, we are unaware of any direct measurement of this effect for AVX2 and AVX-512. Such direct measurements are difficult as the processors implementing these instructions are commonly already limited by their power consumption even when they do not execute any AVX2 or AVX-512 instructions. Consequently, AVX2 and AVX-512

[1] Barredo et al. [17] suggest operating the SIMD unit at a lower frequency than the rest of the CPU core to reduce power. We are unaware of any implementation of such a design.

[2] In Hiroshi Inoue's experiments [112], memory throughput limited the performance of the AVX implementation which therefore sometimes required slightly less power than the SSE implementation.



(a) Throughput (b) Energy per operation (c) Power

Figure 3.1: Experiments conducted at 3.5 GHz on a single CPU core show that AVX2 and AVX-512 instructions can be used to increase throughput (a) while reducing energy consumption (b). The AVX2 and AVX-512 variants of the benchmarks require considerably more power, though (c).

code trigger the power management techniques described in the remainder of this chapter, which hide any power increase caused by these instructions.

We perform such a direct measurement of the impact of AVX2 and AVX-512 on power consumption to demonstrate that frequency reduction is often necessary to prevent power limit violations. To isolate the impact of the choice of instructions, we configure the system to operate at a constant frequency, thereby circumventing the autonomous frequency management performed by the CPU. In particular, we configure a single CPU core of a Xeon Gold 6130 CPU to operate at 3.5 GHz – at this frequency, the CPU is able to execute all types of code if only a single core is active [119, p. 15]. We then execute several real-world applications on this core and measure performance, power, and energy consumption. The three applications used for the experiment are the `benchdnn --rnn` benchmark from OneDNN 2.3.2 which benchmarks OneDNN's code for recurrent neural networks [21], the `openssl speed` benchmark from OpenSSL 1.1.1l [184] configured to benchmark OpenSSL's ChaCha20-Poly1305 cipher, and version 3.5 of the x265 video encoder [256] configured to encode the 1080p50 version of the "old_town_cross" test video [258] at the "medium" quality preset. Where necessary, we modify the benchmarks to be able to configure usage of AVX2 and AVX-512. We normalize throughput and energy consumptions based on the results for configurations where the applications only use SSE4 SIMD instructions. As the goal of our power measurements is to demonstrate the impact of the instructions executed on the CPU, we compare power consumption to that of a setup where the CPU only executes the `PAUSE` instruction in a loop. We subtract the power reported by RAPL [116, pp. 14–37 ff.] for this mostly-idle CPU from that reported for the benchmarks to isolate the impact of the individual instruction sets on power dissipation.[3]

[3] On earlier Intel microarchitectures such as Sandy Bridge, RAPL would not have been suitable for this kind of analysis. On these CPUs, RAPL did not directly measure power but rather counted architectural events and used a model to estimate power [206], leading to systematic over- or underestimation depending on the instructions executed by the CPU. Since the Haswell microarchitecture, RAPL appears to rely on direct power measurements and provides much higher accuracy for arbitrary instruction types, including SIMD instructions [111, pp. 75 ff.].

Figure 3.1 shows the results of these experiments. As expected, the throughput of all benchmarked applications benefits from wider SIMD operations, with AVX-512 providing a speedup of 411% and 241% for OneDNN and OpenSSL, respectively, when comparing to SSE4. The performance improvement for x265 is far less pronounced, but noticeable, which matches the findings of related work [236]. Energy consumption follows a very similar pattern. For both OneDNN and OpenSSL, AVX2 and AVX-512 greatly reduce the energy required per unit of work (i.e., per benchmark iteration or per byte processed). However, it can be seen that the relative gain from AVX2 compared to previous instruction sets is much bigger than the gain from AVX-512 compared to AVX2. This result shows that the energy efficiency of AVX-512 implementations is particularly compromised by the increased power consumption of these complex instructions. Whereas, for example, OneDNN requires 5.3 W more when using SSE4 compared to a program which only executes the `PAUSE` instruction, the AVX2 and AVX-512 implementations require 8.2 W and 10.4 W, respectively.

3.2 Frequency Management for AVX2 and AVX-512

The large power variation caused by AVX2 and AVX-512 poses two challenges for CPU power management. First, as described in Section 2.2.2, sudden increases of power consumption can cause voltage droops which impact system stability. Second, as described in Section 2.3.1, the increased power consumption of AVX2 and AVX-512 instructions can potentially cause the processor to exceed thermal limits. These two problems can be solved by operating the CPU with sufficient margins, i.e., by selecting a frequency low enough that even power-intensive AVX-512 code does not produce excessive heat and by selecting a large voltage guardband so that the system remains stable despite large voltage droops. However, for most workloads, this solution comes at the expense of performance and energy efficiency, as less power-intensive code allows for higher CPU frequencies and lower voltages.

Instead of such a such a simplistic approach, recent Intel CPUs provide an extended variant of Turbo Boost that takes the large power variation caused by AVX2 and AVX-512 instructions into account [213]. These CPUs select different frequencies according to the nature of the currently executed code, with AVX2 and AVX-512 code executed at lower frequencies than other less power-intensive code. Reducing the CPU frequency during power-intensive execution phases not only reduces power dissipation but also increases the available voltage guardband, thereby guaranteeing stable operation. At the same time, executing less power-intensive code at a higher frequency and with a lower voltage guardband improves performance and energy for these workloads.

In the following sections, we describe the voltage and frequency management scheme implemented by these processors in greater detail. We focus on the behavior introduced with the Intel Skylake-SP microarchitecture as this microarchitecture introduced support for AVX-512 instructions – to date, all following Intel server CPUs feature similar behavior. See Section 3.5 for a comparison between Skylake-SP and other Intel microarchitectures.

Table 3.1: Frequencies of the Intel Xeon Gold 6130 server CPU – the CPU reduces its frequency when executing AVX2 or AVX-512 instructions [119].

		Turbo / Active Cores				
Level	Base	1-2 cores	3-4 cores	5-8 cores	9-12 cores	13-16 cores
Non-AVX	2.1 GHz	3.7 GHz	3.5 GHz	3.4 GHz	3.1 GHz	2.8 GHz
AVX 2.0	1.7 GHz	3.6 GHz	3.4 GHz	3.1 GHz	2.6 GHz	2.4 GHz
AVX-512	1.3 GHz	3.5 GHz	3.1 GHz	2.4 GHz	2.1 GHz	1.9 GHz

3.2.1 AVX2 and AVX-512 Frequency Levels

To maximize performance, a power-limited CPU ideally always selects the highest possible frequency that does not cause the CPU to exceed any temperature limits. In other words, individual CPU cores should boost their frequency whenever they do not execute any power-intensive instructions. With the Skylake-SP microarchitecture, Intel introduced such a frequency scaling scheme [118]. These CPUs provide three different frequency ranges; the default set of "non-AVX frequencies" is accompanied by a set of reduced "AVX 2.0 frequencies" as well as a set of even lower "AVX-512 frequencies". Table 3.1 lists the frequencies of a Xeon Gold 6130 server CPU for different numbers of active cores. Note that the frequency differences increase when a larger number of cores is active as chip-level power limits become more restrictive when power needs to be shared among more cores.

Unlike the names imply, AVX2 and AVX-512 frequencies are not simply triggered by AVX2 and AVX-512 instructions, respectively. Instead, Intel categorizes SIMD instructions based on register width as wider operations require more power.[4] The AVX-512 instruction set extension introduced not only 512-bit operations, but also many instructions operating on 256-bit and even 128-bit registers. 256-bit and 512-bit instructions are further classified as either "light" and "heavy" instructions [113, p. 2.13]. Heavy instructions include all floating point instructions as well as multiplications, whereas all other instructions are categorized as light instructions. The CPU frequency is mainly selected based on these categories as well as the rate at which the instructions are executed. Heavy 512-bit instructions trigger a transition to the AVX-512 frequency level when executed at a rate of one instruction per two cycles [150]. Below this rate, they trigger a transition to the AVX2 frequency level, as do all light 512-bit instructions. Similarly, heavy 256-bit instructions only trigger a transition to the AVX2 frequency level when executed at a sufficient rate. Infrequent heavy 256-bit instructions and all light 256-bit instructions do not require any frequency reduction.

In some situations, the CPU deviates from this simple mapping between instruction types and frequency levels. In particular, power consumption depends on the state of the register file and valid 512-bit register content can cause instructions to trigger frequency changes even if they nominally operate on more narrow registers. For example, Intel

[4] Documentation from Intel does not make this classification explicit. Instead, 256-bit instructions are called "AVX2", whereas 512-bit instructions are called "AVX-512" [113, p. 2-13]. However, our experiments show that 256-bit AVX-512 instructions, for example, have a different impact on CPU frequencies than 512-bit AVX-512 instructions.

documentation describes that some mixtures of heavy 256-bit and light 512-bit instructions can trigger a transition to the AVX-512 frequency level [113, p. 2-14]. There have even been reports of scalar – i.e., non-SIMD – floating point code triggering a frequency reduction after the upper half of the 512-bit registers has been modified [67]. Conversely, experiments have shown that very short, yet dense sections of 512-bit multiplications only trigger a transition to the AVX2 frequency level, even though these instructions constitute heavy 512-bit instructions [131].

3.2.2 Frequency Reduction

Whereas traditionally DVFS was mainly used to conserve energy and frequency changes could be arbitrarily delayed by the OS, frequency reduction triggered by AVX2 and AVX-512 plays a vastly different role and therefore features vastly different time constraints. In particular, the frequency is reduced to limit the CPU temperature. While both heat spreader and heat sink are able to buffer some of the excessive heat produced by power-intensive instructions at excessive frequencies, eventually a frequency change is required to prevent overheating. AMD CPUs, for comparison, perform frequency selection mainly based on the temperature margin of the CPU – for these CPUs, Tim Schmidt measured reaction times to load changes of multiple milliseconds [212, pp. 16 ff.]. Intel CPUs need to react far more quickly, because on these CPUs frequency reduction also plays an important role as part of proactive voltage droop mitigation. Essentially, the CPU commonly operates at a substantially reduced voltage guardband while no power-intensive instructions are executed in order to improve energy efficiency. As soon as a processor core starts executing AVX2 or AVX-512 instructions, that core needs to immediately reduce its frequency or increase its operating voltage.

To demonstrate that even short stretches of AVX2 and AVX-512 instructions are a threat to system stability, we disable proactive voltage droop mitigation along with any AVX2 and AVX-512 frequency reduction in a system with an Intel Skylake-X CPU. This CPU features cores identical to those found in Skylake-SP server CPUs, but is targeted at high-end desktop systems and overclocking [197], so it provides closer control over frequency management. In particular, the CPU allows the user to configure the “AVX frequency offset”, which, when set to zero, disables proactive voltage droop mitigation altogether. We let all CPU cores execute a fixed number of 512-bit fused multiply-add (FMA) instructions in parallel and test whether the system remains stable. Our experiment show that often a single 512-bit FMA instruction per core is sufficient to create timing errors that lead to a system crash, which confirms that Intel CPUs operate with very narrow voltage guardbands and that immediate measures are required to prevent instability upon execution of power-intensive instructions.

Even though an immediate frequency change would provide the increased guardband required by AVX2 and AVX-512, recent Intel CPUs slightly delay frequency reduction. In the interim, these CPUs employ throttling to prevent excessive voltage droops [113, p. 2-14]. While multiple sources attempt to document the concrete steps during transitions between the frequency levels, these sources often disagree about the nature and duration of this delay. Intel’s Optimization Manual describes a delay of up to 500 µs as a “power

license" has to be requested from the central power control unit [113, p. 2-14].[5] In contrast, Travis Downs measures a delay of 10 µs and assumes that the delay constitutes a "voltage-only transition" as he observes an increasing voltage during this period in some of his experiments [68]. However, his analysis only considers transitions from the non-AVX to the AVX2 frequency level and in our opinion does not provide a satisfactory explanation of why the CPU delays frequency changes – after all, the voltage does not need to be increased if the frequency is substantially reduced directly afterwards. While Yussuf Khalil's extensive analysis includes other frequency level transitions caused by heavy 512-bit SIMD instructions and observes similarly short delays during all these experiments [131, p. 36], it does not analyze voltage changes.

In an effort to determine the concrete steps undertaken during frequency level transitions caused by different types of SIMD instructions and to resolve the discrepancies listed above, we expand upon the experiments conducted by Travis Downs [68] and Yussuf Khalil [131]. We perform two experiments where in both instances we first execute a specific type of instruction – either 128-bit or 256-bit fused multiply-add (FMA) instructions – in a loop for 1 ms to force the CPU to transition to a well-defined frequency level. We then execute a more power-intensive type of instructions – either 256-bit FMA, 512-bit FMA, 256-bit OR or 512-bit OR instructions – in a second loop for another 100 µs. Both loops are unrolled with as little dependencies between the instructions as possible to maximize throughput and, consequently, power consumption. To minimize its impact on our observations, the sibling hyper-thread executes a `MWAIT` instruction during our observation period. To force the system to operate at all-core turbo levels, we restrict idle power management by disabling all C-states.[6]

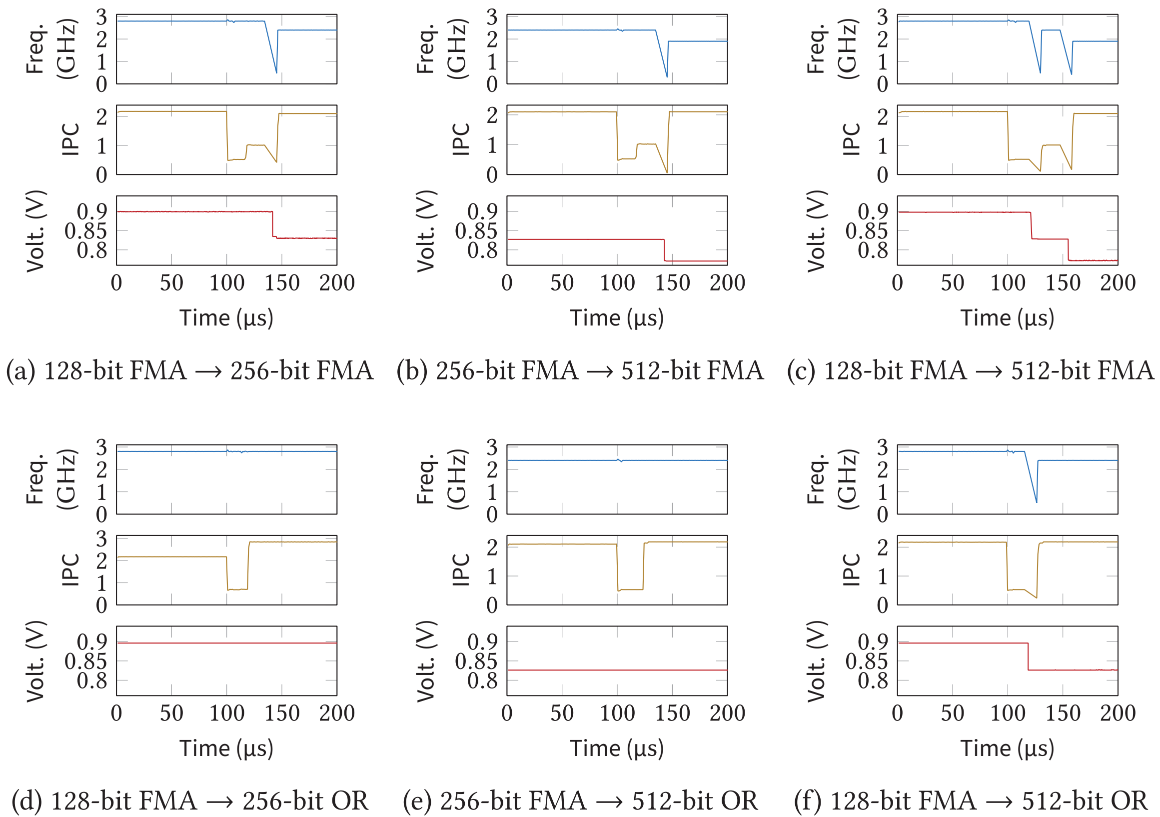
In our first experiment, we configure the performance monitoring unit (PMU) to count CPU cycles and reference cycles – i.e., cycles at a constant frequency – and read the performance counters once per microsecond using the `RDPMC` instruction to measure the CPU frequency. In addition, we read the number of executed instructions to calculate the instructions per cycle (IPC) during each microsecond interval of the frequency level transition.

In our second experiment, we sample the `MSR_PERF_STATUS` register to measure the voltage while the CPU starts executing more power-intensive instructions. Unlike in the previous experiment we do not repeatedly sample the register once per microsecond during a single experiment run as we noticed that the long latency of the required `RDMSR` instruction affects power consumption and CPU frequency selection. Instead, we repeat the experiment and interrupt each repetition after a variable number of instructions to measure the voltage once at that point. To reduce the noise caused by running the code multiple times, we apply a median filter where we calculate the median of 21 consecutive values when plotting the resulting voltages.

Figure 3.2 shows frequency, IPC, and voltage for the different combinations of instructions tested by our experiments. In all experiments, we observe immediate throttling by

[5] The description perfectly matches the behavior of traditional non-AVX turbo boost [95]. We assume that the information was erroneously included in the description of AVX2 and AVX-512 frequency management.

[6] C-states, as defined by the Advanced Configuration and Power Interface (ACPI), can be selected by the operating system to partially or completely disable inactive CPU cores [40]. Intel Turbo Boost only increases the CPU frequency if a sufficient number of cores have been placed in the C3 or C6 states [117].



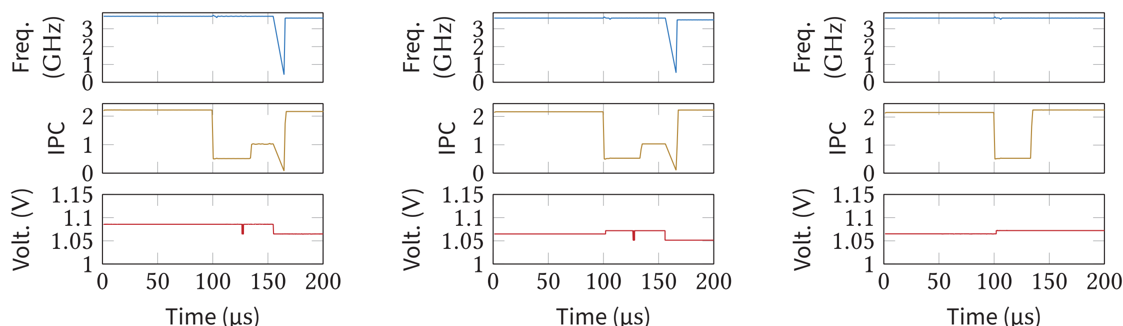
(a) 128-bit FMA ⟶ 256-bit FMA (b) 256-bit FMA ⟶ 512-bit FMA (c) 128-bit FMA ⟶ 512-bit FMA

(d) 128-bit FMA ⟶ 256-bit OR (e) 256-bit FMA ⟶ 512-bit OR (f) 128-bit FMA ⟶ 512-bit OR

Figure 3.2: Plots of typical frequency, IPC, and voltage while a CPU core starts executing more power-intensive instructions at $t = 100\mu s$ – the CPU employs a combination of throttling and frequency reduction to prevent excessive voltage droops as well as overheating. In the depicted situations, all CPU cores of the system were active. The temporary drops in the frequency plots as well as the corresponding drops in the IPC plot are measurement artifacts caused by pauses while the frequency is changed.

the CPU whenever it starts executing more power-intensive instructions. More specifically, the CPU typically operates at a quarter of its maximum IPC for approximately 20 µs. After this initial throttling period, the behavior of the CPU depends on the type of instruction executed:

- If the CPU executes 256-bit or 512-bit FMA instructions, it typically operates at half its maximum IPC for further 20 µs before finally switching to the frequency level required for the instruction type. We are unaware of any available information on why there are two separate throttling phases. We do not observe any temporary voltage boost that may provide enough voltage guardband for operation at higher throughput.
- Frequency level transitions from the non-AVX frequency level to the AVX-512 frequency level such as those caused by a transition from 128-bit to 512-bit FMA instructions involve two separate frequency changes [131, p. 33]. The first frequency



(a) 128-bit FMA ⟶ 256-bit FMA (b) 256-bit FMA ⟶ 512-bit FMA (c) 256-bit FMA ⟶ 512-bit OR

Figure 3.3: If the CPU starts to execute more power-intensive code while only one core is active, a voltage boost can sometimes be observed when the 512-bit register file is activated. The brief drops in the voltage plots are artifacts of the median filter and are caused by variation between different runs.

change occurs between the two aforementioned throttling phases. After the first frequency change, the CPU temporarily operates at the intermediate AVX2 frequency level.

- Some transitions from heavy to light instructions do not involve any frequency change even if the width of the instructions is increased. For example, both 128-bit FMA and 256-bit OR instructions can be executed at the highest frequency level. Nevertheless, the CPU is temporarily throttled when the CPU starts to operate on wider vector registers.

In all experiments, the voltage measured during correlates closely with the CPU frequency – as soon as the frequency is reduced, we measure a reduced voltage. More specifically, we do not measure any voltage boost similar to that observed by Travis Downs [68]. However, during our experiments, all cores are active, whereas his experiment setup only maintained one active core. We therefore repeat our experiment on a system with active C-states and further deactivate all except two cores using the CPU hotplug functionality of Linux to enforce operation at the highest turbo levels. While the results of these experiments are largely very similar to those at all-core turbo levels, two experiments – namely, those testing transitions from 256-bit FMA instructions to 512-bit instructions – show an increase of the voltage as shown in Figure 3.3. In addition, all throttling periods were slightly longer in this two-core configuration, with the initial throttling period consistently requiring approximately 30 µs.

Overall, our observations largely mirror those made by previous work [68, 131]. In particular, we confirm that the throttling period before frequency changes caused by wide SIMD instructions is far shorter than indicated by documentation from Intel [113, p. 2-14]. However, some differences between our results and those of previous work remain. For example, unlike indicated in Travis Downs' analysis, we are unable to trigger voltage boosts via 256-bit OR instructions and measure slightly longer throttling periods. Also, whenever we measure an increased voltage, the change occurs at the beginning of the

throttling period, whereas Travis Downs shows it to occur at the end of the throttling period. These differences are potentially caused by hardware differences, as we use a Skylake-SP server CPU for all our experiments whereas Travis Downs used a mobile CPU when analyzing voltage changes.

Our experimental results do not support the claim that the CPU throttles while waiting for the voltage to rise to a safe level.[7] During most of our experiments, we observe a substantial throttling period but no measurable voltage changes. Usually, no increased voltage is needed, either, as throttling is commonly followed by a frequency reduction. Our experiments show that the frequency reduction lowers the required voltage, so any voltage increase during the throttling period would be superfluous. Intel's documentation instead states that during the throttling period the CPU requests a power license from the central power control unit [113, p. 2-14]. This explanation is hardly more convincing given that the frequency at the AVX2 and AVX-512 frequency levels only depends on the number of active cores which is already known to the core via its previous frequency. The voltage selection does not depend on the activity of other cores either given that each core has its own voltage regulator.

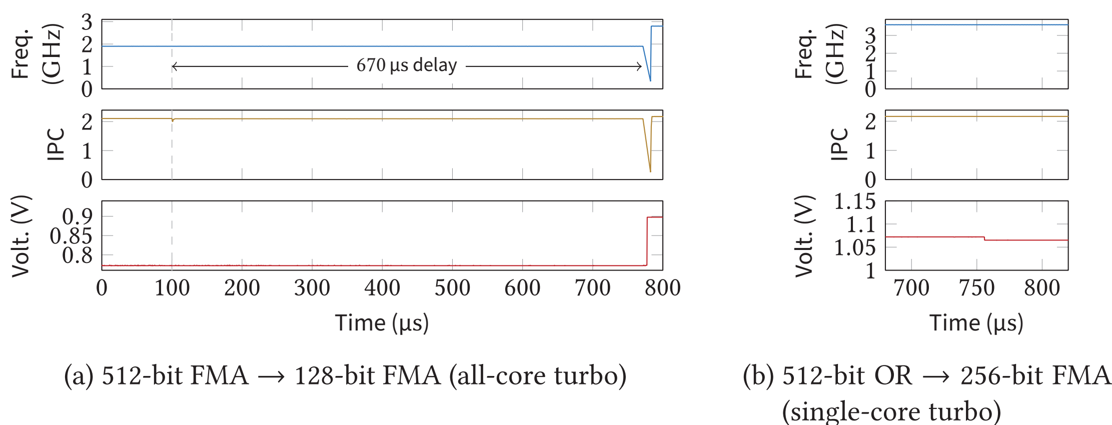
While we are unable to determine the reason for the throttling period with certainty, we identify two potential reasons: First, whenever SIMD functional units or parts of the register file are enabled, they likely provide substantial capacitance that needs to be charged, resulting in large inrush currents. Even if the instructions themselves later do not require a frequency change, this temporary power increase can cause instability, so temporary throttling is required to limit the overall voltage droop. Second, the CPU has to evaluate the behavior of the code before selecting a suitable frequency level. Heavy 256-bit and 512-bit instructions cause less frequency reduction when executed at a low rate, and Yussuf Khalil notes that in his experiments the CPU only transitions to the AVX2 frequency level after the execution of approximately 13000 256-bit FMA instructions [131, p. 35]. Testing whether a frequency reduction is necessary before actually changing the frequency reduces the number of frequency changes and their performance impact, but requires throttling as the system would otherwise become unstable due to the insufficient voltage guardband.

3.2.3 Delayed Frequency Increase

As described above, throttling and frequency reduction need to occur immediately upon execution of power-intensive 256-bit or 512-bit SIMD instructions as these mechanisms are required to achieve sufficient voltage guardbands. In contrast, increasing the frequency can be arbitrarily delayed, as a CPU operating at a low frequency is able to execute arbitrary types of code irrespective of their power consumption. Not immediately reverting to higher frequencies whenever a phase of 256-bit or 512-bit code has finished is associated with a performance penalty, though, as low-power code can be executed at a much higher frequency than power-intensive SIMD code and is therefore unnecessarily slowed down.

Nevertheless, recent Intel CPUs substantially delay such frequency boosts. To demonstrate the behavior of these CPUs when they transition to less power-intensive code, we

[7] Travis Downs calls this phase a *voltage-only transition* [68].



(a) 512-bit FMA → 128-bit FMA (all-core turbo)

(b) 512-bit OR → 256-bit FMA (single-core turbo)

Figure 3.4: Plots of frequency, IPC and voltage during a transition to less power-intensive code measured using the experimental setup described in Section 3.2.2. At t = 100µs the CPU starts executing less power-intensive instructions. It then waits for approximately 670 µs before it reverts the effects caused by the previous power-intensive code.

repeat the experiments from the previous section. In contrast to our previous experiments, we let the system first execute power-intensive 256-bit and 512-bit SIMD instructions and then switch to less power-intensive SIMD instructions. Again, our experiments mirror similar experiments conducted by related work [131, 68, 213], with the main difference that we analyze a wider range of instruction types.

Figure 3.4 shows CPU frequency, IPC and voltage for two representative types of transitions from more power-intensive SIMD instructions to less power-intensive instructions, where the transition between the two occurs at t = 100 µs. If the more power-intensive instruction type requires a frequency reduction, our experiments show that the frequency is increased after a delay of 670 µs (Figure 3.4a). Likewise, if the more power-intensive code requires a voltage boost, this voltage change is reverted after the same delay (Figure 3.4b). The delay matches that reported by related work [131, 68, 213] and again stands in stark contrast to the considerably less accurate delay of "approximately 2 ms" reported by Intel's optimization manual [113, p. 2-14].

The reason for delayed voltage and frequency changes can be found in the overhead caused by throttling and frequency transitions. Both temporarily reduce throughput – while throttling reduces IPC, each frequency change has to temporarily stop the whole CPU core to wait for the clock generator – usually a phase-locked-loop (PLL) – to synchronize to the new frequency [189]. In addition, each frequency change is associated with a voltage change that wastes energy. Before increasing and after lowering its frequency, a CPU core gradually ramps its voltage up and down, respectively, during which time the core operates at a higher voltage than required for its current frequency and is therefore temporarily less energy-efficient [189]. Due to strict power limits, any reduction in energy efficiency commonly translates into reduced performance.

As described in Section 2.2.3, such overhead limits the usefulness of fine-grained frequency scaling, as very short periods of operation at a higher frequency fail to provide any

performance improvement. In the case of Intel CPUs, immediately increasing the CPU frequency whenever the executed instructions allow to do so could potentially result in even lower performance than continuously operating at reduced AVX2 or AVX-512 frequencies. Therefore, recent Intel CPUs use the aforementioned delay as a rate-limiting technique to prevent frequent toggling between different frequencies [68, 204]. The CPUs' behavior is similar to techniques addressing the problem of dynamic power management [22], where temporarily disabling a device saves energy, yet transitions into and from this low-power state cost energy. Previous work in the field of dynamic power management states that a delay equal to the *break-even time*, i.e., the minimum time after which a positive net impact is expected, should be selected to minimize worst-case overhead [127]. In Chapter 5, we show that the delay chosen by Intel does not match the break-even time associated with frequency changes. However, we also show that there is little potential for other policies to improve performance.

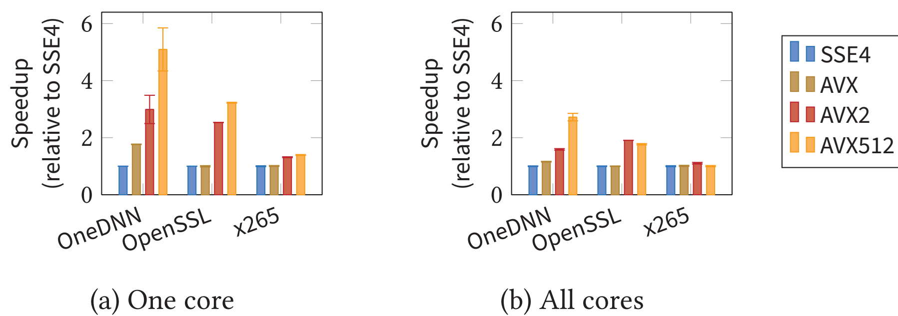
3.3 AVX Overhead

The frequency management scheme implemented by recent Intel CPUs affects both the AVX2/AVX-512 code that causes the CPU frequency to be reduced and – in some scenarios – other, potentially unrelated, less power-intensive code. The former is a *local frequency reduction*, resulting in what we call *local AVX overhead*. The latter, instead, is a remote effect created by the interaction between different software components, resulting in what we call *remote AVX overhead*. In the following sections, we describe both types of AVX overhead and, as a motivation for the remainder of this thesis, present their impact on real-world applications. As this thesis focuses on remote AVX overhead, we explain to which degree existing software is able to mitigate the impact of both local and remote AVX overhead. We show how remote AVX overhead in particular presents challenges not yet solved by existing profiling tools and operating systems.

3.3.1 Local AVX Overhead

Local AVX overhead – i.e., frequency reduction during the execution of power-intensive AVX2 and AVX-512 code – substantially reduces the potential of AVX-512 and, to a lesser degree, AVX2. In Section 3.1, we quantified the performance impact of wide 256-bit and 512-bit SIMD operations using benchmarks based on OpenSSL, OneDNN, and x265. In this previous experiment, we focused on power consumption and ideal throughput in a scenario without power limits and therefore intentionally hid any local AVX overhead by configuring the system to use only core at a fixed frequency of 3.5 GHz. In this setup all benchmarks benefit from both AVX2 and AVX-512 as shown in Figure 3.1. In a more practical setup without such an artificial frequency limit, code using less power-intensive instruction sets could have been executed at higher frequencies, though.

To get an estimate of the the performance impact of local AVX overhead, we therefore repeat the measurements from Section 3.1, but configure the CPU to automatically choose the maximum possible turbo frequency as in most productive environments. We repeat the experiment twice, once restricting the application to only one CPU core, and once



(a) One core

(b) All cores

Figure 3.5: AVX2 and AVX-512 provide much bigger performance gains when OneDNN `benchdnn --rnn` and OpenSSL ChaCha20-Poly1305 are executed on a single CPU core than when all cores are used. The different behavior is explained by different amounts of local AVX overhead – as Table 3.1 shows, there is a larger frequency reduction when all cores are active.

allowing it to use all available CPU cores. Figure 3.5 shows the results of both experiments. As in both cases SSE4 and AVX configurations achieve higher frequencies than their AVX2 and AVX-512 counterparts, the speedup caused by AVX2 and AVX-512 relative to SSE4 is lower than for the fixed-frequency experiment in Section 3.1. For example, whereas our previous experiment showed that AVX-512 yielded a 5.1× throughput improvement for OneDNN, default frequency management reduces the advantage. While AVX-512 still provides a 5.1× performance improvement when the system is operating at single-core turbo frequencies[8], the factor is reduced to 2.7× when all cores are used. For OpenSSL and x265, the local AVX overhead caused by AVX-512 even completely outweighs the advantages of these instructions over AVX2 if the benchmarks are executed on all of the system's cores.

Two effects are responsible for the reduced effectiveness of AVX2 and AVX-512. First, AVX2 and AVX-512 cause a frequency reduction, and this reduction impacts the multi-core configuration more than the single-core configuration due to the larger power consumption when all cores are active. For example, whereas the Xeon Gold 6130 CPU used for the experiment reduces its single-core turbo frequency by only 5.4% when executing AVX-512 instructions, its all-core turbo frequency is reduced by 32.1% [119, pp. 15 f.]. Second, the scalability of SIMD instructions is limited by the available memory throughput, so wider SIMD instructions fail to provide a further advantage if memory is saturated [103, pp. 286 f.]. This effect also increases with higher numbers of cores as memory bandwidth is shared between the cores.

Only the frequency impact constitutes local AVX overhead and is of relevance to this thesis. To distinguish between frequency and memory effects, we analyze the instructions per cycle (IPC) as only the memory effects cause an IPC reduction. We measure the IPC difference between the multi-core and the single-core configuration when the benchmarks use SSE4 and compare it to the IPC difference when the benchmarks use AVX-512. If saturation of the memory bandwidth is responsible for some of the reduced effectiveness of

[8] The expected difference between this and the previous experiment is likely hidden by the variation in the results.

AVX-512 observed above, we expect the AVX-512 version of the benchmarks to experience a substantially larger IPC reduction than the SSE4 version.

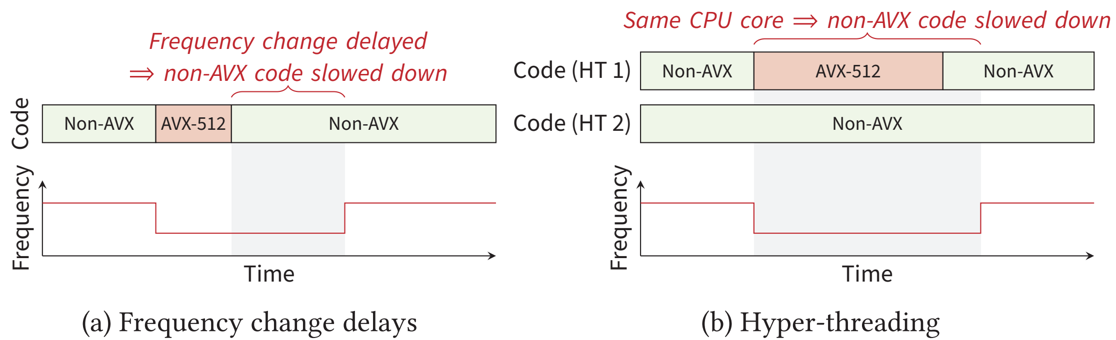
This is not the case for x265, where the instruction throughput of the AVX-512 implementation is even less affected by the number of active cores than the SSE4 implementation. Effects such as memory pressure or microarchitectural conflicts therefore do not explain the reduced effectiveness of AVX-512 for x265. OneDNN and OpenSSL, instead, do experience a more pronounced IPC reduction for AVX-512 than for SSE4. For example, the SSE4 version of OneDNN achieves 67.9% less IPC in the multi-core setup, which is expected given that multiple cores compete for memory throughput. The AVX-512 version experiences an even higher 71.3% multi-core IPC reduction, which would in isolation translate into a 10.6% performance impact. This IPC difference alone is by far not able to explain the behavior observed above, though – executing the AVX-512 version of OneDNN on multiple cores reduced its advantage compared to SSE4 by 46.6%, so frequency changes have a far bigger impact than memory pressure. Similarly, for OpenSSL, the 23.6% performance impact expected due to the different IPC ratios does not explain how AVX-512 is 45.1% less effective in the all-core scenario. Instead, this results – and similar results for AVX2 – again shows that the performance of AVX2 and AVX-512 is to a large degree limited by the associated frequency reduction.

While such local AVX overhead greatly impacts the performance of AVX2 and AVX-512 code, it is not targeted by this thesis for two reasons. First, local AVX overhead is an inevitable side-effect of AVX2 and AVX-512. When software developers directly use AVX2 and AVX-512 instructions, they usually do so knowing that these instructions reduce the frequency. This choice is justified by large throughput improvements which commonly outweigh the corresponding frequency reduction.

Second, local AVX overhead does not pose a particular challenge to software developers. In contrast to *remote AVX overhead* described in the next section, local AVX overhead is a purely local effect, which allows for simple detection and quantification of any resulting performance issues during software development. Optimization using AVX2 and AVX-512 is usually a deliberate effort to improve performance within well-defined sections of code and is usually accompanied by the use of profiling tools which measure the CPU time required for this code. If the benefits of AVX2 and AVX-512 do not outweigh the local AVX overhead, the resulting slowdown is easily observed in the form of increased CPU time. For example, related work describes cases where AVX-512 was shown to fail to provide a performance improvement for specific problems [248] or just specific configurations [236]. We expect that once overall slowdown has been observed it is often easy to mitigate local AVX overhead by reverting the software changes that introduced power-intensive instructions.

3.3.2 Remote AVX Overhead

As described above, local AVX overhead is not particularly problematic as the impact is easy to quantify during software development. However, the impact of AVX frequency reduction is not limited to just AVX2 and AVX-512 code. In many cases, lower frequencies affect other, potentially unrelated code that could have been executed at a higher frequency. In this thesis, we focus on the resulting *remote AVX overhead.* This type of overhead is



(a) Frequency change delays

(b) Hyper-threading

Figure 3.6: The frequency reduction caused by AVX2 and AVX-512 instructions can affect other code without those instructions in two scenarios: When AVX2 or AVX-512 code is followed by code allowing higher CPU frequencies, the frequency increase is delayed (a). When AVX2 or AVX-512 code and less power-intensive code are executed on sibling hyper-threads, the hyper-threads share the same low frequency (b).

often far more problematic for two reasons. First, even short phases of power-intensive AVX2 or AVX-512 code can affect vastly larger amounts of less power-intensive code, amplifying the effect on performance. Second, remote AVX overhead is far harder to predict and quantify as it is caused by the interaction between different applications or software components.

In addition, remote AVX overhead is of particular interest because it is – unlike local AVX overhead – frequently avoidable. As we show in this thesis, the operating system can often negate the specific conditions required for the occurrence of remote AVX overhead without any need for applications to abstain from AVX2 and AVX-512. For example, modified scheduling policies such as the one described in Chapter 6 can greatly reduce the amount of overhead. In the following sections, we describe the causes for remote AVX overhead to provide the necessary technical background for the following chapters of this thesis. We then show the impact of remote AVX overhead on real-world workloads to demonstrate the importance of our work.

3.3.3 Implications of Frequency Change Delays

One of the two sources of remote AVX overhead is the frequency change delay described in Section 3.2.3 which is used by Intel CPUs to prevent frequent toggling between the three frequency levels. These CPUs only restore a higher CPU frequency after 670 µs have passed without AVX2/AVX-512 instructions requiring the lower frequency level. As shown in Figure 3.6a, during this delay, any code is invariably executed at an unnecessarily low frequency.

The resulting overhead especially affects workloads which consist of short stretches of AVX2 or AVX-512 code interleaved with sections of code which can execute at a higher frequency level. Reports of substantial amounts of such remote AVX overhead therefore often represent workloads where only a single software library or component uses AVX2/AVX-512. For example, in 2017, experiments at Cloudflare showed that the performance of a web

server serving web pages via TLS was reduced by 10% if encryption and decryption was performed by OpenSSL's AVX-512 implementation of the ChaCha20-Poly1305 encryption scheme [141]. The magnitude of the slowdown was particularly surprising given that only 2.5% of CPU time was spent in AVX-512 code. OpenSSL developers subsequently disabled AVX-512 on the affected microarchitectures [195] once they noticed that the AVX-512 code could slow some nginx workloads down. When we recreate these web server experiments in the following, we therefore re-enable AVX-512 to demonstrate the original issue. Note that the AVX2 code found in OpenSSL was not disabled even though it also impacts CPU frequencies, albeit to a lesser degree.

Slowdown has also been reported for computer games [80]. Due to the periodic nature of computer games – the game repeatedly calculates new graphics and audio data – a single software component utilizing AVX2 or AVX-512 has the potential to cause substantial slowdown even if only a fraction of each period is spent in the component. The radfft FFT library in particular was shown to cause unacceptable slowdown when using AVX2 and AVX-512 on Intel CPUs [80]. Even the C standard library has the potential to cause remote AVX frequency reduction, as functions such as `memset()` often use SIMD load/store instructions to improve memory throughput. For example, the glibc library temporarily provided AVX-512 implementations of memcpy/memset [161]. Similar to OpenSSL, these implementations were eventually disabled on Intel Skylake CPUs once the potential overall slowdown was noticed.

3.3.4 Implications of Hyper-Threading

The second source of remote AVX overhead stems from the fact that the frequency reduction caused by AVX2 and AVX-512 code often affects more than just the software executed on the same logical CPU. As described in Section 2.1.1, modern Intel CPUs implement simultaneous multi-threading, where each physical CPU cores provides multiple logical CPUs in the form of hyper-threads executing on the same hardware and therefore at identical CPU frequencies. As a result, as shown in Figure 3.6b, software executed on the sibling hyper-thread next to an application using AVX2 or AVX-512 is executed at the same low frequency, even if it could be executed at a higher frequency if it was placed in isolation on a separate core [85].

In isolation, this type of remote AVX overhead has been observed, for example, at Intel where it was shown that latency-critical jobs experienced 10% lower throughput and 30% higher latency when executed alongside deep learning training jobs [154, 55]. The effects of hyper-threading likely also amplified the remote AVX overhead observed in the situations listed in the previous section.

3.3.5 Impact on Real-World Software

To quantify the overhead caused by the two types of remote frequency reduction, we execute a wide range of workloads involving AVX2 and AVX-512 instructions and compare their throughput to the throughput achieved when these power-intensive instructions are disabled. We try to recreate experiments conducted as part of existing reports on remote AVX overhead [141, 154] whenever possible and add further benchmarks to cover a wider

range of workloads. All following experiments are conducted on a system with an Intel Xeon Gold 6130 CPU, 24 GiB of 2666 MHz DDR4 RAM, Fedora 31, and the Linux 5.6.13 kernel. All experiments were repeated ten times.

To recreate the web server experiment performed at Cloudflare [141], we configure the nginx 1.21.3 web server to serve a static page via TLS using the ChaCha20-Poly1305 implementation from OpenSSL 1.1.1l. Whereas the original experiment simulates additional CPU load by removing line breaks and spaces from the page using LuaJIT and then compressing the page using the brotli compression algorithm, we only let the web server compress the page to simplify the experimental setup. We use the wrk benchmark driver [254] – modified to accept brotli-compressed responses – to generate HTTPS requests and to determine the web server's throughput.

We also recreate the two-application scenario mentioned in the previous section [154] by executing a non-AVX application while a power-intensive AVX2 or AVX-512 application runs in the background. We measure the CPU time required by the non-AVX application to determine the remote AVX overhead caused by the power-intensive background task. In contrast to previous work, we vary the power-intensive background task to demonstrate that the problem affects not only deep learning applications but also applications such as video encoders which often have a lower impact on CPU frequencies. We conduct experiments with OneDNN `benchdnn --rnn` as well as x265, each configured as described in Section 3.1, with the exceptions that x265 is configured to encode videos with the "veryslow" quality preset to increase its runtime and that OneDNN is configured to use passive waiting via `OMP_WAIT_POLICY=passive` to improve performance in the presence of other applications.[9] As Figure 3.7 shows, OneDNN causes a substantially larger frequency reduction than x265 which only executes heavy 256-bit/512-bit instructions in select code regions and uses light instructions in most others. We execute both in parallel with a wide range of applications from different fields. Whereas the original report on remote AVX overhead caused by hyper-threading described a workload involving a latency-critical application [154], we select various benchmarks from Parsec 3.0 [26] and the Phoronix Test Suite (PTS) v9.0.1 [194]. Our selection of benchmarks contains benchmarks representative of high-performance computing including many of the Parsec benchmarks as well as benchmarks representative for web workloads such as the PTS redis benchmark and desktop workloads such as the PTS build-linux-kernel and git benchmarks. All applications are configured to use as many threads as there are logical CPUs.

Figure 3.8a shows the results of all the experiments described above. In the figure, the single-application nginx benchmark is labelled "nginx", while the two-application workloads are prefixed with either "x265-" and "onednn-" depending on the background application. The figure shows the remote AVX overhead measured as the CPU time increase of, depending on the workload, either nginx or the Parsec benchmark when AVX-512 or

[9] By default, OneDNN uses an implementation of barrier synchronization that performs active waiting in user space [183]. We argue that this default configuration is problematic because active waiting causes large amounts of overhead whenever one or more threads are preempted. Batch tasks such as machine learning can often be used to utilize spare CPU cycles, in which case they need to coexist with other tasks. `OMP_WAIT_POLICY=passive` selects an implementation that reduces synchronization overhead by, for example, putting threads to sleep when they are blocked.

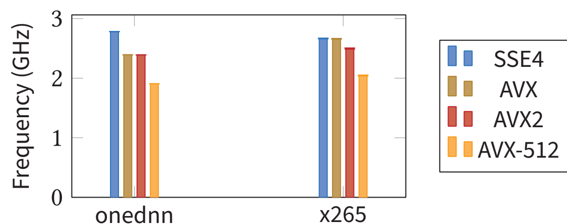


Figure 3.7: OneDNN uses more heavy AVX2 and AVX-512 instructions and therefore executes at lower average CPU frequencies than x265.

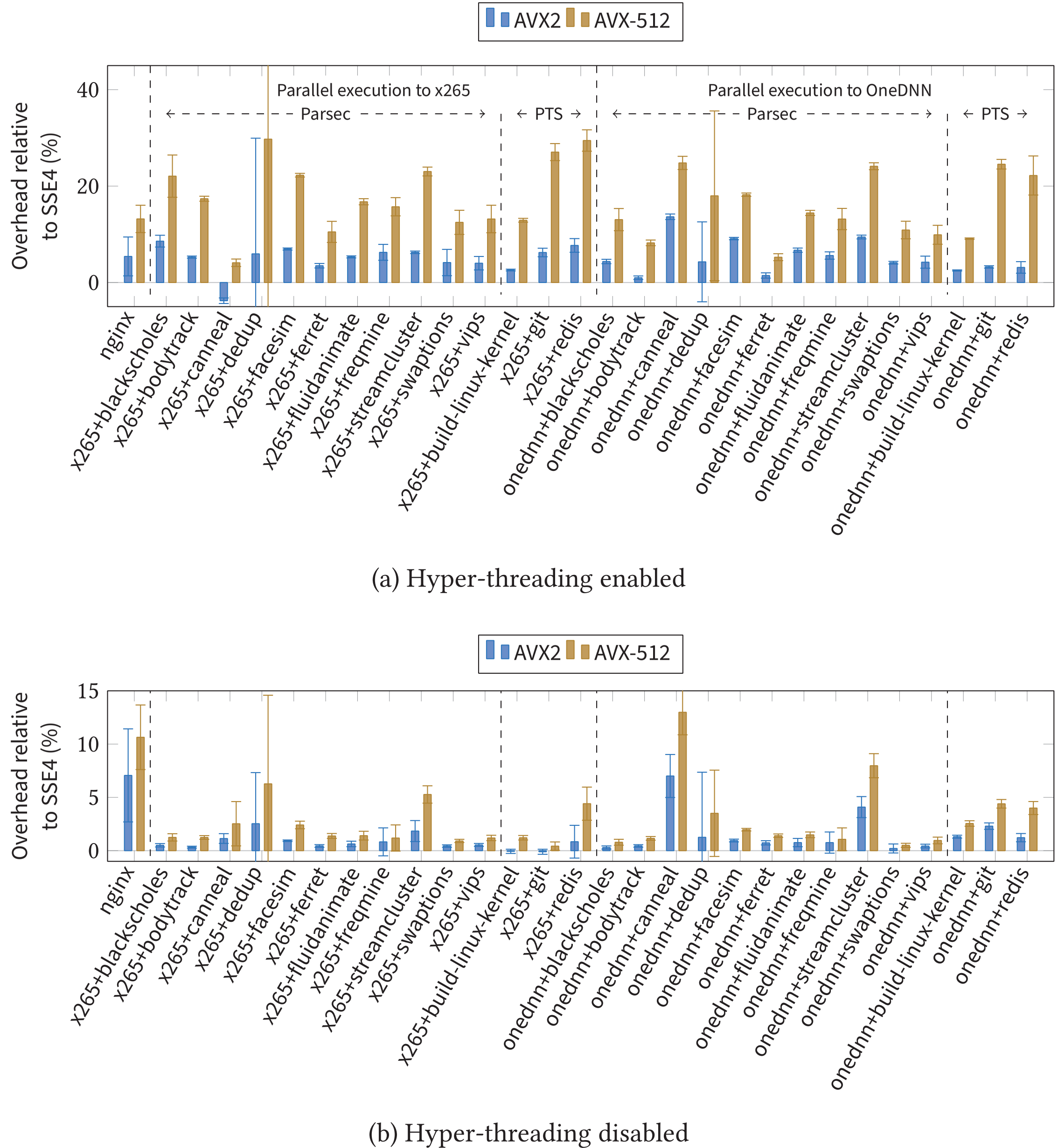
AVX2 instructions are enabled compared to identical configurations when the workload only uses SSE4 instructions:

$$overhead_{\text{AVX-512}} = \frac{t_{\text{CPU,AVX-512}}}{t_{\text{CPU,SSE4}}} - 1 \tag{3.1}$$

$$overhead_{\text{AVX2}} = \frac{t_{\text{CPU,AVX2}}}{t_{\text{CPU,SSE4}}} - 1 \tag{3.2}$$

While the overhead varies from application to application, it can be seen that in almost all cases AVX2 and AVX-512 have a substantial negative performance impact on the non-AVX portion of the workloads. The only exception is Parsec canneal, where a more detailed analysis shows large IPC variation between the different configurations which counteracts some of the impact of frequency changes. On average, AVX2 reduced the measured throughput by 5.1%, whereas AVX-512 caused a 16.8% performance reduction. Counterintuitively, the results also show that x265 and OpenSSL cause similar levels of remote AVX overhead even though they cause different degrees of frequency reduction. Where applicable, our overhead measurements for AVX-512 workloads match those presented by previous reports of remote AVX overhead. For example, Vlad Krasnov described an overhead of slightly less than 10% for a workload involving nginx and OpenSSL [141], while our nginx-based benchmark experiences 13.2% overhead. Also, Aubrey Li observed a 10% impact on throughput and a 30% impact on latency when a web server and an AVX-512 deep-learning workload shared the same cores [154], while in our experiments most applications executed alongside OneDNN showed between 10% and 30% remote AVX overhead.

As described in Sections 3.3.3 and 3.3.4, remote AVX overhead can be caused either by interaction between hyper-threads or by frequency change delays. To show which of the two affects which benchmarks, we repeat the experiments, but effectively disable hyper-threading by configuring all processes to use a CPU affinity mask with only one logical CPU per core. Figure 3.8b contains the results of this experiment. It can be seen that most two-application benchmarks are mainly affected by the effects of hyper-threading, whereas the frequency change delay has little impact on performance. This is expected as the timeslices allocated to the applications by the Linux scheduler are much longer than the frequency change delay, so only small portions of the non-AVX applications are affected by reduced frequencies. Only some applications switch more often between non-AVX and AVX2 or AVX-512 code. For example, the web server example based on



(a) Hyper-threading enabled

(b) Hyper-threading disabled

Figure 3.8: The remote AVX overhead caused by power-intensive AVX2 and AVX-512 instructions reduces the performance of many heterogeneous workloads consisting of AVX2 or AVX-512 code as well as less power-intensive non-AVX code. The figure shows the CPU time increase caused by AVX2 and AVX-512 relative to configurations where AVX2/AVX-512 code is replaced by less power-intensive SSE4 implementations. All workloads experience more overhead when hyper-threading is enabled (a) than when it is disabled (b).

nginx and OpenSSL frequently switches between AVX2/AVX-512 code (encryption and decryption) and non-AVX code (compression, networking, and other web server logic) and is slowed down even in our second experiment where hyper-threading is disabled.[10] Some of the two-application benchmarks also show substantial remote AVX overhead when hyper-threading is disabled, which indicates that these applications cause an increased context switch rate. Whereas, for example, swaptions causes only one context switch per 7.4 ms of CPU time when executed alongside OneDNN and shows very little remote AVX overhead, streamcluster experiences five context switches per millisecond of CPU time and shows substantially higher overhead. The context switch rate is likely mainly caused by the type of synchronization used by the parallelized applications; streamcluster in particular makes intensive use of barrier synchronization [26].[11]

3.3.6 Impact of Speculative Execution

The examples shown in the previous sections involved intentional execution of AVX2 or AVX-512 instructions by parts of the workload. However, remote AVX overhead sometimes occurs even if no task intentionally executes any such power-intensive instructions. In particular, speculative execution can cause a frequency reduction even if AVX-512 instructions are only executed due to misspeculation as indicated by a Ubuntu bug report [1]. While we were able to replicate the issue in a synthetic program, we are unaware of any widespread performance problems caused by such speculative execution of AVX-512 code. In the following chapters, we therefore mostly ignore this source for remote AVX overhead. Some of the techniques inadvertently prevent speculative execution of AVX2 or AVX-512 code, though. For example, the core specialization technique presented in Chapter 6 disables access to 512-bit registers on most cores.

3.4 Information Available at Runtime

As shown in Section 3.3.2, remote AVX overhead is hard to predict when developing or deploying software as it is caused by runtime interaction between different software components. However, once remote AVX overhead has been observed there are a number of techniques to mitigate the effects of remote AVX overhead. Techniques against remote AVX overhead include, for example, replacing AVX2 or AVX-512 with less power-intensive instructions, as has been done by the developers of the OpenSSL or glibc libraries [195, 161], or placing AVX-512 code on other cores than less power-intensive code [85, 154] as described in Chapter 6.

While such techniques can improve throughput in the presence of remote AVX overhead, they often cause overhead themselves if little or no remote AVX overhead is present. In

[10] When hyper-threading is active, the impact of frequency change delays is doubled because each short section of AVX2 or AVX-512 code affects the following code executed on both hyper-threads. Therefore, the nginx benchmark should show less remote AVX overhead when hyper-threading is disabled. In our experiments, this effect was likely hidden by the large variation of the results.

[11] The context switch rate also depends on the length of the scheduling timeslices. In Chapters 6 and 7, we conduct similar experiments but use different schedulers, so the results differ.

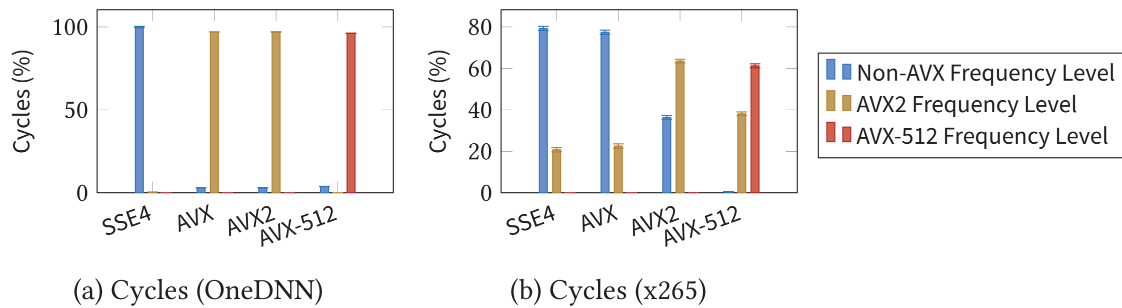
Table 3.2: The CPU provides a number of performance events to count cycles at different frequency levels. We show that the frequency levels documented by Intel can be further subdivided to determine the type of instruction.

Name	Event	Umask	Instructions
Documented Performance Events [116, pp. 19–7 f.]			
CORE_POWER.LVL0_TURBO_LICENSE	28H	07H	≤128-bit, light 256-bit
CORE_POWER.LVL1_TURBO_LICENSE	28H	18H	heavy 256-bit, light 512-bit
CORE_POWER.LVL2_TURBO_LICENSE	28H	20H	heavy 512-bit
Individual Umask Bits (Undocumented)			
-	28H	02H	≤128-bit
-	28H	04H	light 256-bit
-	28H	08H	heavy 256-bit
-	28H	10H	light 512-bit

addition, they sometimes only target specific types of remote AVX overhead. Therefore, any decision on whether to apply such countermeasures requires knowledge of the expected amount of overhead caused by the techniques as well as knowledge of the amount of remote AVX overhead in the absence of any countermeasure. Collecting such information poses a major challenge, though. In particular, while current hardware generally provides powerful performance monitoring facilities, their interfaces lack the performance events required to detect the conditions for remote AVX overhead.

As shown in Table 3.2, Intel documents three performance events which measure the frequency impact of AVX2 and AVX-512 by counting the cycles spent at the three frequency levels [116, pp. 19–7 f.]. The performance events `CORE_POWER.LVL0_TURBO_LICENSE`, `CORE_POWER.LVL1_TURBO_LICENSE`, and `CORE_POWER.LVL2_TURBO_LICENSE` count the cycles spent at the non-AVX, AVX2 and AVX-512 levels, respectively. As we showed in Section 3.2.2, the CPU differentiates between more than three types of instructions – for example, the CPU's behavior during transitions from 128-bit to 256-bit FMA instructions differs from that during transitions from 128-bit FMA to 512-bit OR instructions. To determine whether there are undocumented performance events which capture these differences, we vary the Umask value which selects between different related events. We set individual bits in the Umask value while executing different SIMD instructions in a loop to test which events are triggered by which instructions. As shown in the lower half of Table 3.2, we find that each bit corresponds to a different type of instruction, with separate bits existing for light 256-bit and 512-bit as well as heavy 256-bit instructions.

The performance events allow the user to measure the time during which the frequency is reduced by power-intensive instructions. As an example, Figure 3.9 shows the fraction of cycles spent at the three frequency levels as reported by the PMU for x265 and OneDNN.



(a) Cycles (OneDNN)

(b) Cycles (x265)

Figure 3.9: Recent Intel CPUs provide performance events which represent the cycles spent at the three frequency levels. When these events are counted for OneDNN (a) and x265 (b), it can be seen that OneDNN causes a stronger frequency reduction.

The measurements show that OneDNN makes more intensive use of heavy 256-bit and 512-bit instructions and therefore as mentioned earlier causes a larger frequency reduction. However, the PMU does not give any information on whether the frequency reduction is actually required for the currently executing code, i.e., it is not able to identify the impact of frequency change delays. The PMU is not able to attribute the frequency reduction to individual hyper-threads either – instead, it reports the same cycle counts for both hyper-threads even if only one of them executes AVX2 or AVX-512 code. The interface can therefore not be used directly to identify or measure remote AVX overhead.

Instead, detecting and quantifying remote AVX overhead – i.e., unnecessarily low CPU frequencies – requires not only information on CPU frequencies but also on whether the currently executed code uses power-intensive SIMD instructions. While recent Intel CPUs provide a number of performance events to count 256-bit or 512-bit floating point operations [116, pp. 19–20 f.], they lack events for other power-intensive instructions such as SIMD integer multiplications. In the absence of suitable hardware interfaces, previous work has experimented with a wide range of methods to detect whether specific functional units are in use. For example, static analysis is able to identify the regions of the control flow graphs in which the program uses specific CPU components. You et al. [260] use this information to implement efficient power gating of individual functional units. The main drawbacks of static analysis is that it cannot determine the time spent in individual program regions, though. Kumar et al. [142] therefore use dynamic profiling to determine whether individual program regions are long enough to warrant use of SIMD instructions or whether the SIMD unit should remain power-gated. In both cases, the referenced work employs the techniques to reduce leakage power, as power-gating itself is only effective when applied over sufficiently long time spans. However, the information can likely also be used to measure the remote AVX overhead. When information about SIMD usage is correlated with the aforementioned cycle counts reported by the PMU, phases without SIMD instructions but with reduced frequencies constitute remote AVX overhead.

While such a profiler based on code analysis and dynamic profiling may be viable for some workloads, the approach chosen by Kumar et al. [142] has an important drawback. In particular, dynamic profiling is implemented via instrumentation of the application using a just-in-time compiler. Such instrumentation not only has the potential to introduce substantial overhead but also prevents use of the approach in scenarios where applications

cannot be modified. A cloud provider, for example, is not able to add arbitrary instrumentation to virtual machines provided by customers. In Chapter 4, we therefore present a simple sampling profiler able to measure remote AVX overhead for arbitrary workloads at runtime with low overhead. We also show how simple hardware modifications will make quantification of remote AVX overhead far more simple and accurate.

As described in the introduction to this section, selecting a suitable countermeasure against remote AVX overhead not only requires knowledge of the amount of remote AVX overhead but also requires an estimate of the expected overhead of the countermeasure itself. Given the wide range of potential countermeasures, no single technique to determine this overhead can be given. Previously, remote AVX overhead has most commonly been met by replacing the power-intensive code responsible for the remote AVX overhead with a less power-intensive implementation. For example, when the OpenSSL developers, as mentioned above, noticed performance regressions due to the use of AVX-512, they completely disabled 512-bit SIMD instructions on the affected systems [195]. Similarly, glibc's AVX-512 implementations of string functions were disabled after remote AVX overhead was observed [161]. Such code changes not only prevent remote AVX overhead but also cause a local slowdown as wide SIMD instructions are replaced by less performant instructions, which may lead to reduced performance in some situations. While, for example, disabling AVX-512 in OpenSSL improved performance for many web server configurations, our experiments show that other configurations such as web servers that serve static uncompressed files are substantially slowed down [83]. Predicting the overhead caused by such code changes therefore requires an estimate of the local speedup brought by AVX2 and AVX-512 in the given scenario.

We are unaware of any generic method that produces reliable estimates of the local slowdown if AVX2 or AVX-512 are replaced by less power-intensive instructions. One could assume that, for example, counting all 512-bit SIMD operations and multiplying the instruction count by four would yield a usable estimate of the number of 128-bit SIMD instructions required. This and similarly simplistic approaches are not practical, though, mainly because, as described above, the CPU does not provide the required instructions counts. Also, such approaches may overestimate the slowdown as wider vector operations are more heavily affected by memory stalls [112]. Zhu et al. [263] show how it is possible to extrapolate cache efficiency and SIMD utilization from measurements performed with reduced problem sizes, but such approaches require application-level knowledge. Overall, in our eyes, the lack of methods to predict the impact of program changes such as those described above on OpenSSL and glibc means that such changes come with an inherent high risk to cause performance regressions. In this thesis, we therefore instead focus on techniques which mitigate the impact by remote AVX overhead without any changes to applications. We discuss changes to the CPU's DVFS policy in Chapter 5 and describe suitable scheduler modifications in Chapters 6 and 7.

3.5 Applicability to Other Microarchitectures

The sections above describe the behavior of the Intel Skylake-SP and Intel Skylake-X microarchitectures. Similarly, the following chapters will focus on processors using these

microarchitectures. However, as we show in the following, our work also applies to other existing CPUs, albeit to varying degrees. More importantly, we expect a rising number of CPUs to feature similar behavior to the Intel CPUs covered by this thesis as power limits become more and more stringent.

The first Intel processors to feature different frequency levels depending on the instruction set used were the Intel Haswell-EP processors introduced in 2014 [76]. Whenever one processor core executed AVX2 code, these processors reduced the frequencies of all their cores [77]. This is in contrast to the processors used in this thesis which select different frequencies for individual cores depending on instruction set usage. As a result, while our work on profiling and fair scheduling in Chapters 4 and 7 largely applies to these processors, albeit with minor modifications, our work on core specialization presented in Chapter 6 does not.

The first microarchitecture to implement separate AVX frequency management for individual CPU cores was the subsequently released Broadwell microarchitecture [77]. In contrast to the Skylake-SP and Skylake-X microarchitectures described in this chapter and similar to Haswell, it did not support AVX-512, yet. The third AVX-512 frequency level was then introduced by Skylake, which proved to be a comparably long-lived microarchitecture. To the best of our knowledge its derivatives Cascade Lake-SP and Cooper Lake feature identical frequency management [188]. Subsequent Cannon Lake laptop processors with support for AVX-512 also provide three different frequency levels [58]. The first microarchitecture that deviated from this frequency management scheme was Ice Lake which was released for laptops in 2019 and for server systems in 2021. This microarchitecture provided optimized frequency synthesizers – the processor cores do not need to halt while the PLL changes its frequency – to reduce frequency change overhead [79, 188]. Also, Ice Lake reduced the number of frequency levels to two, where heavy AVX2 and light AVX-512 code is able to execute at the same frequency level as non-AVX code. On these CPUs, heavy AVX-512 code still has to be executed at reduced frequencies, so the qualitative results of our work remain valid.

As described in Section 2.3.1, other CPU manufactures have yet to introduce similar techniques to execute code using power-intensive instructions at reduced frequencies, most likely because Intel's implementation is the subject of multiple patents [29, 204, 201]. We expect the manufacturers to eventually follow suit, as increasingly stringent power limits mean that exploiting leftover power headroom remains as the primary source for performance improvements. Potentially, the number of different frequency levels implemented by upcoming CPUs will even grow to enable better utilization of power budgets for a wider range of code. We therefore expect the results of this thesis to be applicable to increasing numbers of CPUs in the future.

4 Runtime Profiling of AVX2 and AVX-512 Overhead

In the previous chapter, we showed how AVX2 and AVX-512 code often causes other code to be slowed down. These experiments demonstrate the need for a profiler that quantifies this remote AVX overhead. In particular, the experiments show the large performance impact and show how the origin of the remote AVX overhead can be non-obvious given that the affected code is often completely unrelated to the AVX2 or AVX-512 code. In this chapter, we present a profiler which is able to quantify remote AVX overhead and which provides information about the reasons for remote AVX overhead. Developers and system administrators both benefit from having such information. While developers require knowledge of the amount of overhead to make design decisions such as whether or not to use AVX2 or AVX-512, system administrators need to be able to make informed decisions on whether to deploy countermeasures such as those described in the following chapters.

Our profiler is often even suited for use directly in production environments, unlike the technique used to measure remote AVX overhead in the previous chapter. Our previous experiments relied heavily on the ability to configure the use of AVX2 and AVX-512 in the workloads as we compared the CPU time of variants of the same workload with and without these instructions. This reconfiguration is problematic in most real-world situations where developers or system administrators want to analyze their own workloads. In particular, the experiments are rather labor-intensive as each measurement not only requires multiple runs of a benchmark but may in some cases even require manual modification of the applications involved to make use of AVX2 and AVX-512 configurable. We, for instance, needed to modify the OpenSSL library to reenable AVX-512 implementations that had previously been deactivated by the OpenSSL developers. Also, the experimental setup requires the workload to be restarted to reconfigure the use of AVX2 and AVX-512. Often, such intervention in the operation of applications is outright impossible. If, for example, a cloud provider wants to investigate remote AVX overhead caused by the colocation of virtual machines from different customers, the cloud provider is unable to modify and restart these virtual machines at runtime. Finally, even if restarting applications is possible, it may not be desirable. For example, taking a web service offline often results in a degraded user experience even if the service is only stopped for a brief moment.

In contrast, in this chapter we describe a non-intrusive profiler that is able to quantify remote AVX overhead at runtime with very little negative impact on the system [87]. We start the chapter by reviewing existing related approaches to profiling in Section 4.1. In Section 4.2, we then describe *remote AVX overhead sampling* as the underlying working principle of our profiler. The profiler periodically pauses the software running on individual CPU cores and analyzes the resulting frequency changes to determine whether

any frequency reduction present on the CPU is actually required by the current task. We show how this technique can be extended to differentiate between the remote frequency reduction caused by sibling hyper-threads and that caused by the frequency change delay. We then show how the frequency reduction translates into overhead in Section 4.3, presenting a model that predicts instruction throughput at different CPU frequencies for the current workload. In Section 4.4, we describe a technique to identify AVX2 and AVX-512 processes that can potentially cause a frequency reduction. This technique can be used to determine the processes responsible for the overhead detected by the profiler. We then present an evaluation of the profiler based on the benchmarks from the previous chapter in Section 4.5. Our evaluation indicates that our profiler can determine the remote AVX overhead in a wide range of benchmarks with an average error of only 2.2% percentage points. Our experiments also show that our profiler can be used in many production environments with negligible impact on quality of service as it only slows the system down by 2.1% on average. Finally, we discuss limitations of our profiler as well as potential improvements in Section 4.6.

4.1 Existing Profilers

In many situations, software developers and users require information about nonfunctional application properties. In particular, information about resource usage helps finding bottlenecks that restrict performance and allows selective optimization of those bottlenecks. Profilers – tools to collect such information about nonfunctional properties at runtime – were originally proposed as an optimization aid that determined how often individual parts of programs were executed [135]. Since then, profilers were extended to cover many different properties ranging from physical properties such as energy consumption [214] to microarchitectural events that allow software developers to determine the nature of bottlenecks [259]. While there are many different approaches to profiling depending on the supported hardware and the information to be collected, most profilers are either instrumentation-based or sampling profilers.

Instrumentation-based profilers are tools that modify programs to insert code that collects the required information [135]. For example, if each basic blocks of a program is modified to increase a separate counter, the resulting information can be used to determine the execution frequency of all statements of the program which allows the developer to identify frequently executed code. Alternatively, the inserted code can utilize the performance monitoring unit (PMU) found in recent CPUs such as those covered by this thesis [116, pp. 18.1 ff.]. The PMU can be configured to count a range of microarchitectural events with implications for performance. If the profiler, for example, inserts code that uses the PMU to read the cumulative number of cache misses at various points in the program, the resulting information can be used to identify particularly memory-heavy parts of the program [7].

The main advantage of instrumentation-based profilers is that they are able to produce a very detailed description of the behavior of the program if it is instrumented with sufficiently high granularity. However, doing so can introduce substantial overhead both in terms of program size and execution time. If a profiler wants to measure the execution

frequency of all basic blocks, even optimal placement of the inserted code results a 2× slowdown for some programs [14]. Such overhead limits usage scenarios where a specific minimum performance is required – for example, it is likely rarely viable to employ instrumentation-based profiling in a production server environment. The overhead also causes what is commonly called *instrumentation perturbation* [165]: As additional code is executed, the timing behavior of the software changes resulting in modified memory access patterns and event reordering, both of which affect the quality of the profiling results.[1]

Even though instrumentation-based profiling is able to use the PMU for flexible microarchitectural analysis, such a profiler is not usable to measure remote AVX overhead due to the reasons presented in Section 3.4. In particular, although the PMUs of recent Intel CPUs are able to count the cycles spent at the three frequency levels – non-AVX, AVX2, and AVX-512 frequencies – they do not provide any information about whether a frequency reduction was necessary (local AVX overhead) or unnecessary (remote AVX overhead) for the currently executed part of the workload.

In contrast to instrumentation-based profilers, *sampling profilers* achieve far lower overhead, sometimes even enabling continuous profiling on production systems [8]. Sampling profilers periodically interrupt the program, each time recording the location where the program was interrupted [135]. The profilers then translate the resulting samples into information similar to that provided by instrumentation-based profilers. If we assume that sampling occurs at fixed time intervals, then the number of interrupts within a code section correlates with the time spent in that section. On most systems, the same approach can also be used to measure the occurrence of other performance events such as cache misses [8] as the PMU can be configured to generate an interrupt after a fixed number of events. In this case a large number of interrupts signal a large number of such events in the corresponding part of the program. As described above, this approach is not usable to measure remote AVX overhead, though, as the CPU lacks appropriate performance events.

Although literature does not describe any alternative to measure remote AVX overhead, some related work strives to achieve similar goals. For example, the Linux kernel provides an interface to detect whether individual tasks use AVX-512 [235]. During each context switch, the kernel tests whether the 512-bit vector registers contain valid content and marks the task as an AVX-512 task if they do. While this interface can be used to identify tasks that could potentially cause remote AVX overhead, the kernel does not actually measure the overhead. In fact, the mechanism would not be usable as part of a profiler which measures remote AVX overhead for three reasons. First, as described by Linux documentation [235], the values reported depend on the rate of context switches. Second, temporary use of 512-bit vector instructions remains unnoticed by the kernel if the application clears the registers using the `VZEROUPPER` instruction before the next context switch [115, p. 5-550] as suggested by documentation from Intel [113, p. 17-57].[2] Third, the mechanism is unable to differentiate between light and heavy 512-bit instructions, even if they cause different degrees of frequency reduction.

[1] Other sources use the terms invasiveness [178] or intrusiveness [10] to describe the impact of the profiler on application behavior.

[2] The Linux kernel documentation states that performance counters can be used to obtain precise information [235]. As we show in Section 3.4, this is not the case.

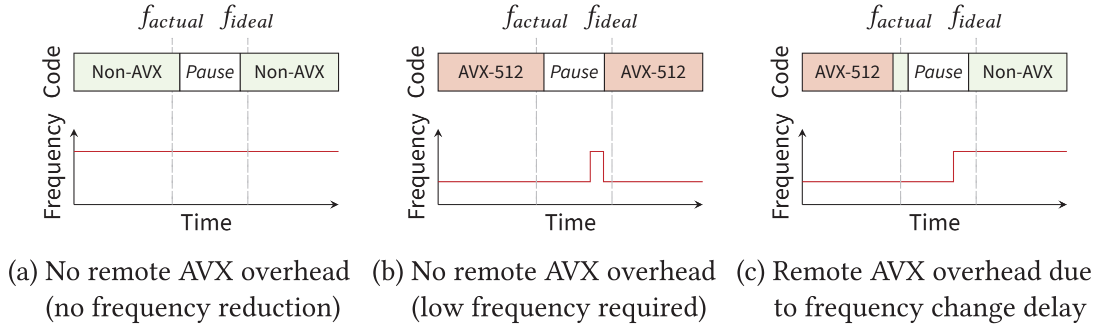
The work performed by Kumar et al. [142] is more similar to our goal of determining the amount of remote AVX overhead. Kumar et al. present a technique to determine whether code should use SIMD instructions or whether the corresponding function units should be power-gated. Their approach uses instrumentation-based profiling based on a runtime translation layer. The layer adds code to all basic blocks which counts how often individual blocks are executed and which uses a bit mask to track whether any recently executed instructions were SIMD instructions. To conserve energy, basic blocks are then devectorized by the translation layer if too few recently executed instructions were SIMD instructions. While we assume that the instrumentation-based profiler could potentially be modified to measure remote AVX overhead, the approach suffers from two substantial drawbacks. First, as with any instrumentation-based profiler, the approach requires modifications to application code which may be difficult in, for example, cloud computing or any other multi-tenant environment. Second, the analysis only covers a single application and does not cover the operating system. In our case, the performance of system calls in particular is often negatively affected by remote AVX overhead. In the case of the nginx benchmark with AVX-512-enabled OpenSSL described in Chapter 3, our experiments showed that the network stack was commonly affected by a frequency reduction caused by preceding calls to OpenSSL. Therefore, the sampling profiler presented in the next sections covers not only all applications executed on the system but also large parts of the operating system kernel.

4.2 Detecting Unnecessary Frequency Reduction

In Chapter 3, we defined remote AVX overhead as the performance impact of remote frequency reduction, i.e., it is the impact of 256-bit and 512-bit SIMD code on the frequency of other less power-intensive code. In relative terms, remote AVX overhead can be expressed as the *actual CPU time* required by a task divided by its *ideal CPU time*, i.e., the CPU time that would be required by the task if there was no remote AVX overhead. These times, in turn, are determined by the average *actual CPU frequency* during execution of the task and the average *ideal CPU frequency* at which the task could have been executed in the absence of any AVX2 or AVX-512 frequency reduction. Therefore, the remote AVX overhead can be expressed as follows, where $\Delta_{perf}(f_1, f_2)$ is a function that calculates the performance change caused by a given frequency change:

$$o = \frac{t_{actual}}{t_{ideal}} = \Delta_{perf}(f_{actual,avg}, f_{ideal,avg}) \tag{4.1}$$

Recent Intel CPUs do not provide any low-overhead method to continuously monitor the ideal CPU frequency. Therefore, we propose a sampling profiler which periodically selects an individual logical CPU (i.e., a hyper-thread) in a round-robin fashion to sample the *unnecessary frequency reduction* (i.e., f_{actual} and f_{ideal}) at the given time [87]. For each task, our profiler stores the number of samples during execution of the task. The profiler also maintains two sums corresponding to the approximate actual performance (p_{actual})



(a) No remote AVX overhead (no frequency reduction)
(b) No remote AVX overhead (low frequency required)
(c) Remote AVX overhead due to frequency change delay

Figure 4.1: Unnecessary frequency reduction – the difference between actual CPU frequency and the ideal frequency for the currenty executed code – is measured by temporarily pausing individual CPU cores and then resuming a single hyper-thread [87]. During this process, the CPU first switches to the non-AVX frequency level and then to the ideal frequency for that hyper-thread. In the absence of remote AVX overhead, frequency measurements f_{actual} and f_{ideal} before and after these steps are identical (a-b). Differences indicate remote AVX overhead (c).

and the approximate performance at ideal frequencies (p_{ideal}) and uses those two sums to create an estimate of the remote AVX overhead:

$$o \approx \frac{p_{ideal}}{p_{actual}} = \frac{\sum f_{actual} \Delta_{perf}(f_{actual}, f_{ideal})}{\sum f_{actual}} \tag{4.2}$$

Unnecessary frequency reduction – i.e., a difference between f_{ideal} and f_{actual} – occurs if a task does not use power-intensive SIMD instructions, yet the CPU core has recently executed power-intensive SIMD instructions or the other hyper-thread of the physical CPU core currently executes such instructions. In the following, we describe a technique to sample both frequencies as well as a technique to differentiate between overhead caused by hyper-threading and overhead caused by frequency change delays. Afterwards, in Section 4.3, we describe how to determine the performance reduction caused by a frequency change, i.e., we describe our implementation of Δ_{perf}.

4.2.1 Frequency Reduction Sampling

Whereas the actual frequency can easily be measured using performance events supported by the PMU, the CPU lacks any performance event that allows direct measurements of the ideal frequency. Instead, we propose a scheme where, whenever sampling a hyper-thread, the profiler forces its physical CPU core to temporarily return to the non-AVX frequency level. If the CPU then does not directly return to its previous lower frequency, the frequency reduction is not necessary for the currently executed code. As shown in Figure 4.1, this scheme involves a number of steps that are executed whenever a hyper-thread is sampled:

1. The profiler measures the actual frequency (marked as f_{actual} in Figure 4.1). We measure the frequency by executing 1000 `NOP` instructions while the PMU is configured to count CPU cycles as well as the fixed-frequency reference cycles. The

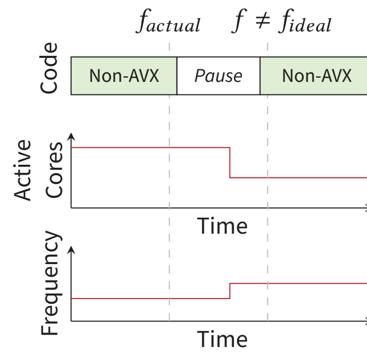


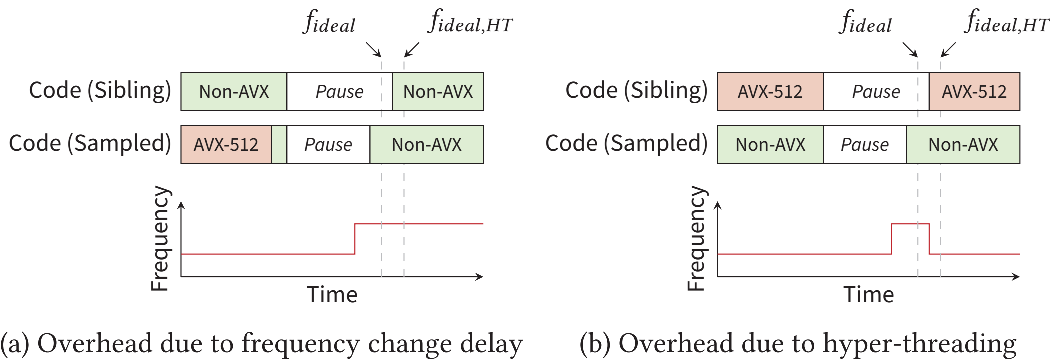
Figure 4.2: If the number of active cores changes during the pause induced by the profiler, the difference between the CPU frequencies before and after the pause is not representative of the remote AVX overhead anymore. In the example shown in the figure there is no remote AVX overhead, yet the frequency increases during the pause. To isolate the impact of AVX instructions, our profiler only measures f_{actual}, and uses this frequency together with the AVX frequency level to calculate f_{ideal} under the assumption that the number of active cores does not change.

ratio between the two values is proportional to the CPU frequency. In addition, the profiler configures the PMU to count the cycles at the three frequency levels as described in Section 3.4. Most commonly, only one of the three counters produces a non-zero result, which indicates the current frequency level of the CPU.

2. The profiler sends an inter-processor interrupt to the sibling of the sampled hyper-thread to instruct it to spin in the kernel while executing only non-AVX instructions. The sampled hyper-thread itself then proceeds by spinning for 700 µs while interrupts are disabled on both hyper-threads. As a result, the physical core does not execute any SIMD instructions for a period longer than the AVX frequency change delay, forcing it to transition to the non-AVX frequency level if it was previously operating at a lower frequency level.

3. While the sibling hyper-thread continues spinning, the hyper-thread to be sampled continues regular operation. As the previously executed task continues, any power-intensive SIMD instructions executed by the task, if present, quickly force the CPU to assume a lower frequency level. Our profiler sleeps for additional 70 µs to allow the workload to affect the CPU frequency. After this brief delay, the CPU frequency equals the ideal frequency for the currently executed code (marked as f_{ideal} in Figure 4.1).

   Usually, the remote frequency reduction can be calculated as the difference between this frequency after the pause and the frequency before the pause. There is one exception, though, as shown by Figure 4.2. If the number of active CPU cores changes during the pause, the turbo level changes, so the frequency difference does not represent the impact of AVX2 and AVX-512 anymore. This example shows that our profiler is not actually interested in the ideal frequency after the pause, but rather has to determine what would have been the ideal CPU frequency at the time of step



(a) Overhead due to frequency change delay (b) Overhead due to hyper-threading

Figure 4.3: To determine the source of remote AVX overhead, our profiler optionally measures the frequency again 70 µs after resuming the sibling hyper-thread. A further frequency change, i.e., a difference between f_{ideal} and $f_{ideal,HT}$ indicates that the sibling hyper-thread causes remote AVX overhead.

1. We therefore do not simply measure the frequency again after the pause. Instead, the profiler measures the frequency level – non-AVX, AVX2 or AVX-512 – using the technique from step 1 again. The profiler then uses both frequency levels as well as the initial frequency measurement to determine what would have been the ideal frequency before the pause.

   This calculation is performed with the help of a table of the CPU frequencies at the different turbo levels such as Table 3.1. The frequency measurement taken in step 1 is first compared to all frequencies for the initial frequency level to determine the number of active cores, and the second frequency level is then used to lookup the ideal frequency for this number of cores.

4. Finally, the sibling hyper-thread resumes regular operation.

While the steps above allow the profiler to determine the remote frequency reduction, they do not provide any information on the reason for the frequency reduction. As described in Section 3.3.2, remote frequency reduction can be caused either by frequency change delays or by the sibling hyper-thread's parallel execution of AVX2 or AVX-512 code. Often, the user profits from knowing which of the two mechanisms contributes to remote AVX overhead, as techniques to mitigate remote AVX overhead may only target overhead caused by one of them. For example, core scheduling [154] only mitigates remote AVX overhead caused by hyper-threading, whereas modifications to the CPUs DVFS policy such as that presented in the next chapter mainly influence the impact of frequency change delays [86]. To be able to differentiate between the two types of overhead, our profiler optionally adds an additional step to the steps described in the previous section:

5. As shown in Figure 4.3, the profiler waits for additional 70 µs and measures the frequency level again. Based on this frequency level, it calculates the ideal CPU frequency $f_{ideal,HT}$ for the code on both hyper-threads combined. As any frequency reduction after the previous step is caused by the code running on the sibling hyper-thread, this final frequency enables the profiler to differentiate between the two

sources for remote AVX overhead. Whereas Figure 4.3a shows a situation where all reduction is caused by frequency change delays and the CPU frequency remains high, Figure 4.3b shows a situation where resuming the sibling hyper-thread causes the frequency to drop to its original level. Consequently, the remote frequency reduction caused by hyper-threading can be calculated as the difference between $f_{ideal,HT}$ and f_{ideal}, while the impact of frequency change delays can be calculated as the difference between $f_{ideal,HT}$ and f_{actual}.

As the additional timer interrupt slightly increases the overhead caused by the profiler and we expect the additional information to be only infrequently required, we disable this final step by default and only enable it when the user requests an analysis of the sources of overhead.

Once these steps have completed, the next logical CPU to be sampled is selected in a round-robin fashion. A high-resolution timer is configured to interrupt that logical CPU at a random time between 10 and 30 ms in the future. Once elapsed, this timer causes the selected logical CPU to sample its frequency reduction. Similar to Anderson et al. [8], we randomize the time between samples to prevent aliasing effects – if the workload showed periodic behavior, a fixed sampling frequency may fail to sample parts of the workload. We select an average time of 20 ms between samples as we found the resulting sample rate to provide accurate profiling results at acceptable overhead. See Section 4.6.1 for a more complete discussion of the trade-off between accuracy and overhead.

This timer provides a non-obvious source for systematic measurement error when implemented incorrectly. In particular, the implementation must ensure that the timer interrupt does not activate an additional CPU core. Such additional active cores would artificially increase the amount of remote AVX overhead because the difference between the frequency levels depends on the number of active cores [119]. Whereas, for example, the Intel Xeon Gold 6130 CPU reduces its frequency by 400 MHz when four CPU cores are active, this difference increases to 1 GHz for five active cores. While our profiler, as described above, tries to compensate for turbo level changes during the pause, it is unable to detect the impact of additional active CPU cores if the whole sampling process is affected, i.e., if the CPU switches to a lower turbo level before the initial frequency measurement. In Linux, high-resolution timers are local to the logical CPU on which they are started, so we prevent additional active CPU cores by always starting the timers on the logical CPU that is supposed to be sampled next.

Another source for systematic measurement error stems from scheduling – if a new task is selected during the sampling process, the ideal frequency determined after the pause does not correspond to the task executed before the pause. We therefore skip samples where we detect a context switch between steps 1 and 4.

4.3 DVFS Performance Prediction Model

Once the amount of unnecessary frequency reduction has been calculated, the frequency reduction needs to be translated into an estimate of the performance reduction (Δ_{perf} in Equation 4.2). While the throughput of some applications scales linearly with CPU

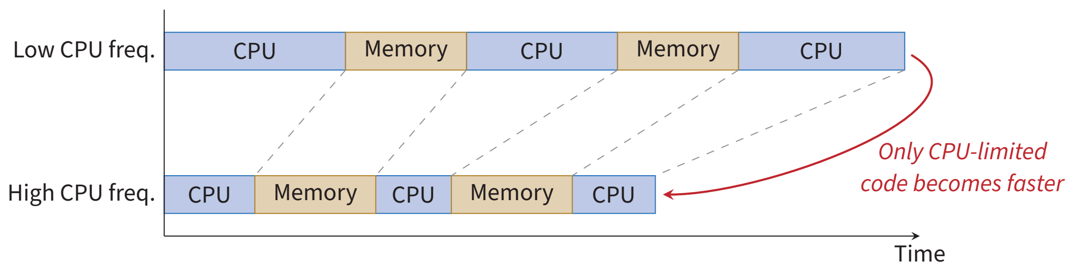


Figure 4.4: Depiction of the CPU time required to execute a program that alternates CPU-heavy phases with memory accesses – whenever the CPU is waiting for memory, frequency changes do not impact performance. The figure is simplified – in practice, CPUs overlap memory accesses with other instructions.

frequency, for others, it does not [101]. As we described earlier in Section 2.2.3, the reason for this behavior of different workloads can be found in *memory slack*. Whenever, as shown in Figure 4.4, the CPU is stalled waiting for resources whose latency does not vary based on the CPU frequency, this frequency does not have any impact on performance. Such resources include memory as well as large high-latency last-level caches that are operated at a frequency different to the frequency of the CPU cores. As a result, memory-intensive applications show particularly little performance changes when the CPU frequency is changed.

The share of the time spent waiting for memory depends not only on the number of memory accesses but also on the nature of the accesses. Even memory accesses with identical latency do not cause the CPU to stall for the same amount of time, as modern microarchitectures are often able to hide the latency of memory accesses via speculative execution, out-of-order execution, store buffers, and support for multiple outstanding loads. In addition, the latency of memory accesses is influenced by external factors such as memory pressure generated by other cores. For example, most SPEC benchmarks show less performance change when executed multiple times in parallel as the additional copies contribute to memory pressure [101]. All these factors make it rather hard to predict the performance of an application at a different frequency.

4.3.1 Existing Models

Existing approaches to DVFS performance prediction commonly try to reduce complexity by creating simplified models of the system's behavior. *Empirical models* form one class of such models [229]. These models are created using a black-box approach and exploit the correlation between performance changes caused by DVFS and performance events reported by the hardware. For example, Weissel et al. [251] characterize system behavior with a wide range of microbenchmarks which vary both in the rate of memory accesses and total instructions executed per cycle. They manually determine the frequency at which these benchmarks are slowed down by 10% and use the resulting data to predict the performance impact of frequency changes on other applications based on their characteristics.

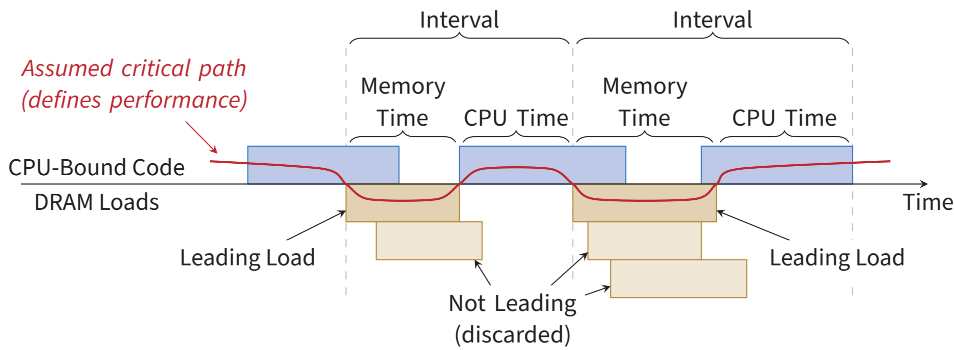


Figure 4.5: The leading-loads model divides time into intervals consisting of a leading load and subsequent CPU-bound code. Any load started during the leading load is ignored. The model assumes that the time during leading loads remains constant, whereas the remaining time scales inversely to the CPU frequency. Figure adapted from [229].

Snowdon et al. [224] instead divide total time into CPU time, memory time, and bus time and calculate the latter two times as a linear combination of a number of performance counters. They calculate suitable coefficients for this linear combination via linear regression and select appropriate performance events using an exhaustive search. Similarly, Lee et al. [147] notice that the cycles per instructions (CPI) at a different frequency depend on the CPI at the current frequency as well as the ratio between memory instructions and total instructions. The resulting model effectively divides time into CPU-bound time and memory-bound time.

Empirical models have two main drawbacks resulting from their need for training data for calibration of model parameters. First, collecting this training data can be time consuming, which is particularly problematic as the resulting model is specific to the system configuration. Any changes not only to the CPU but also to hardware parameters such as memory latency or throughput affect the model's accuracy and may require constructing a new model. Second, and perhaps more importantly, the correlation between performance events and the impact of frequency changes may only be present for specific types of workload, whereas the model may greatly over- or underestimate performance changes for other applications. For example, models such as that by Snowdon et al. [224] which divide time into purely CPU-bound time and purely memory-bound time are able to produce good predictions for workloads where CPU-bound code and memory accesses show little overlap. Modern out-of-order CPUs are often able to overlap the two to hide the latency of memory accesses, though.

It is hard to model the impact of overlap between CPU-bound code and memory accesses using empirical models as the impact depends on the structure of the program and the properties of memory accesses. While, for example, Li et al. [155] describe an approach to DVFS prediction which takes overlaps into account, their design produces a separate model for each individual application. This requirement for separate models stems from the fact that the models estimate the impact of overlap time based on the computational intensity of individual applications.

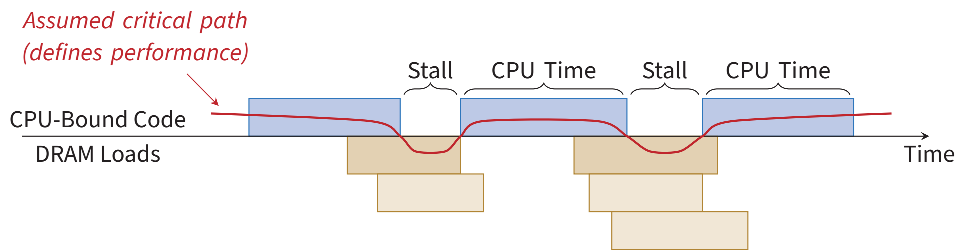


Figure 4.6: Stall cycle counting is a model which assumes that time while the CPU is stalled remains constant whereas all other time scales inversely to the CPU frequency. In contrast to the leading-loads model, overlaps between memory instructions and other instructions are ignored.

In contrast to empirical models, *mechanistic models* are based directly on the mechanisms that influence the impact of frequency changes on performance [229]. By modelling these mechanisms using corresponding hardware events, empirical models commonly eliminate the need for calibration. One mechanistic model that has shown to provide good accuracy for recent superscalar out-of-order CPUs is the *leading-loads* model [72, 130, 207]. The leading-loads model is based on the observation that overlapping loads commonly do not contribute to the number of stall cycles. Instead, performance is defined by the time required for a critical path through the program consisting of non-overlapping loads and the CPU-intensive code between them, depicted by the red line in Figure 4.5. When this critical path leads through non-overlapping memory accesses, CPU frequency changes are assumed to have no impact on performance, while all remaining time is assumed to scale according to the frequency. The non-overlapping memory accesses are called *leading loads*.

To identify overlapping loads, the leading-loads model makes the simplifying assumption that, as shown in Figure 4.5, the CPU time of any piece of code is defined by a series of intervals consisting of a memory access – the leading load – followed by CPU-bound code that depends on this memory access [72, 207]. Any memory accesses started while the leading load is still in flight are discarded by the model. Instead, once the leading load has completed, the next load access is classified as a leading load.

While this model is able to capture the parallelism provided by superscalar processors, it requires performance monitoring events that are not widely available. The only practical application of a leading-loads model to date has been described by Su et al. who show that the *miss address buffer* provided by some AMD CPUs can be used to identify leading loads [229]. No similar mechanism is available for the Intel CPUs covered by this thesis, making the leading-loads model impractical on such systems. The same applies to extensions to the leading-loads model such as CRIT+BW [169] which includes factors such as variable memory access latencies, prefetching and limited memory bandwidth.

Stall cycle counting [72, 130] as shown in Figure 4.6 is a simpler model that can be implemented using the existing performance counters found in almost all recent CPUs. The model assumes that there is negligible overlap between memory accesses and other instructions and that therefore all time while the CPU is not stalled due to memory accesses scales inversely to the CPU frequency. In contrast to empirical models that perform a

similar classification of time, stall cycle counting uses performance events that directly measure stall cycles and therefore does not require any calibration. Otherwise, these models are fairly similar – unlike the leading-loads model, neither models the overlap between memory accesses and other instructions. Nevertheless, it has been shown that stall cycle counting is able to predict performance with an error of only 2.1% on a simulated out-of-order CPU [130].

4.3.2 Stall Cycle Counting for Intel Skylake-SP

For our profiler, we implement a model based on stall cycle counting as the technique has shown to be easy to implement, yet provides estimates with acceptable prediction errors. The main challenge when implementing stall cycle counting is the choice of suitable performance events. On Intel Skylake-SP CPUs, the L2 cache operates at the same frequency as the CPU core while the L3 cache does not [231, 95], so similar to existing models [130] we choose the performance event that represents stalls due to L2 misses caused by demand loads – `CYCLE_ACTIVITY.STALLS_L2_MISS` [116, p. 19-15] – to count the stall cycles due to memory loads. In addition, we noticed that some benchmarks frequently stall due to full store buffers, whereas existing models ignore stalls due to store instructions. Unfortunately, the hardware does not provide any performance event that directly counts stall cycles while the store buffer is full. Instead, the CPU only provides `EXE_ACTIVITY.BOUND_ON_STORES` [116, p. 19-16] which counts all cycles where the store buffer is full yet there is no outstanding load. The event includes cycles when the CPU is not stalled, so we make the simplifying assumption that the CPU is always stalled when the store buffer is full. Note that the performance counter also does not differentiate between the different cache levels and therefore includes stores that miss L1 but hit the L2 cache. We assume that due to the low latency of the L2 cache most of the stalls are caused by L2 misses, though.

We calculate the sum of the two performance events and divide it by the total number of CPU cycles to calculate the time during which frequency changes have an impact on performance. This time is then used to predict the expected speedup or slowdown after an assumed frequency change according to the following equation where f_{now} and f_{next} denote the current and the future frequency, respectively, c_{total} denotes the total CPU cycle count, and $c_{l2-stall}$ and $c_{store-bound}$ correspond to the two aforementioned performance events.

$$\Delta_{perf}(f_{now}, f_{next}) = \frac{f_{next}}{\frac{c_{l2-stall}+c_{store-bound}}{c_{total}} * (f_{next} - f_{now}) + f_{now}} \tag{4.3}$$

To test the quality of the resulting model, we compare its predictions to direct measurements of the corresponding slowdown. For our experiment, we let the model predict the performance change for a range of SPEC CPU 2017 benchmarks when the frequency is changed from 2.8 GHz to 1.9 GHz. We use these frequencies as they represent the largest difference between the AVX-512 and non-AVX frequency levels on the Intel Xeon Gold 6130 CPU [119] that is used for the evaluation of our prototype. SPEC CPU 2017 is suitable for our experiment as it was shown to contain both benchmarks that scale well with increasing CPU frequency such as 538.imagick_r as well as benchmarks whose performance

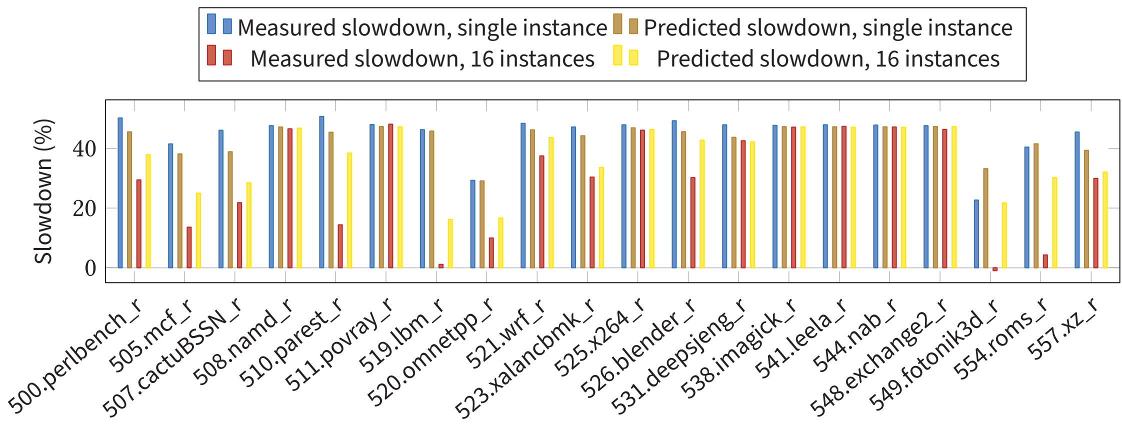


Figure 4.7: We evaluate the accuracy of our DVFS model by comparing its predictions to direct measurements of the impact of a frequency change from 2.8 GHz to 1.9 GHz on benchmark completion times. Our model consistently overestimates the impact of frequency changes as it does not take overlap between memory accesses and other instructions into account.

is hardly affected by frequency changes such as 549.fotonik3d_r [101]. We repeat our experiments with one SPEC benchmark instance as well as with 16 instances to simulate situations with different amounts of memory pressure as the number of instances was shown to have substantial impact on the behavior of the benchmarks [101].

Executing the benchmarks at 2.8 GHz is only possible if they do not execute AVX2 or AVX-512 instructions. As some of the SPEC CPU 2017 benchmarks use the libmvec library which provides SIMD implementations for math functions [158], we used the environment variable `GLIBC_TUNABLES` to force the library to select implementations that do not cause any frequency reduction.

Figure 4.7 shows the results of our experiments. For many benchmarks – especially largely CPU-bound benchmarks – our model is able to closely predict the slowdown caused by the frequency reduction. On average, our model predicts the performance with an error of 4.3%. The error varies substantially across different benchmarks, though. For memory-intensive benchmarks in particular, our model consistently overestimates the impact of frequency changes on performance. This is expected given that our model ignores the overlap between memory accesses and CPU-bound code – instead, CPU performance during phases with overlap is assumed to scale linearly with the CPU frequency.

Previous work by Keramidas et al [130] evaluated stall cycle counting on a simulated out-of-order CPU and therefore faced the same limitation. Nevertheless, the researchers report a better average error of only 2.1%. We see two main reasons for the worse results or our model. First, the CPU used for our experiments is far more complex than the CPU simulated by Keramidas et al. For example, they simulated a CPU with a 64-entry reorder buffer, whereas the reorder buffer of the Skylake-SP microarchitecture provides space for 224 entries.[3] Skylake-SP CPUs are therefore likely more often able to overlap

[3] Keramidas et al. state that they varied parameters such as the reorder buffer size and achieved similar results, but do not provide any concrete values [130].

memory accesses with other instructions. Second, our model assumes that the latency of memory accesses does not change if the CPU frequency is changed. This is mostly true for accesses to external memory, but not for accesses to the last-level cache. In particular, Intel Haswell CPUs were shown to increase their uncore frequency – and thereby, consequently, the frequency of the last-level cache – when the CPU core frequency was reduced [95]. We assume that the CPU used in our experiments implements a similar scheme to utilize thermal headroom. In our case, such behavior results in higher uncore frequencies and consequently lower L2 cache hit latencies during benchmark execution at 1.9 GHz. The resulting performance improvement at this lower frequency causes our model to further overestimate the performance impact of frequency changes for memory-intensive applications which frequently hit the L3 cache.

While we assume that variable uncore frequencies impact the quality of our performance model, we expect their impact to be much lower in real-world workloads with AVX2 and AVX-512. Whereas the frequency reduction in the experiments shown in this section likely resulted in substantial power reduction, AVX2 and AVX-512 reduce the frequency because the CPU would otherwise violate power limits. Workloads with AVX2 and AVX-512 would therefore leave far less thermal headroom for the CPU to increase its uncore frequency. Note that uncore frequency changes likely have a similar impact on most other types of models. We tried creating an empirical model based on an exhaustive search of suitable performance events using the methodology of Snowdon et al. [224], but were unable to improve upon the prediction quality of stall cycle counting.

In our profiler, we apply our performance model after each sample as shown by Equation 4.2. We count the required performance events during the 70 μs window after the pause introduced by our profiler. We substitute f_{now} and f_{next} with the actual and ideal frequency, respectively, as shown in Equation 4.2.

4.4 Source of Overhead

While the design described above allows us to determine both the amount of unnecessary frequency reduction as well as the resulting remote AVX overhead affecting individual tasks, it does not provide any information on the origin of the remote AVX overhead. Often, this information is crucial to mitigation the overhead, though. A system administrator faced with substantial remote AVX overhead could, for example, move tasks to a separate CPU core if they affect the performance of other tasks.

There are existing mechanisms designed to identify problematic AVX-512 tasks that can cause remote AVX overhead, yet all fail to identify the source of remote AVX overhead with sufficient accuracy. For example, as described in Section 4.1, the Linux kernel tests whether the upper halves of the 512-bit vector registers hold valid contents during each context switch [235]. This technique identifies some AVX-512 tasks, but does not provide any information on whether the task executed heavy or light 512-bit SIMD instructions. However, this information is also important as the frequency levels caused by the two types of instructions differ. If we assume a system executing a task with heavy 512-bit instructions and another task with light 512-bit instructions, the former task causes remote AVX overhead while the latter task is negatively impacted by the overhead. In addition,

the interface often fails to notice tasks as register contents are only infrequently checked as described in Section 4.1. It is possible to solve the latter problem by restricting access to the 512-bit register file so that the first 512-bit SIMD instruction during each time slice triggers an exception [85]. While we use this mechanism in Chapters 6 and 7, it is not able to differentiate between heavy and light SIMD instructions, either.

The performance events implemented by the CPU do not provide any simple method to identify tasks that cause remote AVX overhead, either. The `CORE_POWER.THROTTLE` performance event [116, p. 19-8], for example, is often observed during execution of tasks that cause remote AVX overhead as it counts the throttle cycles caused by frequency level changes. The performance event does not differentiate between the hyper-threads of a single physical core, though, so any of the applications running on the core during the occurrence of the event can be the task responsible for the remote AVX overhead. In addition, if multiple subsequent tasks execute, for example, heavy 512-bit SIMD instructions, only the first triggers a frequency reduction, so the performance event is never triggered for the other tasks even if they contribute equally to remote AVX overhead.

Instead of these incomplete approaches, we exploit the fact that our profiler is already able to detect frequency reduction caused by AVX2 and AVX-512. In Section 4.2, we described how our profiler samples the frequency level before and after a pause. Different frequency levels indicate remote AVX overhead, as they show that the current task can be executed at a higher frequency than present before the pause. To detect tasks which cause remote AVX frequency reduction, we invert this logic – if the frequency remains low even after the pause, the current task requires a frequency reduction and, consequently, may cause other tasks to experience remote AVX overhead. Whenever our profiler detects this situation, it flags the current task according to the frequency level required.

While this technique likely provides greater accuracy as the alternatives listed above, it is not perfect, either. In particular, insufficient sampling rates or observation periods may caused the profiler to misidentify tasks that infrequently execute very small amounts of AVX-512 code. In Section 4.6 we discuss improved hardware/software interfaces that allow for continuous monitoring of tasks and therefore do not suffer from the statistical error introduced by sampling.

4.5 Evaluation

To be useful in a wide range of situations, profilers not only need to provide accurate results. In addition, they must not introduce excessive overhead themselves, as such overhead impacts application behavior and prevents use in production environments where performance changes impact the quality of service. To determine how well our design is able to fulfill these two goals, we implement a prototype in the form of a loadable module for the Linux kernel. As floating-point math is generally discouraged in the kernel [31, p. 114], we implement our DVFS prediction model using fixed-point math with 16 fractional bits.

In the following, we describe our evaluation to quantify accuracy and overhead before presenting additional experiments that show the impact of different design decisions. All experiments described in this section are executed on a system with a Xeon Gold 6130

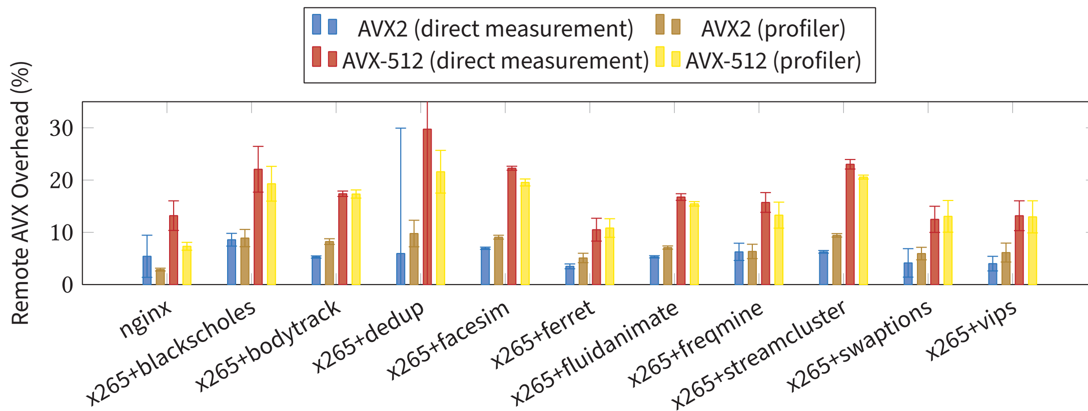


Figure 4.8: A comparison to direct measurements of the remote AVX overhead shows that our profiler is commonly able to determine the remote AVX overhead with very little error.

CPU, 24 GiB DDR4 RAM operating at 2666 MHz, Fedora Linux 31, and the Linux 5.6.13 kernel.

4.5.1 Profiler Accuracy

To measure the accuracy of the output of our profiler, we compare it to the results of the experiments described in the previous chapter. As these results were obtained using direct measurements of remote AVX overhead, they provide a suitable ground truth. We therefore replicate the exact experimental setup described in Section 3.3.5 where we execute benchmarks from the Parsec benchmark suite alongside the x265 video encoder and configure the video encoder to either use AVX2 or AVX-512 instructions. We choose x265 over OneDNN as the behavior of x265 is less homogeneous and therefore likely provides a bigger challenge for our profiler. We exclude Parsec canneal as our previous analysis shows that the IPC variation during execution of the benchmark causes performance changes of the same order of magnitude as those caused by remote AVX overhead. In addition, we replicate the experiment involving the nginx web server and OpenSSL and vary the SIMD instructions used by OpenSSL. We instruct our profiler to measure the remote AVX overhead inflicted on the Parsec benchmarks and the nginx web server.

Figure 4.8 shows both direct measurements of the remote AVX overhead generated in the previous chapter as well as the results provided by our profiler. The experiments show that our profiler is able to reliably determine the order of magnitude of the remote AVX overhead present in the workloads. Often, this information alone is sufficient for the user to make informed decisions on, for example, the use of countermeasures against remote AVX overhead.

While such decisions most likely rarely require precise information on the amount of remote AVX overhead, the experiment also shows that our profiler is commonly able to quantify the overhead with very high precision. For AVX-512, the output of our profiler has an average error of only 2.4 percentage points. For AVX2, our profiler achieves an

almost identical average error. Overall, our profiler tends to overestimate the impact of AVX2 and tends to underestimate the impact of AVX-512.

Besides instrumentation perturbation – which we analyze in the following section – we identify two potential sources for error in the core mechanism of our profiler. First, each sample takes considerable amounts of time as the CPU is first paused for 700 µs and then the application is monitored for further 70 µs. The length of the latter phase defines the temporal resolution with which the profiler can detect remote AVX overhead. If the application did not use AVX-512 before the pause but then starts to use AVX-512 instructions during the second phase, the frequency is not deemed to be unnecessarily low at the time of the sample even though the application could have been executed at a higher frequency before the pause. In theory, such situations should cause our profiler to underestimate the remote AVX overhead of workloads such as nginx that frequently switch between AVX-512, AVX2, and other code.

Second, our profiler fails to take into account that sometimes more than one application thread is required to cause a frequency reduction. In particular heavy 256-bit and 512-bit instructions only cause a frequency reduction when executed at a rate of more than one instructions per two cycles [150] as described in Section 3.2.1. Even if a single thread does not exceed this rate, two identical threads may exceed it if they are executed in parallel on the same physical CPU core. In this situation, the application does not experience remote AVX overhead as its two threads collectively cause the frequency reduction. However, our profiler only executes a single thread during the second phase of each sample, so it does not observe any frequency reduction. It therefore falsely concludes that there is remote AVX overhead.

4.5.2 Impact of Instrumentation Perturbation

Neither of the two effects listed above is responsible for the comparably high error for the nginx workload. Instead, a closer analysis shows that the profiler result indeed hardly deviates from the actual remote AVX overhead that is present during the experiment. The apparent error is caused by the fact that the web server experiences far less remote AVX overhead when the profiler is active than when it is disabled. In the experiment described in Chapter 3, AVX-512 reduced the throughput of nginx from 6783 to 5992 requests per second, corresponding to 10.6% remote AVX overhead. While our profiler was running, the throughput was instead reduced from 6509 to 5991 requests per second which closely matches the 7.3% remote AVX overhead detected by our profiler.

As described in Section 4.1, this effect that a profiler changes the behavior of the profiled workload is called instrumentation perturbation and is most often caused by overhead caused by the profiler itself. In the case of the nginx workload, the overhead of our profiler was the highest when no AVX2 or AVX-512 was used, whereas a configuration with AVX-512 experienced no substantial slowdown. Our profiler likely causes different amounts of remote AVX overhead because it pauses CPU cores for a fixed amount of time when sampling the frequency reduction. If an application executes at a low frequency at the time of the sample, its performance is less affected by the pause than if the application executes at a high frequency, as in both cases execution resumes at a high CPU frequency after the pause. This effect is only visible in workloads such as nginx affected by frequency change

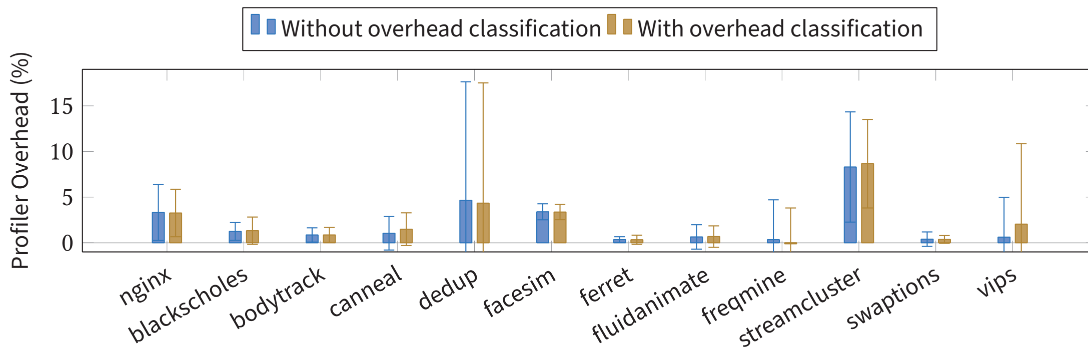


Figure 4.9: For most workloads, our profiler causes very little overhead. The additional sample phase required for the classification of remote AVX overhead does not cause significant additional overhead.

delays and does not affect the remote AVX overhead caused by hyper-threading – if the code executed on the sibling hyper-thread immediately reduces the frequency again after it is unpaused, the pauses do not improve the average CPU frequency during execution of the application.

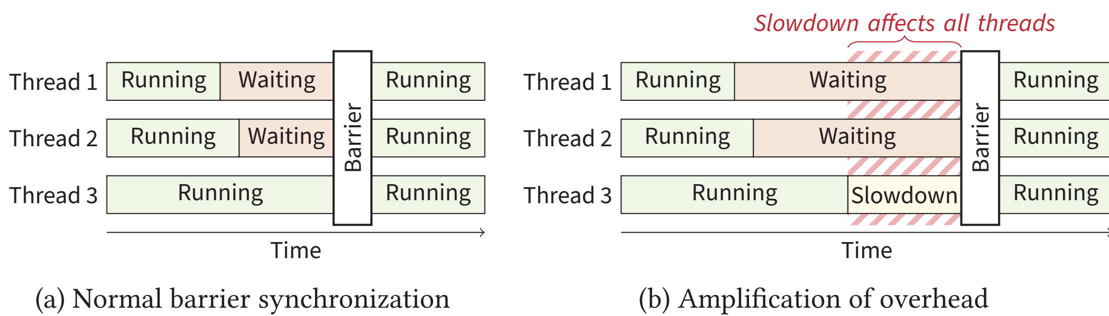
4.5.3 Profiler Overhead

The overhead introduced by the profiler not only impacts its accuracy. Excessive overhead would also make using the profiler with live production workloads difficult, as the overhead can affect the quality of service of the workloads. Our profiler is implemented as a kernel module and does not require any modifications to the kernel itself, which means that the profiler does not cause any overhead while this kernel module is not loaded. While the profiler is active, frequency reduction sampling does cause some overhead, though.

To measure the overall overhead caused by our profiler, we execute individual benchmarks while our profiler is active and compare their completion time to that of runs without the profiler. Figure 4.9 shows the overhead measured using this experiment. We repeat the experiment with and without the optional additional timer interrupt to determine the cause of remote AVX overhead (step 5 in Section 4.2.1) and find that the additional step has negligible impact on performance. In both cases, only few benchmarks show statistically significant overhead, with a maximum overhead of 8.3% for streamcluster. On average, our profiler causes only a 2.1% overhead. This low impact on throughput shows that our profiler is directly usable in most production environments.

Nevertheless, the overhead is far larger than we initially expected. Specifically, we expected overhead to mainly stem from the pauses required to let individual CPU cores recover their non-AVX frequency. As our profiler pauses one out of 16 cores for 700 µs every 20 ms, these pauses directly result in approximately 0.2% overhead.[4] Some additional overhead is caused by timer interrupts and by the fact that the sibling hyper-thread is paused for additional 70 µs, yet even this additional overhead does not explain our substantially larger measurements.

[4] We calculate the expected overhead caused by pauses as $\frac{1}{16} \cdot \frac{700\mu s}{20ms} = 0.22\%$.



(a) Normal barrier synchronization

(b) Amplification of overhead

Figure 4.10: Barrier synchronization is a synchronization technique where threads only continue execution once all involved threads have reached the barrier (a) [65]. When individual threads are slowed down, this synchronization scheme amplifies the negative performance impact as all threads have to wait for the thread that is slowed down (b).

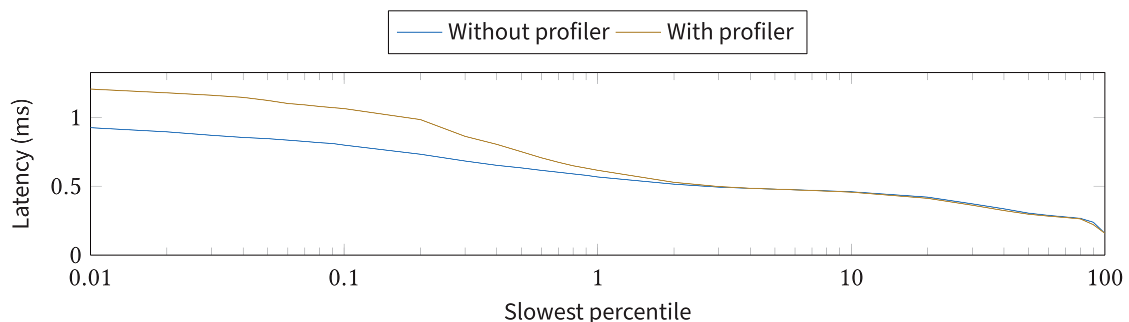


Figure 4.11: The TailBench masstree benchmark [128] shows a substantially increased tail latency when our profiler is active, as the CPU cores are periodically paused.

Instead, the increased overhead for some workloads is likely caused by the chosen synchronization technique. Streamcluster in particular employs *barrier synchronization* [26] which requires all participating threads to wait for the slowest thread to reach the barrier [48]. As Figure 4.10 shows, barrier synchronization amplifies the impact if individual threads are slowed down – if, in our case, one thread is paused by our profiler, all other threads have to wait for 700 µs at the next barrier even if they otherwise require the same amount of unpaused CPU time.

For latency-critical applications, low throughput degradation is not sufficient, though. These applications also require that individual pauses introduced by our profiler do not introduce excessive latency spikes. If we assume a latency-critical web service, for example, any request processed by the service during the pause is delayed until the end of the pause. We use the masstree benchmark from the TailBench 0.9 benchmark suite [128] to measure the impact of our profiler on the latency of web services. We execute the benchmark configured to perform 100000 requests using 32 threads. We plot the resulting latency distribution with and without our profiler in Figure 4.11. The figure shows that our profiler hardly affects the average latency of the key-value store, but increases the tail latency, as a small fraction of the requests is slowed down by pauses.

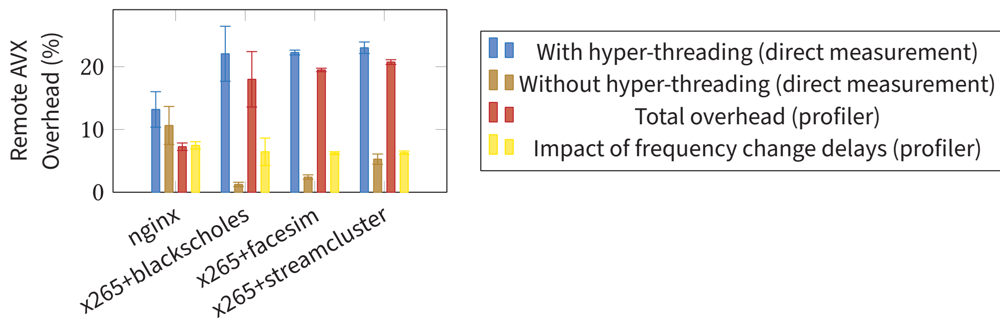


Figure 4.12: When overhead classification is enabled, our scheduler estimates how much of the total remote AVX overhead is caused by frequency change delays. While the estimates often deviate substantially from direct overhead measurements in a configuration without hyper-threading, our scheduler is able to identify nginx as a benchmark that is impacted particularly heavily by frequency change delays.

While the amount of slow responses may seem hardly noticeable, the architecture of large-scale online services often amplifies the problem [62]. If such web services parallelize the processing of individual requests on multiple servers, the response time of the slowest server defines the response time of the whole web service. Therefore, responses from the web server are frequently delayed even if each individual server is only infrequently slowed down, similar in nature to the problem caused by barrier synchronization described above.

The pauses that are responsible for most of the overhead caused by the profiler are a central part of the mechanism used to measure remote AVX overhead and can therefore not be avoided completely. We expect the overhead – or, in the case of latency-sensitive web services, the percentile of affected requests – to scale linearly with the sample rate. Reducing the sample rate likely reduces overhead at the cost of reduced accuracy for short observation periods, though, as previous work has shown that the accuracy of sampling profilers depends on the number of samples [8]. Besides sample rate reduction, another mechanism to reduce overhead is to reduce the length of individual pauses whenever possible. See Section 4.6 for a description of such and other potential optimizations as well as hardware features that would enable profiling with less overhead.

4.5.4 Overhead Source Analysis

Our profiler is not only designed to quantify the overall remote AVX overhead affecting individual applications but is also supposed to identify whether the overhead is caused by hyper-threading or by frequency change delays. To test the accuracy of the latter information, we repeat the experiment described in Section 4.5.1 for a subset of the benchmarks, but enable the additional step during sampling that allows the profiler to differentiate between the two sources of remote AVX overhead. We test nginx as an application that is mainly affected by the frequency change delay, streamcluster as an applications that is affected by both hyper-threading and the frequency change delay,

and blackscholes and facesim as two applications where remote AVX overhead is almost exclusively caused by hyper-threading.

Figure 4.12 shows direct measurements of the remote AVX overhead for the AVX-512 versions of the workloads both with and without hyper-threading. The figure also shows the total overhead determined by our profiler as well as the fraction of overhead that our profiler assumes to be caused by frequency change delays. Our profiler is able to correctly identify that almost all overhead experienced by the nginx benchmark is caused by frequency change delays. The profiler is also able to determine that the remaining benchmarks are mainly slowed down by the frequency impact of hyper-threading. However, it appears that our profiler greatly overestimate the impact of frequency change delays for the Parsec benchmarks.

This discrepancy does not necessarily indicate a problem of our design – instead, the error is largely the result of our sub-optimal experimental setup. This setup misrepresents the results of our profiler because we let it characterize runs with hyper-threading, but compare the results to runs without hyper-threading. Hyper-threading potentially doubles the impact of each individual frequency change delay, though – for example, if a single hyper-thread executes AVX-512 instructions, both hyper-threads are slowed down. Therefore, our experimental setup underestimates the amount of remote AVX overhead that is actually present while the profiler is executed, resulting in an apparent overestimation of the impact of frequency change delays.

While we expect our experimental setup to be responsible for a majority of the discrepancy in our measurements, a limitation of our prototype likely also impacts the accuracy of our profiler. Specifically, our profiler currently does not detect context switches on the sibling hyper-thread during the sampling process. This is problematic when, for example, our profiler pauses a core to measure the remote AVX overhead on one hyper-thread, but the sibling hyper-thread switches from an AVX2 or AVX-512 task to a non-AVX task directly after the pause. In this situation, our profiler may misclassify the overhead as being caused by frequency change delays, even though it was actually caused by software executed on the sibling hyper-thread.

4.6 Discussion

Our evaluation shows that the profiler can measure remote AVX overhead with an average error of 2.4 percentage points for AVX-512 and 2.4 percentage points for AVX2. The error varies substantially between the different applications. Whereas most Parsec benchmarks show very little error, our profiler greatly underestimates the remote AVX overhead of nginx. We argue that even such inaccuracies do not impact the usefulness of the profiler, though. Instead, we expect our profiler to be mainly used to determine whether to apply countermeasures against remote AVX overhead. In this case, users are mainly interested in the order of magnitude of remote AVX overhead, which our profiler is able to determine with high reliability.

Our evaluation identifies one weakness of our design, though. In particular, the pauses during frequency reduction sampling have some potential to change application behavior and to cause substantial overhead. In the following, we discuss potential optimizations to

frequency reduction sampling as well as hardware interfaces that could reduce perturbation and increase accuracy.

4.6.1 Optimized Frequency Reduction Sampling

The pauses introduced by our profiler pose a problem both for latency-critical workloads as well as workloads which use synchronization patterns which amplify the overhead introduced by the pauses. In both cases, the resulting performance impact of our design is mainly determined by the sampling rate – if virtual CPUs are sampled more frequently, pauses occur at a faster rate. In addition, the length of individual pauses defines the overhead per pause or, in the case of latency-critical applications, the maximum latency increase caused by the profiler, so a reduced pause length would positively impact performance.

In our design, the sampling rate is a parameter that can be changed easily. As with any sampling profiler, higher sample counts result in higher accuracy [8], i.e., lower sampling rates require more time to yield the same profiling accuracy, though, which limits the profiler's viability for short-running workloads. Our profiler randomly samples random parts of the workload, and low sample counts may lead to over- or underestimates of remote AVX overhead depending on when the workload is paused. We did not evaluate the impact of the sampling rate on accuracy as the results likely greatly depend on the completion time of the benchmarks in use. Given that there is no sample rate that provides both high accuracy and low overhead for arbitrary workloads, we propose a configurable sampling rate similar to how Linux perf lets the user choose the sampling rate via a command line parameter [191]. A configurable sampling rate allows use of the profiler with both short workloads that require high accuracy as well as long-running workloads that require low profiling overhead. System administrators or software developers employing our profiler likely know the approximate duration of their workload as well as the acceptable impact on performance and are therefore able to select a suitable sampling rate.

In contrast to the sampling rate, the length of individual pauses cannot be arbitrarily modified as it is dictated by the underlying working principle of the profiler. Each pause needs to be long enough so that the CPU regains its maximum non-AVX frequency during the pause, a process which can take up to 670 µs. In our prototype, we therefore simply pause physical cores for 700 µs. One method to reduce the length of the pause would be to monitor the frequency level during the pause and to end the pause as soon as the CPU has reached the non-AVX frequency level. This method is particularly effective if the CPU core spends substantial time executing non-AVX code before most pauses. We decided against implementing this optimization for several reasons.

First, the method is only effective for certain workloads. In particular, the method is ineffective at times when remote AVX overhead is caused by hyper-threading – whenever the sibling hyper-thread was executing AVX2 or AVX-512 code at the time of the sample, the CPU waits for the full 670 µs before changing its frequency. As shown in Chapter 3, hyper-threading effects are often responsible for the majority of remote AVX overhead. Also, the method is ineffective for workloads that make intensive use of AVX2 or AVX-512 instructions as such code requires the full pause duration. For example, we show that streamcluster suffers from increased profiler overhead due to its use of barrier synchro-

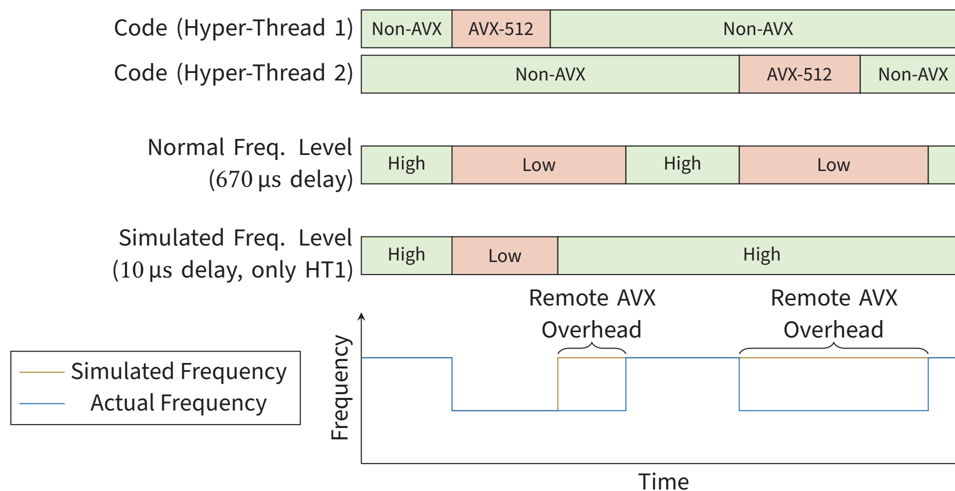


Figure 4.13: To detect and quantify remote frequency reduction, the CPU could simulate a DVFS policy with much shorter frequency change delays for the code running on individual hyper-threads. Remote frequency reduction can then be identified by comparing the frequency selected by this simulated policy with the actual frequency of the CPU core.

nization, yet this benchmark also highly profits from AVX-512 [41]. An AVX-512-enabled version of streamcluster would therefore not benefit from variable pause durations.

Finally, we expect that adaptive pause durations have the potential to introduce systematic error to the profiling results. Adaptive pauses cause AVX2 and AVX-512 tasks to be slowed down more than non-AVX tasks, as pauses during AVX2 or AVX-512 code always last the full 700 µs, whereas pauses during non-AVX code may be abbreviated. Longer completion times of AVX2 and AVX-512 tasks may artificially increase the amount of remote AVX overhead reported by the profiler if they elongate the timespan during which these tasks are executed in parallel with other tasks.

4.6.2 Proposed Hardware Changes

While it may be possible to reduce the length of pauses introduced by the profiler for some workloads, the pauses itself remain a necessity as the hardware does not provide any better mechanism to determine unnecessary frequency reduction. In particular, as described in Section 3.4, the CPU only provides performance events for cycles spent at the individual frequency levels, but does not provide any fine-grained information about whether a frequency reduction is necessary for the software running on individual hyper-threads. Instead, the only information that can be drawn from the performance events is the lowest frequency level of any software running on all sibling hyper-threads combined during the previous 670 µs.

We expect that improved hardware-software interfaces can alleviate the need for intrusive profiling mechanisms such as the one described in this chapter. As future work, we therefore propose the implementation of specialized performance monitoring events to identify remote AVX overhead. Similar to the existing events, these events count cycles at

different frequency levels. However, in contrast to the existing events which represent the actual CPU frequency of a physical CPU core, the proposed events represent the ideal frequency level for the code running on an individual hyper-thread. The difference between these sets of events then can be used to calculate the remote frequency reduction and, consequently, the remote AVX overhead.

Given that the proposed performance events convey similar information as the existing frequency level events, the proposed events can be implemented similarly to the existing DVFS policy. For each individual hyper-thread, we propose implementing a simulation of a slightly modified DVFS policy, essentially replicating the hardware used to implement the existing policy. As shown in Figure 4.13, this simulated policy differs in two ways from the existing DVFS policy: First, the CPU uses a far shorter frequency change delay such as 10 µs to detect the end of AVX2 and AVX-512 code. Second, the simulated policy only takes the instructions executed on a single hyper-thread into account.

As shown in Figure 4.13, the frequency levels determined by this simulated DVFS policy – in essence, a DVFS policy that changes frequencies shortly after the type of instructions changes – are a much closer representation of the frequency level that is actually required at each point in time. Whenever the corresponding events therefore indicate a lower frequency than the one actually chosen by the CPU, the difference represents unnecessary frequency reduction and, consequently, remote AVX overhead.

Such performance events allow the construction of a profiler that does not rely on sampling – and therefore does not need to pause any CPU cores – but rather continuously monitors the difference between required and actual CPU frequencies. This profiler would likely provide both lower overhead as well as increased accuracy. We argue that the required performance events are likely easy to implement, as the CPU already contains logic to detect power-intensive instructions and to determine whether a frequency reduction is required. It is likely not difficult to differentiate between hyper-threads, either, as the CPU already already implements a wide range of hyper-thread-specific performance events [116, pp. 19.3 ff.].

One challenge that has to be solved by future work, though, is that it is somewhat difficult to predict whether the code executing on a single hyper-thread would cause a frequency reduction if the other hyper-thread was idle. Often, a minimum rate of power-intensive instructions is required to trigger a frequency reduction. For example, as described in Section 3.2.1, recent Intel CPUs do not use the AVX-512 frequency level unless they execute more than one heavy 512-bit instructions per two cycles [150]. A program may not exceed this rate when executed alongside another application on the same core, but may exceed it when executed in isolation. In this case, a profiler should not misinterpret low instruction rates and assume that the program is affected by remote AVX overhead. We propose that the CPU simply multiplies the rate of power-intensive instructions by two when predicting necessary frequency reduction in order to prevent overestimates of remote AVX overhead.

As described in Section 4.5.1, a similar problem also affects our profiler which monitors only one hyper-thread in isolation when determining the required frequency level. A multi-threaded application may not exceed the instruction rate necessary for a frequency reduction if only one thread is scheduled on a CPU core. If the application occupies both hyper-threads at the same time, it may cause increased frequency reduction, though. As

our profiler pauses one hyper-thread and only detects the impact of the software running on the other hyper-thread, it may overestimate the amount of remote AVX overhead present. Our evaluation shows that the error caused by this effect does not result in unacceptable accuracy.

5 Viability of Improved DVFS Policies

In Chapter 3, we showed that power-intensive 256-bit and 512-bit SIMD instructions can slow down other code that does not use these instructions [141, 154]. This remote AVX overhead is caused by two mechanisms: First, the dynamic voltage and frequency scaling (DVFS) policy implemented by the CPU delays frequency changes, causing more cores to run at low frequencies than necessary. Second, reduced frequencies affect all hyper-threads of a physical CPU core equally even if only one hyper-thread uses power-intensive SIMD instructions. The impact of frequency change delays in particular suggests that improvements to the DVFS policy could reduce the remote AVX overhead affecting code that does not use AVX2 or AVX-512.

In this chapter we demonstrate that for most workloads the potential of improved DVFS policies is very limited. Specifically, we show that improved DVFS policies are unable to mitigate remote AVX overhead caused by hyper-threading which constitutes the majority of all remote AVX overhead. In Section 5.1, we start by describing the parallels between DVFS policies and existing work on dynamic power management (DPM) and identify techniques used for DPM that may be applicable to DVFS. We also show that in both fields the break-even time defines whether an operation – changing the frequency in the case of DVFS, shutting a device down in the case of DPM – yields a net benefit.

One important design goal common to both DPM and DVFS policies is to provide as little overhead in worst-case scenarios as possible. It was shown that simple timeout-based policies optimize worst-case overhead if the timeout equals the break-even time [163]. We therefore describe an experiment to quantify the break-even time of frequency changes triggered by AVX2 or AVX-512 code in Section 5.2 and show that the policy implemented by Intel provides sub-optimal worst-case performance. In particular, in situations where most cores are active, the break-even time of AVX-512 frequency changes can be as low as 156 µs, much lower than the delay of 670 µs implemented by Intel CPUs.

We simulate improved DVFS policies on a selection of workloads in Section 5.3. We implement both a timeout-based policy based on the previously measured break-even time as well as an ideal oracle algorithm that performs frequency changes without delay whenever changes will be beneficial. On systems with hyper-threading, the improvement achieved by the policies is very limited as frequency change delays play only a minor role. While the original DVFS policy results in 18.2% average remote AVX overhead, the optimized timeout-based policy reduces this value to 15.2%. Even with the oracle policy 14.2% average overhead remains, showing the limited potential of improvements to the DVFS policy. Only when hyper-threading is disabled, both improved policies almost completely mitigate remote AVX overhead.

In Section 5.4, we discuss these results. As our experiments in Chapter 3 show, most remote AVX overhead is commonly caused by hyper-threading resource sharing. We therefore argue that DVFS policy improvements alone cannot be a sufficient solution to

the problem of remote AVX overhead and should be paired with techniques such as core scheduling [154] or our OS-level solution presented in Chapter 6 that cover the impact of hyper-threading instead.

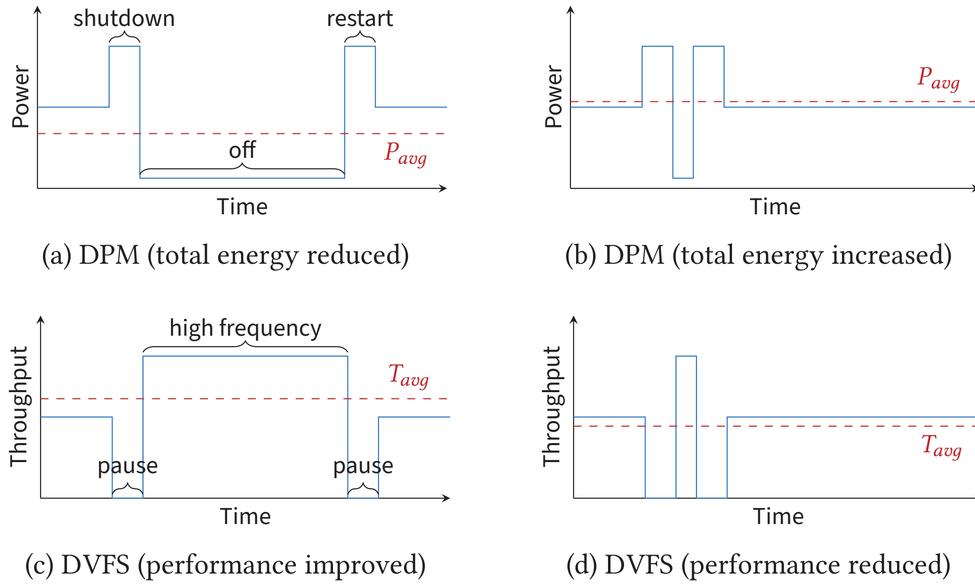
5.1 Parallels to Dynamic Power Management

As described in Section 2.2.3, frequency changes have the desired effect – either energy reduction or performance increase – only if the frequency is not changed again shortly after. Otherwise, the overhead of the frequency changes dominates, both in terms of energy because the CPU temporarily operates at sub-optimal voltages during a frequency change [189] as well as in terms of performance because the CPU needs to be paused while the clock synthesizer changes its frequency. For this thesis, this pause is of particular interest as we mainly focus on the performance impact of AVX2 and AVX-512. Intel Ivy Bridge CPUs, for example, were shown to pause for approximately 10 µs [167] during frequency changes, while we measure similar pauses for Intel Skylake CPUs in Chapter 3. In the case of AVX2 and AVX-512, the throttling period during frequency reduction adds to these costs.

When trying to avoid frequent clock changes, DVFS policies for AVX2 and AVX-512 instructions can only decide whether and when to increase the CPU frequency. Reducing the CPU frequency is mandatory to prevent excessive power consumption and voltage droops as described in Section 3.2.2. As a result, recent Intel CPUs immediately reduce their frequency when executing power-intensive instructions, but delay frequency boosts by 670 µs to limit the worst-case overhead caused by frequency changes. However, any unnecessary delay comes at the cost of performance as less power-intensive code is executed at sub-optimal frequencies.

The resulting trade-off – frequency change overhead versus performance degradation at reduced frequency levels – closely matches the trade-off posed by *dynamic power management* (DPM), where a system can transition individual devices into a low-power state to save energy [22]. Such state transitions commonly cause overhead in terms of energy consumption. For example, while a hard disk drive requires almost no power when the platters are stopped, accelerating them to their operational speed requires substantial energy. The system therefore has to decide when to employ low-power states to save energy during idle periods and when to leave devices active to prevent excessive state transition overhead. As an example, Figure 5.1a shows a situation where disabling a device reduces energy consumption, whereas Figure 5.1b shows a situation where energy consumption is increased. Figures 5.1c and 5.1d demonstrate the close similarity between DPM and AVX frequency management. While Figure 5.1c shows a situation where temporarily increasing the CPU frequency increases performance, Figure 5.1d shows a frequency change that reduces overall performance. As the figures show, the two problems are closely related, which suggests that existing research targeted at reducing the overhead caused by power state transitions can be applied to DVFS to reduce remote AVX overhead caused by frequency change delays.

In the case of DPM, the correct decision regarding a power state transition is defined by the *break-even time* of that transition [163]. If a device remains in the low-power state for



(a) DPM (total energy reduced)

(b) DPM (total energy increased)

(c) DVFS (performance improved)

(d) DVFS (performance reduced)

Figure 5.1: Dynamic power management and DVFS provide similar trade-offs. Placing a device in a low-power mode can result in energy savings (a), but may increase energy due to mode transition overhead if the device is soon reactivated again (b). Similarly, increasing the CPU frequency can improve performance (c), but may fail to do so if the frequency is quickly reduced again, as then the frequency transition overhead dominates (d).

longer than the break-even time, the system experiences a net energy reduction, whereas energy consumption is increased if the low-power state does not last as long. If we assume that $E_{trans} = E_{shutdown} + E_{wakeup}$ and $T_{trans} = T_{shutdown} + T_{wakeup}$ are the energy and the time required for power state transitions and P_{sleep} and P_{wake} are the power during low-power states and during normal operation, respectively, the break-even time T_{be} is calculated as follows:

$$T_{be} = \frac{E_{trans} - P_{sleep}T_{trans}}{P_{wake} - P_{sleep}} \tag{5.1}$$

Again, this formula is very similar to the break-even time observed for a change from a low to a high CPU frequency, where it is only beneficial to increase the frequency if it can remain high for longer than this break-even time. If we assume p_{low} to be the application performance at the low frequency, $W_{nochange} = p_{low}T$ is the work performed by the application during the timespan T if no frequency change occurs. If we further assume p_{high} to be the application performance at the high frequency level and assume $T_{overhead} = T_{down} + T_{up}$ to be the additional CPU time required by frequency reduction (T_{down}) and frequency boost (T_{up}), the application can perform the work $W_{change} = p_{high}(T - T_{overhead})$ when the frequency is increased for the given duration. Solving $W_{nochange} = W_{change}$ for T results in the following formula for the break-even time [86]:

$$T_{be} = \frac{p_{high}T_{overhead}}{p_{high} - p_{low}} \tag{5.2}$$

The main challenge of DPM is that it is usually not known in advance when individual devices will be required again. In those cases, it is therefore not possible to know with certainty whether a power state transition is desirable or not. Instead, power management policies have to predict the length of the next idle period. Existing work presents many different approaches to DPM with different prediction accuracy [163]. A simple, yet widely used policy is to simply delay transitions to low-power states by a fixed timeout, similar in nature to the DVFS policy of recent Intel CPUs. Such a policy limits the rate of state transitions and therefore the maximum overhead but is still able to exploit long idle times to conserve energy. However, the timeout increases energy consumption as devices invariably remain active for longer than necessary.

More energy can be saved by exploiting information about the workload. For example, Srivastava et al. [228] propose predicting future idle times based on a nonlinear regression function calculated from past active and idle times. They also observe that in some workloads long idle times commonly follow long active periods and therefore propose immediate transitions to low-power states for these workloads only if the preceding active period is below a workload-dependent threshold. It is also possible to model systems and workloads as stochastic processes using Markov chains and then, for example, to use linear programming to find an optimal solution that minimizes energy [22].

These policies require the workload to be known in advance and are therefore not viable if the workload changes over time. Dynamic situations instead require DPM to be adaptive. One possible approach is to create multiple policies with different characteristics and to select the policy best suited to the current workload. Such selection can be guided either by a characterization of the workload at runtime using parameter learning [22] or by an evaluation of the different policies based on how well they would have predicted recent idle times [66]. In both cases, the policy that is deemed to be best for the workload – or, alternatively, an interpolation of the best policies – is used to perform future DPM decisions. Finally, it is also possible to train a neural network at runtime using the idle times of the workload to predict future idle times [162].

The different approaches to DPM differ substantially in their energy consumption [163]. Stochastic approaches, in particular, were shown to provide good energy efficiency. In practice, however, simple timeout-based approaches are prevalent. For example, the ATA/ATAPI command set provides a standby timer which places disk drives into a standby mode when it expires [249, p. 34]. Similarly, the X Window System provides a configurable timeout which specifies the idle time after which displays are disabled [149] while Linux commonly places wireless network connections into a low-power state after a fixed period without any network traffic [160]. Simple timeout-based policies combine reduced development costs with an energy efficiency that rivals that of more complex techniques if the timeout is well-tuned to the system [22]. In many cases, an even more important reason to select a timeout-based policy is that it results in predictable worst-case behavior, though, both in terms of energy consumption as well as performance. Such bounds on overhead are important given that both dynamic power management and DVFS policies can potentially generate arbitrarily large overhead. If we assume, for example, a dynamic power management scheme that places devices in a low-power state as soon as they become idle, arbitrarily short idle periods lead to frequent state transitions and thereby to unbounded overhead. Frequency changes can have similar unbounded overhead, which

demonstrates the need for frequency change delays such as that implemented by recent Intel CPUs. While one of Intel's patents describes a "hysteresis timer" used to limit the rate at which functional units are power-gated [29], another describes a similar hysteresis mechanism for frequency changes associated with wide SIMD instructions such as AVX2 or AVX-512 [201].

Commonly, the worst-case overhead is determined via comparison to an ideal off-line policy – commonly called *oracle* – which has perfect knowledge of future system behavior. At the beginning of each idle period, this policy can therefore determine whether to immediately place devices in low-power states or whether to keep them active until they are used again with perfect accuracy. The ratio between the energy required by a dynamic power management policy and the energy required by the oracle policy for a worst-case scenario is called the *competitiveness* of the former policy [127]. If we assume a simple timeout policy, for example, this worst-case scenario is when the idle time equals the timeout and therefore the device is reactivated directly after it was placed in a low-power state. Ideal competitiveness is achieved when the timeout is configured to equal the break-even time, as then the energy consumed by state transitions equals the energy consumed during the idle period in the absence of any state transition. Such a policy is 2-competitive – it consumes at most twice as much energy as the oracle policy.

5.2 Characterization of AVX Frequency Changes

The frequency change delay implemented by Intel CPUs is not 2-competitive. Instead, the CPU commonly spends far more time at lower frequency levels than necessary to minimize worst-case performance impact. This behavior contributes to the remote AVX overhead demonstrated by our analysis in Chapter 3 for workloads that often switch between AVX and non-AVX code.

In the following, we demonstrate that the DVFS policy implemented by Intel CPUs is not 2-competitive by showing that the break-even time of frequency changes severely differs from the frequency change delay actually used by the CPU. To determine the break-even time, we first measure the overhead of all possible frequency level transitions separately. These measurements cover both increasing and reducing the frequency. We then use Equation 5.2 and the performance differences between high and low frequency levels to calculate the resulting break-even times. Our experiments to measure frequency change overhead closely mirror those performed as part of our previous work [86], with only minor changes to the experiment setup.[1]

5.2.1 Frequency Reduction Overhead

To measure the overhead when the frequency is changed from a higher frequency level (non-AVX or AVX2 frequencies) to a lower frequency level (AVX2 or AVX-512 frequencies),

[1] During the experiments to measure the overhead during frequency reduction, we execute a piece of AVX2 or AVX-512 code twice in quick succession. Our previous experiments left more time between the two executions, thereby increasing the chance that turbo level changes affect the results [86]. Also, our previous experiments did not differentiate between light AVX-512 and heavy AVX2 instructions.

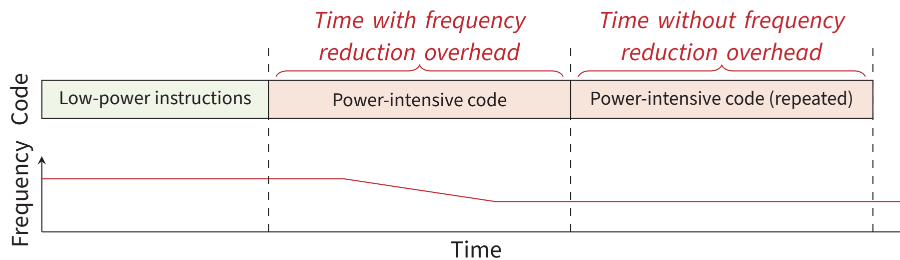


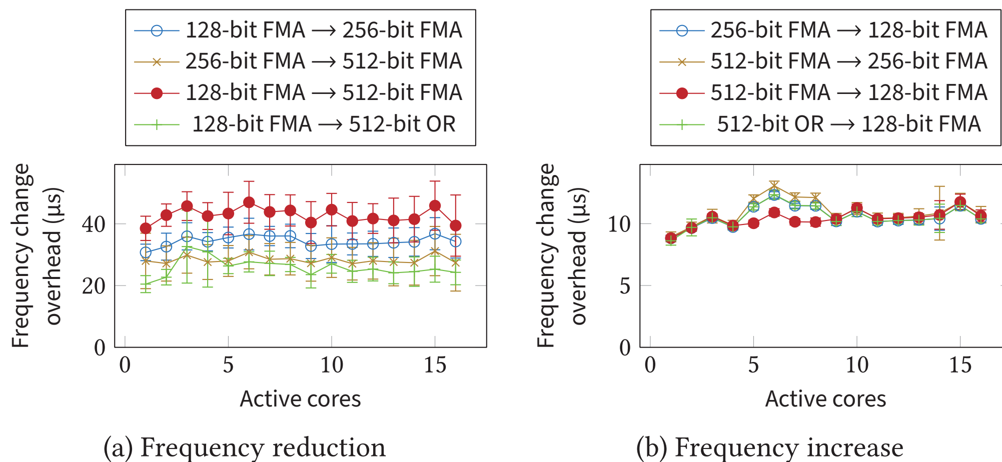
Figure 5.2: To measure the frequency reduction overhead, we execute the same code consisting of more power-intensive instructions twice. We measure the time required for both executions. The difference between the two times is the frequency reduction overhead incurred during the first execution of the power-intensive code.

we perform the experiment shown in Figure 5.2. We first execute a large amount of instructions that can be executed at the former frequency level. We then measure the time required to execute a fixed number of instructions requiring the latter frequency level. In both cases, the number of instructions is chosen so that the code takes long enough that the system has arrived at the lower frequency level by the time the code has finished. We then repeat executing the latter code, with the only difference being that by now the system is already at the appropriate frequency level for the given instructions. We again measure the time required for execution and then calculate the overhead of the frequency level change as the difference between the two times. As both the frequency difference and, consequently, the expected overhead vary based on the number of active cores, we repeat this experiment while executing between one and 15 threads that keep additional physical cores busy. We not only measure the frequency reduction overhead caused by FMA instructions of different sizes but also include 512-bit OR instructions in our experiment as our previous experiments in Section 3.2 showed that light AVX-512 instructions cause different types of frequency level transitions compared to heavy AVX-512 instructions. For each configuration, we repeat the experiment 100000 times to determine the variance of the results.

Great care must be taken to reduce noise caused by external factors. In particular, we discard all results that indicate a CPU time difference of more than 100 µs as doing so has shown to be a simple yet effective method to filter out outliers caused by our program being preempted. We also use the Linux CPU hotplugging functionality to deactivate all additional cores as the turbo frequency would change if additional cores were intermittently activated, for example, to process any interrupts.[2] We configure Linux to process interrupts on the first logical CPU whenever possible, while performing our runtime measurements on another core or, in the case of single-core configurations, on the second hyper-thread of the that core.

Figure 5.3a shows the overhead of frequency reduction as measured using this experiment. The results closely match the findings of our analysis of frequency level transitions

[2] At the time of writing the Linux kernel contains a bug that affects the CPU frequencies of cores after they have been temporarily disabled [176]. We therefore reboot the system before performing experiments for a different number of active cores.



(a) Frequency reduction

(b) Frequency increase

Figure 5.3: Both reducing and increasing the frequency is associated with overhead. The former causes more overhead than the latter as it involves a throttling period with reduced instruction throughput.

in Chapter 3.2.2 which shows that the overhead is mainly defined by two factors. First, the number of frequency changes during a frequency level transition varies, with each frequency change causing a stall of approximately 10 µs. Second, the number of throttling periods to prevent voltage droops varies – while all transitions involve a throttling period that causes a fourfold throughput reduction, some involve a second throttling periods with twice the throughput.

As expected from these observations, transitions from 128-bit to 512-bit FMA instructions incur the highest average cost of 42.7 µs as they involve two separate frequency changes as well as two throttling periods. Transitions which involve only one frequency change cause less overhead. For example, transitions from 128-bit to 256-bit FMA instructions cost 34.4 µs on average while transitions from 256-bit to 512-bit FMA instructions cost 28.4 µs on average. Transitions from 128-bit FMA to 512-bit OR instructions cause the lowest average overhead of only 25.9 µs as they only involve a single throttling period. Note that the overhead hardly varies based on the number of active cores. This result stands in contrast to our expectation of slightly lower overhead for larger numbers of active cores. We expected that, as the CPU operates at a lower frequency level during the throttling period when many cores are active, throttling should cause slightly less overhead. Any such effect is likely hidden by the variance of our results.

5.2.2 Frequency Boost Overhead

Transitions from low to high frequency levels do not require any throttling and therefore cause less overhead. However, despite frequency boosts being conceptually simpler, measuring the associated overhead is far more complicated than in the case of transitions from high to low frequency levels. In the previous section, we executed the same code twice, while only one of the two runs included a frequency level transition. The CPU time difference of the two runs represented the overhead of the transition. Applying the same principle to frequency boosts is impractical, though, because frequency increases

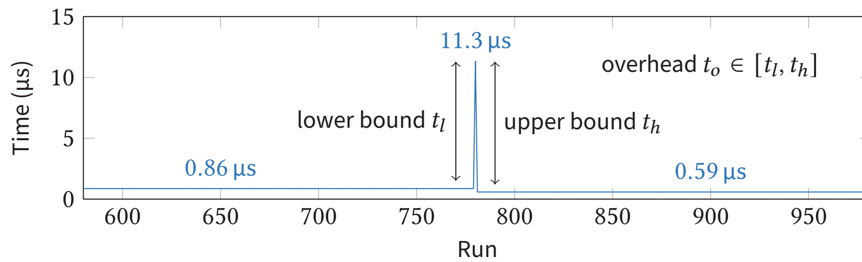


Figure 5.4: We measure the overhead of frequency boosts by repeatedly measuring the time required for a very short section of code. Before and after the frequency change, the time correlates to the frequency. During the frequency change the time is dominated by the pause while the CPU's PLL is reconfigured.

are delayed by 670 µs. In our previous experiment, we tried to minimize the CPU time of our test program to minimize CPU time variation caused by external factors, yet the delay stipulates much longer CPU times. In addition, the delay makes comparisons between the CPU times of the two runs much more complicated. Runs without any frequency level transition are only performed at the higher frequency level, while runs with a frequency level transition are partially affected by the lower frequency level. Calculating the overhead therefore requires precise knowledge of the point in time at which the frequency level transition starts. Due to the difficulty of determining this time with sub-microsecond accuracy, we deem the approach impractical for obtaining precise estimates of the frequency change overhead.

Instead, we exploit the fact that increasing the frequency is substantially less complex than reducing it. Whereas frequency reduction involves a throttling period as described above, the only source for overhead when increasing the frequency is the pause while the CPU core's PLL is reconfigured for the new clock speed. We therefore use a technique described by Mazouz et al. [167] who determine the length of this pause by repeatedly measuring the time required for a short section of `ADD` instructions. We modify this technique to measure AVX frequency level transitions by initially forcing the system to operate at the AVX2 or AVX-512 frequency level and then repeatedly measuring the time required for a fixed number of 128-bit or 256-bit FMA instructions. This code snippet requires less than 1 µs to execute; we repeat it for a total duration of 1.5 ms.[3] As the instructions allow operation at higher frequencies, the system will eventually trigger a frequency change to the non-AVX or AVX2 frequency level.

As shown in Figure 5.4, the time required to execute the code snippet corresponds to the frequency, with a higher frequency level resulting in lower CPU times. The frequency change itself poses an exception, as the resulting pause substantially increases the CPU time required. We therefore select the highest runtime measured during the experiment and then calculate the average runtime before and after this sample. The runtime before the sample is generally higher than that afterwards, so the difference between the highest

[3] While, as shown in Section 3.2.3, frequency boosts commonly occur after a delay of 670 µs, we observed longer delays during our experiments involving 512-bit OR instructions, similar to observations by Yussuf Khalil [131]. We are unaware of the cause for these varying delays.

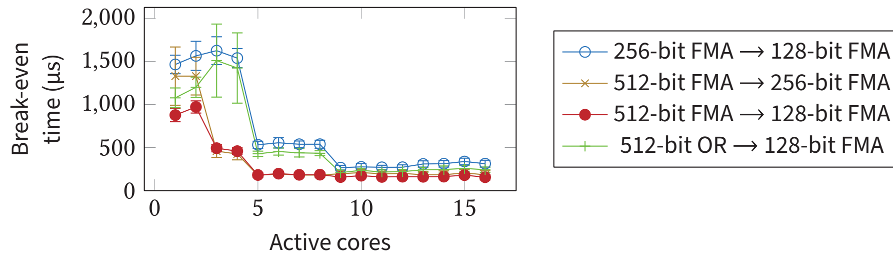


Figure 5.5: The break-even times of AVX2 and AVX-512 frequency level transitions depend on the number of active cores. While the frequency change overhead is fairly constant across different core counts, the performance difference increases as more cores become active.

runtime and the previous runtime forms a lower bound for the frequency change overhead whereas the difference between the highest runtime and the following runtime forms an upper bound. For our experiment, we simply calculate the frequency change overhead as the mean of the two values, assuming that the most of the resulting statistical error is cancelled out when we repeat the experiment 100000 times. As above, we repeat this experiment for various numbers of active cores while we deactivate inactive cores and let the first logical CPU process most interrupts.

Figure 5.3b shows the frequency level transition overhead determined by this experiment. Similar to above, we observe little variation of the overhead depending on the number of active CPU cores. In contrast to frequency reduction, the cost of frequency boosts does not depend on the frequency levels involved, either, with all types of frequency level transitions requiring approximately 10 µs. This result is expected given that the overhead is mainly caused by a single fixed-duration pause during the frequency change itself.

5.2.3 Break-Even Time of Frequency Changes

Computing the break-even time requires not only the overhead caused by frequency changes but also information on the performance degradation at low frequencies (p_{low} and p_{high} in Equation 5.2). To simplify the calculation, we assume that CPU performance is perfectly proportional to CPU frequency which allows substituting the performance variables p_{low} and p_{high} with the frequencies f_{low} and f_{high} at the lower and higher frequency level. In Section 4.3, we showed that this assumption is invalid especially for memory-intensive applications which experience stall durations independent from the CPU frequency. We also described techniques to achieve a more accurate estimate of the performance impact of frequency changes. However, in this section we do not need this level of accuracy. We merely want to demonstrate the sub-optimal behavior of the fixed-timeout policy of existing Intel CPUs. Even though our analysis consequently only covers workloads with negligible memory stalls, we show that the delay implemented by these CPUs usually deviates so drastically from the break-even time that our results likely also apply to most other types of workloads affected by remote AVX overhead.

We insert the frequencies for different numbers of active cores as well as the measured overhead into Equation 5.2 to calculate the break-even time. Figure 5.5 shows the results of

this calculation. Given that the frequency change overhead is fairly constant for different core counts, the break-even time mainly varies based on the frequency swing caused by AVX2 and AVX-512. When only one core is active, these instructions cause very little frequency reduction – on the Intel Xeon Gold 6130 CPU, the CPU frequency is only reduced by 200 MHz by AVX-512 [119] as the single active core can use all the available thermal budget. Therefore, the resulting break-even times are high, ranging from 876 µs for transitions between AVX-512 and non-AVX frequencies to 1464 µs for transitions between AVX2 and non-AVX frequencies. These values are reduced to 156 µs and 313 µs, respectively, when all cores are active, as the higher frequency reduction – 900 MHz in the case of the Xeon Gold 6130 CPU and AVX-512 code [119] – means that frequency increases more quickly compensate for frequency change costs.

The low break-even times for large numbers of active cores demonstrate the problem of Intel's DVFS policy. The 670 µs delay implemented by Intel CPUs provides good worst-case competitiveness for single-threaded workloads using heavy 512-bit instructions as the delay roughly matches the corresponding break-even time, yet such scenarios are hardly representative for server workloads. Instead, most server workloads are multi-threaded. When more than half of the CPU cores are active, the break-even time generally is much lower than the frequency change delay, resulting in far lower competitiveness and increased overhead. We demonstrated the resulting impact in Chapter 3 where we showed that some workloads are slowed down by more than 10%.

Note that recent Intel CPUs provide only a single constant delay for different numbers of active cores, even though our experiments show that the break-even time varies. To provide optimal worst-case behavior, the CPUs should select different delays for different numbers of active cores. In the following, we concentrate on measuring the potential of improved delays for scenarios where all cores are active and leave implementing such adaptive behavior as future work. We argue that it should be simple to extend the DVFS policy of existing CPUs as the policy already takes the number of active cores into account when selecting a suitable turbo frequency level.

5.3 Simulating Improved Frequency Scaling

Our previous analysis showed that the existing DVFS policy implemented by existing Intel CPUs is sub-optimal. However, the analysis does not quantify the improvement expected from improved DVFS policies. In the following, we therefore compare three different DVFS policies to demonstrate the remaining potential for performance improvements. First, as a baseline, we determine the overhead caused by a policy that delays all frequency boosts by 670 µs, similar to that implemented by Intel CPUs. Second, we determine the overhead caused by a policy that instead uses a delay of only 150 µs. This delay roughly matches the break-even time of transitions between the AVX-512 and non-AVX frequency levels and should therefore be approximately 2-competitive. Finally, we compare these policies to an oracle policy that always performs frequency changes without any delay if the time that can be spent at the higher frequency level is long enough that a frequency change is beneficial. For the latter two policies, no hardware implementation is available. We decide against implementing all three policies in a full system simulator such as gem5 [27]. As

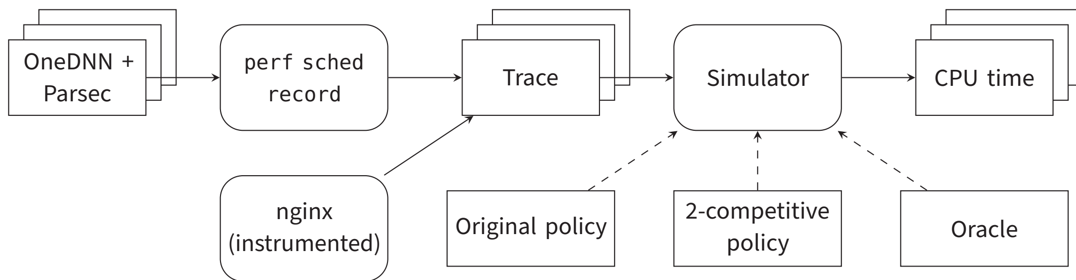


Figure 5.6: To evaluate DVFS policies, we use a simulator which scales CPU time according to the CPU frequency selected by the policies. The simulator operates on traces that provide information on AVX-512 usage.

the CPU microarchitecture implemented by such simulators generally does not match that of the Intel CPUs covered by this thesis, it would have been unclear whether the timing produced by these simulators would be representative of existing CPUs. Instead, we implement a simple simulator that operates on traces generated from workloads executed on a Skylake-X CPU to capture realistic system behavior.

Figure 5.6 shows the design of this setup. The input to the simulator consists of a trace of CPU activity which contains information about which application is running on each logical CPU and whether the applications use AVX-512. The simulator itself simulates the impact of the DVFS policies on performance by scaling the length of trace entries according to the CPU frequency selected by the policies as shown in Figure 5.7. In addition, the simulator inserts a delay whenever the CPU frequency is changed to simulate frequency change overhead. Whenever the trace indicates that the CPU executes AVX-512 code, the simulator forces a transition to the low AVX-512 frequency level. Whenever the trace indicates that no AVX-512 instructions are used, the simulated DVFS policy is applied to increase the CPU frequency.

As the simulator divides the duration of individual trace entries by the simulated CPU frequency, the trace itself needs to be recorded at a constant CPU frequency. Recording the trace on a system that uses Intel's DVFS policy would negatively affect the accuracy of the simulation results as the DVFS policy would make AVX-512 phases appear to require disproportionally more time. In all other aspects, the trace should be as representative of server CPUs as possible. In particular, it should be recorded at a frequency similar to the non-AVX frequency of server CPUs as otherwise reduced memory stalls would affect the recorded times.

To simulate a CPU which constantly operates at the non-AVX frequency of the Xeon Gold 6130 server CPU used for most of our previous experiments, we executed the workloads on a system with an Intel Core i9-7940X CPU. As described in Section 3.2.2, this CPU in combination with a mainboard targeted at overclocking[4] allows deactivating any AVX frequency reduction. In addition, the CPU is able to execute AVX-512 code at higher frequencies than the non-AVX frequencies of most server CPUs. We configure the system

[4] We generate our traces on a system with a Intel Core i9-7940X CPU, the Asus TUF X299 Mark 2 mainboard, and 32 GiB 2666 MHz DDR4 RAM.

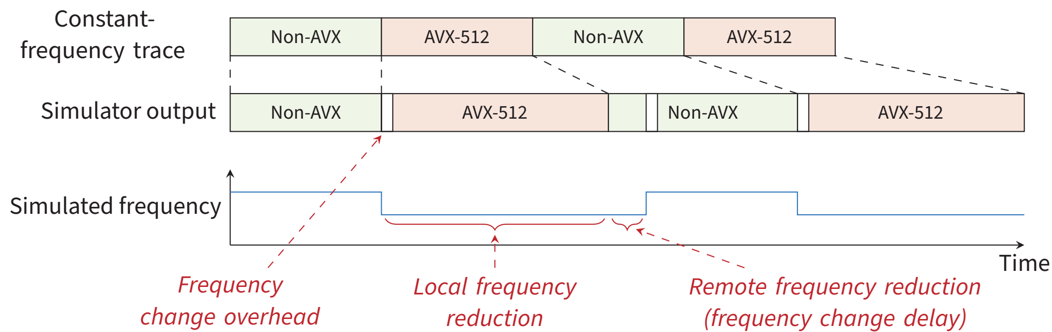


Figure 5.7: To simulate different DVFS policies, our simulator scales the duration of individual trace entries by the simulated frequency. Whenever the simulated frequency changes, frequency change overhead is added to the simulator output. The example shows a timeout-based DVFS policy.

so that all parts of the workload to operate at the same constant CPU frequency of 2.8 GHz, thereby matching the non-AVX frequency of the Intel Xeon 6130.

While we are able to prevent frequency changes, we are, unfortunately, unable to remove other artifacts of the CPU's power management policy such as overhead due to throttling. As a result, some parts of the trace appear slightly slower than they would be in the absence of autonomous CPU power management. We do not expect this behavior to have any substantial impact on our simulation results as the issue affects all simulated DVFS policies equally. We mainly compare the different policies, while absolute remote AVX overhead results are of less importance. In the following, we also show that the time unnecessarily spent at lower frequencies by Intel's original DVFS policy has a much larger impact on performance than any such frequency change overhead.

We generate traces of two types of workloads, similar to our experiments performed in Section 3.3.5 to quantify remote AVX overhead. First, we generate traces of workloads where various Parsec benchmarks are executed in parallel to OneDNN benchdnn. We use `perf sched record` [192] while these workloads are running to generate scheduler traces. All traces are recorded while OneDNN benchdnn is configured to use AVX-512. In addition, we generate a trace using an instrumented version of the nginx benchmark described in Chapter 3. We configure nginx to serve a compressed static file via HTTPS and instrument OpenSSL to write timestamps to a file whenever the library starts or finishes encryption of data blocks. These timestamps represent use of AVX-512 and form the input to our simulator. All workloads are restricted to four physical cores to reduce the size of the resulting traces.

Similar to the experiments in Chapter 3, we use the CPU time of the benchmark – either nginx or a Parsec benchmark – as the target metric to determine the performance achieved by the DVFS policy. More specifically, to calculate the slowdown caused by frequency changes, we compare this CPU time to the CPU time required if the benchmark is executed at the non-AVX frequency and no frequency changes occur. Figure 5.8 shows the slowdown determined by our simulation. We make three main observations:

First, in the case of the original DVFS policy (labelled "Stock" in the figure), the absolute overhead differs from that determined during our experiments described in Chapter 3.

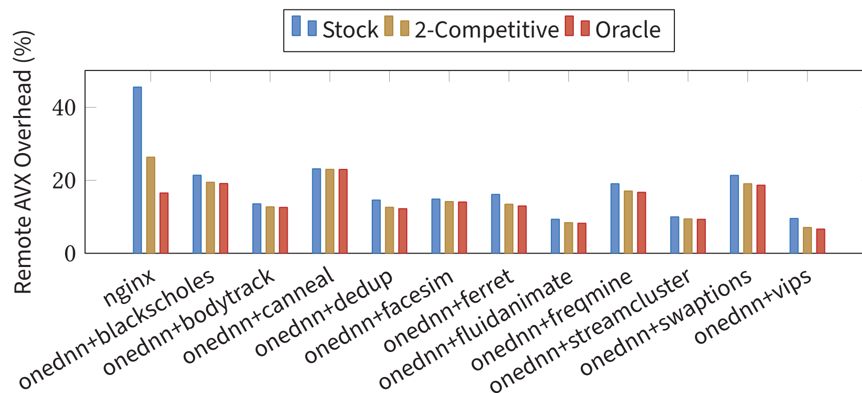


Figure 5.8: The 2-competitive DVFS policy substantially reduces remote AVX overhead for the Parsec benchmarks executed alongside OneDNN `benchdnn --rnn`, almost to the level achieved by the ideal oracle policy. However, the overhead caused by hyper-threading generally dominates, so the impact of the DVFS policies is limited. In contrast, nginx is substantially affected by excessive frequency change delays, so the improved policies are able to reduce the remote AVX overhead.

The reason for this discrepancy mostly likely lies in the different number of threads which influences the length of the time slices allocated by the scheduler. As we described in Section 3.3.5, the time slice length has a substantial impact on remote AVX overhead. For example, in our simulations, Parsec ferret shows a much higher context switch rate than during the experiments described in Chapter 3. Therefore, the simulator measures more remote AVX overhead. As described above, such discrepancies are not relevant for our analysis as we are mainly interested in a comparison between the different DVFS policies.

Second, 2-competitive frequency scaling with a lower frequency change delay consistently outperforms the original frequency scaling policy provided by the CPU. The impact is mostly very small, though, with the exception of the nginx single-application benchmark that was already shown to especially suffer from the frequency change delay in Section 3.3.5. Whereas the original DVFS policy results in 18.2% overhead on average, 2-competitive frequency scaling achieves an average overhead of 15.2%.

Third, the oracle policy consistently provides even better performance at an average overhead of 14.2%. This is expected given that the oracle policy has knowledge of future system behavior and can minimize overhead by increasing the CPU frequency without delay. Again, the impact is very small, though, which demonstrates that the performance potential of improved frequency scaling policies is very limited for two-application workloads. This result is consistent with our results presented in Chapter 3 which showed that most remote AVX overhead in these workloads is caused by hyper-threading, with little overhead directly attributed to frequency change delays.

As our experiments simulated a system with hyper-threading, we perform an additional experiment to show the impact of improved DVFS policies in systems without hyper-threading. While such CPUs are rare in current server systems, they may become more

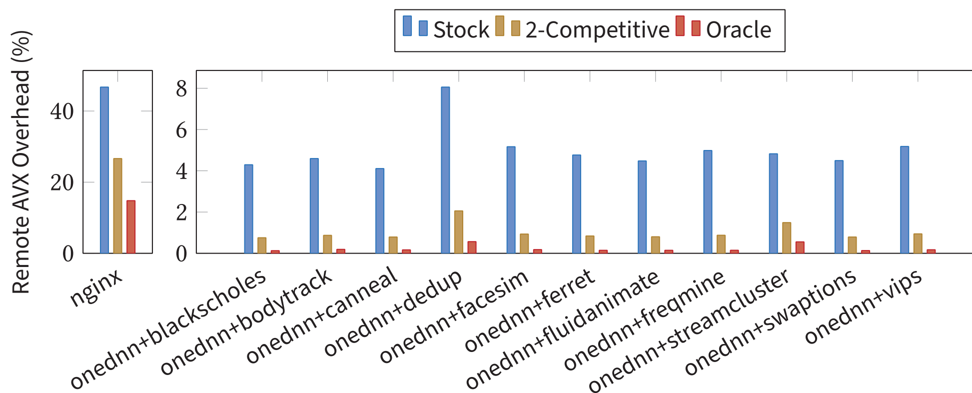


Figure 5.9: On a system without hyper-threading, improved DVFS policies have a greater impact on remote AVX overhead. In this setup, all remote AVX overhead is caused by the suboptimal frequency change delay implemented by the original policy.

common in future many-core or heterogeneous CPUs.[5] We repeat our simulations on traces generated using only one hyper-thread per physical core. Figure 5.9 shows the results of this experiment. In this configuration, both the 2-competitive DVFS policy and the oracle policy are able to mitigate almost all remote AVX overhead. A detailed analysis shows that this is because most of the original policy's remote AVX overhead is caused by excessively long frequency change delays whereas the frequency change overhead itself is hardly noticeable. On average, the original policy causes 8.5% overhead, of which 0.1 percentage points can be attributed to frequency change overhead, while the remainder is caused by the execution of non-AVX code at the AVX-512 frequency level. In the given configuration, only nginx is affected by far more frequency change overhead as it switches very frequently between non-AVX and AVX-512 code. Consequently, it is the only benchmark where substantial overhead remains even with the improved DVFS policies.

5.4 Discussion

Our experiments uncover a deficiency in Intel's DVFS policy and show a straightforward solution to improve performance. In many scenarios, the frequency change delay is too long and should be reduced to match the break-even time. Perhaps the CPU should even take the number of active CPU cores into account when selecting a suitable frequency change delay. While our analysis shows that such changes have a positive effect on most workloads suffering from remote AVX overhead, the impact is low for workloads where hyper-threading causes most remote AVX overhead. Nevertheless, we expect that these improvements are worth pursuing given that the effort required to optimize frequency change delays is likely low. As the remaining remote AVX overhead is often substantial,

[5] At the time of writing, Intel integrates CPU cores without hyper-threading in its heterogeneous desktop CPUs [205] and has announced server CPUs with such cores [4].

improved DVFS policies should be paired with software techniques such as that described in Chapter 6 to further reduce remote AVX overhead.

Our analysis has two main limitations which we discuss in the following:

Impact of Memory Stalls. Our calculation of break-even times does not take memory stalls into account. Applications that spend most of their time waiting for memory have longer break-even times as these applications experience less performance improvement when the CPU frequency is increased. Future work has to determine whether timeout-based DVFS policies should choose the frequency change delay according according to model-based predictions of the performance at higher CPU frequencies [229]. Given how large the difference between Intel's delay and the break-even time is in the absence of memory stalls, we argue that such complex policies are likely unnecessary. Instead, we expect that an implementation with a single static optimized frequency change delay is already able to reap most of the potential performance benefits. Note that most benchmarks used in our simulation were shown to provide throughput in proportion to the CPU frequency [86], so the lack of a DVFS performance model does not have much impact on our simulation results.

Impact of Voltage Changes. When we calculate the break-even time, we only measure the direct impact of frequency changes and throttling to determine the frequency change overhead. We do not measure the impact of voltage changes on energy consumption, even though voltage changes were shown to impact energy efficiency [189]. During any frequency change, the CPU temporarily operates at a sub-optimal voltage level and therefore consumes more energy. In a power-limited system, reduced energy efficiency directly translates into reduced performance, so voltage changes contribute to frequency change overhead. The resulting performance impact is likely negligible as both our experiments as well as those performed by Travis Downs [68] show the voltage to change by only a few millivolts. In addition, even in the worst case, this sub-optimal voltage setting is only used for few dozens of microseconds per frequency change [167].

Voltage changes also impact the latency of frequency boosts as the voltage needs to be increased before the frequency can be changed [167]. This impact on frequency change latency does not pose any issue for the frequency changes caused by AVX2 and AVX-512, though, as the frequency change delay means that frequency increase is scheduled far in advance. If, for example, the voltage change requires 20 µs and the CPU uses a frequency change delay of 100 µs, the CPU can simply start changing the voltage 80 µs after the start of the frequency change delay.

6 Separating AVX-512 and Non-AVX-512 Code

Remote AVX overhead has substantial negative impact on overall system performance if power-intensive AVX2 and AVX-512 code is executed in temporal or spatial proximity to other less power-intensive code. As shown in the previous chapter, improved frequency scaling policies implemented by the CPU have very little potential to mitigate this performance impact. In this chapter, we therefore present a software approach which is able to mitigate most remote AVX overhead without restricting where and how software developers can use AVX-512 instructions to improve performance.

Our design restricts co-scheduling of AVX-512 and non-AVX-512 code. In contrast to the frequency scaling policies described in the previous chapter, this software approach is able to target both sources of remote AVX overhead and therefore presents a solution for a wider range of applications. Improved frequency scaling policies are able to reduce the impact of frequency change delays, but are unable to mitigate remote AVX overhead caused by hyper-threading. While a technique to mitigate the latter type of remote AVX overhead exists in the form of core scheduling [154], it in turn does not prevent remote AVX overhead caused by frequency change delays.

Our approach is based on *core specialization*, i.e., it designates individual CPU cores for specific types of code. In particular, it specifies a number of *AVX-512 cores* and restricts execution of AVX-512 code to these cores [85]. We select the number of these cores dynamically based on the number of threads executing AVX-512 code so that the cores rarely execute non-AVX-512 code. The resulting separation of different types of code results in two main effects: First, AVX-512 and non-AVX-512 code are rarely co-scheduled on sibling hyper-threads at the same time, so core specialization prevents most remote AVX overhead caused by hyper-threading. Second, as non-AVX-512 code is mainly executed on other cores, it is less likely to directly follow AVX-512 code on the same core, which reduces the remote AVX overhead caused by frequency change delays.

The chapter is structured as follows: First, we revisit existing approaches to mitigate remote AVX overhead (Section 6.1). We describe basic requirements and make a case why mitigation techniques should be implemented in the OS to benefit from the available information about runtime conditions. We then show how core specialization can be used to mitigate remote AVX overhead (Section 6.2). We describe a suitable scheduling policy that differentiates between AVX-512 and non-AVX-512 tasks and we present methods to detect power-intensive AVX-512 code. While core specialization can be implemented in different ways, we present a concrete implementation in the Linux kernel (Section 6.3). Our prototype replicates the scheduler's run queues to differentiate between different types of tasks and uses task stealing to move tasks to appropriate cores. As the resulting migration of tasks between cores can negatively impact performance, we also provide a

mechanism to gauge the overall impact of core specialization on performance (Section 6.4). We evaluate our prototype with a range of workloads and show that it can substantially reduce the performance impact of remote AVX overhead (Section 6.5). Whereas AVX-512 slows the workloads down by 12.3% on average with an existing scheduler, core specialization reduces the average remote AVX overhead to 1.2%. We also show that in most cases the additional complexity introduced by core specialization causes very little overhead itself. We conclude the chapter with a discussion of the limitations of our approach and sketch potential improvements (Section 6.6). In particular, as the concepts presented in this chapter are not limited to AVX-512 but instead will likely also apply to many power-intensive instruction sets introduced by future CPUs, we describe extensions to the CPU which would make core specialization more effective and efficient.

6.1 Existing Mitigation Techniques

Existing work has tried a variety of techniques to solve problems associated with rare use of power-intensive instructions. Few of these techniques specifically target the performance impact of AVX2 or AVX-512 instructions. Due to different reasons, all existing work falls short of providing a practical, generic mitigation technique for remote AVX overhead. In this section, we describe the approaches as well as their drawbacks and derive a set of requirements for our approach.

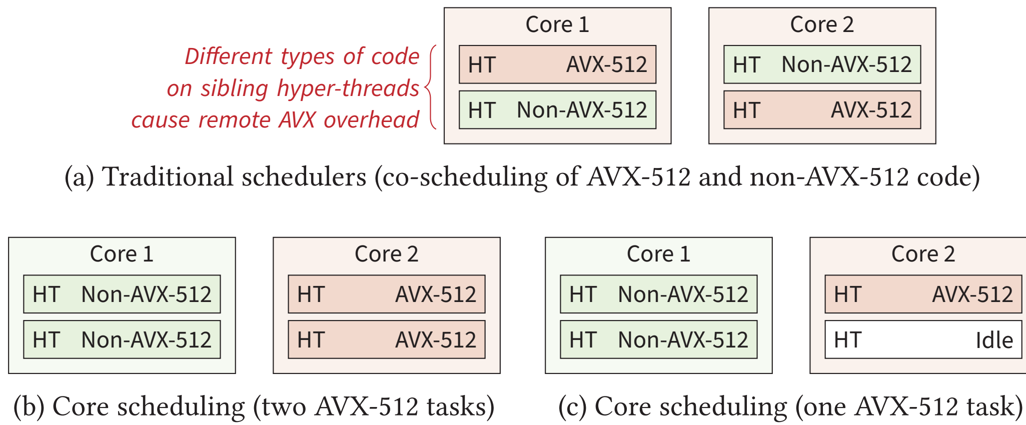
As described in Section 3.3.3, remote AVX overhead was first reported for workloads involving the nginx web server and the OpenSSL library [141]. Shortly after the report, the OpenSSL developers chose to resolve the problem by completely disabling AVX-512 for all Skylake-SP and Skylake-X processors [195], likely due to a lack of practical alternatives. Preventing remote AVX overhead by removing any power-intensive instructions from a program has two major drawbacks, though. First, the technique is rather labor-intensive. For each program that uses AVX-512 or other power-intensive instructions, the overall performance impact in a range of expected workloads has to be assessed by the developers who have to compare the performance gain brought by the instructions with the amount of remote AVX overhead. We argue that any mitigation technique against remote AVX overhead should be automatic. It should not require any manual modification of individual programs. Second, the technique is unable to adapt to individual workloads. In the case of OpenSSL, some workloads – in particular workloads which spend much of their time encrypting or decrypting data – are able to benefit from the substantial local speedup caused by AVX2 and AVX-512 [82, 81], whereas others, as described in Chapter 3, are not. An application that uses OpenSSL with AVX-512 is optimized for the former type of workloads, whereas an application where AVX-512 has been disabled is optimized for the latter type of workloads, but neither application performs equally well in both scenarios. Inability to adapt to the situation is particularly problematic in the case of software libraries such as OpenSSL which are expected to be used in a wide variety of applications and usage scenarios. We argue that any mitigation technique against remote AVX overhead should not prevent the use of power-intensive instructions in scenarios where their use is beneficial.

As it is commonly only known at runtime whether or not power-intensive instructions are beneficial, this requirement in fact rules out any technique that selects a specific implementation when or before launching an application. The amount of remote AVX overhead as well as the speedup brought by power-intensive instructions often depends on external input. For example, the type of web server requests is not known in advance, but the complexity of the requests may decide whether AVX-512-enabled cryptography routines are beneficial or not. Moreover, such external input can change over time, thereby changing the nature of the workload. To adapt to runtime conditions, mitigation techniques that select different implementations to maximize performance would have to include a profiler to determine the impact of power-intensive instructions. While we are not aware of any such technique in the context of remote AVX overhead, *selective devectorization* [142] uses a similar approach for power-gating. It employs runtime profiling to determine whether individual parts of the program are executed in sufficiently SIMD-heavy program phases to warrant the use of power-intensive SIMD units. If this profiler detects code that is used in phases when there is little other SIMD code, a just-in-time compiler devectorizes the code so that the SIMD unit can be disabled.

Any such profiling in user space has another substantial drawback. Often, remote AVX overhead is the product of interaction between different applications – for example, one application running on one hyper-thread slows the application on the other hyper-thread down. If a power-intensive application consists almost completely of power-intensive AVX2 or AVX-512 code, a user-space profiler analyzing this application likely observes a large speedup associated with these instructions and does not disable them, even if other applications, unbeknownst to the profiler, are slowed down. We argue that due to this distributed nature of remote AVX overhead any mitigation technique should either be implemented in the OS or, if implemented in other parts of the software stack, should be provided with information by the OS on whether power-intensive instructions provide a system-wide benefit. Such a setup is required as only the OS can potentially have sufficient information over all applications and their operating frequencies and instruction usage.

To the best of our knowledge, the only existing OS-level mitigation technique to date has been described by Aubrey Li who showed that *core scheduling* can be used to mitigate remote AVX overhead caused by hyper-threading [154]. Core scheduling as provided by the Linux kernel is a mechanism which restricts the co-scheduling of different tasks on sibling hyper-threads and which was originally proposed to mitigate hardware side-channel vulnerabilities [53]. It is implemented by assigning a *core_sched cookie* to each task. The scheduler then only allows co-scheduling of tasks with identical cookie values [56]. Aubrey Li showed that setting cookies based on AVX-512 usage can be used to restrict co-scheduling of AVX-512 and non-AVX-512 tasks as shown in Figure 6.1.[1] His prototype uses the simple – yet, as we show in Section 4.1, defective – mechanism provided by the Linux kernel to identify AVX-512 applications based on register contents during context switches. Any application that has recently used 512-bit vector registers is assumed to

[1] The approach by Aubrey Li [154] is similar in nature to *voltage smoothing* by Reddi et al. [202] where co-scheduling on different physical CPU cores is restricted to reduce voltage droops. Whereas Aubrey Li's prototype co-schedules power-intensive AVX-512 tasks, voltage smoothing prevents co-scheduling of power-intensive tasks to limit peak power consumption.



(a) Traditional schedulers (co-scheduling of AVX-512 and non-AVX-512 code)

(b) Core scheduling (two AVX-512 tasks)

(c) Core scheduling (one AVX-512 task)

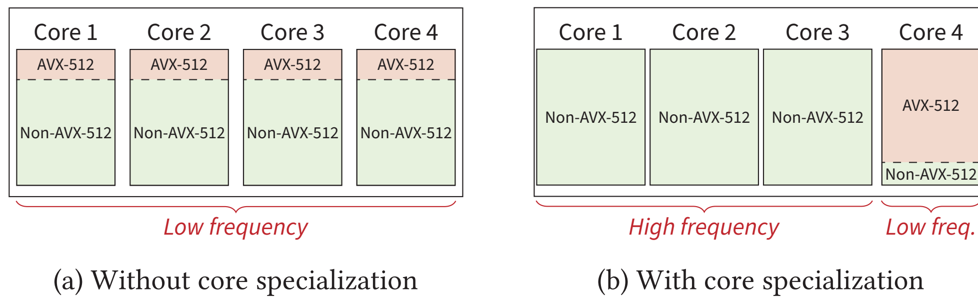
Figure 6.1: Core scheduling can be used to mitigate remote AVX overhead caused by hyper-threading. Whereas most schedulers arbitrarily co-schedule AVX-512 and non-AVX-512 tasks (a), core scheduling only allows execution of tasks of the same type on sibling hyper-threads (b). The technique has the drawback that sometimes hyper-threads have to remain idle (c).

potentially cause a frequency reduction. Whenever such a task is scheduled, its sibling hyper-thread is only allowed to execute tasks that are also known to use AVX-512.

Aubrey Li's approach has two main limitations that we aim to improve upon in our work: First, core scheduling is able to mitigate remote AVX overhead caused by the effects of hyper-threading, but is not able to prevent slowdown in programs which frequently change between AVX-512 and non-AVX code such as the web server example described above. We improve upon core scheduling by identifying AVX-512 tasks with a finer temporal granularity and via modifications to scheduling to prevent remote AVX overhead caused by frequency change delays. Second, as shown in Figure 6.1c, core scheduling leaves individual hyper-threads idle if there is no suitable AVX-512 task ready to be scheduled [154]. If, for example, there was only one runnable AVX-512 task, but a sufficient number of runnable non-AVX tasks to fully utilize all logical CPUs, core scheduling would not co-schedule any of these tasks on the same physical CPU core as the AVX-512 task. As a result, its sibling hyper-threads would remain unoccupied. Leaving individual hyper-threads idle commonly reduces performance. While there are scenarios where it is beneficial to schedule only a single task on a physical CPU core, hyper-threading commonly provides substantial performance improvements as it improves resource utilization [210, 39, 237, 232]. We therefore argue that any technique against remote AVX overhead should be work-conserving and should utilize all available logical CPUs. Our approach explicitly allows co-scheduling of AVX-512 tasks and non-AVX-512 tasks when necessary to achieve work-conserving scheduling.

6.2 Core Specialization for AVX-512 Applications

Our mitigation technique against remote AVX overhead is based on *core specialization*. Core specialization is a software technique where the OS designates individual cores



(a) Without core specialization (b) With core specialization

Figure 6.2: Existing operating systems colocate AVX-512 and non-AVX-512 applications (a), which may result in a frequency reduction for all cores. Core specialization limits use of AVX-512 to a subset of the physical CPU cores instead (b). All other cores are not affected by frequency reduction, so the code running on these cores is not affected by remote AVX overhead.

to perform different types of work. In the past, core specialization has been shown to be beneficial for asymmetric multi-core systems where some tasks are better executed on few fast cores whereas others are better suited for execution on a larger number of small low-power cores [209] or where cores provide different functionality [94, 6]. Core specialization has also been employed to reduce cache misses on symmetric multi-core systems. For example, the system can be configured to let cores execute different parts of the application [146] or to execute system calls on separate cores [225].

As remote AVX overhead is mainly caused by either spatial or temporal proximity between power-intensive and less power-intensive code, we use core specialization to separate the two types of code. Our approach is inspired by suggestions from Tiwari et al. [236] and Daniel Lemire [151] who propose reducing the number of cores used for AVX2 and AVX-512 code without describing a concrete design. We let the OS designate some cores as *AVX-512 cores* which are allowed to execute any type of code, whereas all other cores are designated as *non-AVX-512 cores* and are not allowed to execute any AVX-512 instructions [85]. Figure 6.2 shows the impact of this design on CPU frequencies. If, as shown in Figure 6.2a, all cores are able to execute both AVX-512 and non-AVX-512 code, non-AVX-512 code is slowed down by frequency reduction on all cores, resulting in remote AVX overhead. If, as shown in Figure 6.2b, most cores are instead limited to non-AVX-512 code, these cores will never transition to the AVX-512 frequency level and will never suffer from the resulting remote AVX overhead. Only the designated AVX-512 cores will reduce their frequencies, but these execute comparatively less non-AVX-512 code that could be affected by remote AVX overhead. As more code is executed at its ideal frequency, the CPU cores are able to better utilize their power budget. Finally, less CPU time and energy is lost to transitions between different frequency levels as the type of code executed by individual cores does not change as often. Whereas non-AVX-512 cores never switch to the AVX-512 frequency level, AVX-512 cores rarely switch to other frequency levels as they mostly execute AVX-512 code.

Our implementation of core specialization for AVX-512 workloads consists of a number of components which are shown in Figure 6.3 and which we describe in the following sections. The central component of our design is a modified scheduler ① which differ-

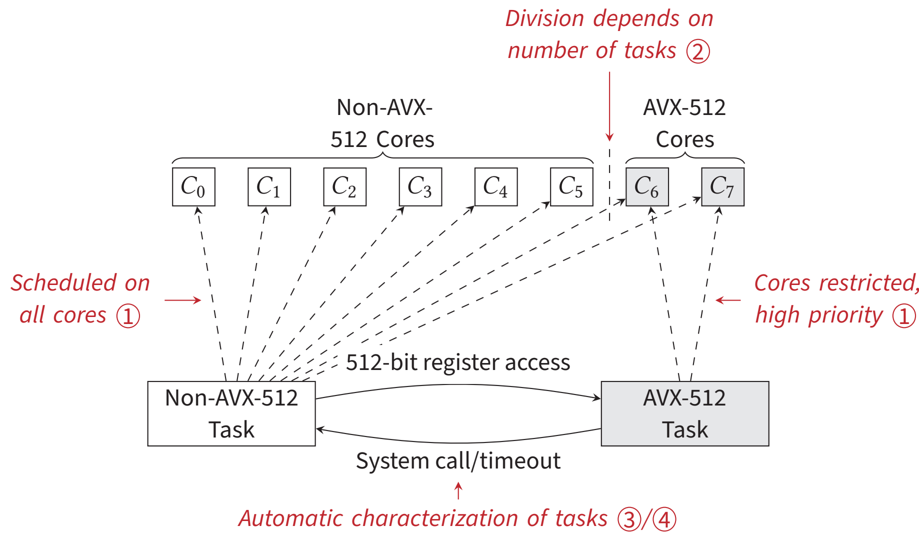


Figure 6.3: Our approach to core specialization for AVX-512 workloads consists of several building blocks; a scheduler which restricts 512-bit SIMD operations to specific cores (①), a policy to determine a good number of cores for AVX-512 code (②), and mechanisms to determine whether individual tasks use AVX-512 (③/④).

entiates between *AVX-512 tasks* and *non-AVX-512* tasks and which prevents execution of AVX-512 tasks on non-AVX-512 cores. We present our modifications to the scheduling policy in Section 6.2.1. The scheduler also has to determine the number of AVX-512 and non-AVX-512 cores based on the number of runnable AVX-512 and non-AVX-512 tasks ② as we describe in Section 6.2.2. Both techniques require knowledge about whether individual tasks execute AVX-512 code or not. In Section 6.2.3, we therefore describe a technique to detect the beginning of execution phases involving AVX-512 code based on register accesses ③ while Section 6.2.4 contains a description of heuristics to identify the end of AVX-512 phases ④.

6.2.1 Scheduling Policy

As we describe above, there are two main requirements for the scheduling policy. First, the scheduler should minimize remote AVX overhead, i.e., it should minimize co-scheduling of AVX-512 tasks and non-AVX-512 tasks. Second, as described in Section 6.1, the scheduler should be work-conserving, i.e., it should not let individual hyper-threads idle if there are any runnable tasks. These two requirements are often conflicting. For example, if there is only one AVX-512 task, a work-conserving scheduler has to execute a non-AVX-512 task on its sibling hyper-thread, yet such task placement causes remote AVX overhead. Whereas core scheduling [154] potentially leaves hyper-threads idle during execution of AVX-512 tasks, our approach implements a different compromise between the two requirements. In particular, our scheduler never tries to leave hyper-threads idle even if the result is increased remote AVX overhead, because hyper-threading commonly improves resource utilization and therefore performance. Our scheduler operates according to the following principles [85]:

- The scheduler never allows non-AVX-512 cores to execute AVX-512 tasks as even short sections of AVX-512 code potentially cause remote AVX overhead for the following 670 µs.

- In contrast, the scheduler allows AVX-512 cores to execute any type of code. While, as described above, the resulting co-scheduling increases remote AVX overhead, it also improves resource utilization.

- On AVX-512 cores, the scheduler prioritizes AVX-512 tasks and only executes non-AVX-512 tasks if no additional AVX-512 task is runnable.[2] Minimizing non-AVX-512 code running on AVX-512 cores minimizes co-scheduling of the two types of tasks and therefore also minimizes remote AVX overhead.

The resulting policy frequently requires tasks to be migrated to different cores. In particular, if a task becomes an AVX-512 task, it needs to be migrated to an AVX-512 core to continue running. Similarly, if a task becomes a non-AVX-512 task, it should be quickly migrated to a non-AVX-512 core as it is otherwise preempted by AVX-512 tasks with higher priority and may be starved of CPU time. Modern systems with support for large numbers of logical CPUs commonly maintain separate queues of runnable tasks – for example, one such *run queue* per each individual logical CPU – to reduce lock contention [60]. On such systems, tasks that are not suited for execution would remain in the run queue of their previous logical CPU, requiring explicit migration to a different queue.

Commonly, migration is done by a load-balancing mechanism that moves tasks from longer queues to shorter queues. Our scheduler features modified load balancing that only migrates tasks if the destination CPU is able to execute the tasks, i.e., AVX-512 tasks are never migrated to non-AVX-512 CPUs. If load balancing is responsible for moving tasks to suitable CPU cores, the load balancing mechanism needs to react quickly to task type changes and changing CPU load. Otherwise, tasks may be slowed down due to placement on an unsuitable CPU core, while logical CPUs may become idle due to a lack of suitable tasks. We describe the impact of this requirement on the implementation of our prototype in Section 6.3.1.

Our scheduler does not try to minimize the number of active CPU cores. Each task migration has the potential to activate a CPU core that was previously idle, which in turn could reduce the CPU's turbo frequency level. As described in Section 2.2.3, in this thesis we focus on fully utilized systems which operate at the lowest turbo level. Future work should analyze the behavior of our approach on lightly loaded systems to determine whether it should be combined with techniques such as those proposed by Lawall et al. [145] to reduce the number of active CPU cores.

[2] Such a prioritization scheme has previously been proposed by Shen et al. for heterogeneous multiprocessors, where the scheduler selects the task with the highest instruction set requirements runnable on the core [218]. Our approach is conceptually similar except that we essentially create heterogeneity in software and are able to dynamically vary the number of AVX-512 cores.

6.2.2 Number of AVX-512 Cores

Good utilization of all CPU cores and effective mitigation of remote AVX overhead does not only depend on good placement of tasks but also on the number of AVX-512 cores. In particular, whenever there is an excessive number of AVX-512 cores, they frequently have to execute non-AVX-512 tasks to keep utilization high, resulting in increased co-scheduling of AVX-512 and non-AVX-512 tasks and therefore increased remote AVX overhead. In contrast, when there is an excessive number of non-AVX-512 cores the AVX-512 cores will experience high load while the non-AVX-512 cores will often idle as there are no suitable non-AVX-512 tasks.

Our design determines the number of AVX-512 cores based on the number of AVX-512 and non-AVX-512 tasks.[3] We define the *AVX-512 load* as the number of AVX-512 tasks divided by the number of AVX-512 cores:

$$load_{AVX} = \frac{n_{tasks,AVX}}{n_{cores,AVX}} \tag{6.1}$$

The *non-AVX-512 load* is similarly defined as the number of non-AVX-512 tasks divided by the number of non-AVX-512 cores:

$$load_{non-AVX} = \frac{n_{tasks,non-AVX}}{n_{cores,non-AVX}} \tag{6.2}$$

Large differences between the two values indicate sub-optimal scheduling. If, for example, the AVX-512 load is larger than the non-AVX-512 load, it is more likely that the non-AVX-512 cores are underutilized, while a larger non-AVX-512 load indicates a higher chance that AVX-512 cores execute non-AVX-512 tasks. In addition, different loads indicate that either AVX-512 or non-AVX-512 tasks are given more CPU time, whereas identical loads indicate fair scheduling.

The aim of our scheduler is therefore to select a number of AVX-512 cores which results in approximately equal loads.[4] Whenever a task becomes an AVX-512 task, the scheduler compares $load_{AVX}$ and $load_{non-AVX}$. Whenever the former is higher than the latter or whenever there are idle non-AVX-512 cores, the current core should be marked as an additional AVX-512 core. The same check also occurs during each scheduler invocation on non-AVX-512 cores. The scheduler also compares the load values during scheduler invocations on AVX-512 cores if the scheduler would have to choose a non-AVX-512 task. At this point, if $load_{AVX}$ would still be lower than $load_{non-AVX}$ after an AVX-512 core was removed, the current core should be marked as a non-AVX-512 core.

The scheduler only applies these changes to the core type if the previous change to any core is more than 20 ms in the past to limit the rate of changes. Higher rates would impact the effectiveness of our approach – whenever an AVX-512 core becomes a non-AVX-512

[3] In addition to the load factor, schedulers for latency-critical applications have taken the CPU utilization into account [198]. Such schedulers often limit CPU utilization to prevent latency spikes due to varying load. Our scheduler tries to maximize CPU utilization and therefore ignores CPU utilization when determining the number of AVX-512 cores.

[4] The algorithm described in this section leads to a slightly lower load on AVX-512 cores. This is intentional as AVX-512 cores can execute non-AVX-512 tasks if necessary, whereas lower load on non-AVX-512 cores may lead to idle cores and underutilization of the CPU.

core, it is potentially affected by remote AVX overhead during the next 670 µs as it has recently executed AVX-512 code. In addition, the scheduler will always ensure that there is at least one AVX-512 core and one non-AVX-512 core in the system, as otherwise one of the corresponding types of tasks could be starved.

6.2.3 Detecting AVX-512 Code

Our scheduler can only restrict AVX-512 code to AVX-512 cores if the tasks containing the code are marked accordingly. While we experimented with manual annotations in the applications denoting AVX-512 code regions in previous work [83, 187], such manual annotation is labor-intensive. Therefore, in this thesis we propose automatic detection of 512-bit SIMD operations based on register accesses [85].

As described in Section 4.1, Linux already contains a technique to identify tasks that use AVX-512 by sampling the register state during context switches: If there is valid 512-bit register content, the kernel records the timestamp [235]. This approach exploits the behavior of the `XSAVE` instruction used in recent Intel CPUs to save the floating-point register state during context switches [114, pp. 13–1 ff.]. Due to the large size of the floating-point register file – the 32 512-bit registers introduced by AVX-512 alone require 2 KiB of memory when saved – the `XSAVE` instruction determines which parts of the register file are in active use and only saves those parts. If `XSAVE` saves 512-bit register state during a context switch, these registers have been filled with valid state by the application, so the application has recently used AVX-512.

Sampling the register state during context switches is not a suitable technique for our design as short sections of AVX-512 code, despite their potential to reduce CPU frequencies, often go unnoticed. To reduce power and context switch overhead Intel suggests executing the `VZEROUPPER` instructions after each section of AVX2 or AVX-512 code [113, p. 17-57]. This instruction clears all SIMD register content except for the lowest 128 bits of the first eight SIMD register, which causes the 512-bit register state to show up as unused during the next context switch. Furthermore, the approach operates only on whole tasks and can not distinguish between different parts of a program consisting of both AVX-512 and non-AVX-512 code such as a web server using AVX-512-enabled cryptography.

While it would alternatively be possible to mark all pages containing valid 512-bit SIMD instructions and then use page faults to identify AVX-512 code similar to the approach pursued by region scheduling [146], the coarse granularity imposed by page sizes risks that non-AVX-512 code is marked as AVX-512 code. Any such mischaracterization of code potentially causes excessive thread migrations between the two types of cores. As migrating a thread is associated with overhead, these false positives during detection of AVX-512 code should be minimized.

Instead of these techniques, we take inspiration from work by Li et al. [157] who disabled the floating-point unit of some of their CPU cores to simulate an asymmetric multiprocessor system on existing symmetrical off-the-shelf CPUs. Whenever a thread tries to execute a floating-point instruction on a core where support is disabled, the core triggers an undefined instruction exception. Li et al. show that this exception can be used as part of a *fault-and-migrate* mechanism to migrate the thread to a core with support for the instruction. We use a modified version of this approach not only to prevent

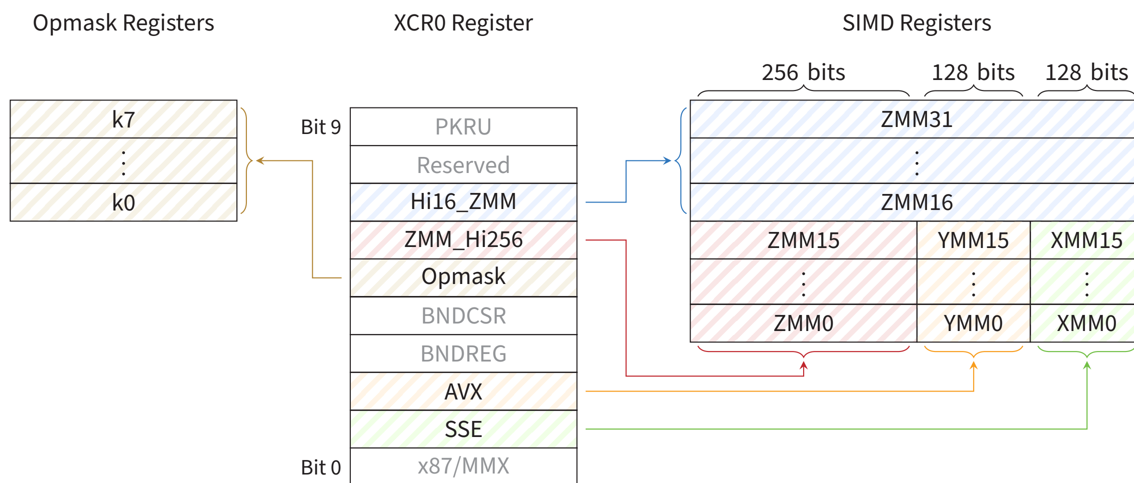


Figure 6.4: The `XCR0` register defines which parts of the register set can be saved using the `XSAVE` instruction [116, p. 2-20]. If an instruction accesses a register while the corresponding bit in `XCR0` is 0, an exception is raised. We clear the `ZMM_Hi256`, and `Hi16_ZMM` registers to detect execution of 512-bit SIMD instructions.

execution of 512-bit SIMD instructions on non-AVX-512 cores but also to mark any thread executing such an instruction as an *AVX-512 task* [85]. The main difference between our implementation and that of Li et al. is that we do not completely disable the floating-point unit but rather selectively disable access to 512-bit registers.

We exploit the fact that individual CPU cores prevent execution of SIMD instructions if the operating system does not implement context switching for the required registers [114, ch. 13]. In particular, the aforementioned `XSAVE` instruction stores register state into a memory region specified by the OS and therefore relies on the OS to allocate sufficient memory. To inform the CPU core about the set of registers that can be safely written, the OS uses a bitmap in the `XCR0` register to instruct the CPU core about the registers that shall be saved. As shown in Figure 6.4, the `XCR0` register contains a bit representing the upper 256 bit of the first 16 `ZMM` registers – these were extended by AVX-512 from 256 bit to 512 bit – as well as a bit representing an additional set of 16 512-bit `ZMM` registers introduced by AVX-512. Whenever an instruction modifies registers that are not saved and restored during context switches, the data is potentially lost, so the CPU does not allow 512-bit register accesses if either of these two bits is not set[5]. Instead, the CPU raises an undefined instruction exception. We therefore clear both bits on all non-AVX-512 cores. Whenever an undefined instruction exception is triggered by these cores, the OS then marks the current task as an AVX-512 task and, as described above, the load balancing mechanism migrates the task to a suitable AVX-512 core.

This mechanism correctly identifies all instructions that could potentially cause transitions to the AVX-512 frequency level and prevents execution of such instructions on non-AVX-512 cores, thereby completely preventing any frequency reduction to the AVX-512 frequency level. However, the mechanism causes false positives in two situations

[5] In addition, AVX-512 introduced a set of mask registers. As AVX-512 instructions of all sizes require these registers, the corresponding bit in `XCR0` cannot be used to detect particularly power-intensive instructions.

where tasks are flagged as AVX-512 tasks even though they can be executed at a higher frequency level:

1. Other undefined instruction exceptions such as those triggered by invalid opcodes also cause the task to be marked as an AVX-512 task. To minimize overhead in the common case – execution of AVX-512 instructions – our design intentionally does not differentiate between different causes for exceptions. Instead, the operating system simply resumes the task on an AVX-512 core. There, any invalid opcode will simply cause a second exception which will then be handled properly by the OS. Performance-wise, this behavior is not problematic as illegal instructions are rare and commonly cause the application to be terminated.

2. Only 512-bit multiplications and floating-point operations trigger a transition to the lowest frequency level [113, p. 2-13]. Ideally only tasks executing these instructions would be flagged as AVX-512 tasks, yet our approach intercepts all 512-bit register accesses and therefore also flags code that only triggers transitions to the intermediate AVX2 frequency level. This behavior exposes an inherent limitation of the hardware-software interface provided by current Intel CPUs, as the coarse-grained control provided by `XCR0` does not allow differentiation between different types of operations. While this limitation reduces the effectiveness of our approach, we expect false positives to be less problematic than false negatives, as a single false negative can slow a non-AVX-512 core down for 670 µs, causing substantial remote AVX overhead in the process. Note that the limitation essentially prevents the approach to be used to identify 256-bit SIMD instructions, as the bits in `XCR0` that prevent access to 256-bit registers cause widely used instructions such as light 256-bit SIMD operations or instructions such as `VZEROUPPER` to trigger exceptions [115, pp. 2-32, 5-550]. Neither of these instructions cause any frequency reduction. See Section 6.6.4 for a discussion of this limitation as well as of an improved hardware-software interface.

Our manipulation of `XCR0` on non-AVX-512 cores not only triggers undefined instruction exception but also general protection faults. In particular, the `XRSTOR` instruction used to restore the register state causes such faults if the program tries to restore the state of registers that were disabled. The faults pose a problem as use of `XRSTOR` is not restricted to kernel space. For example, the dynamic linker uses `XSAVE` and `XRSTOR` during lazy binding, where references to symbols from shared libraries are resolved once the corresponding functions are first called [18]. The dynamic linker implements lazy binding by initially configuring all references to shared libraries to point to its own symbol-binding function instead of the referenced library symbols. When this function is first called for any specific reference, it looks up the target address and rewrites the reference accordingly before calling the referenced function directly for the first time. During the process, this symbol-binding function must ensure that none of the arguments provided by the application are lost, i.e., it needs to save and later restore the corresponding registers. In glibc's dynamic linker, the symbol-binding function `_dl_runtime_resolve` uses `XSAVE` to save any floating-point arguments to the function to be resolved[6], which causes general protection

[6] glibc 2.37, `sysdeps/x86_64/dl-trampoline.h`, line 121

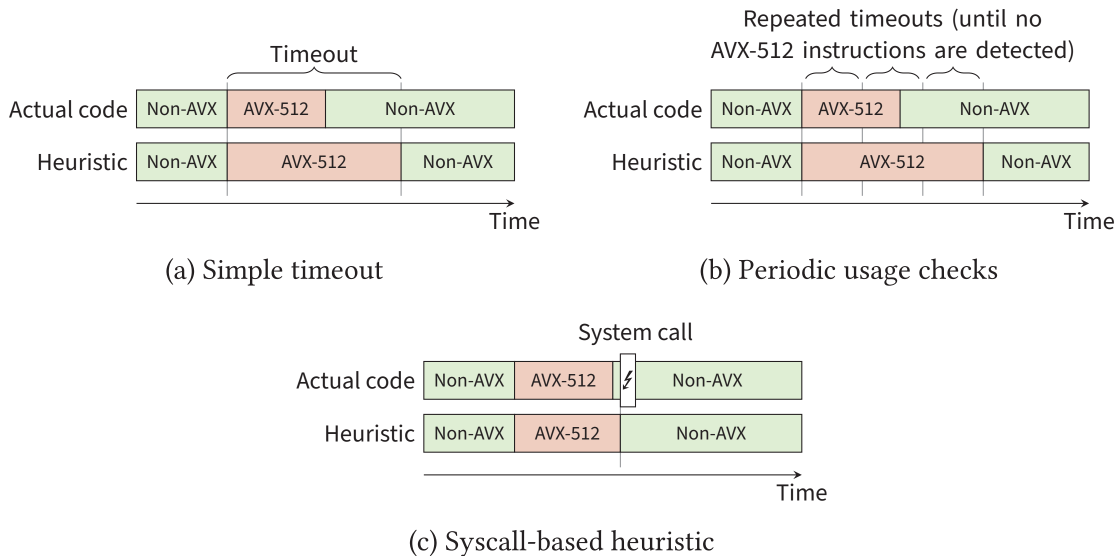
faults when `XCR0` is modified to prevent execution of 512-bit SIMD operations. We handle this fault by simply temporarily allowing 512-bit register access on the specified core, before reinstating our changes to `XCR0` at the time of the next context switch – naturally, if there is valid 512-bit register content at this time, the current task needs to be marked as an AVX-512 task. While this change temporarily allows undetected execution of power-intensive 512-bit SIMD operations on non-AVX-512 cores, we expect lazy binding to be a rare event in most workloads that is mostly restricted to application startup.

After such a situation or when the number of AVX-512 cores is reduced, the OS changes `XCR0` on the affected cores to prevent further execution of 512-bit SIMD operations. At this point, the OS has to ensure that there is no valid 512-bit register state left on the core. Such register state would become inaccessible once `XCR0` has been modified, yet the corresponding parts of the register file cannot be deactivated, causing even narrow SIMD operations to affect the CPU frequency. The change to `XCR0` therefore has to be accompanied by the `VZEROUPPER` or `VZEROALL` instructions to clear the problematic register state.

6.2.4 Detecting Non-AVX-512 Code

AVX-512 code is often limited to specific program phases. During these phases, the task should be marked as an AVX-512 task using the mechanism described above, but after the phase the task should be marked as a non-AVX-512 task again and should be migrated to a non-AVX-512 core to prevent remote AVX overhead. For example, if we take the web server scenario described by Vlad Krasnov [141], the cryptography routines of the web server should be executed on AVX-512 cores, whereas the remainder of the web server should be executed on non-AVX-512 cores at high CPU frequencies. The mechanism described in the previous section does not provide any information about the end of AVX-512 program phases, though, as no further exceptions are triggered on AVX-512 cores. In fact, there is no practical mechanism provided by the CPU to automatically detect the absence of any such type of instruction. Therefore, the OS has to resort to heuristics to detect the end of AVX-512 program phases.

Li et al. propose a simple timeout after which the application is automatically assumed to not use any instruction of concern anymore [157] (Figure 6.5a). Alternatively, they test whether during this timeout the application has made use of floating-point instructions again and extend the timeout whenever it does (Figure 6.5a). While these two approaches are simple, they present a tradeoff between effectiveness and low overhead. In particular, short timeout durations are associated with overhead from additional timer interrupts, exceptions, and perhaps even unnecessary task migration between non-AVX-512 and AVX-512 cores if the timeout expires while the task is still using AVX-512. In contrast, long timeouts substantially impact the ability to prevent remote AVX overhead, as any non-AVX code executed before the timeout expires is likely executed at reduced frequencies as the task is still running on an AVX-512 core. Li et al. evaluate their prototype with timeouts in the range of multiple milliseconds resulting in acceptable overhead, yet these timeouts are far longer than the AVX frequency change delay. For such configurations, experiments conducted by Peter Brantsch [35] show timeout-based approaches to be ineffective for the mitigation of remote AVX overhead. His prototypes only provided performance gains



(a) Simple timeout

(b) Periodic usage checks

(c) Syscall-based heuristic

Figure 6.5: Several heuristics can be used to detect the end of AVX-512 phases. The OS can assume that the phase has ended after a fixed timeout has expired (a) as suggested by Li et al. [157]. Alternatively, the OS can repeat the timeout until no further AVX-512 instructions have been executed (b). We mainly employ the heuristic proposed by Peter Brantsch [35] where we mark tasks as non-AVX tasks as soon as they execute any system call.

if timeouts were in the range of few microseconds, but such short timeouts likely cause excessive overhead for any workload with long AVX-512 phases.

As an alternative, Peter Brantsch therefore suggests marking tasks as non-AVX-512 tasks whenever they execute system calls [35] as shown in Figure 6.5c. The rationale behind this heuristic is that AVX-512 is commonly used in very computationally heavy parts of a program, whereas such parts commonly do not perform any I/O. Any system call therefore has a high probability of marking the end of a computationally heavy phase and therefore the end of AVX-512 code. We employ this approach in our prototype, combined with a simple timeout mechanism as a fallback in case programs do not execute any system calls for extended periods of time after the end of AVX-512 program phases.

We extend the approach with a second heuristic that detects very short non-AVX-512 phases for which a migration to a non-AVX-512 core is more costly than execution of the non-AVX-512 phase at lower frequency levels. In particular, we measure the CPU time during individual non-AVX-512 phases. Whenever most recent non-AVX-512 phase have been shorter than 100 µs, system calls do not cause any change to the task type. Instead, we assume that the task will quickly execute AVX-512 code again after the system call. To determine whether most recent non-AVX-512 phases have been short, we map the CPU time to 1 or 0 depending on whether it was shorter than 100 µs and then calculate an exponential moving average of that value. If this heuristic falsely determines that a task type change is detrimental to performance, the timeout mechanism described above will eventually rectify the mistake and mark the task as a non-AVX-512 task.

6.3 Implementation

The design described above is not limited to any specific instruction set and the concepts are also applicable to other CPUs with similar frequency management. In particular, the design can likely be used on future CPUs with other power-intensive instructions if, for example, similar mechanisms to intercept these power-intensive instructions are available. For current Intel CPUs with support for AVX-512, we created a concrete prototypical implementation based on the Linux 5.9 kernel. In the following, we present our implementation and discuss additional aspects of this prototype which are specific to the underlying scheduler and operating system as well as the specific hardware. In particular, we first explain why we constructed our scheduler on top of the MuQSS scheduler instead of the more common CFS scheduler. We then describe the required modifications to the scheduler's data structures. Finally, as the mechanism described in Section 6.2.3 breaks CPU feature detection on x86 CPUs, we describe a technique to create the illusion that all cores continuously support AVX-512.

6.3.1 Choice of Scheduler

The Linux *Completely Fair Scheduler* (CFS) [173] is the most widely used scheduler on Linux systems. While our evaluation would therefore benefit from basing our scheduler on CFS, we found that CFS provides far too slow load balancing. As described in Section 6.2.1, migration of tasks to suitable cores is commonly performed by the scheduler's load balancing mechanism. Core specialization requires particularly quick load balancing as tasks placed on unsuitable cores are unable to make progress. CFS performs periodic load balancing where the scheduler compares the length of the ready queues of different logical CPUs and transfers tasks to a different CPU to equalize load. To limit overhead, this load balancing is rather infrequent, which means that non-AVX-512 cores would frequently idle while AVX-512 cores would frequently execute non-AVX-512 tasks.

Other schedulers provide faster load balancing. For example, the *Multiple Queue Skiplist Scheduler* (MuQSS) [139] is an out-of-tree scheduler for Linux which bases scheduling decisions on virtual deadlines. In this scheduler, each logical CPU is associated with one run queue. The scheduler not only considers tasks in this queue but, depending on the configuration, also tasks from other run queues to improve response times. In particular, in the "interactive" configuration the scheduler migrates the task with the earliest deadline to the current CPU on each invocation. As this fast load balancing mechanism fulfills the requirements of core specialization, we base our prototype on MuQSS 0.204 running in interactive mode for the Linux 5.9 kernel [138].

Unlike CFS, MuQSS implements *run queue sharing* where multiple logical CPUs can be associated with the same run queue [140]. By default, MuQSS lets all logical CPUs on a single chip share one run queue for even shorter response times – if all logical CPUs share one run queue, no migration of tasks between different queues is needed at all. As our experiments indicate that this setup is slower than setups with less run queue sharing [88], we configure MuQSS to only share run queues among sibling hyper-threads instead.

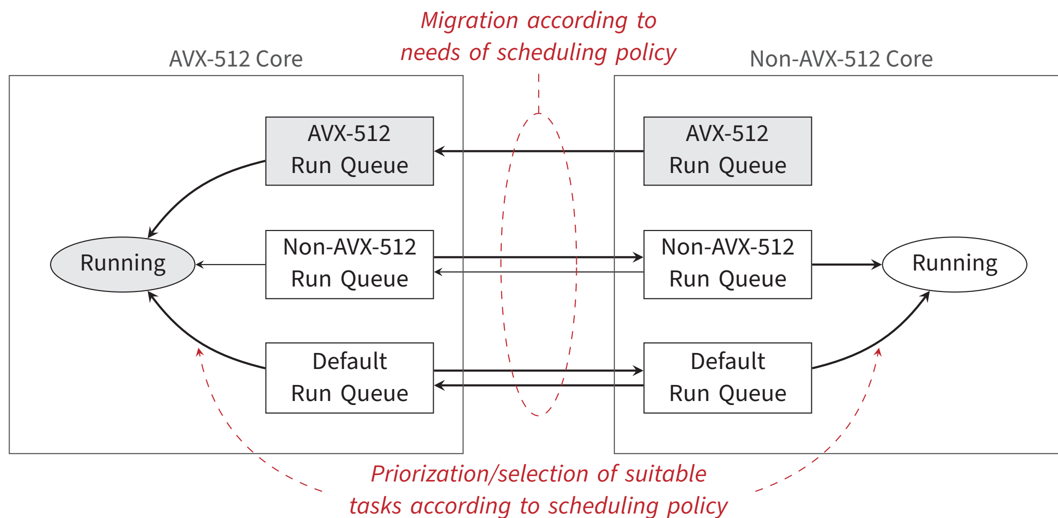


Figure 6.6: Our prototype implements different handling of the task types via replicated run queues. On non-AVX-512 cores, the main scheduling algorithm never picks tasks from the run queue containing AVX-512 tasks. On AVX-512 cores, in contrast, non-AVX-512 tasks are only executed if the other run queues are empty.

6.3.2 Tripled Run Queues

We modify MuQSS to differentiate between three different types of tasks [85]. As described in Section 6.2.1, the scheduler treats AVX-512 and non-AVX-512 tasks differently. In addition, we introduce a third *default task* type. Default tasks are tasks which are not affected by core specialization. These tasks are executable on all CPU cores and are executed with the same priority as AVX-512 tasks when executed on AVX-512 cores. The third task type is needed to prevent starvation of system tasks that are pinned to individual CPUs. During execution of a default task, the OS enables support for AVX-512 even if the task is executed on a non-AVX-512 core.

Initially, all tasks in the system are default tasks. Only when a task is explicitly marked as an AVX-512 or non-AVX-512 task via system call, core specialization will be applied when scheduling the task. Whenever such a non-default task is forked to create a child task, the child task inherits its non-default task type. We expect all OS shells to automatically mark themselves as non-AVX-512 tasks after login to enable core specialization for all tasks created directly by users. During our evaluation we did not modify the shell as such lasting modifications could have interfered with other experiments on the system. Instead, we constructed a separate application whose sole purpose is to launch another command with a non-default task type.

To enable the scheduler to differentiate between the three task types, we simply replicate all run queues three times as shown in Figure 6.6 [85]. One queue per physical CPU – the AVX-512 run queue – holds the runnable AVX-512 tasks, while the other two queues – the non-AVX-512 run queue and the default run queue – hold the runnable non-AVX-512 and default tasks respectively. Non-AVX-512 cores only select tasks from the non-AVX-512

and the default run queue. In contrast, AVX-512 cores select tasks from the AVX-512 and the default run queues, but also fetch tasks from the non-AVX-512 run queue if no other tasks are available, thereby implicitly prioritizing AVX-512 tasks over non-AVX-512 tasks. The same logic applies when the scheduler fetches tasks from remote run queues with an earlier deadline than the tasks in the local run queues, which results quick migration of tasks suitable for execution on the local CPU.

6.3.3 AVX-512 Feature Detection

As described in Section 6.2.3, tasks are placed into the appropriate run queue based on register use. One important problem of this fault-and-migrate mechanism used to detect 512-bit SIMD operations is that it breaks CPU feature detection. In particular, applications first have to test whether both the CPU and the OS support AVX-512 before using its instructions. The method proposed by Intel to detect whether 512-bit SIMD operations are safe to use consists of three steps [114, p. 15-5]. First, the application has to use the `CPUID` instruction to check whether the processor supports the `XGETBV` instruction. Second, the application has to execute `XGETBV` to retrieve the value of the `XCR0` register and has to test whether the bits are set which signal that the OS provides context switch support for 512-bit registers. Third, the application has to use `CPUID` to check whether the CPU supports AVX-512. As described earlier, our approach clears two relevant bits in `XCR0` which causes existing applications to wrongly assume that AVX-512 is not supported by the OS if the applications are executed on non-AVX-512 cores.

This problem could easily be solved by adding another level of indirection. If all applications are forced to execute a system call to retrieve the list of supported CPU features, the OS can return identical information on all CPU cores. Such an interface is, for example, implemented by macOS which only enables context switch support for AVX-512 after the first AVX-512 instruction has executed by the application.[7] However, existing applications for other operating systems use the standard method described by Intel to detect whether AVX-512 is available and need to be modified to use other methods. While such modifications are likely simple for newly-built applications or existing open-source applications, closed-source applications are often hard to modify. We therefore argue that there is a need for a technique to fix CPU feature detection for unmodified programs.

Instead of introducing system calls, related work has proposed to intercept the existing instructions used for CPU feature detection. Specifically, Reddy et al. suggest that processors should be configurable to raise exceptions upon execution of instructions such as `CPUID` [203]. On Intel CPUs, exceptions for `CPUID` can be enabled by setting a bit in the `MSR_MISC_FEATURES_ENABLES` register.[8] In the exception handler, the OS can then emulate the instruction to return the desired results.

[7] See Apple Darwin xnu-7195.121.3, `osfmk/i386/fpu.c`, l. 174ff.
`https://github.com/apple/darwin-xnu/blob/xnu-7195.121.3/osfmk/i386/fpu.c#L174`

[8] This bit does not appear to be documented by Intel, although it is used by the Linux kernel. See `set_cpuid_faulting()` in Linux 5.9, `arch/x86/kernel/process.c:250`. In some of our experiments, we use this bit via the libvirtcpuid library [245] to prevent applications from using AVX2 and AVX-512. If the bit is not available, it is possible to intercept `CPUID` by running all processes in a virtualization environment as suggested by Belay et al. [19] because the `CPUID` instruction triggers a VM exit [116, p. 25-2].

Our fault-and-migrate implementation does not affect the results of CPUID. Instead, it modifies XCR0, so our prototype needs to intercept the XGETBV instruction. This instruction is harder to intercept as the CPU does not provide any viable mechanism that causes the instruction to trigger exceptions.[9] We therefore propose static binary rewriting of programs to replace any XGETBV found by a disassembler with an invalid opcode to trigger an undefined instruction exception. The exception handler then inspects the faulting instruction to determine whether the fault was triggered by a replaced XGETBV instruction and, if it was, emulates the instruction in the kernel returning a result that indicates support for AVX-512. Such modifications can be automatically performed on all application binaries and libraries. We only perform these modifications on a copy of the applications used during our experiments as we want the system to be usable with a stock Linux kernel to generate reference measurements for our evaluation.

6.4 Estimation of Effectiveness

While our scheduler is able to mitigate remote AVX overhead in many cases, it is not suited for all kinds of applications. In particular, the task migrations caused by our scheduler require application state to be transferred from one core to another [52]. In some cases, the additional cache misses cost more performance than is gained via improved CPU frequencies. To measure the impact of task migrations on particularly memory-heavy applications, we construct a synthetic worst-case example. Our program first performs 50000 dependent random read accesses to a memory region with a fixed size. After each read access, it also performs a write access to the same cache line to invalidate copies of the cache line in the local caches of other cores. The pairs of read and write accesses are interleaved with 512-bit SIMD instructions so that the code is executed on an AVX-512 code. The program then executes a system call to force the task to be marked as a non-AVX-512 task. It then performs the same memory accesses again, but without any AVX-512 instructions. During this phase, the program is most likely executed on a non-AVX-512 core, so the working set has to be transferred to this core. All steps are repeated 20000 times to trigger approximately 40000 task type changes.

Figure 6.7 shows the performance improvement caused by our prototype when the program is executed 48 times in parallel on a 16-core CPU with hyper-threading, with negative values instead indicating increased overhead. We vary the size of the memory region accessed by the program to vary the number of potential cache misses per thread migration. Regardless of the size of the memory region, our prototype fails to improve performance. This result is expected given that the program is particularly memory heavy and therefore hardly profits from increased CPU frequencies. In addition, the program suffers from overhead caused by core specialization. For small working sets, this overhead is mainly caused by the scheduler itself as it is invoked far more frequently than without core specialization. For large working sets, even more overhead is caused by cache line bouncing as the data is frequently moved between cores.

[9] XGETBV triggers exceptions if the OS disables support for XSAVE. This method of intercepting XGETBV is impractical, though, as many other instructions including all AVX and AVX2 instructions become unavailable as well [114, pp. 14–15 f.].

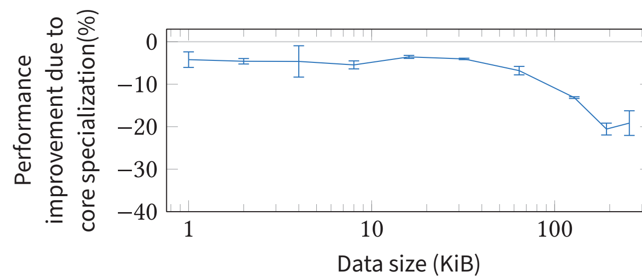


Figure 6.7: Core specialization has a negative performance impact on a particularly memory-heavy program that mainly consists of dependent, unpredictable memory accesses. Large working sets in particular cause lots of cache line bouncing as the task is migrated between cores. Even small working sets do not show any improvement when executed with core specialization, though, as the memory-heavy code is hardly effected by frequency changes.

The experiment shows that there are situations where core specialization is harmful to performance. In these cases, other techniques to mitigate the impact of remote AVX overhead such as our scheduler modifications described in Chapter 7 should be selected instead. Deciding between different countermeasures against remote AVX overhead requires a mechanism to gauge the effectiveness of our prototype. We propose a program which performs the following steps to measure the overall performance impact of core specialization:

1. The program counts the total number of instructions executed in user-space within a fixed duration using performance counters in order to get a baseline measurement of system throughput. Instructions executed in kernel-space must be not be counted as they may be caused by the scheduler itself. If these instructions were included, high thread migration rates would result in higher instruction rates and, consequently, exaggerated throughput readings.

2. The program disables core specialization and configures the scheduler to choose from all run queues on all cores without any prioritization between the different types of tasks.

3. The program then repeats the measurement from the first step again to measure throughput when core specialization is disabled. The difference between the two instruction rates is an indicator for whether core specialization is beneficial for performance or not.

This mechanism depends on the assumption that the workload behaves identically during the first and the third step and that there are no instruction rate differences caused by execution phase changes. To reduce the error introduced by such execution phase changes, the steps outlined above should be repeated several times. It is unlikely that disabling and re-enabling core specialization repeatedly coincides with an execution phase change.

6.5 Evaluation

We evaluate our scheduler with the intent to demonstrate its ability to mitigate remote AVX overhead. To do so, we first perform direct measurements of remote AVX overhead and conduct a comparison between core specialization and the core scheduling design proposed by Aubrey Li [154]. We then describe further experiments to show that the efficacy of our prototype stems from increased average CPU frequencies. Finally, we also quantify the overhead caused by our prototype and show that the technique described in Section 6.4 can be used to predict whether our prototype results in an overall performance improvement.

As in most previous chapters, all experiments are conducted on a system with an Intel Xeon Gold 6130 CPU, 24 GiB DDR4 RAM, and the Fedora 31 Linux distribution. For comparisons with the unmodified MuQSS scheduler, we use MuQSS 0.204 and Linux 5.9 as our prototype is based on this version of the scheduler. As Linux 5.9 does not support core scheduling, experiments with core scheduling use the Linux 5.14 kernel.[10] During our experiments, we use the same versions of the benchmarks previously described in Section 3.3.5. All benchmarks are repeated ten times unless specified otherwise.

6.5.1 Effectiveness

To demonstrate the effectiveness of our prototype, we measure the remote AVX overhead in our prototype and compare it to the remote AVX overhead when the same system uses the MuQSS scheduler. For both configurations, we repeat a range of the experiments from Section 3.3.5 where we measured the CPU times of workloads with AVX-512 and compared it to identical workloads without AVX-512 to determine the amount of remote AVX overhead. The benchmarks include benchmarks from the Parsec 3.0 [25] benchmark suite executed alongside OneDNN benchdnn either configured to use AVX-512 or only SSE4 instructions. These benchmarks mainly demonstrate the impact of our prototype on remote AVX overhead caused by hyper-threading – if OneDNN uses AVX-512 and is executed on the sibling hyper-thread next to a Parsec thread, the Parsec benchmark is slowed down by remote frequency reduction. As in Chapter 3, we configure the Parsec benchmarks and benchdnn to use 32 threads each so both are able to utilize all CPU resources available to them. We measure the CPU time of the Parsec benchmarks to calculate the amount of remote AVX overhead. To demonstrate the impact on remote AVX overhead caused by frequency change delays, we also repeat the web server benchmark derived from experiments by Vlad Krasnov [141]. We configure the nginx web server to serve a static page with on-the fly compression using OpenSSL for encryption. The web server is configured to use the ChaCha20-Poly1305 encryption scheme as its implementation in OpenSSL was shown to trigger a reduction to the AVX-512 frequency level. We execute

[10] We were unable to use the same kernel version for all experiments as MuQSS is not available for the 5.14 kernel [136] whereas core scheduling was introduced in this kernel version [54]. While the version difference may impact the comparison of our approach to related work in Section 6.5.1, we argue that any such impact is likely small. Specifically, each calculation of remote AVX overhead only involves a division of two CPU time measurements from a single kernel version. Performance changes between the two kernel versions that are unrelated to scheduling should have little impact on the resulting relative value.

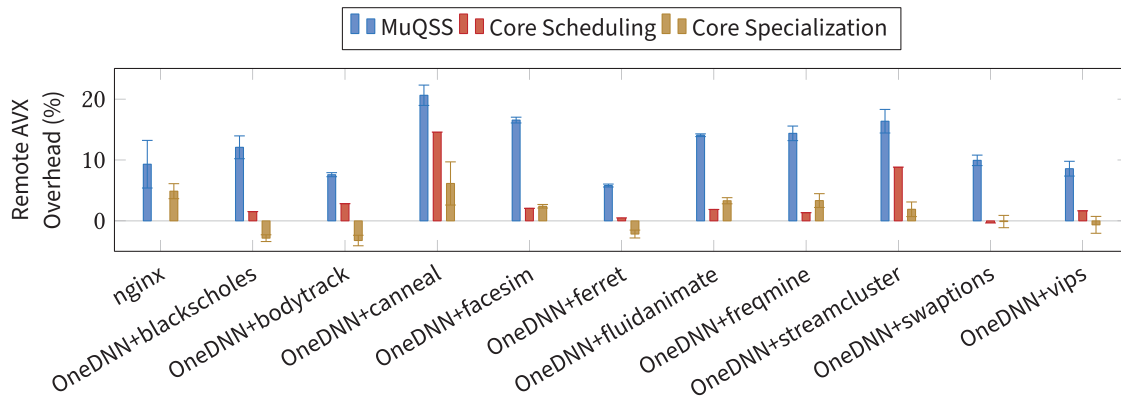
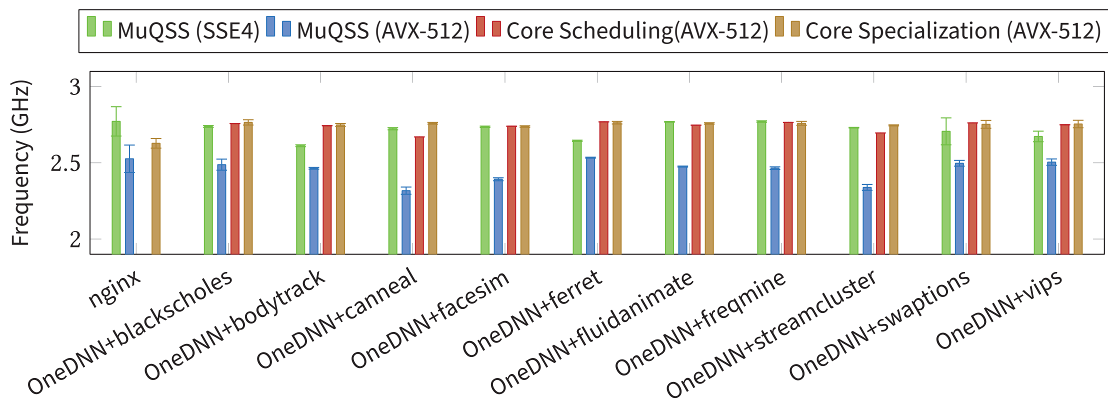


Figure 6.8: We compare the CPU time of a range of applications with and without AVX-512 to calculate the remote AVX overhead during execution with MuQSS, an approach based on core scheduling, and our core specialization prototype. It can be seen that both core scheduling and core specialization are able to mitigate most remote AVX overhead. Core scheduling cannot be applied to single-application workloads such as nginx, though.

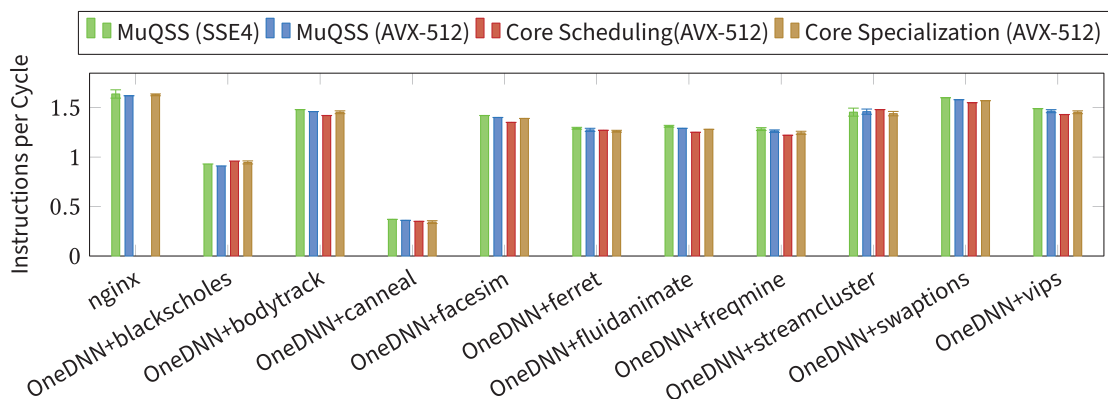
the wrk [254] load generator on two physical cores while the web server is restricted to one hyper-thread of each of the remaining 12 cores to isolate the impact of frequency change delays and to prevent remote AVX overhead caused by hyper-threading. Similar to our experiments in Section 4.3.2, we disabled AVX and AVX2 instructions in glibc to prevent nginx and the Parsec benchmarks from reducing the CPU frequency themselves during our experiments.

In addition to the comparison between our prototype and MuQSS, we compare our approach to a simplified version of Aubrey Li's core scheduling approach [154]. As described in Section 6.1, this approach uses core scheduling cookies to prevent co-scheduling of AVX-512 and non-AVX-512 tasks on the same physical core at the same time. As Aubrey Li's prototype uses the existing mechanism found in the Linux kernel to detect AVX-512 tasks which we deem unreliable, we manually assign different core scheduling cookies to OneDNN and the Parsec benchmarks to achieve reliable separation of the two applications. Our core scheduling experiments do not include the nginx workload – as the workload only consists of a single application and all threads frequently switch between AVX-512 and non-AVX code, manual marking as described above is not possible. While a more fine-grained classification of threads may be possible, nginx is mainly affected by frequency change delays which are not covered by core scheduling as we describe in Section 6.1.

Figure 6.8 shows the remote AVX overhead during experiments with unmodified MuQSS, core scheduling, and our prototype. As expected, our prototype is effective at mitigating remote AVX overhead. While co-scheduling caused by the unmodified MuQSS scheduler results in an average remote AVX overhead of 12.3% when the workloads use AVX-512, only a 1.2% slowdown remains during runs in our prototype compared to an identical setup without AVX2 or AVX-512. Our prototype also performs better than core scheduling which is unable to improve the performance of nginx and features a remaining average remote AVX overhead of 3.5% for the other workloads.



(a) Average CPU frequency during execution of the benchmark (note the restricted y axis range)



(b) Instructions per second

Figure 6.9: Core specialization improves average CPU frequencies (a). In some cases, the frequencies achieved by our prototype for AVX-512 workloads are even higher than those measured when OneDNN only uses SSE4. Overall, our prototype has little impact on instructions per cycle (b) despite the increased code size and more thread migrations.

While core specialization generally appears effective at mitigating remote AVX overhead, not all benchmarks benefit equally. In particular, our prototype is only partially effective for some benchmarks such as nginx or canneal while other benchmarks counterintuitively even become faster when OneDNN uses AVX-512. To determine the reasons for the differences between the workloads, we analyze the CPU frequency during execution of nginx and the Parsec benchmarks. Figure 6.9a shows the average frequency during the runs conducted as part of our previous experiment. Our frequency measurements show why some workloads become faster when executed with core specialization. In particular, workloads such as bodytrack or canneal operate at a reduced frequency level with SSE4 and MuQSS, whereas our prototype causes the system to almost reach 2.8 GHz as expected in the absence of AVX frequency reduction. This result is counterintuitive given that the 128-bit operations of SSE4 themselves are not expected to cause any frequency reduction. Further analysis shows that SSE4 only causes a frequency reduction when the

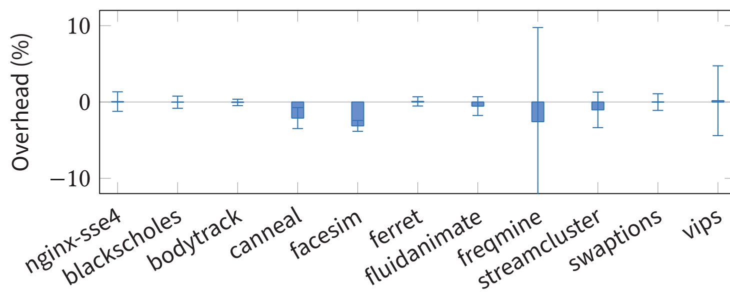


Figure 6.10: To measure the overhead of core specialization itself, we compared the completion time of various Parsec benchmarks and the throughput of nginx without AVX-512 to MuQSS. In these scenarios, core specialization hardly has any impact on performance.

two applications are co-scheduled on sibling hyper-threads. We assume that the frequency reduction is caused by a mixture of light 256-bit SIMD instructions executed by the Parsec benchmarks and heavy 128-bit SIMD instructions in OneDNN. Documentation from Intel indicates that such a mixture can cause a frequency reduction even if the individual instructions are able to be executed at the highest frequency level [113, p. 2-14].

Our frequency measurements also show why substantial remote AVX overhead remains when nginx is executed in our prototype. In particular, the frequency achieved by core specialization remains below the frequency achieved in the absence of AVX-512 instructions. This result is caused by a sub-optimal number of AVX-512 cores. As the task types frequency change, the number of AVX-512 tasks shows high variation, which poses a challenge for our method to allocate AVX-512 cores described in Section 6.2.2. Whenever our prototype allocates more AVX-512 cores then necessary, non-AVX-512 tasks are frequently executed on AVX-512 cores, resulting in substantial remaining remote AVX overhead. Additional experiments where we manually designate only a single core as an AVX-512 core result in improved frequencies and lower remote AVX overhead.

Our frequency measurements fail to explain the results for other benchmarks. In particular, benchmarks such as blackscholes, fluidanimate, or freqmine experience substantial remaining overhead even though their average frequency is close to the ideal non-AVX frequency. To explain these results, we measure the instructions per cycle (IPC) during execution of the benchmarks. The results of this experiment are shown in Figure 6.9b. It can be seen that the benchmarks suffer from slightly reduced IPC when executed with our prototype. Such reduced IPC can, for example, be caused by increased cache line bouncing as tasks and their working sets are frequently migrated to different cores. Also, the increased size of the scheduler code may cause the scheduler to more frequently displace application data from the cache. While our experiments do not show why IPC is reduced, they show that the impact of our prototype on IPC is generally minor. Instead, CPU frequency improvements are responsible for most of the performance impact of our prototype.

Listing 6.1: Synthetic workload to trigger task type changes at varying rates depending on the value of N.

```
1  int fd = open("/dev/zero", O_RDONLY);
2  while (true) {
3          /* very short AVX-512 phase */
4          asm volatile("vpord zmm0, zmm0, zmm0");
5          /* system call to end the AVX-512 phase */
6          char c;
7          read(fd, &c, 1);
8          /* very little non-AVX code */
9          for (int i = 0; i < N; i++) {
10             asm volatile("nop");
11         }
12 }
```

6.5.2 Overhead

While or previous experiments show the beneficial impact of core specialization on workloads with AVX-512, they do not show the performance impact in the absence of such power-intensive instructions. As the latter case is potentially more common than the former, the increased complexity of our scheduler and the resulting increased cache footprint must not cause any substantial slowdown for non-AVX-512 applications. To measure the overhead caused by our scheduler for such workloads, we measure the completion time of individual Parsec applications in isolation – without any parallel AVX-512 application – and compare it to the completion time achieved with MuQSS. We also compare the throughput of the nginx web server with OpenSSL configured to use SSE4. Figure 6.10 shows the overhead caused by our prototype calculated as the increase of completion time increase or, in the case of nginx, throughput reduction. The results show that any performance impact of core specialization on such non-AVX-512 workloads is very limited.

6.5.3 Short Non-AVX-512 Phases

An important reason for the limited overhead caused by core specialization is that our prototype prevents task type changes if it detects very short average non-AVX-512 phases as described in Section 6.2.4. We demonstrate the importance of this heuristic using the program shown in Listing 6.1 which is designed to frequently switch between AVX-512 and non-AVX-512 code. The program executes a loop consisting of a 512-bit SIMD instruction, a system call, and a varying number of non-AVX instructions defined by the variable N. The AVX-512 instruction causes our prototype to mark the task as an AVX-512 task and to migrate it to an AVX-512 core. The system call then potentially causes the task type change to be immediately reverted again, likely triggering a migration back to a non-AVX-512 core. This code in the system call handler is only executed when our prototype detects that the average non-AVX-512 phase is sufficiently long.

Figure 6.11 shows the performance improvement over MuQSS caused by our prototype for 48 instances of the program executed in parallel, where negative numbers indicate

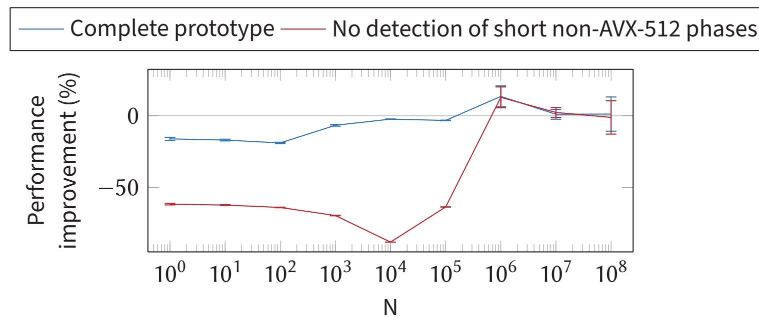


Figure 6.11: We demonstrate the necessity for detection of short non-AVX-512 phases by measuring the impact of core specialization on the program shown in Listing 6.1. When the detection of short non-AVX-512 phases is disabled, the large rate of thread migrations severely impacts performance.

overhead caused by core scheduling. We also repeat the experiment with a modified version of our prototype where the heuristic to detect short non-AVX-512 phases is disabled. In both cases, our prototype fails to provide any performance benefit for very short non-AVX-512 phases while improving performance for lower task migration rates. The overhead differs between the two configurations, though.

When the heuristic is enabled, our prototype correctly detects when the non-AVX phase is too short, in which case it always categorizes the tasks as AVX-512 tasks. As a result, the scheduler marks almost all cores as AVX-512 cores. Our prototype reserves one core for non-AVX-512 tasks to prevent starvation for these tasks, so the performance of workloads completely consisting of AVX-512 code is slightly reduced. When the heuristic is disabled, the overhead caused by core specialization is far larger. In this case, the scheduler is invoked during each system call and each AVX-512 instruction, thereby massively increasing the total number of instructions executed. The difference between the two configurations shows that the heuristic to detect short non-AVX-512 phases is essential to limit worst-case overhead.

6.5.4 Estimation of Effectiveness

Even with the heuristic to detect short non-AVX-512 phases, our prototype is detrimental to the performance of some workloads. Therefore, we described a tool to determine whether core specialization results in a net performance improvement in Section 6.4. To show that this tool is able to detect both types of excessive overhead – overhead stemming from cache line bouncing as described in Section 6.4 and overhead stemming from excessive task type change rates as described in Section 6.5.3 – we test the tool using the programs described in the two sections. In both cases, we compare the output of the tool with the results of direct measurements shown in Figures 6.7 and 6.11. To determine the absolute accuracy of the output in real-world scenarios, we also use the tool to predict the impact of core specialization on the nginx workload used in the previous sections configured to use AVX-512.

Our experiments demonstrate the reliability of the tool. It correctly shows that the memory-heavy workload described in Section 6.4 never benefits from core specialization and that configurations with large working sets are particularly badly affected. The tool also shows that the program shown in Figure 6.1 only benefits from core specialization during runs with $N = 10^6$ and $N = 10^7$. Such reliable estimates of the outcome of core specialization mean that the tool can be used to determine whether core specialization should be activated or whether other less intrusive techniques such as that described in the next chapter should be used instead. Note that this use case does not require precise absolute estimates of the performance impact. Nevertheless, an evaluation with nginx shows high accuracy. While our tool measures a performance improvement by 4.3%, a direct comparison of the throughput with MuQSS and our prototype shows an improvement by 4.2%.

6.6 Discussion

As our evaluation shows, core specialization can be used to mitigate the remote AVX overhead in many workloads. However, its effectiveness is limited both by the overhead of task type changes and the resulting thread migrations and by imprecise detection of power-intensive program phases. In the following, we therefore first discuss potential techniques to reduce the cost of thread migrations, arguing that these techniques are likely either ineffective or difficult to implement. As *non-uniform memory access* (NUMA) systems potentially increase the cost of thread migration, we also describe how our approach needs to modified to support NUMA systems. We then describe the limitations of fault-and-migrate when used to detect power-intensive instructions such as AVX-512. As fault-and-migrate is the best mechanism available on current CPUs, we conclude the chapter by outlining a hardware-software interface that would enable more precise detection of power-intensive program phases on future CPUs.

6.6.1 Reducing Migration Overhead

As described above, the additional overhead introduced by core specialization is one of the main limiting factors for the effectiveness of our design. More efficient thread migration would make the approach usable with much shorter non-AVX-512 phases. There are three main sources for overhead. First, during each scheduler invocation, more code is executed to check the replicated run queues and to determine whether AVX-512 cores should be added or removed. Second, the scheduler is potentially invoked far more often if there are frequent transitions between AVX-512 and non-AVX-512 program phases. Third, the task migrations themselves potentially increase cache line bouncing as the working set of the tasks needs to be transferred between different CPU cores.

While the impact on cache line bouncing is unavoidable, the overhead caused by scheduler invocations itself can potentially be reduced. One potential approach would be to separate core scheduling from the kernel's scheduler and use a hybrid threading model where the application performs core specialization completely in user space. Ioannis Papamanoglou combined such an approach with manual instrumentation of the target

application and achieved good results for a nginx benchmark similar to that used in our evaluation [187]. We expect it to be possible to combine user-level core specialization with automatic detection of AVX-512 code as described in this thesis and with a upcall mechanism to inform the application about the number and location of AVX-512 cores. Such a hybrid approach would likely introduce less overhead because specialized, less complex user-level scheduling algorithms could be used and because less CPU context would have to be transferred during thread migration. We leave construction of such a hybrid approach as future work due to the increased implementation complexity.

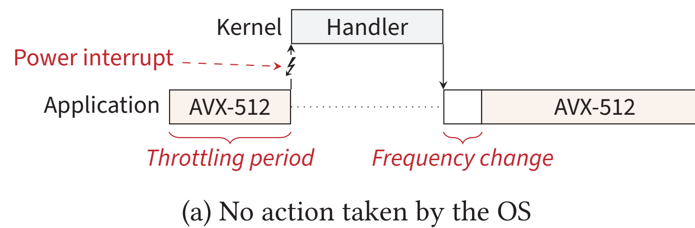
Alternatively, very short AVX-512 phases of a thread could be emulated on the thread's current non-AVX-512 core instead of migrating the thread to an AVX-512 code. This technique is commonly called *trap-and-emulate* [3] and is commonly used in virtual machine monitors to emulate privileged instructions [196]. Similarly, *proxy execution* [99] is a technique where individual instructions are executed on a different CPU core, but the thread is then resumed on the original core. Both mechanisms cause some overhead themselves – e.g., because emulation of individual instructions is always far more costly than direct execution – but may cause less overhead than full migration of the thread for very short sections of AVX-512 code. However, as AVX-512 is mainly used for computationally heavy code, we do not expect such short AVX-512 phases to be common, so we do not expect any substantial performance improvements from trap-and-emulate or proxy execution implementations.

6.6.2 NUMA Support

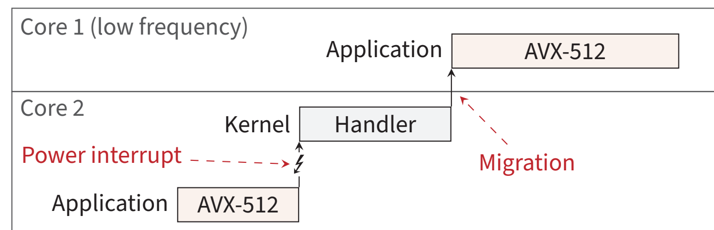
To simplify development, our prototype only targeted systems with uniform memory access. On NUMA systems, task migration between different NUMA nodes often incurs an extra penalty as accessed data has to be fetched from memory attached to a different NUMA node [43]. The load balancing of existing schedulers therefore commonly tries to minimize migration across NUMA nodes. Our scheduler currently does not ensure that there are AVX-512 cores available on each NUMA node, so tasks may be forced to migrate to a different NUMA node if there is no AVX-512 core on their current NUMA node. To improve NUMA support in our prototype, it is therefore necessary to modify AVX-512 core allocation to reserve a sufficient number of cores on all NUMA nodes. In addition, the core scheduler algorithm needs to be modified to prefer tasks from the current NUMA node.

6.6.3 Imprecise Detection of Power-Intensive Code

Another important limitation of our prototype is that it is unable to differentiate between heavy and light 512-bit instructions. Whereas the former instruction type causes a transition to the lowest AVX-512 frequency level, the latter only triggers the intermediate AVX2 frequency level [113, p. 2-13]. Our method to detect AVX-512 code only relies on the register size to categorize tasks. The prototype therefore co-schedules heavy and light 512-bit SIMD code, even though doing so exposes the latter type of code to remote AVX overhead.



(a) No action taken by the OS



(b) Frequency change is prevented via migration to a different core

Figure 6.12: Interrupts triggered before the CPU reduces its frequency (*power interrupts*) would enable the OS to precisely detect power-intensive tasks responsible for frequency reduction (a). In the interrupt handler, the OS could prevent further execution of power-intensive code to prevent the frequency reduction (b).

Similarly, our prototype is not able to prevent the remote AVX overhead caused by heavy 256-bit SIMD instructions such as many AVX2 floating-point instructions. Our method to detect power-intensive instructions could be extended to cover 256-bit register accesses as well. See Section 7.4.1 in the next chapter for a description of the corresponding modifications. However, the modified mechanism would then trigger exceptions during execution of many instructions that do not cause any frequency change such as light 256-bit operations and even instructions such as `VZEROUPPER`. These instructions are more common than 512-bit operations, so the resulting task migrations would often cause excessive overhead.

6.6.4 Proposed Hardware Changes

As detecting power-intensive code based on register accesses is inaccurate, we argue that future CPUs should provide enhanced mechanisms to detect power-intensive code. We expect such hardware-software interfaces to become increasingly important as CPUs become more and more power-limited as outlined in Section 2.3. In particular, we expect future CPUs to provide more and more instruction set extensions that increase power variability as such specialized hardware has shown to be energy-efficient [234]; we therefore expect these CPUs to use similar frequency scaling policies as those found on current Intel CPUs. A hardware-software interface to detect power-intensive code in order to enable techniques such as core specialization has to fulfill two main requirements not fulfilled by existing mechanisms:

1. Information provided by the CPU has to be accurate. Migrating tasks to a different core even though they do not cause any remote frequency reduction is harmful, as is not migrating tasks even though they cause a frequency reduction.

2. Information provided by the CPU has to be actionable. The OS has to be able to prevent any frequency changes by, for example, migrating tasks to a different core. In this sense, the `CORE_POWER.THROTTLE` performance event [116, p. 19-8] is a somewhat accurate indicator of frequency reduction, but does not provide actionable information. When the event is detected, the CPU core has already begun reducing its frequency.

We argue that existing hardware is easily modified to provide a hardware-software interface that fulfills these two requirements. Currently, before a CPU core changes its frequency level, there is a throttling period during which the core decides on whether a frequency change is required as described in Section 3.2.2. As shown in Figure 6.12, a *power interrupt* can be added to this throttling period which signals that the CPU has determined that a frequency change is necessary. We suggest delivering the power interrupt only to the hyper-thread executing the power-intensive code that triggered the interrupt. Techniques such as that described in Section 4.6.2 can be used to identify this hyper-thread. In cases where a frequency reduction is only necessary due to the combination of the code running on both hyper-threads, the interrupt can be delivered to both hyper-threads.

In the interrupt handler, the OS can prevent the frequency change by moving the task to a different CPU core or by scheduling a different task, in which case it instructs the CPU core to cancel the frequency change as no more power-intensive code is expected. Alternatively, the OS can return from the interrupt handler and let the CPU continue executing the power-intensive task, in which case the CPU immediately performs the frequency change. System stability is not endangered if the OS instructs the CPU core to cancel the frequency change and then continues to execute power-intensive code on that core. In the case of such a misprediction, the CPU simply restarts the throttling period upon execution of the next power-intensive instruction, so no excessive voltage droops are possible.

7 Scheduling for Improved Performance Isolation

In the previous chapters, we described remote AVX overhead primarily as a system-wide performance problem. It is also a performance isolation problem, though, as AVX2 and AVX-512 tasks slow other tasks down. Such remote slowdown can be problematic even if the benefits of AVX2 and AVX-512 mean that system-wide performance is not reduced. In particular, if the affected tasks have real-time requirements, the remote slowdown may cause the tasks to miss their deadline. Also, cloud customers commonly rent virtual CPUs or fractions of virtual CPUs in the expectations that these CPU shares represent specific throughput, yet remote AVX overhead may cause users to receive less performance for their money than expected. In this chapter, we therefore describe a technique that specifically improves performance isolation in heterogeneous workloads with AVX2 or AVX-512 by reducing the impact of AVX tasks on other tasks. In particular, we describe a scheduler which prioritizes tasks affected by remote AVX overhead at the expense of the tasks which cause the overhead. This policy shifts the effective performance impact of remote AVX overhead so that mainly those tasks which cause it are slowed down.

We start by describing existing approaches to fair scheduling and performance isolation (Section 7.1). These approaches commonly try to achieve performance isolation by guaranteeing that specific tasks receive at least a certain share of the CPU time [239, 45, 227]. We then show how these approaches fail to provide performance isolation in the presence of remote AVX overhead as they fail to take frequency changes into account (Section 7.2).

As time-based scheduling is unable to provide performance isolation, we describe alternative scheduling metrics for AVX2, AVX-512, and other power-limited systems (Section 7.3). While existing scheduling concepts based on energy usage exist, we explain why they cannot be implemented for AVX2 and AVX-512 workloads on current hardware and argue for simpler modifications to existing time-based scheduling instead. In particular, we show how modifying CPU time accounting can improve performance isolation if the task causing remote AVX overhead is billed for the CPU time essentially lost to remote AVX overhead, while the task suffering from remote AVX overhead is not. We then describe how to implement such a scheme in existing time-based schedulers (Section 7.4). As part of our design, we describe a technique to identify tasks using AVX2, AVX-512, or similar power-intensive instruction set extensions derived from our fault-and-migrate implementation described in Chapter 6. In addition, we describe a technique to calculate remote AVX overhead based on information from the PMU and we describe how to modify CPU time accounting to prioritize tasks affected by this overhead. We also describe a concrete implementation in the Linux kernel based on the MuQSS scheduler as our approach requires fast load balancing to be effective (Section 7.5).

We evaluate the resulting prototype based on a range of workloads (Section 7.6) and show how it greatly improves performance isolation for workloads that are able to utilize the additional CPU time allocated by our scheduler. Overall, for a wide range of benchmarks, our scheduler reduces the remote performance impact of AVX2 from 7.3% to 2.1%, while the remote performance impact of AVX-512 is reduced from 23.5% to 6.4%. Our evaluation also shows that our scheduler has very low overhead, making the approach viable in situations where core specialization, the approach described in the previous chapter, causes excessive overhead. Finally, we discuss limitations of our prototype and possible areas for future improvement (Section 7.7).

7.1 Fairness and Performance Isolation

Performance isolation and quality of service have long been identified as an important goal of scheduling algorithms [50]. Since then, previous work has described a wide range of techniques to ensure performance isolation to varying degrees. For example, priority scheduling with hard priorities is an early scheduler algorithm that was designed to guarantee low latency and high throughput for specific high-priority applications [50]. However, its susceptibility to starvation of low-priority tasks means that it provides very weak performance isolation for these tasks and is therefore rarely used in modern systems.

Instead, many modern systems use “fair scheduling” which allocates equal resources to individual tasks [129]. For example, the Linux *Completely Fair Scheduler* (CFS) allocates equal CPU time to individual tasks of equal priority [173]. CFS implements *fair queueing* [63] which is a simple and flexible method to achieve fairness and which can be applied not only to CPU times but to other metrics as well which we exploit in the following. Previous work has, for example, applied fair queueing to networking [63] and disk operations [38]. CFS implements fair queuing by measuring the *virtual runtime* (*vruntime*) which corresponds to the time individual tasks spend on the CPU [121]. During each scheduling decision, CFS then always selects the task with the lowest virtual runtime of execution. If we assume that the virtual runtime of a task is identical to the CPU time used by the task – which is, as we describe in following, not always the case – each task is allocated an identical share of CPU time, resulting in completely fair scheduling. Such fair scheduling achieves only a very weak form of performance isolation, though. In particular, the CPU shares given to individual tasks depend on the total number of tasks in the system. Whenever the number of tasks rises, the CPU time available to individual tasks is reduced accordingly.

Many use cases require stronger performance isolation [239]. For example, real-time applications may require a fixed share of CPU time to perform their task. Similarly, cloud providers commonly want to reserve minimum shares of CPU time to individual customers. In both cases, the scheduler must adhere to these limits despite arbitrary numbers and types of concurrently running tasks. Jones et al. propose *CPU reservations* [122] as a technique to let users declare their CPU time requirements. Their scheduler precalculates a cyclical schedule that adheres to all specified CPU reservations. Such precalculated schedules are inflexible in the presence of I/O operations, though, as individual tasks are always scheduled at a regular interval.

Instead of using precalculated schedules, many recent approaches instead implement guaranteed CPU time allocations by modifying fair scheduling. Linux CFS weights the virtual runtime of individual tasks by their priority so that tasks with a higher priority are allocated more CPU time [121]. The Xen Borrowed Virtual Time scheduler uses the same principle to implement CPU share guarantees by weighting the CPU time of individual virtual machines according to the desired CPU time shares [45]. If, for example, one virtual machine is supposed to be allocated 50% of CPU time whereas two other virtual machines shall be allocated 25% each, the CPU time of the first is effectively divided by two to calculate the virtual CPU time used for scheduling decisions. The same technique is used by the newer Xen Credit Scheduler [227] and the group scheduling provided by CFS [239]. Both not only are able to guarantee minimal CPU time allocations but also provide additional mechanisms to limit the maximum CPU usage of tasks or groups of tasks. All these approaches demonstrate the flexibility of fair scheduling to provide different performance isolation policies, which is why the approach described in the remainder of this chapter uses a similarly modified scheduler based on fair queuing. The approaches also have in common that they purely use CPU time as the underlying metric to quantify CPU throughput. They therefore completely discard the impact of CPU frequency on throughput. Tasks executed at a lower CPU frequency obtain less than their fair share of CPU time.

In power-limited systems, this is particularly problematic as frequency changes are unavoidable and cannot be controlled by the operating system. As an example, we assume a non-AVX application that requires half of the available CPU time to provide acceptable quality of service when executed at normal CPU frequencies. The approaches described above will allocate half of the CPU time to the application independently from the CPU frequency, which is sufficient if the application is scheduled alongside another non-AVX application. If the application is scheduled alongside an AVX-512 application, though, its CPU time share is not sufficient to provide acceptable quality of service due to the reduced frequency.

While there are a number of approaches that modify time-based scheduling to take frequency changes into account, none of them target power-limited systems where one application affects the performance of another application. Instead, they commonly target frequency changes issued by the operating system. For example, Dynamic Time-Slice Scaling (DTS) [120] stretches time slices to improve fairness when different applications are executed at different frequencies. Similarly, Hagimont et al. [96] propose scaling of the credits in a credit-based scheduler to implement CPU quotas that translate into fixed throughput despite frequency changes.[1] These approaches cannot mitigate the performance isolation problems caused by AVX2 and AVX-512 as they are unable to differentiate between local and remote AVX frequency reduction. On the contrary, the schedulers would increase the CPU time allocated to AVX2 and AVX-512 applications executing at low frequencies and would therefore amplify the performance isolation problem caused by these applications.

[1] Wen et al. [252] similarly modify credit-based scheduling. While their evaluation focuses on a reduction of frequency changes, their prototype likely also improves performance isolation.

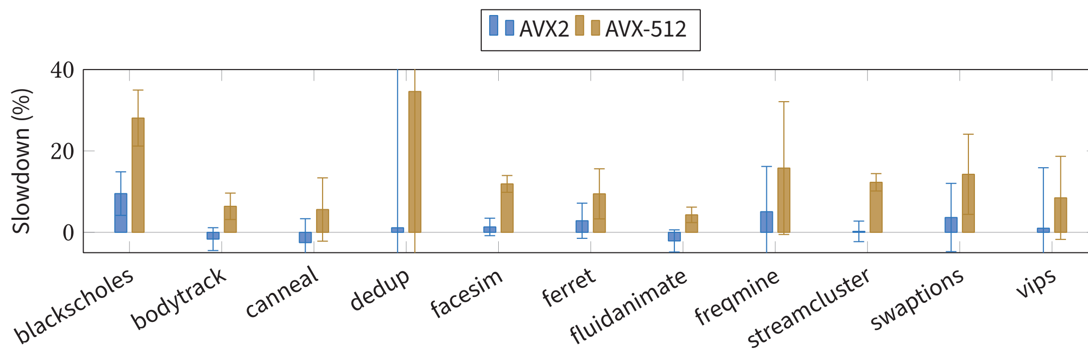


Figure 7.1: The completion time of benchmarks executed alongside x265 configured to use AVX2 or AVX-512 is generally substantially higher than if x265 uses no such power-intensive instructions. The completion time change demonstrates the lack of performance isolation between the applications.

7.2 Quantifying AVX2 and AVX-512 Performance Isolation Problems

To quantify the impact of remote AVX overhead on performance isolation in systems with existing time-based fair schedulers, we repeat the experiment from Section 3.3.5 where we execute a number of Parsec benchmarks alongside the x265 video encoder. We deviate from our earlier experiment in one key aspect as we do not measure the CPU time as a metric for performance. Instead, we measure the impact of AVX2 and AVX-512 on the completion time of the benchmarks as changes to the completion time indicate reduced performance isolation.

Figure 7.1 shows the increase of completion time when x265 uses AVX2 or AVX-512 instructions. The figure shows that AVX-512 in particular causes severely decreased performance isolation as the benchmarks are slowed down by 13.7% on average. For AVX2, the average slowdown is lower (1.7%), with some applications even experiencing increased throughput, even though our earlier experiment in Chapter 3 indicated increased CPU time. This counter-intuitive result is caused by the fact that x265 keeps fewer threads busy in parallel when using AVX2 than when using only SSE4 instructions. As a result, our experimental setup underestimates the performance change caused by remote AVX overhead.

Whereas four instances of x265 configured to use eight threads each use 24.6 logical CPUs on average when configured to use only SSE4, AVX2 reduces the average CPU utilization to 20.8 logical CPUs. As a result, the benchmark executed alongside x265 is allocated a larger share of the CPU when x265 uses AVX2, which compensates for the CPU time increase due to remote AVX overhead. The reduced scalability of x265 is likely caused by the dependencies between different operations performed by the video encoder. If the performance of some operations is increased, the corresponding thread waits longer for other threads to finish their work until it can continue.

For comparison, Figure 7.2 shows the results of an identical experiment, except for a different background task. Instead of x265, the benchmarks are executed alongside an

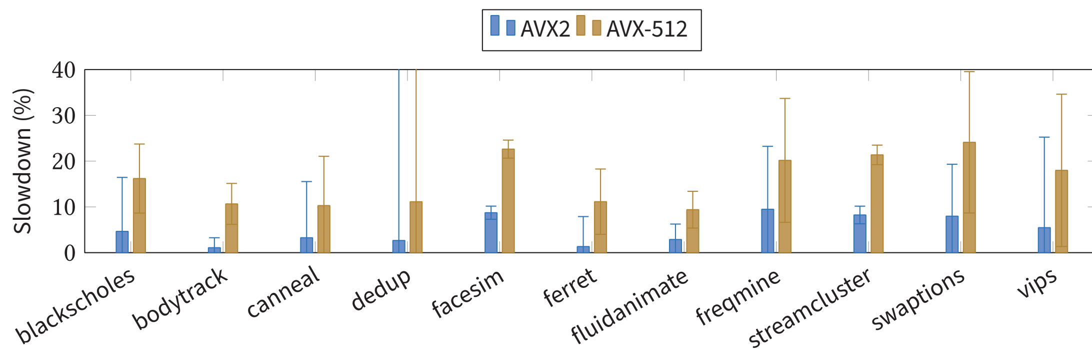


Figure 7.2: When the benchmarks are executed alongside a synthetic loop executing the specified instruction type instead of x265, its completion times consistently increase. The results in Figure 7.1 therefore underestimate the impact of remote AVX overhead on performance isolation.

application that executes instructions of the specified type in a loop and can therefore scale well to the specified number of threads. As expected, this experiment shows the completion time of all benchmarks to increase when executed alongside the AVX2 version of the application. In this setup, the benchmarks were slowed down by 5.1% on average, while the AVX-512 version of the application slowed the benchmarks down by 15.9% on average. Both results – those for x265 and those for the synthetic background application – show that AVX2 and AVX-512 instructions significantly impact performance isolation on systems with existing time-based schedulers.

7.3 Metrics for Performance Isolation

Our experiment shows that time is not suitable as the sole underlying metric for performance isolation techniques as it is only loosely correlated with throughput in power-limited systems. Instead, on these systems, scheduling also needs to take power draw into account as a task with high power draw potentially affects the CPU frequencies experienced by other tasks. In the following, we describe alternatives to time-based scheduling policies and discuss their drawbacks. We then present *overall remote performance impact* as the scheduling metric used by scheduler described in this chapter.

One potential approach for improved performance isolation in power-limited systems would be to base scheduling on energy consumption instead of CPU time utilization, because the performance impact of remote AVX overhead is accompanied by a lack of energy isolation. In particular, the power consumption of affected tasks is reduced during execution at reduced frequencies, so a scheduler that allocated a fixed share of the power budget to these tasks would mitigate the performance impact of remote AVX overhead on the tasks. Such a scheduler would increase their CPU time share at the expense of tasks responsible for remote AVX overhead.

Previous work describes a number of designs which provide scheduling based on energy consumption. For example, ECOSystem [262] and Currentcy [261] show how to implement proportional scheduling under energy constraints where tasks receive shares of the power

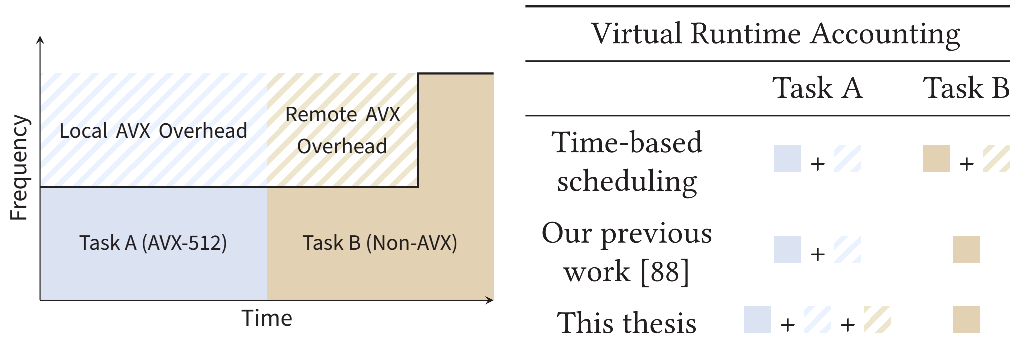


Figure 7.3: Modifications to virtual runtime accounting can improve performance isolation between AVX tasks and tasks impacted by remote AVX overhead. Our previous work [88], for example, does not bill tasks for the CPU time effectively lost due to remote AVX (represented by the hatched brown area). In this thesis, the task that caused the overhead – in this case, task A – is instead billed for this time.

budget. Bellosa et al. [20] apply this approach to power-limited systems with thermal constraints, while Cinder [208] extends the approach and introduces further mechanisms to manage energy allocations and to improve isolation between different tasks. If it was possible to determine the CPU power consumption of individual tasks with sufficient accuracy and temporal resolution, similar approaches would likely be able to solve the performance isolation problems caused by remote AVX overhead.

Task-level accounting of power consumption poses a substantial challenge, though. In particular, sensors which measure the power consumption of individual physical CPU cores are unable to differentiate between code running on sibling hyper-threads, so the operating system instead has to rely on CPU energy consumption models. We are unaware of any such model which takes the increased power draw of AVX2 and AVX-512 instructions into account. While some previous approaches to energy-based scheduling completely ignore that different programs cause different CPU power consumption [262, 208], others estimate the energy consumption using linear models based on performance counters [20]. However, there are no performance counters on current hardware that track use of AVX2 or AVX-512 instructions on a per-hyper-thread basis as shown in Section 3.4. We expect it to be impossible to construct an energy model that covers the impact of these instructions on current CPUs, making it impossible to mitigate the impact of remote AVX overhead via scheduling based on energy consumption.

As an alternative to energy-based scheduling, our own previous work [88] therefore proposed a modification of simple time-based fair scheduling to take remote AVX overhead into account. Instead of picking the task with the lowest CPU time, the proposed scheduler bases its decision on the CPU time of individual tasks minus the remote AVX overhead experienced by the tasks, i.e., it scales virtual runtimes according to remote AVX overhead. As an example, Figure 7.3 shows a context switch from an AVX-512 application (blue) to a non-AVX application (brown). In this case, the scheduler bills the AVX-512 application for its full CPU time (hatched and solid blue area), whereas the non-AVX application is not billed for the CPU time effectively lost to remote AVX overhead (hatched brown area). As

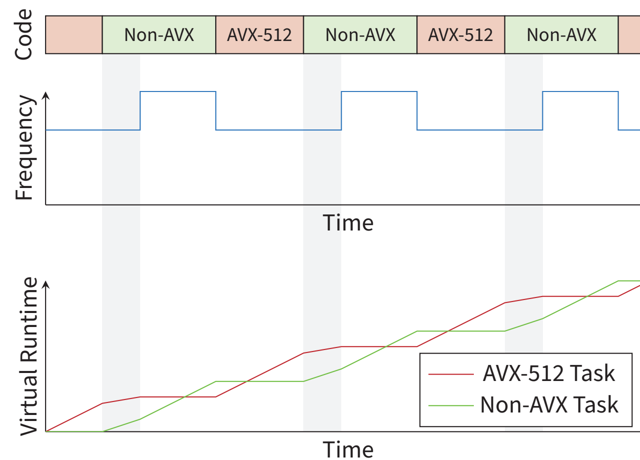


Figure 7.4: Our proposed scheduler bases its decision on overall remote performance impact. Remote AVX overhead – indicated by the shaded regions in the figure – is added to the virtual runtime of the task causing the overhead and is subtracted from the virtual runtime of the task that is slowed down. As a result, the latter task is allocated more CPU time.

a result, the non-AVX task is scheduled more frequently. While the resulting scheduler achieves a substantially fairer distribution of CPU time compared to previous time-based schedulers, it fails to provide strong performance isolation between AVX-512 and non-AVX tasks. Specifically, it fails to take into account that overall system performance is degraded by remote AVX overhead, similar to how time-based fair schedulers reduce the performance of individual tasks if additional tasks are created.

In this thesis, we propose to base scheduling on what we term the *overall remote performance impact* of applications on other applications instead, taking both CPU time usage and frequency reduction into account. We define this overall remote performance impact of an application as the performance improvement that all other applications combined would experience if the application was removed from the system.

Traditionally, if two applications are scheduled on the same core, each is allocated half the CPU time, therefore reducing the potential CPU throughput of the other application by the equivalent of 0.5 CPU cores. As Figure 7.3 shows, remote AVX overhead in particular changes this value. In the depicted situation, the effective performance impact of the AVX-512 application not only includes its own execution at the AVX-512 frequency level, represented by the solid blue area in the figure, but also both local and remote AVX overhead, shown as the hatched areas. Removing the AVX-512 application would more than double the performance of the non-AVX application as it would no longer be affected by remote AVX frequency reduction. In contrast, removing the non-AVX application would only free the brown solid area for use in the AVX-512 application, resulting in less than doubled performance for the latter application.

As depicted in Figure 7.4, a scheduler based on overall remote performance impact would therefore calculate the CPU time effectively lost to remote AVX overhead, i.e., the brown hatched area in the figure, and would subtract it from the virtual runtime of the

affected task, similar to our previous work [88]. In contrast to our previous work, the scheduler would also add the time to the virtual runtime of the task that caused the remote AVX overhead. Such a scheduler completely mitigates the effect of remote AVX overhead. If, for example, we assume a single-CPU scenario with one AVX-512 task and one non-AVX task, the scheduler will prioritize the latter task to distribute CPU time in a way that both tasks have an overall remote performance impact equivalent to 0.5 CPU cores. As a result, the throughput of the non-AVX task is exactly as if it was executed alongside a second non-AVX task.

7.4 Modified CPU Time Accounting

Implementing a fair scheduler based on overall remote performance impact instead of CPU time is conceptually simple. Existing schedulers already deviate from purely using CPU time for their scheduling decisions, instead factoring in task properties such as priority. For example, the Linux CFS scheduler maintains counters of the tasks' *virtual runtime* as described above which represent CPU time scaled by the tasks' priority [121]. During each invocation, the scheduler then selects the task with the lowest virtual runtime. For fair scheduling based on overall remote performance impact, such a scheduler merely has to be modified to select the task with the lowest accumulated remote performance impact instead. In this case, the main challenge is to estimate the remote performance impact of individual tasks.

In the following, we describe a solution to this challenge consisting of three main steps. First, the scheduler observes SIMD register usage to differentiate between tasks which potentially cause AVX frequency reduction and tasks which may suffer from remote AVX overhead (Section 7.4.1). Second, after each time slice of the latter tasks, the scheduler estimates the remote AVX frequency reduction based on performance counters and calculates the resulting performance impact (Section 7.4.2). Third, for tasks negatively affected by remote AVX overhead, the scheduler takes the overhead into account by recording less virtual runtime than was actually used by the application (Section 7.4.3). Whenever such a deduction from the virtual runtime occurs, the scheduler also adds the difference to the virtual runtime of the last task that used the corresponding SIMD registers as the remote AVX overhead contributes to the task's overall remote performance impact. In the following, we term the latter two steps *frequency reduction compensation*.

7.4.1 Attribution of Frequency Changes

To correctly modify the virtual runtimes to improve performance isolation, the scheduler needs to be able to differentiate between tasks that potentially cause remote AVX overhead and tasks that experience remote AVX overhead. We differentiate between three different classes of tasks according to the lowest possible frequency level during execution of the tasks:

1. *AVX-512 tasks* can potentially cause a reduction to the AVX-512 frequency level and are assumed to never suffer from remote AVX overhead.

2. *AVX2 tasks* can only cause a reduction to the AVX2 frequency level and may suffer from remote AVX overhead caused by AVX-512 tasks.

3. *Non-AVX tasks* can not cause any AVX frequency reduction and can therefore suffer from remote AVX overhead caused by any AVX2 or AVX-512 tasks.

As we described in the previous chapters, differentiating between the different types of tasks is non-trivial. For our scheduler, we employ the technique described in Section 6.2.3 which temporarily disables access to 512-bit registers to cause the corresponding AVX-512 to trigger exceptions. The technique involves clearing the `ZMM_Hi256` and `Hi16_ZMM` bits in the `XCR0` register [85] which disables saving and restoring 512-bit registers during context switches. To prevent data loss, the CPU consequently prevents the execution of any instruction accessing these registers. In the previous chapter, we used the resulting exceptions to identify AVX-512 tasks.

In this chapter, we slightly extend the technique as our scheduler also needs to be able to identify AVX2 tasks based on their access to 256-bit registers [88]. During each context switch, our scheduler not only clears the aforementioned bits but also the 2 of `XCR0` which controls context switching for 256-bit SIMD registers [114, p. 13-10]. As above, clearing this bit prevents the execution of 256-bit SIMD instructions. As this technique affects CPU feature detection, it requires the OS to intercept the `CPUID` instruction and accesses to the `XCR0` register as previously described in Section 6.3.3

Directly after each context switch, the newly scheduled task is regarded as a non-AVX task. Once the task executes a 256-bit or 512-bit SIMD instruction and triggers an exception, the operating system sets the `AVX` bit and marks the task as an AVX2 task. Until the next context switch, subsequent 256-bit instructions do not cause further exceptions. The next 512-bit instruction, in contrast, triggers a second exception and causes the OS to mark the task as an AVX-512 task. In response to this exception, the OS also re-enables the remaining bits in `XCR0` that were cleared during the previous context switch so that no further AVX-512 instructions cause unnecessary exceptions.

7.4.2 Estimating Remote AVX Overhead

Whenever a task operates at a lower frequency than what could have possibly been caused by the task itself, the virtual runtimes of the tasks need to be modified according to the resulting remote AVX overhead. Therefore, whenever the scheduler notices such a lower frequency during a context switch, it needs to calculate the amount of remote AVX overhead. As we described in Section 4.2, the remote AVX overhead can be expressed as the actual time required for a task divided by the ideal time, i.e., the CPU time for execution at the ideal frequency, of the same task:

$$o = \frac{t_{actual}}{t_{ideal}} = \Delta_{perf}(f_{actual,avg}, f_{ideal,avg}) \tag{7.1}$$

In this chapter, for simplicity, we instead express the remote AVX overhead as the CPU time difference Δt between t_{actual} and t_{ideal}, where t_{actual} is the length of the previous time slice:

$$\Delta t = t_{actual} - t_{ideal} = t_{actual}\left(1 - \frac{1}{\Delta_{perf}(f_{actual,avg}, f_{ideal,avg})}\right) \tag{7.2}$$

In this equation, Δ_{perf} is the model that is used to predict the performance change caused by the given frequency change. We employ the mechanistic model developed in Section 4.3.2 which splits time into memory-bound and CPU-bound phases and which assumes that only CPU-bound phases are affected by frequency changes. The model estimates the split based on two performance counters that count the L2 stall cycles and the cycles while the CPU is bound by stores. Inserting the model into Equation 7.2 and simplifying yields the following equation which the scheduler uses to calculate remote AVX overhead:

$$\begin{aligned}\Delta t &= t_{actual}\left(1 - \frac{\frac{c_{l2-stall}+c_{store-bound}}{c_{total}} * (f_{ideal,avg} - f_{actual,avg}) + f_{actual,avg}}{f_{ideal,avg}}\right) \\ &= t_{actual}\frac{(c_{total} - c_{l2-stall} - c_{store-bound})(f_{ideal,avg} - f_{actual,avg})}{c_{total}f_{ideal,avg}}\end{aligned} \tag{7.3}$$

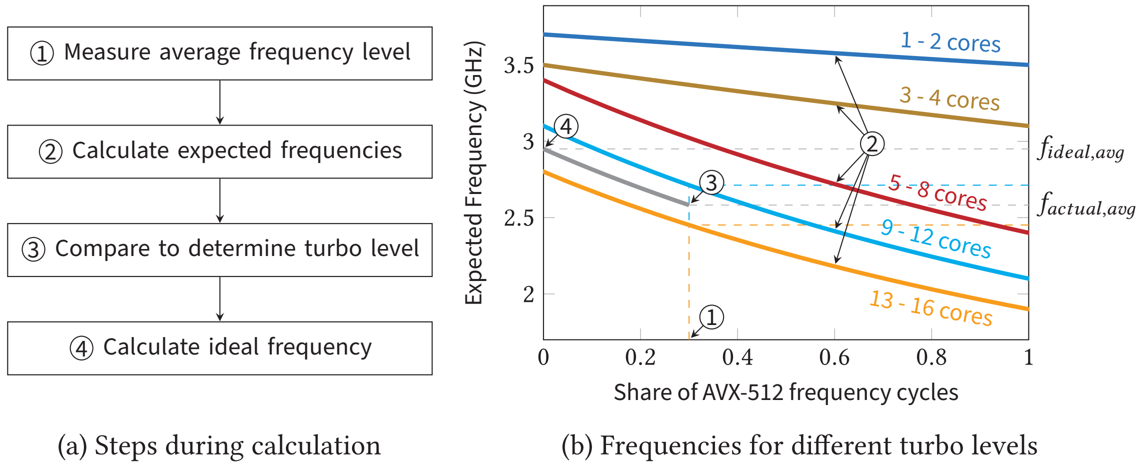
Calculating Δt requires the scheduler to know both the average actual frequency during the time slice as well as the average ideal frequency. Measuring the former frequency is trivial as the PMU provides both the number of CPU clock cycles as well as the number of cycles of a fixed-frequency reference clock as fixed-function performance counters [116, pp. 19.2 f.]. The scheduler divides the CPU clock cycles by the reference clock cycles to calculate the average actual CPU frequency.

In contrast, the ideal frequency cannot be measured directly but rather has to be calculated from the actual frequency. The difference between the ideal and actual CPU frequencies depends on a number of factors. First, it depends on the frequency levels involved – for example, a non-AVX application running at AVX-512 frequencies experiences a bigger frequency reduction than an application running at AVX2 frequencies. Second, the difference depends on the turbo level which in turn depends on the number of active CPU cores. As shown in Table 3.1 using the example of the Intel Xeon Gold 6130 CPU, larger numbers of active cores cause a larger difference between the three frequency levels.

Both these factors can change at any point between successive scheduler invocations, which complicates calculating the difference between the average actual and ideal frequencies. Only parts of the time between successive scheduler invocations may have been affected by remote AVX overhead and cores may have become active at any point in time. Therefore, in our experience, counting the number of active cores and measuring the frequency level only once during the scheduler invocation does not provide a sufficiently precise estimate of the ideal frequency. Instead, our scheduler performs the four steps shown in Figure 7.5a to continuously monitor the frequency level and to derive the average ideal frequency [88]:

① The scheduler continuously monitors the fraction of cycles spent at the three frequency levels via configurable performance counters.[2] Note that two of the four available performance counters are already in use by the DVFS performance prediction model. We therefore use the remaining two performance counters to count the

[2] Both RDMSR and RDPMC can be used to read the performance counters [115, pp. 4–533 f.]. Only the latter instruction should be used in frequently executed scheduler code due to its far lower latency, though.



(a) Steps during calculation

(b) Frequencies for different turbo levels

Figure 7.5: The ideal frequency needs to be calculated from the measured actual CPU frequency, which involves several steps. The scheduler compares the actual CPU frequency with the expected frequencies at the different turbo levels to determine the average turbo level during the time slice before using this information to calculate the corresponding ideal frequency [88].

cycles at the AVX2 (c_2) and AVX-512 (c_1) frequency levels and calculate the cycles at the non-AVX frequency level (c_0) as the difference between the total CPU cycles (c_{total}) and these two cycle counts:

$$c_0 = c_{total} - c_1 - c_2 \tag{7.4}$$

② If the cycle counts indicate that there is remote AVX overhead, the scheduler calculates the expected average frequencies for different numbers of active cores at the given distribution of cycles. In general, the average frequency f of a system that spends the times t_i at the frequency levels f_i can be calculated as follows:

$$f = \frac{1}{t_{\text{total}}} \sum t_i f_i \tag{7.5}$$

The scheduler does not measure time, though, but rather counts CPU cycles which take a varying amount of time depending on the CPU frequency. We therefore substitute $t_i = c_1/f_i$, where c_i are the cycles at the three AVX frequency levels as described above, while f_0, f_1, and f_2 are the corresponding frequencies for a given number of active cores as published by Intel [119]. After simplification, we arrive at the following equation for the expected CPU frequency at a given turbo level:

$$f(c_0, c_1, c_2) = \frac{f_0 f_1 f_2 (c_0 + c_1 + c_2)}{f_1 f_2 c_0 + f_0 f_2 c_1 + f_0 f_1 c_2} \tag{7.6}$$

Further substituting $c_0 = c_{total} - c_1 - c_2$ and as well as $r_1 = c_1/c_{\text{total}}$ and $r_2 = c_2/c_{\text{total}}$ as the share of cycles spent at the AVX2 and AVX-512 frequency levels yields the following final formula:

$$f(r_1, r_2) = \frac{f_0 f_1 f_2}{(f_0 - f_1) f_2 r_1 + (f_0 - f_2) f_1 r_2 + f_1 f_2} \tag{7.7}$$

Figure 7.5b plots this expected frequency for the different turbo levels of an Intel Xeon Gold 6130 under the assumption that $r_1 = 0$, i.e., the system only operates at either the non-AVX or the AVX-512 frequency level.

③ The scheduler then compares the measured actual CPU frequency $f_{actual,avg}$ with the expected frequencies at the different turbo levels as calculated above to identify the turbo level present during the time slice. In many cases, the measured frequency will not match any of the expected frequencies directly, especially if the number of active cores changed during the time slice. In this case, we select the closest two frequencies and calculate an interpolation factor that yields the measured frequency. This interpolation factor corresponds to the average turbo level during the time slice. In the example shown in Figure 7.5b, we assume that the system spends half of the time at the lowest turbo level, while the other half is spent at the second lowest turbo level.

④ Finally, the scheduler uses the collected information to calculate the ideal frequency during the time slice. In particular, the scheduler evaluate the function in Equation 7.7 for the detected turbo level again, but change the fraction of cycles at the AVX2 and AVX-512 frequency levels to ideal conditions without any remote AVX overhead. If, for example, an AVX2 application executed predominantly at the AVX-512 frequency level, the scheduler would calculate the frequency for $c'_0 = c_0$, $c'_1 = c_1 + c_2$, and $c'_2 = 0$, effectively assuming that all AVX-512 frequency level cycles were instead spent at the AVX2 frequency level. If, as described above, none of the turbo levels perfectly matches the measured frequency, we instead evaluate the function twice for the closest turbo levels and interpolate between the results to calculate the ideal CPU frequency. In Figure 7.5b, for example, $f_{ideal,avg}$ is calculated as the mean value of the frequencies at the lowest and the second lowest frequency level.

7.4.3 Frequency Reduction Compensation

Once both $f_{actual,avg}$ and $f_{ideal,avg}$ are known, the remote AVX overhead Δt can be calculated using Equation 7.3. The scheduler then in turn uses Δt to calculate the overall remote performance impact from the CPU time of individual tasks. In particular, as described in Section 7.3, Δt is subtracted from the virtual runtime of the task suffering from the remote AVX overhead and is added to the task that is assumed to have caused the overhead. The modification of the virtual runtime of the task affected by the overhead is performed directly during the context switch away from this task [88]. Before reinserting the task into the ready queue, the scheduler simply subtracts Δt from its virtual runtime. As the task is in a known state and local to the CPU that calculated the remote AVX overhead, direct manipulation of the task is trivial. Adding Δt to the virtual runtime of the task that caused the remote AVX overhead is more complicated as it presents two challenges:

- The first challenge is that the scheduler has to identify the tasks that cause the remote AVX overhead. Precise tracking of tasks that cause AVX2 or AVX-512 is impractical for two reasons. First, multiple tasks are often jointly responsible for remote AVX overhead if all executed AVX2 or AVX-512 instructions during the last 670 µs.

Tracking and memorizing all AVX2 and AVX-512 code phases is computationally expensive, so we instead completely attribute the remote AVX overhead to the last task that used AVX2 or AVX-512.

Second, identifying this last task is often difficult. In particular, the exceptions used to detect AVX2 or AVX-512 code only mark the beginning of AVX2 or AVX-512 phases, whereas the precise end of such phases is unknown. On a system with hyper-threading, it is therefore often not known which of the two sibling hyper-threads of a physical CPU core has most recently executed AVX2 or AVX-512 code.

We therefore use only an approximation where our scheduler attributes all remote AVX overhead to the task which most recently caused an exception due to use of AVX2 or AVX-512 on the given physical core or which was most recently descheduled after using AVX2 or AVX-512. Implementing this approximation is computationally efficient as the scheduler can simply place the current task's process ID in a per-core variable whenever the task triggers an exception by executing AVX2 code or AVX-512 code as well as whenever the scheduler is invoked following such an exception.

Even though this approximated policy may misattribute remote AVX overhead to the wrong task, we expect misattribution to have limited long-term impact on how the scheduler distributes CPU time. In particular, we expect that in most workloads all AVX2 and AVX-512 tasks to be equally affected by misattribution, so the effects on virtual runtimes cancel each other out.

- The second challenge is that the scheduler has to efficiently manipulate the virtual runtime of tasks that are not currently running. Our scheduler has to add the remote AVX overhead to the virtual runtime of the task that most likely caused the overhead, but the state of that task is unknown and direct manipulation of the virtual runtime requires expensive synchronization as the change may otherwise cause inconsistencies. In particular, for tasks placed in a sorted ready queue, the task's position in the queue needs to reflect the task's virtual runtime, so any change to the variable holding the task's virtual runtime needs to be accompanied with an update of the ready queue. As the task may have been migrated to another CPU core, locking this ready queue may limit scalability of the scheduler.

  To alleviate the need to modify the ready queue of other CPUs, we add an additional field to the task structure that holds the accumulated penalty due to remote AVX overhead caused by the task. This field is only accessed via atomic instructions to remove the need for synchronization. The scheduler only adds the accumulated penalty to the task's virtual runtime when other modifications to the virtual runtime are performed so that the existing scheduler code ensures correct placement in the ready queue.

7.5 Thread Mobility

Frequency reduction compensation as described above results in a scheduler whose virtual runtimes equal the overall remote performance impact of the tasks. In most scenarios,

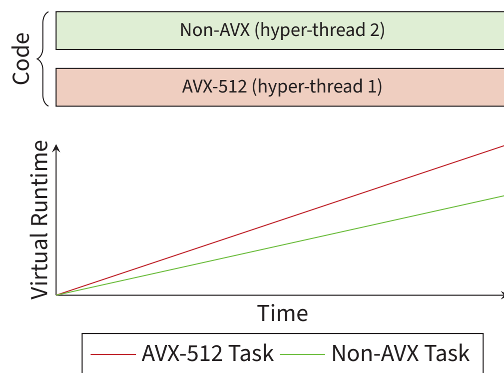


Figure 7.6: The virtual runtime of tasks slowed down by remote AVX overhead progresses at a lower rate than that of other tasks. If the scheduler is unable to choose between a sufficient number of different tasks on each logical CPU, the virtual runtime of different logical CPUs can diverge. To minimize the virtual runtime differences and to achieve good performance isolation, the scheduler has to periodically migrate tasks between different CPUs.

this scheduler achieves good performance isolation as it prioritizes tasks slowed down by remote AVX overhead over other tasks. In some situations, no such prioritization is possible. In particular, schedulers commonly use separate ready queues for individual logical CPUs to improve scalability on multi-core systems [60]. Individual ready queues may not provide the scheduler with different types of tasks to choose from. Figure 7.6 shows a situation where a physical CPU core executes only a single AVX-512 task on one hyper-thread and a single non-AVX task on the other hyper-thread, with no other runnable tasks placed in the ready-queue of the two logical CPUs. In this situation, the scheduler is forced to execute the only runnable task on each hyper-thread and is unable to provide good performance isolation. Instead, the virtual runtime of the AVX-512 task quickly outgrows the virtual runtime of the non-AVX task.

Whenever the runnable tasks outnumber the logical CPUs, this problem is easily solved by increasing the mobility of tasks across CPUs.[3] In the example given above, the scheduler could migrate a non-AVX task from another CPU core to the hyper-thread used by the AVX-512 task, thereby reducing the CPU time allocated to the latter task.

As the goal of such migrations is to equalize the virtual runtime of all tasks in the system, the scheduler should always prefer to migrate tasks with a lower virtual runtime than those already on the destination CPU. Therefore, the virtual runtimes of tasks running on different logical CPUs need to be comparable. These requirements – good thread mobility and comparable virtual runtimes – mean that the CFS scheduler is not suitable as the basis for our scheduler. In particular, as described in Section 6.3.1, CFS only infrequently performs load-balancing to migrate threads to cores with lighter load. This load balancing does not compare virtual runtimes but only takes the number of tasks on the individual

[3] The requirement that there need to be more tasks than logical CPUs does not impact the usability of our design. Instead, if the number of tasks is equal or less than the logical CPUs, a technique such as that presented in the previous chapter can be used to place AVX tasks and non-AVX tasks on separate cores, thereby eliminating remote AVX overhead.

CPUs into account. In fact, the virtual runtimes maintained by CFS are not comparable across different CPUs. Instead, the virtual runtimes of a CPU that was temporarily idle remain lower than those on a CPU that was always fully loaded. Whenever a task is migrated to a different CPU, its virtual runtime is changed to match that of other tasks at the destination CPU.[4]

As described in Section 6.3.1, other schedulers behave differently. The MuQSS scheduler is a scheduler based on virtual deadlines which targets low latency and high responsiveness. As a result, MuQSS provides mechanisms for increased thread mobility which provide a good starting point for our prototype. Most importantly, MuQSS implements much faster load balancing between different logical CPUs. Whereas CFS only periodically compares CPU load, MuQSS checks whether to migrate tasks from other CPUs at each scheduler invocation.[5] The underlying load balancing criteria of MuQSS are configurable at runtime. In its non-interactive mode, MuQSS behaves similar to CFS in that the scheduler elects to execute a task from a different CPU if the run queue of that CPU holds more tasks and if the task has a lower virtual deadline than the tasks on the current CPU. In the interactive mode, the run queue length comparison is removed – instead, the scheduler always checks all runqeues and selects the task with the lowest virtual deadline. As checking all run queues results in the behavior described above where tasks with lower virtual runtime are migrated to replace AVX2 or AVX-512 tasks, we select MuQSS in its interactive mode as the basis for our prototype.

In addition, MuQSS can be configured so that multiple logical CPUs share a single run queue [140]. In the non-interactive mode, such run queue sharing improves responsiveness as all CPUs sharing a run queue always pick the task with the earliest deadline even if that task previously executed on a different CPU. In the interactive mode, run queue sharing has little impact – instead, increased lock contention on the run queues potentially results in higher overhead. We therefore disable run queue sharing in our prototype. Attempts in our previous work to employ run queue sharing [88] did not yield any positive impact.

While, as described above, the core mechanisms of MuQSS are suitable for our prototype, we had to perform extensive further modifications to the scheduler. In particular, our experiments showed that the deadline-based policy implemented by MuQSS fails to achieve good fairness even in the absence of remote AVX overhead. MuQSS would therefore not be able to provide good performance isolation for workloads involving AVX2 or AVX-512, either. To create a scheduler that combines good fairness with fast load balancing, we modify MuQSS to use a scheduling policy very similar to the policy of CFS. In particular, we replace the virtual deadlines of MuQSS with the virtual runtimes used by CFS. Finally, we implement frequency reduction compensation as described above within the resulting fair scheduler.

7.6 Evaluation

We perform an evaluation of our prototype with a subset of the two-application workloads presented in Section 3. The main goal of our evaluation is to show that our scheduler

[4] Linux 5.9, `kernel/sched/fair.c`, lines 4160 ff.

[5] Linux 5.9 with MuQSS 0.204 [138], `kernel/sched/MuQSS.c`, lines 3844 ff.

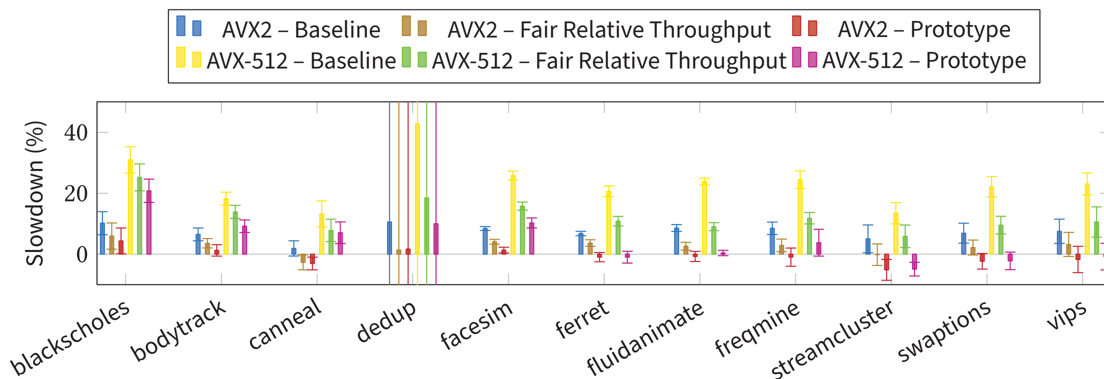


Figure 7.7: We evaluate performance isolation by measuring the slowdown caused by enabling AVX2 or AVX-512 in the x265 video encoder when x265 is executed in parallel to a benchmark application. For many benchmarks, our prototype can greatly reduce the slowdown.

provides improved performance isolation in situations with remote AVX overhead, i.e., that AVX2 or AVX-512 tasks have less influence on the performance of other tasks. Section 7.6.2 describes the result of an experiment to measure this performance isolation.

As our prototype increases the complexity of the scheduling routines, we also perform an analysis of the overhead of our scheduler compared to a similar scheduler without frequency reduction compensation in Section 7.6.3. To demonstrate that our scheduler provides competitive performance compared to CFS despite its radically different implementation based on MuQSS, we also present a comparison to CFS in Section 7.6.4.

7.6.1 Setup

All following experiments are conducted on the same system that was used for the experiments in Chapters 3 and 6. This system is equipped with an Intel Xeon Gold 6130 CPU and 24 GiB DDR4 RAM. All experiments are performed using Fedora 31 and kernels derived from Linux 5.9. In particular, our prototype is built upon MuQSS 0.204 [138] and has all available mitigations for CPU vulnerabilities enabled. As in earlier chapters, we use the Parsec 3.0 benchmark suite [25]. Unless specified otherwise, all experiments are repeated ten times.

7.6.2 Performance Isolation

To measure the impact of frequency reduction compensation on performance isolation, we execute a range of non-AVX benchmarks alongside the x265 video encoder configured to use either AVX-512, AVX2, or only SSE4 instructions. Ideally, in a scenario with ideal performance isolation, the benchmark is not slowed down when the video encoder causes remote AVX overhead. We therefore measure the slowdown of the benchmarks as the ratio between the completion time of runs with AVX2 or AVX-512 compared to the completion time of the benchmark when x265 uses SSE4. This setup differs from the experiments described in Chapter 3 where we measured CPU time. Our scheduler is supposed to

compensate for increased CPU time by prioritizing the non-AVX benchmark so that completion time remains constant.

We compare three different schedulers. The baseline for our experiments is provided by our prototype, but with frequency reduction compensation disabled. Without frequency reduction compensation, this scheduler basically implements the same scheduling algorithm also used by the ubiquitous CFS scheduler found in the mainline Linux kernel, but is based on MuQSS as described in Section 7.5. In particular, this baseline scheduler does not take remote AVX into account when calculating virtual runtimes. The second scheduler – labelled "Prototype" in Figure 7.7 – is our prototype, but with frequency reduction compensation enabled. This scheduler is designed to completely eliminate any impact of remote AVX overhead on the completion time of non-AVX tasks. This prototype differs from our earlier work which only reduced the virtual runtimes of tasks slowed down by remote AVX overhead, but which did not increase the virtual runtimes of the tasks causing the overhead [88]. To demonstrate the impact of the improvements described in this thesis, we also include a third scheduler in our comparison which consists of our prototype minus the code responsible to increase the virtual runtime of AVX2 and AVX-512 tasks. This third scheduler is labelled "fair relative throughput" in Figure 7.7. Note that this previous variant of frequency reduction compensation was not designed to achieve good performance isolation, but rather targeted a different fairness metric where the impact of remote AVX overhead was evenly spread across all tasks in the system.

Figure 7.7 shows the slowdown of our benchmarks for all three schedulers. The figure shows that our prototype provides much lower slowdown – and, consequently, much better performance isolation – than the other two schedulers. On average, the runs with the baseline scheduler show an average slowdown of 7.3% when x265 uses AVX2 and of 23.5% for AVX512. Our prototype, instead, shows only an average absolute performance difference[6] reduces this slowdown to 2.1% and 6.4%, respectively. As expected, the improvement is commonly twice as large as that provided by our previous fair scheduling algorithm which only results in a slowdown of 2.9% for AVX2 and 12.6% for AVX-512.

Not all benchmarks are equally affected by frequency reduction compensation, though. While our scheduler is able to completely mitigate the performance impact of remote AVX overhead on benchmarks such as ferret, fluidanimate, or swaptions, it is unable to ensure good performance isolation for benchmarks such as blackscholes, bodytrack, or canneal. Further analysis shows that the former group of benchmarks scales better than the latter. While all benchmarks were configured to create 32 threads, canneal, for example, only used 7 logical CPUs on average, whereas swaptions was able to use 31.3 logical CPUs. This difference shows an inherent limitation of our design – prioritization of tasks suffering from remote AVX overhead only improves performance isolation if the tasks are able to make use of the additional CPU resources.

To demonstrate that this limitation is indeed responsible for the remaining slowdown, we repeat the experiments, but limit execution to two physical cores. We configure both the benchmark and x265 to launch four threads each. Figure 7.8 shows the results of this experiment. The figure shows that performance isolation is greatly improved. While

[6] Speedup, as experienced by some of the benchmarks, is also unwanted, so we calculate an average of the absolute values.

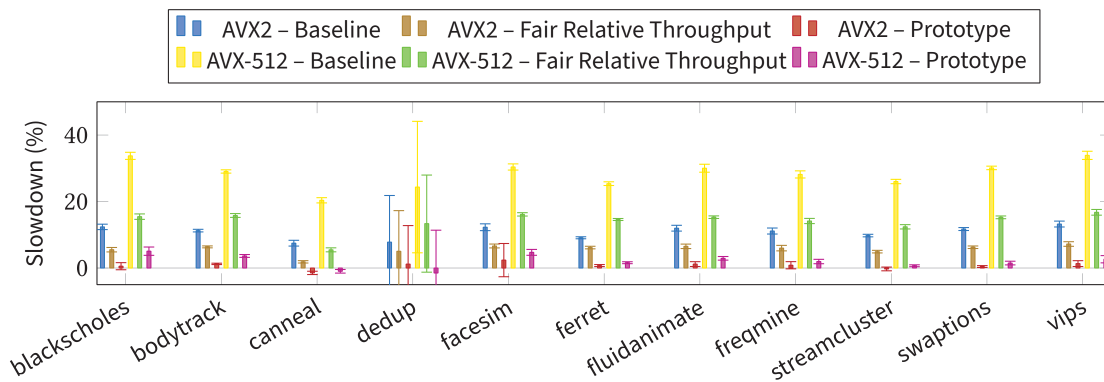


Figure 7.8: As we expect that limited scalability is responsible for the bad performance isolation experienced by some of the benchmarks in our earlier experiment, we repeat the experiment, but restrict the tasks to two physical CPU cores. In this configuration, performance isolation is greatly improved.

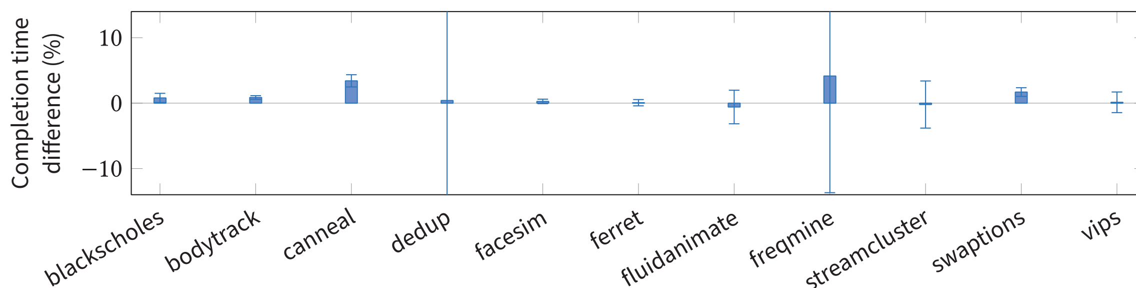


Figure 7.9: Comparing our prototype to an identical scheduler without frequency reduction compensation shows that frequency reduction compensation adds very little overhead.

AVX-512 causes an average slowdown of 28.3% for a configuration with the baseline scheduler, our prototype achieves almost complete performance isolation, with only an average slowdown of 2.5% remaining.

Finally, we repeat this previous set of experiments but replace the x265 video encoder with a synthetic program executing SSE4, AVX2 or AVX-512 instructions as described in Section 7.2, as our earlier experiments showed that the limited scalability of x265 slightly affected the results. The experiments with the synthetic background load showed similar behavior as those with x265. Frequency reduction compensation reduced the average absolute performance change caused by AVX2 from 8.7% to 1.2%, while the performance change caused by AVX-512 was reduced from 26.9% to 2.3%.

7.6.3 Overhead

The scheduler must not only provide good performance isolation, but must also provide good performance overall. Therefore, the scheduler itself must not introduce excessive overhead. There are two main reasons why the techniques described in this chapter

increase the CPU time required by the scheduler, though. First, whenever the scheduler has to evaluate the DVFS performance model and has to calculate the remote frequency reduction, the additional code increases the cost of schedule invocations. Most of this code is only executed for workloads involving AVX2 or AVX-512 tasks, though. For workloads without these instructions, the schedule only reads the two performance counters that count the cycles spent at the lower frequency levels and then concludes that no further work is necessary. Second, detecting AVX2 and AVX-512 tasks involves up to two additional exceptions between successive context switches. These exceptions also only occur during execution of AVX2 and AVX-512 tasks.

To quantify the overhead of frequency reduction compensation for both workloads with and without AVX frequency reduction, we perform experiments where we measure the completion time of applications in our prototype and compare it to execution in a baseline scheduler. As described above, this baseline scheduler is identical to our prototype, but with all code required for frequency reduction compensation removed, including the code to detect AVX2 and AVX-512 tasks. We test both non-AVX applications from the Parsec benchmark suite as well as the x265 video encoder as an example for an AVX2/AVX-512 application. We repeat the dedup benchmark 100 times to improve the accuracy of the result.

Figure 7.9 shows the overhead caused by frequency reduction calculated as the completion time increase during runs using our prototype. On average, frequency reduction compensation causes 1% overhead, with many benchmarks experiencing statistically insignificant overhead. In most cases, we expect the benefits of improved performance isolation to outweigh this overhead.

7.6.4 Comparison with CFS

The overhead experiment above uses our own fair scheduler as a reference to determine the overhead introduced by frequency reduction compensation. This scheduler is based on MuQSS, which is not representative for most Linux systems – instead, almost all Linux systems use Linux' default scheduler, CFS. CFS may provide significantly different performance characteristics, though.

To show that our prototype is a viable alternative to CFS and that the experiments above do not misrepresent the prototype's performance, we therefore also need to show that our reference scheduler provides competitive performance compared to CFS. MuQSS itself has already been shown to provide similar performance to CFS. For example, D. Shakoori Gustafsson [217] measured slightly worse Hackbench [74] performance, but slightly increased performance when compiling software, while Con Kolivas, the author of MuQSS, reported similar performance for a wider range of workloads [137]. We perform additional experiments to show that our modifications to MuQSS to achieve fair scheduling to not carry any excessive performance impact.

In particular, we repeat the overhead experiments from the previous section, but compare our reference scheduler without frequency reduction compensation to CFS. Figure 7.10 shows the result of this experiment. Our observations mirror those made about MuQSS. While there are performance differences between our scheduler and CFS, these differences are generally small. Note that while CFS experienced years of development and was likely

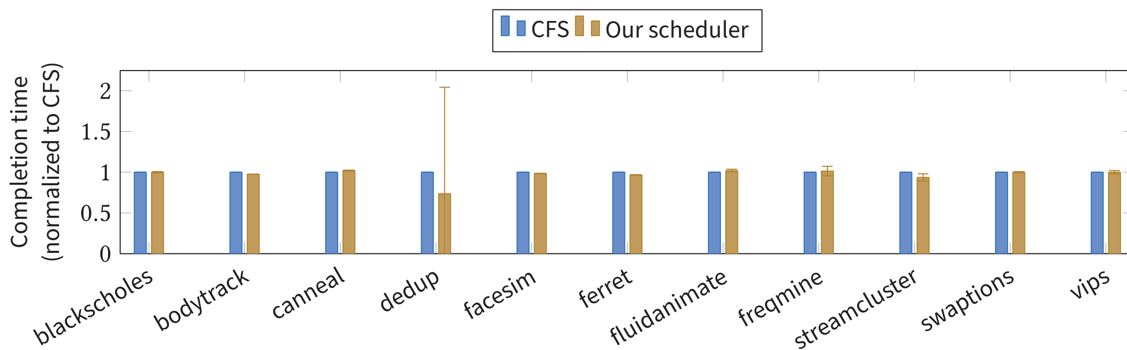


Figure 7.10: Comparing our baseline scheduler – i.e., our prototype, but without frequency reduction compensation – to CFS shows that our scheduler generally provides competitive performance.

extensively optimized, our scheduler did not experience the same treatment, so there is probably potential for further performance improvement.

7.7 Discussion

Our evaluation shows that our scheduler is able to almost completely mitigate the slowdown caused by AVX2 and AVX-512 for many benchmarks. Even for those benchmarks where significant slowdown remains, our scheduler is able to improve performance isolation. This improvement comes at very little cost – the prototype causes very little additional overhead, which demonstrates the general viability of the approach. There is still much potential for future improvements, though, as our prototype still has a number of limitations which we describe in the following.

7.7.1 Attribution of Remote AVX Overhead

One limitation of our prototype is its tendency to mischaracterize tasks. Similar to the techniques described in the previous chapter, our scheduler identifies tasks causing remote AVX overhead only based on the size of registers used by the tasks. The scheduler is not able to distinguish between heavy and light SIMD instructions, which can cause the scheduler to attribute remote AVX overhead to the wrong task. For example, applications executing only light 512-bit instructions are never compensated for any slowdown due execution at the AVX-512 frequency level, but may instead be billed for remote AVX overhead caused by other tasks executing heavy 512-bit instructions. In such situations, performance isolation is impacted.

We argue that only very specific workloads are affected by this problem, while our evaluation shows that our scheduler successfully improves performance isolation for other workloads. As detailed in Section 4.1, the CPU does not provide any better mechanism to differentiate between the different types of code. Our scheduler would therefore likely benefit from the changes to the PMU described in Section 4.6.2, where we propose perfor-

mance events that track whether the code currently executed on the CPU needs specific frequency levels.

As described in Section 3.4, the hardware does not provide precise information on which hyper-thread made most recent use of AVX2 and AVX-512, either. Instead, our scheduler bases attribution of remote AVX overhead on its information on recent exceptions and context switches. The resulting heuristic, as previously described, is often inaccurate if both hyper-threads use AVX2 or AVX-512. As a result, the scheduler may not treat multiple AVX2 or AVX-512 threads fairly and may instead bill some of the threads for more remote AVX overhead than they actually caused. We expect collecting more accurate information in hardware to be very inexpensive once support for the additional performance events described above has been added. Whenever code is detected that requires a frequency reduction, the CPU merely has to update a hardware register to indicate the corresponding hyper-thread.

7.7.2 Throttle Cycles

Another slight inaccuracy during calculation of remote AVX overhead is that our scheduler only takes cycles spent at the different frequency levels into account but ignores throttling periods that occur during transitions between the levels. In some situations, one thread may trigger a frequency change, while the following thread is slowed down by the resulting throttling. Our prototype already uses the maximum number of configurable performance counters – two for frequency level tracking, two for the DVFS performance model – and is therefore not able to additionally track throttle cycles. As the throttling periods are generally very short and will, in most cases, only affect the task causing the frequency reduction, we expect the resulting impact on performance isolation to be small. Our evaluation supports this thesis as our scheduler is able to completely mitigate the performance impact of x265 for many benchmarks.

7.7.3 Compatibility with Profiling Tools

The limited number of configurable performance counters does not only impact the calculation of the remote AVX overhead, but also impacts the compatibility between our scheduler and existing profiling tools. Our current prototype assumes exclusive access to the PMU, but even if it was designed with parallel access in mind, profilers such as perf [193] would require additional configurable performance counters to be available.

To improve compatibility between our scheduler and existing profilers, we propose adding a mechanism to temporarily free up performance counters. For example, frequency reduction compensation could be completely disabled while a profiler requires performance counters, thereby making all configurable and fixed performance counters available to the profiler. Doing so would temporarily affect performance isolation, which we expect not to be a problem in most cases. In particular, profiling is an infrequent activity that is mostly performed during software development, in which cases the performance impact caused by remote AVX overhead may be acceptable. However, such situations are an example for instrumentation perturbation as the presence of the profiler changes system behavior and affects profiler results. Instead of completely disabling frequency reduction

compensation, we therefore propose to only disable the DVFS performance model and to assume that performance is proportional to CPU frequency. Doing so frees up two of the four performance counters. At the same time, performance isolation is somewhat impacted for memory-bound workloads which do not scale less with changing CPU frequencies. Nevertheless, most of the code for frequency reduction is still active, so we expect far less instrumentation perturbation than if frequency reduction compensation is completely disabled.

7.7.4 NUMA-Support

Finally, our prototype does not support NUMA systems because we strived for keeping software complexity as low as possible and because we lacked access to suitable multi-socket systems. Task migration between different NUMA nodes is often more costly than migration within one NUMA node [43]. In particular, if the working set of a task is local to a specific NUMA node, migrating the task to a different node causes many expensive memory accesses across NUMA nodes. Our approach requires frequent load balancing which likely causes more of such expensive migrations than, for example, CFS. Future work should quantify the resulting overhead and should, if necessary, implement stricter limits to when the scheduler migrates tasks across NUMA nodes.

8 Conclusion

In recent years, CPUs have become more and more power limited to the point where performance is mainly limited by power. As different instructions require different amounts of energy, CPUs have therefore started to execute different types of code at different frequencies. Such behavior allows the CPU to execute code only consisting of simple instructions with little energy consumption at very high frequencies, fully utilizing the available power budget. Power-intensive instructions, in contrast, require the system to temporarily reduce its CPU frequency.

In some cases, this frequency reduction can potentially affect other code running on the same CPU core, thereby slowing this other code down and preventing the CPU from fully exploiting its power budget. In the case of Intel CPUs where AVX2 and AVX-512 instructions show such effects, the resulting slowdown – remote AVX overhead – is up to 30% for less power-intensive programs running in parallel with AVX-512 programs [55] and up to 10% for web servers which use AVX-512 to encrypt or decrypt data [141]. Future CPUs are projected to remain power-limited, so the same effects are likely also observed in many additional future CPU microarchitectures.

In this thesis, we showed that the impact of remote AVX overhead can often be mitigated. Specifically, we described a set of tools usable in scenarios with remote AVX overhead. We presented a profiler that is able to quantify remote AVX overhead with an average error of 2.2 percentage points by temporarily pausing individual CPU cores and analyzing the resulting frequency changes. We then showed that improved frequency scaling policies implemented by the CPU itself can almost completely eliminate remote AVX overhead for many workloads executed on systems without any hardware multi-threading such as Intel hyper-threading. However, such improved frequency scaling policies are not available on current CPUs and, more importantly, do not provide any substantial performance advantage on systems with hardware multi-threading. We therefore also showed how remote AVX overhead can be reduced in software by limiting co-scheduling of AVX-512 and non-AVX-512 tasks and restricting the former tasks to a small set of AVX-512 cores. This software approach can reduce the performance impact of remote AVX overhead by 90.4% on average for a wide range of workloads, but again suffers from limitations imposed by existing CPUs – in particular, it is only applicable to AVX-512, but not AVX2, and it is unable to differentiate between different types of AVX-512 instructions. While we therefore deem it impractical to completely mitigate remote AVX overhead on current systems, it is possible to at least prevent it from having a substantial impact on performance isolation. To this end, we presented a modification to existing time-based fair schedulers which prioritizes tasks affected by remote frequency reduction in order to counteract their slowdown due to remote AVX overhead. This scheduler reduces the performance impact of an AVX2 application on parallel non-AVX applications from an average 7.3%

slowdown down to a 2.1% slowdown, with similar improvements achieved for workloads with AVX-512 applications.

The evaluation of our prototypes demonstrates their viability, but also leads to a number of additional conclusions. Most importantly, our experiments show that the frequency reduction caused by power-intensive code should be at least partially managed by software. In particular, given that the OS has the most complete information on which tasks are affected by the resulting slowdown and has the most control over scheduling, we argue that the OS should be more involved in frequency management for power-limited systems. Our experiments also uncover a range of limitations of the hardware that limit the effectiveness of our designs. We therefore sketch a number of improved hardware-software interfaces for future CPU designs that provide the OS with more information on currently executed code as well as more control over frequency changes.

8.1 Future Work

While this thesis presents a comprehensive work on AVX frequency effects and as such answers a wide range of associated research questions, it also uncovers a number of questions that remain unanswered. Most of these questions stem from limitations of our work – we discuss these limitations in the previous chapters (see Sections 4.6, 5.4, 6.6, and 7.7). Many of these limitations are the result of deliberate efforts to reduce the complexity of our prototypes and future work is likely to achieve improved results with more refined approaches. In addition, we demonstrate how the interfaces provided by existing CPUs are inadequate for our work and how improved CPUs likely allow for better mitigation of remote AVX overhead. We propose that future work should experimentally implement improved hardware-software interfaces to determine the potential for performance improvements.

Finally, the limited scope of our work leaves a substantial related research area for future work. Specifically, this thesis only covers a specific type of power-intensive accelerators. The SIMD units of AVX2 and AVX-512 are tightly coupled accelerators [51] directly connected to individual CPU cores, whereas other accelerators such as GPUs are commonly implemented as loosely coupled accelerators connected to buses external to the CPU cores. Such accelerators potentially cause similar problems as power-intensive closely coupled accelerators, yet require different approaches.

For example, many recent CPUs for desktop or mobile systems have an integrated GPU and employ techniques to dynamically allocate parts of the power budget to CPU cores and the GPU [34]. If the GPU of these systems is idle, the full power budget is available to the CPU cores, while the CPU cores reduce their frequency when an application uses the GPU. Such power management can lead to performance isolation problems similar to those covered in Chapter 7. Future work should test whether scheduling based on energy or schedulers that deprioritize the CPU portion of GPU applications similar to our approach described in Chapter 7 can improve performance isolation.

Bibliography

[1] *[GLIBC] Fix glibc AVX frequency problems.* `https://bugs.launchpad.net/linux/+bug/1727136`.

[2] Karim Abbas. *Handbook of Digital CMOS Technology, Circuits, and Systems.* Springer Nature, 2020.

[3] Ole Agesen et al. “The Evolution of an x86 Virtual Machine Monitor”. In: *ACM SIGOPS Operating Systems Review* 44.4 (2010), pp. 3–18.

[4] Paul Alcorn. *Intel Unveils New Xeon Roadmap, E-Cores Coming to the Data Center.* `https://www.tomshardware.com/news/intel-unveils-new-xeon-roadmap-brings-e-cores-to-the-data-center`. Feb. 2022.

[5] *AMD Ryzen™ Technology: Precision Boost 2 Performance Enhancement.* `https://www.amd.com/en/support/kb/faq/cpu-pb2`.

[6] Alexandre Aminot et al. “Floating Point Units Efficiency in Multi-Core Processors”. In: *Proceedings of the 28th International Conference on the Architecture of Computing Systems (ARCS 2015).* VDE. 2015, pp. 1–8.

[7] Glenn Ammons, Thomas Ball, and James R. Larus. “Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling”. In: *ACM Sigplan Notices* 32.5 (1997), pp. 85–96.

[8] Jennifer M. Anderson et al. “Continuous Profiling: Where Have All the Cycles Gone?” In: *ACM Transactions on Computer Systems (TOCS)* 15.4 (1997), pp. 357–390.

[9] Murali Annavaram, Edward Grochowski, and John Shen. “Mitigating Amdahl’s Law Through EPI Throttling”. In: *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA’05).* IEEE. 2005, pp. 298–309.

[10] Ziya Aral and Ilya Gertner. “Non-Intrusive and Interactive Profiling in Parasight”. In: *ACM Sigplan Notices* 23.9 (1988), pp. 21–30.

[11] *Arm DynamIQ Shared Unit-110 Technical Reference Manual – Cluster configurations.* `https://developer.arm.com/documentation/102639/0201/The-DynamIQ-Shared-Unit-110/Cluster-configurations`. ARM.

[12] Muhammad Ali Awan and Stefan M. Petters. “Race-to-halt energy saving strategies”. In: *Journal of Systems Architecture* 60.10 (2014), pp. 796–815.

[13] Saisanthosh Balakrishnan et al. “The Impact of Performance Asymmetry in Emerging Multicore Architectures”. In: *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA’05).* IEEE. 2005, pp. 506–517.

[14] Thomas Ball and James R. Larus. "Optimally Profiling and Tracing Programs". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16.4 (1994), pp. 1319–1360.

[15] Antonio Barbalace et al. "Popcorn: Bridging the Programmability Gap in Heterogeneous-ISA Platforms". In: *Proceedings of the Tenth European Conference on Computer Systems (EuroSys'15)*. 2015, pp. 1–16.

[16] John Bardeen and Walter Hauser Brattain. "The Transistor, A Semi-Conductor Triode". In: *Physical Review* 74.2 (1948), p. 230.

[17] Adrian Barredo et al. "Efficiency Analysis of Modern Vector Architectures: Vector ALU Sizes, Core Counts and Clock Frequencies". In: *The Journal of Supercomputing* (2019), pp. 1960–1979.

[18] David M. Beazley, Brian D. Ward, and Ian R. Cooke. "The Inside Story on Shared Libraries and Dynamic Loading". In: *Computing in Science & Engineering* 3.5 (2001), pp. 90–97.

[19] Adam Belay et al. "Dune: Safe User-level Access to Privileged CPU Features". In: *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association. 2012, pp. 335–348.

[20] Frank Bellosa et al. "Event-Driven Energy Accounting for Dynamic Thermal Management". In: *Proceedings of the Workshop on Compilers and Operating Systems for Low Power (COLP'03)*. Vol. 22. 2003.

[21] *benchdnn – OneDNN 2.3.2*. `https://github.com/oneapi-src/oneDNN/blob/v2.3.2/tests/benchdnn/README.md`.

[22] Luca Benini, Alessandro Bogliolo, and Giovanni De Micheli. "A Survey of Design Techniques for System-Level Dynamic Power Management". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 8.3 (2000), pp. 299–316.

[23] Malini K. Bhandaru and Eric J. DeHaemer. *Providing energy efficient turbo operation of a processor*. US Patent 9,354,689. May 2016.

[24] Sharath K. Bhat et al. "Harnessing Energy Efficiency of Heterogeneous-ISA Platforms". In: *ACM SIGOPS Operating Systems Review* 49.2 (2016), pp. 65–69.

[25] Christian Bienia. "Benchmarking Modern Multiprocessors". Ph.D. Thesis. Princeton University, Jan. 2011.

[26] Christian Bienia et al. "The PARSEC Benchmark Suite: Characterization and Architectural Implications". In: *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT'08)*. 2008, pp. 72–81.

[27] Nathan Binkert et al. "The gem5 Simulator". In: *ACM SIGARCH Computer Architecture News* 39.2 (2011), pp. 1–7.

[28] Emily Blem, Jaikrishnan Menon, and Karthikeyan Sankaralingam. "Power Struggles: Revisiting the RISC vs. CISC Debate on Contemporary ARM and x86 Architectures". In: *Proceedings of the 19th International Symposium on High Performance Computer Architecture (HPCA'13)*. IEEE. 2013, pp. 1–12.

[29] Nadav Bonen et al. *Performing local power gating in a processor.* US Patent 9,772,674. Sept. 2017.

[30] Shekhar Borkar. “Low Power Design Challenges for the Decade”. In: *Proceedings of the 2001 Asia and South Pacific Design Automation Conference (ASP-DAC’01).* 2001, pp. 293–296.

[31] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel: From I/O Ports to Process Management.* 3rd ed. O’Reilly Media, Inc., 2005.

[32] Keith A. Bowman, Steven G. Duvall, and James D. Meindl. “Impact of Die-to-Die and Within-Die Parameter Fluctuations on the Maximum Clock Frequency Distribution for Gigascale Integration”. In: *IEEE Journal of Solid-State Circuits* 37.2 (2002), pp. 183–190.

[33] Keith A. Bowman et al. “A 16 nm All-Digital Auto-Calibrating Adaptive Clock Distribution for Supply Voltage Droop Tolerance Across a Wide Operating Range”. In: *IEEE Journal of Solid-State Circuits* 51.1 (2015), pp. 8–17.

[34] Alexander Branover, Denis Foley, and Maurice Steinman. “Amd Fusion APU: Llano”. In: *IEEE Micro* 32.2 (2012), pp. 28–37.

[35] Peter Brantsch. “Core Specialization for AVX-512 Using Fault-and-Migrate”. Master’s Thesis. Operating Systems Group, Karlsruhe Institute of Technology (KIT), Germany, July 2019.

[36] Ferdinand Braun. “Ueber die Stromleitung durch Schwefelmetalle”. In: *Annalen der Physik* 229.12 (1875), pp. 556–563.

[37] Dominik Brodowski et al. “CPU frequency and voltage scaling code in the Linux (tm) kernel”. In: *Linux kernel documentation* (2013). `https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt`, p. 66.

[38] John Bruno et al. “Disk Scheduling with Quality of Service Guarantees”. In: *Proceedings of the 1999 IEEE International Conference on Multimedia Computing and Systems (ICMCS’99).* Vol. 2. IEEE. 1999, pp. 400–405.

[39] James R. Bulpin and Ian A. Pratt. “Multiprogramming Performance of the Pentium 4 with Hyper-Threading”. In: *Second Annual Workshop on Duplicating, Deconstruction and Debunking (WDDD).* 2004, p. 53.

[40] *C-States – ACPI.* `https://en.wikichip.org/wiki/acpi/c-states`.

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

[42] Timothy J. Chainer et al. “Improving Data Center Energy Efficiency With Advanced Thermal Management”. In: *IEEE Transactions on Components, Packaging and Manufacturing Technology* 7.8 (2017), pp. 1228–1239.

[43] Rohit Chandra et al. “Scheduling and Page Migration for Multiprocessor Compute Servers”. In: *ACM SIGOPS Operating Systems Review* 28.5 (1994), pp. 12–24.

[44] Anantha P. Chandrakasan, Samuel Sheng, and Robert W. Brodersen. “Low-Power CMOS Digital Design”. In: *IEICE Transactions on Electronics* 75.4 (1992), pp. 371–382.

[45] Ludmila Cherkasova, Diwaker Gupta, and Amin Vahdat. “Comparison of the Three CPU Schedulers in Xen”. In: *ACM SIGMETRICS Performance Evaluation Review* 35.2 (2007), pp. 42–51.

[46] Minki Cho et al. “Postsilicon Voltage Guard-Band Reduction in a 22 nm Graphics Execution Core Using Adaptive Voltage Scaling and Dynamic Power Gating”. In: *IEEE Journal of Solid-State Circuits* 52.1 (2016), pp. 50–63.

[47] George Chrysos. “Intel® Xeon Phi coprocessor (codename Knights Corner)”. In: *Hot Chips 24 Symposium (HCS)*. IEEE. 2012, pp. 1–31.

[48] James Cipar et al. “Solving the straggler problem with bounded staleness”. In: *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems (HotOS’13)*. USENIX Association, 2013.

[49] Lucian Codrescu et al. “Hexagon DSP: An Architecture Optimized for Mobile Multimedia and Communications”. In: *IEEE Micro* 34.2 (2014), pp. 34–43.

[50] Edward G. Coffman Jr. and Leonard Kleinrock. “Computer scheduling methods and their countermeasures”. In: *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference (AFIPS’68 (Spring))*. 1968, pp. 11–21.

[51] Katherine Compton, Scott Hauck, et al. “An Introduction to Reconfigurable Computing”. In: *IEEE Computer* 9 (2000).

[52] Theofanis Constantinou et al. “Performance Implications of Single Thread Migration on a Chip Multi-Core”. In: *ACM SIGARCH Computer Architecture News* 33.4 (2005), pp. 80–91.

[53] Jonathan Corbet. *Core scheduling*. `https://lwn.net/Articles/780703/`. Feb. 2019.

[54] Jonathan Corbet. *Core scheduling lands in 5.14*. `https://lwn.net/Articles/861251/`. July 2021.

[55] Jonathan Corbet. *Many uses for Core scheduling*. `https://lwn.net/Articles/799454/`. Sept. 2019.

[56] *Core Scheduling*. `https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/core-scheduling.html`.

[57] Marius Cornea. “Intel® AVX-512 Instructions and Their Use in the Implementation of Math Functions”. In: *Intel Corporation* (2015).

[58] Ian Cutress. *Intel’s 10nm Cannon Lake and Core i3-8121U Deep Dive Review*. `https://www.anandtech.com/show/13405/intel-10nm-cannon-lake-and-core-i3-8121u-deep-dive-review`. Jan. 2019.

[59] Ian Cutress. *The AMD 2nd Gen Ryzen Deep Dive: The 2700X, 2700, 2600X, and 2600 Tested*. `https://www.anandtech.com/show/12625/amd-second-generation-ryzen-7-2700x-2700-ryzen-5-2600x-2600`. Apr. 2018.

[60] Sivarama P. Dandamudi. “Reducing Run Queue Contention in Shared Memory Multiprocessors”. In: *Computer* 30.3 (1997), pp. 82–89.

[61] Andrew Danowitz et al. “CPU DB: Recording Microprocessor History”. In: *Communications of the ACM* 55.4 (2012), pp. 55–63.

[62] Jeffrey Dean and Luiz André Barroso. “The Tail at Scale”. In: *Communications of the ACM* 56.2 (2013), pp. 74–80.

[63] Alan Demers, Srinivasan Keshav, and Scott Shenker. “Analysis and Simulation of a Fair Queueing Algorithm”. In: *ACM SIGCOMM Computer Communication Review* 19.4 (1989), pp. 1–12.

[64] Robert H. Dennard et al. “Design of Ion-Implanted MOSFET’s with Very Small Physical Dimensions”. In: *IEEE Journal of Solid-State Circuits* 9.5 (1974), pp. 256–268.

[65] Jay Desai. *Barrier Synchronization in Threads*. https://medium.com/@jaydesai36/barrier-synchronization-in-threads-3c56f947047. June 2020.

[66] Gaurav Dhiman and Tajana Simunic Rosing. “Dynamic Power Management Using Machine Learning”. In: *Proceedings of the 2006 IEEE/ACM International Conference on Computer-Aided Design (ICCAD’06)*. 2006, pp. 747–754.

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

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

[69] Markus Dreseler et al. “Fused Table Scans: Combining AVX-512 and JIT to Double the Performance of Multi-Predicate Scans”. In: *2018 IEEE 34th International Conference on Data Engineering Workshops (ICDEW)*. IEEE. 2018, pp. 102–109.

[70] *Dual-Core Intel® Xeon® Processor 5100 Series Datasheet*. Intel Corporation, Aug. 2007.

[71] Hadi Esmaeilzadeh et al. “Dark Silicon and the End of Multicore Scaling”. In: *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA’11)*. IEEE. 2011, pp. 365–376.

[72] Stijn Eyerman and Lieven Eeckhout. “A Counter Architecture for Online DVFS Profitability Estimation”. In: *IEEE Transactions on Computers* 59.11 (2010), pp. 1576–1583.

[73] Krisztián Flautner, Steve Reinhardt, and Trevor Mudge. “Automatic Performance Setting for Dynamic Voltage Scaling”. In: *Wireless networks* 8.5 (2002), pp. 507–520.

[74] Matt Fleming. *A survey of scheduler benchmarks*. https://lwn.net/Articles/725238/. June 2017.

[75] Andrei Frumusanu. *Intel 3rd Gen Xeon Scalable (Ice Lake SP) Review: Generationally Big, Competitively Small*. https://www.anandtech.com/show/16594/intel-3rd-gen-xeon-scalable-review. Apr. 2021.

[76] Johan de Gelas. *Intel Xeon E5 Version 3: Up to 18 Haswell EP Cores*. https://www.anandtech.com/show/8423/intel-xeon-e5-version-3-up-to-18-haswell-ep-cores-. Sept. 2014.

[77] Johan de Gelas. *The Intel Xeon E5 v4 Review: Testing Broadwell-EP With Demanding Server Workloads*. https://www.anandtech.com/show/10158/the-intel-xeon-e5-v4-review. Mar. 2016.

[78] Patrick P Gelsinger. "Microprocessors for the New Millennium: Challenges, Opportunities, and New Frontiers". In: *2001 IEEE International Solid-State Circuits Conference (ISSCC). Digest of Technical Papers*. IEEE. 2001, pp. 22–25.

[79] Alex Gendler, Ernest Knoll, and Yiannakis Sazeides. "I-DVFS: Instantaneous Frequency Switch During Dynamic Voltage and Frequency Scaling". In: *IEEE Micro* 41.5 (2021), pp. 76–84.

[80] Fabian Giesen. *avx_sigh.md – why doesn't radfft support AVX on PC?* https://gist.github.com/rygorous/32bc3ea8301dba09358fd2c64e02d774. May 2018.

[81] Martin Goll and Shay Gueron. "Vectorization of Poly1305 Message Authentication Code". In: *Proceedings of the 12th International Conference on Information Technology: New Generations (ITNG'15)*. IEEE. 2015, pp. 145–150.

[82] Martin Goll and Shay Gueron. "Vectorization on ChaCha Stream Cipher". In: *Proceedings of the 11th International Conference on Information Technology: New Generations (ITNG'14)*. IEEE. 2014, pp. 612–615.

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

[84] Mathias Gottschlag and Frank Bellosa. "Reducing AVX-Induced Frequency Variation With Core Specialization". In: *The 9th Workshop on Systems for Multi-core and Heterogeneous Architectures(SFMA)*. Dresden, Germany, Mar. 2019.

[85] Mathias Gottschlag, Peter Brantsch, and Frank Bellosa. "Automatic Core Specialization for AVX-512 Applications". In: *Proceedings of the 13th ACM International Systems and Storage Conference (SYSTOR'20)*. 2020, pp. 25–35.

[86] Mathias Gottschlag, Yussuf Khalil, and Frank Bellosa. *Dim Silicon and the Case for Improved DVFS Policies*. 2020. arXiv: 2005.01498 [cs.OS].

[87] Mathias Gottschlag, Tim Schmidt, and Frank Bellosa. "AVX Overhead Profiling: How Much Does Your Fast Code Slow You Down?" In: *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys'20)*. 2020, pp. 59–66.

[88] Mathias Gottschlag et al. "Fair Scheduling for AVX2 and AVX-512 Workloads". In: *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC '21)*. USENIX Association. 2021, pp. 745–758.

[89] Redha Gouicem et al. "Fewer Cores, More Hertz: Leveraging High-Frequency Cores in the OS Scheduler for Improved Application Performance". In: *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association. 2020, pp. 435–448.

[90] Peter Greenhalgh. *Big.LITTLE processing with ARM Cortex-A15 & Cortex-A7*. https://www.eetimes.com/big-little-processing-with-arm-cortex-a15-cortex-a7/. Oct. 2011.

[91] Aaron Grenat et al. "Increasing the Performance of a 28nm x86-64 Microprocessor Through System Power Management". In: *Proceedings of the 2016 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE. 2016, pp. 74–75.

[92] Ed Grochowski et al. "Best of Both Latency and Throughput". In: *IEEE International Conference on Computer Design: VLSI in Computers and Processors, 2004. ICCD 2004. Proceedings*. IEEE. 2004, pp. 236–243.

[93] Amina Guermouche and Anne-Cécile Orgerie. "Experimental analysis of vectorized instructions impact on energy and power consumption under thermal design power constraints". In: (2019). Research Report.

[94] Vishakha Gupta et al. "Kinship: Efficient Resource Management for Performance and Functionally Asymmetric Platforms". In: *Proceedings of the ACM International Conference on Computing Frontiers*. ACM. 2013, p. 16.

[95] Daniel Hackenberg et al. "An Energy Efficiency Feature Survey of the Intel Haswell Processor". In: *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW)*. IEEE. 2015, pp. 896–904.

[96] Daniel Hagimont et al. "DVFS Aware CPU Credit Enforcement in a Virtualized System". In: *Proceedings of the ACM/IFIP/USENIX 14th International Middleware Conference (Middleware 2013)*. Springer. 2013, pp. 123–142.

[97] Per Hammarlund et al. "Haswell: The Fourth-Generation Intel Core Processor". In: *IEEE Micro* 34.2 (2014), pp. 6–20.

[98] Jie Han and Michael Orshansky. "Approximate Computing: An Emerging Paradigm For Energy-Efficient Design". In: *2013 18th IEEE European Test Symposium (ETS)*. IEEE. 2013, pp. 1–6.

[99] Richard A. Hankins et al. "Multiple Instruction Stream Processor". In: *Proceedings of the 33rd International Symposium on Computer Architecture (ISCA'06)*. IEEE. 2006, pp. 114–127.

[100] Nikos Hardavellas et al. "Toward Dark Silicon in Servers". In: *IEEE Micro* 31.4 (2011), pp. 6–15.

[101] Ranjan Hebbar SR and Aleksandar Milenković. "Impact of Thread and Frequency Scaling on Performance and Energy Efficiency: An Evaluation of Core i7-8700K Using SPEC CPU2017". In: *2019 SoutheastCon*. IEEE. 2019, pp. 1–7.

[102] Ranjan Hebbar SR and Aleksandar Milenković. "SPEC CPU2017: Performance, Event, and Energy Characterization on the Core i7-8700K". In: *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*. 2019, pp. 111–118.

[103] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. 5th ed. Morgan Kaufmann, 2012. ISBN: 9780123838728.

[104] Seongmoo Heo, Kenneth Barr, and Krste Asanović. "Reducing Power Density Through Activity Migration". In: *Proceedings of the 2003 International Symposium on Low Power Electronics and Design*. 2003, pp. 217–222.

[105] Gael Hofemeier and Robert Chesebrough. “Introduction to Intel AES-NI and Intel Secure Key Instructions”. In: *Intel, White Paper* (2012).

[106] H. Peter Hofstee. “Power Efficient Processor Architecture and The Cell Processor”. In: *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA’05).* IEEE. 2005, pp. 258–262.

[107] Mark Horowitz, Thomas Indermaur, and Ricardo Gonzalez. “Low-Power Digital Design”. In: *Proceedings of 1994 IEEE Symposium on Low Power Electronics.* IEEE. 1994, pp. 8–11.

[108] Chung-Hsing Hsu and Ulrich Kremer. “The Design, Implementation, and Evaluation of a Compiler Algorithm for CPU Energy Reduction”. In: *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation.* 2003, pp. 38–48.

[109] Song Huang, Shucai Xiao, and Wu-chun Feng. “On the Energy Efficiency of Graphics Processing Units for Scientific Computing”. In: *Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing (IPDPS’09).* IEEE. 2009, pp. 1–8.

[110] Wei Huang et al. “Scaling with Design Constraints: Predicting the Future of Big Chips”. In: *IEEE Micro* 31.4 (2011), pp. 16–29.

[111] Thomas Ilsche. “Energy Measurements of High Performance Computing Systems: From Instrumentation to Analysis”. Ph.D. Thesis. TU Dresden, July 2020.

[112] Hiroshi Inoue. “How SIMD Width Affects Energy Efficiency: A Case Study on Sorting”. In: *2016 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS XIX).* IEEE. 2016, pp. 1–3.

[113] *Intel® 64 and IA-32 Architectures Optimization Reference Manual.* Intel Corporation, Sept. 2019.

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

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

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

[117] *Intel® Turbo Boost Technology in Intel® Core™ Microarchitecture (Nehalem) Based Processors.* Tech. rep. `http://web.archive.org/web/20081221055920/http://download.intel.com/design/processor/applnots/320354.pdf?iid=tech_tb+paper`. Nov. 2008.

[118] *Intel® Xeon® Processor E5 v3 Product Family – Specification Update.* Intel Corporation, Feb. 2016.

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

[120] Gangyong Jia et al. "DTS: Using Dynamic Time-slice Scaling to Address the OS Problem Incurred by DVFS". In: *Proceedings of the 2012 IEEE International Conference on Cluster Computing Workshops*. IEEE. 2012, pp. 65–72.

[121] M. Jones. *Inside the Linux 2.6 Completely Fair Scheduler*. `https://lwn.net/Articles/725238/`. Sept. 2018.

[122] Michael B. Jones, Daniela Roşu, and Marcel-Cătălin Roşu. "CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities". In: *ACM SIGOPS Operating Systems Review* 31.5 (1997), pp. 198–211.

[123] Russ Joseph, David Brooks, and Margaret Martonosi. "Control Techniques to Eliminate Voltage Emergencies in High Performance Processors". In: *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture (HPCA-9 2003)*. IEEE. 2003, pp. 79–90.

[124] Jim Kahle. "The Cell Processor Architecture". In: *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*. IEEE. 2005, pp. 3–3.

[125] Manuel Kalmbach et al. *TurboCC: A Practical Frequency-Based Covert Channel with Intel Turbo Boost*. 2020. arXiv: `2007.07046 [cs.CR]`.

[126] Vijay Kiran Kalyanam et al. "A Proactive Voltage-Droop-Mitigation System in a 7nm Hexagon™ Processor". In: *2020 IEEE Symposium on VLSI Circuits*. IEEE. 2020, pp. 1–2.

[127] Anna R. Karlin et al. "Competitive Randomized Algorithms for Nonuniform Problems". In: *Algorithmica* 11.6 (1994), pp. 542–571.

[128] Harshad Kasture and Daniel Sanchez. "TailBench: A Benchmark Suite and Evaluation Methodology for Latency-Critical Applications". In: *2016 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2016, pp. 1–10.

[129] Judy Kay and Piers Lauder. "A Fair Share Scheduler". In: *Communications of the ACM* 31.1 (1988), pp. 44–55.

[130] Georgios Keramidas, Vasileios Spiliopoulos, and Stefanos Kaxiras. "Interval-Based Models for Run-Time DVFS Orchestration in SuperScalar Processors". In: *Proceedings of the 7th ACM International Conference on Computing Frontiers*. 2010, pp. 287–296.

[131] Yussuf Khalil. "Analysis and Optimization of Dynamic Voltage and Frequency Scaling for AVX Workloads Using a Software-Based Reimplementation". Bachelor's Thesis. Operating Systems Group, Karlsruhe Institute of Technology (KIT), Germany, Sept. 2019.

[132] Jack S. Kilby. "Invention of the Integrated Circuit". In: *IEEE Transactions on Electron Devices* 23.7 (1976), pp. 648–654.

[133] Nam Sung Kim et al. "Leakage Current: Moore's Law Meets Static Power". In: *Computer* 36.12 (2003), pp. 68–75.

[134] Young Duk Kim et al. “A 7nm High-Performance and Energy-Efficient Mobile Application Processor with Tri-Cluster CPUs and a Sparsity-Aware NPU”. In: *2020 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE. 2020, pp. 48–50.

[135] Donald E. Knuth. “An Empirical Study of FORTRAN Programs”. In: *Software: Practice and Experience* 1.2 (1971), pp. 105–133.

[136] Con Kolivas. *5.14 and the future of MuQSS and -ck once again*. `http://ck-hack.blogspot.com/2021/08/514-and-future-of-muqss-and-ck-once.html`. Aug. 2021.

[137] Con Kolivas. *First MuQSS Throughput Benchmarks*. `http://ck-hack.blogspot.com/2016/10/first-muqss-throughput-benchmarks.html`. Oct. 2016.

[138] Con Kolivas. *linux-5.9-ck1, MuQSS version 0.204 for linux-5.9*. `https://ck-hack.blogspot.com/2020/10/linux-59-ck1-muqss-version-0204-for.html`. Oct. 2020.

[139] Con Kolivas. *MuQSS - The Multiple Queue Skiplist Scheduler*. `http://ck.kolivas.org/patches/muqss/sched-MuQSS.txt`.

[140] Con Kolivas. *Runqueue sharing experiments with MuQSS*. `https://ck-hack.blogspot.com/2017/11/runqueue-sharing-experiments-with-muqss.html`. Nov. 2017.

[141] Vlad Krasnov. *On the dangers of Intel’s frequency scaling*. `https://blog.cloudflare.com/on-the-dangers-of-intels-frequency-scaling/`. Oct. 2017.

[142] Rakesh Kumar, Alejandro Martinez, and Antonio Gonzalez. “Efficient Power Gating of SIMD Accelerators Through Dynamic Selective Devectorization in an HW/SW Codesigned Environment”. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 11.3 (2014), p. 25.

[143] Rakesh Kumar et al. “Heterogeneous Chip Multiprocessors”. In: *Computer* 38.11 (2005), pp. 32–38.

[144] Viren Kumar and Alexandra Fedorova. “Towards Better Performance Per Watt in Virtual Environments on Asymmetric Single-ISA Multi-core Systems”. In: *ACM SIGOPS Operating Systems Review* 43.3 (2009), pp. 105–109.

[145] Julia Lawall et al. “OS Scheduling with Nest: Keeping Tasks Close Together on Warm Cores”. In: *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys’22)*. 2022, pp. 368–383.

[146] Min Lee and Karsten Schwan. “Region Scheduling: Efficiently Using the Cache Architectures via Page-level Affinity”. In: *ACM SIGARCH Computer Architecture News*. Vol. 40. 1. ACM. 2012, pp. 451–462.

[147] Sang-Jeong Lee, Hae-Kag Lee, and Pen-Chung Yew. “Runtime Performance Projection Model for Dynamic Power Management”. In: *Asia-Pacific Conference on Advances in Computer Systems Architecture*. Springer. 2007, pp. 186–197.

[148] Charles R. Lefurgy et al. “Active Management of Timing Guardband to Save Energy in POWER7”. In: *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. IEEE. 2011, pp. 1–11.

[149] Rob Lembree. *X Display Power Management Signaling (DPMS) Extension Protocol Specification*. https://www.x.org/releases/X11R7.7/doc/xextproto/dpms.html. 1996.

[150] Daniel Lemire. *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/. Aug. 2018.

[151] Daniel Lemire and Travis Downs. *AVX-512: when and how to use these new instructions*. https://lemire.me/blog/2018/09/07/avx-512-when-and-how-to-use-these-new-instructions/. Sept. 2018.

[152] Ari Lemmetti et al. "AVX2-optimized Kvazaar HEVC intra encoder". In: *2016 IEEE International Conference on Image Processing (ICIP)*. IEEE. 2016, pp. 549–553.

[153] Jingwen Leng et al. "Safe Limits on Voltage Reduction Efficiency in GPUs: a Direct Measurement Approach". In: *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-48)*. IEEE. 2015, pp. 294–307.

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

[155] Bo Li, Edgar A. León, and Kirk W. Cameron. "COS: A Parallel Performance Model for Dynamic Variations in Processor Speed, Memory Speed, and Thread Concurrency". In: *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*. 2017, pp. 155–166.

[156] Sheng Li et al. "CACTI-P: Architecture-Level Modeling for SRAM-based Structures with Advanced Leakage Reduction Techniques". In: *Proceedings of the 2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'11)*. IEEE. 2011, pp. 694–701.

[157] Tong Li et al. "Operating System Support for Overlapping-ISA Heterogeneous Multi-core Architectures". In: *Proceedings of the 16th International Symposium on High Performance Computer Architecture*. IEEE. 2010, pp. 1–12.

[158] *Libmvec – glibc wiki*. https://sourceware.org/glibc/wiki/libmvec.

[159] Ankur Limaye and Tosiron Adegbija. "A Workload Characterization of the SPEC CPU2017 Benchmark Suite". In: *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2018, pp. 149–158.

[160] *Linux Wireless – Dynamic power save*. https://wireless.wiki.kernel.org/en/users/documentation/dynamic-power-save. 2015.

[161] H. J. Lu. *Bug 21396 - Use AVX2 memcpy/memset on Skylake server*. https://sourceware.org/bugzilla/show_bug.cgi?id=21396. Apr. 2017.

[162] Huaxiang Lu et al. "SOC Dynamic Power Management Using Artificial Neural Network". In: *International Conference on Natural Computation*. Springer. 2006, pp. 555–564.

[163] Yung-Hsiang Lu et al. "Quantitative Comparison of Power Management Algorithms". In: *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition 2000 (DATE'00)*. IEEE. 2000, pp. 20–26.

[164] Philipp Machauer. "Faires Scheduling unter Beachtung von AVX-512-Frequenzeffekten". Bachelor Thesis. Operating Systems Group, Karlsruhe Institute of Technology (KIT), Germany, Sept. 2020.

[165] Allen D. Malony, Daniel A. Reed, and Harry A. G. Wijshoff. "Performance Measurement Intrusion and Perturbation Analysis". In: *IEEE Transactions on Parallel & Distributed Systems* 3.04 (1992), pp. 433–450.

[166] Giuseppe Massari et al. "Towards Fine-Grained DVFS in Embedded Multi-core CPUs". In: *Proceedings of the 31st International Conference on Architecture of Computing Systems (ARCS 2018)*. Springer. 2018, pp. 239–251.

[167] Abdelhafid Mazouz et al. "Evaluation of CPU Frequency Transition Latency". In: *Computer Science - Research and Development* 29.3-4 (2014), pp. 187–195.

[168] Chris McClanahan. *History and Evolution of GPU Architecture*. `https://mcclanahoochie.com/blog/wp-content/uploads/2011/03/gpu-hist-paper.pdf`. 2010.

[169] Rustam Miftakhutdinov, Eiman Ebrahimi, and Yale N Patt. "Predicting Performance Impact of DVFS for Realistic Memory Systems". In: *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. IEEE. 2012, pp. 155–165.

[170] Damian Miralles and Jessica Iwamoto. "Accelerating software radios by means of SIMD Instructions. A case for the AVX2 and AVX512 Extensions". In: *Proceedings of the GNU Radio Conference*. Vol. 3. 1. 2018.

[171] Jeffrey C. Mogul et al. "Using Asymmetric Single-ISA CMPs to Save Energy on Operating Systems". In: *IEEE Micro* 28.3 (2008), pp. 26–41.

[172] Daniel Molka et al. "Characterizing the Energy Consumption of Data Transfers and Arithmetic Operations on x86-64 Processors". In: *International Conference on Green Computing*. IEEE. 2010, pp. 123–133.

[173] Ingo Molnar. "Modular scheduler core and completely fair scheduler [CFS]". In: *Linux-Kernel mailing list* (Apr. 2007).

[174] G. E. Moore, C. T. Sah, and F. M. Wanlass. "Metal-Oxide-Semiconductor Field-Effect Devices for Micropower Logic Circuitry". In: *Micropower Electronics* (1964), pp. 41–55.

[175] Gordon E. Moore. *Cramming More Components onto Integrated Circuits*. 1965.

[176] Guillaume Morin. *[BUG] incorrect scaling_max_freq with intel_pstate after offline/online*. `https://www.spinics.net/lists/kernel/msg3814544.html`. linux-kernel mailing list.

[177] Vijay Nagarajan et al. "A Primer on Memory Consistency and Cache Coherence". In: *Synthesis Lectures on Computer Architecture* 15.1 (2020), pp. 1–294.

[178] Oscar Naım and Anthony J. G. Hey. “Invasiveness of Performance Instrumentation Measurements on Multiprocessors”. In: *Proceedings of the IFIP WG10.3 Working Conference on Applications in Parallel and Distributed Computing* (1994), pp. 319–328.

[179] John Nickolls. “GPU Parallel Computing Architecture and CUDA Programming Model”. In: *Hot Chips 19 Symposium (HCS)*. IEEE. 2007, pp. 1–12.

[180] Eriko Nurvitadhi et al. “Accelerating Binarized Neural Networks: Comparison of FPGA, CPU, GPU, and ASIC”. In: *2016 International Conference on Field-Programmable Technology (FPT)*. IEEE. 2016, pp. 77–84.

[181] Gerard O’Regan. *A Brief History of Computing*. 3rd ed. Springer, 2021. ISBN: 978-3-030-66599-9.

[182] Takanori Okuma, Tohru Ishihara, and Hiroto Yasuura. “Real-Time Task Scheduling for a Variable Voltage Processor”. In: *Proceedings of the 12th International Symposium on System Synthesis (ISSS’99)*. IEEE. 1999, pp. 24–29.

[183] *OPENMP API Specification: Version 5.0 November 2018 – OMP_WAIT_POLICY*. `https://www.openmp.org/spec-html/5.0/openmpse55.html`.

[184] *OpenSSL – Cryptography and SSL/TLS Toolkit*. `https://www.openssl.org/`.

[185] Santiago Pagani et al. “TSP: Thermal Safe Power - Efficient power budgeting for Many-Core Systems in Dark Silicon”. In: *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis*. 2014, pp. 1–10.

[186] Ajit Pal. *Low-Power VLSI Circuits and Systems*. Springer, 2015. ISBN: 978-81-322-1937-8.

[187] Ioannis Papamanoglou. “Constructing a Library for Mitigating AVX-Induced Performance Degradation”. Master’s Thesis. Operating Systems Group, Karlsruhe Institute of Technology (KIT), Germany, Mar. 2019.

[188] Irma Esmer Papazian. “New 3rd Gen Intel® Xeon® Scalable Processor (Codename: Ice Lake-SP)”. In: *Hot Chips 32 Symposium (HCS)*. 2020, pp. 1–22.

[189] Sangyoung Park et al. “Accurate Modeling of the Delay and Energy Overhead of Dynamic Voltage and Frequency Scaling in Modern Microprocessors”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32.5 (2013), pp. 695–708.

[190] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. 4th ed. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann, 2012. ISBN: 9780123747501.

[191] *perf-record - Run a command and record its profile into perf.data*. perf Manual – PERF-RECORD(1). Apr. 2021.

[192] *perf-sched - Tool to trace/measure scheduler properties (latencies)*. perf Manual – PERF-SCHED(1). Nov. 2021.

[193] *perf: Linux profiling with performance counters*. `https://perf.wiki.kernel.org/index.php/Main_Page`.

[194] *Phoronix Test Suite*. https://phoronix-test-suite.com/.

[195] Andy Polyakov. *crypto/x86_64cpuid.pl: suppress AVX512F flag on Skylake-X*. https://github.com/openssl/openssl/commit/79337628702dc5ff5570f02d6b92eeb02a310e18. Dec. 2017.

[196] Gerald J. Popek and Robert P. Goldberg. "Formal Requirements for Virtualizable Third Generation Architectures". In: *Communications of the ACM* 17.7 (1974), pp. 412–421.

[197] *Product Brief: Intel® Core™ X-Series Processor Family*. https://www.intel.com/content/www/us/en/products/docs/processors/core/x-series-processor-family-brief.html.

[198] Henry Qin et al. "Arachne:Core-Aware Thread Management". In: *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. USENIX Association. 2018, pp. 145–160.

[199] Arun Raghavan et al. "Computational Sprinting". In: *Proceedings of the 18th IEEE International Symposium on High Performance Computer Architecture*. IEEE. 2012, pp. 1–12.

[200] Bharathwaj Raghunathan et al. "Cherry-Picking: Exploiting Process Variations in Dark-Silicon Homogeneous Chip Multi-Processors". In: *Proceedings of the 2013 Design, Automation & Test in Europe Conference & Exhibition (DATE'13)*. IEEE. 2013, pp. 39–44.

[201] Daniel J. Ragland et al. *Processors, methods, and systems to adjust maximum clock frequencies based on instruction type*. US Patent App. 15/055,578. Aug. 2017.

[202] Vijay Janapa Reddi et al. "Voltage Smoothing: Characterizing and Mitigating Voltage Noise in Production Processors via Software-Guided Thread Scheduling". In: *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-43)*. IEEE. 2010, pp. 77–88.

[203] Dheeraj Reddy et al. "Bridging Functional Heterogeneity in Multicore Architectures". In: *ACM SIGOPS Operating Systems Review* 45.1 (2011), pp. 21–33.

[204] Nir Rosenzweig, Zeev Sperber, and Efraim Rotem. *Method and apparatus to control current transients in a processor*. US Patent 9,411,395. Aug. 2016.

[205] Efraim Rotem et al. "Alder Lake Architecture". In: *Hot Chips 33 Symposium (HCS)*. IEEE. 2021, pp. 1–23.

[206] Efraim Rotem et al. "Power-Management Architecture of the Intel Microarchitecture Code-Named Sandy Bridge". In: *IEEE Micro* 32.2 (2012), pp. 20–27.

[207] Barry Rountree et al. "Practical Performance Prediction Under Dynamic Voltage Frequency Scaling". In: *2011 International Green Computing Conference and Workshops*. IEEE. 2011, pp. 1–8.

[208] Arjun Roy et al. "Energy Management in Mobile Devices with the Cinder Operating System". In: *Proceedings of the Sixth Conference on Computer Systems (EuroSys'11)*. 2011, pp. 139–152.

[209] Juan Carlos Saez et al. "A Comprehensive Scheduler for Asymmetric Multicore Systems". In: *Proceedings of the 5th European Conference on Computer Systems (EuroSys'10)*. ACM. 2010, pp. 139–152.

[210] Subhash Saini et al. "The Impact of Hyper-Threading on Processor Resource Utilization in Production Applications". In: *2011 18th International Conference on High Performance Computing*. IEEE. 2011, pp. 1–10.

[211] Michael S. Schlansker and B. Ramakrishna Rau. "EPIC: Explicitly Parallel Instruction Computing". In: *Computer* 33.2 (2000), pp. 37–45.

[212] Tim Schmidt. *Covert Channel based on AMD Precision Boost 2*. Bachelor Thesis. Oct. 2019.

[213] Robert Schöne et al. "Energy Efficiency Features of the Intel Skylake-SP Processor and Their Impact on Performance". In: *2019 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE. 2019, pp. 399–406.

[214] Simon Schubert et al. "Profiling Software for Energy Consumption". In: *2012 IEEE International Conference on Green Computing and Communications*. IEEE. 2012, pp. 515–522.

[215] Joseph Schuchart et al. "The Shift from Processor Power Consumption to Performance Variations: Fundamental Implications at Scale". In: *Computer Science - Research and Development* 31.4 (2016), pp. 197–205.

[216] Christian Schwarz. "Stage-Aware Scheduling in a Library OS". Bachelor's Thesis. Operating Systems Group, Karlsruhe Institute of Technology (KIT), Germany, Mar. 2018.

[217] David Shakoori Gustafsson. *Linux CPU Schedulers: CFS and MuQSS Comparison*. 2021.

[218] 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 (ASP-DAC'09)*. IEEE Press. 2009, pp. 733–738.

[219] William Shockley. "The Theory of p-n Junctions in Semiconductors and p-n Junction Transistors". In: *Bell System Technical Journal* 28.3 (1949), pp. 435–489.

[220] Teja Singh et al. "Zen: A Next-Generation High-Performance x86 Core". In: *2017 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE. 2017, pp. 52–53.

[221] Magnus Själander, Margaret Martonosi, and Stefanos Kaxiras. "Power-Efficient Computer Architectures: Recent Advances". In: *Synthesis Lectures on Computer Architecture* 9.3 (2014), pp. 1–96.

[222] *Skylake (client) - Microarchitectures - Intel*. `https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client)`.

[223] *Skylake (server) - Microarchitectures - Intel*. `https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server)`.

[224] David C. Snowdon et al. "Accurate Run-Time Prediction of Performance Degradation under Frequency Scaling". In: *Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2007)*. 2007, p. 58.

[225] 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 (OSDI'10)*. USENIX Association. 2010, pp. 33–46.

[226] Avinash Sodani et al. "Knights Landing: Second-Generation Intel Xeon Phi Product". In: *IEEE Micro* 36.2 (2016), pp. 34–46.

[227] Gaurav Somani and Sanjay Chaudhary. "Application Performance Isolation in Virtualization". In: *2009 IEEE International Conference on Cloud Computing*. IEEE. 2009, pp. 41–48.

[228] Mani B. Srivastava, Anantha P. Chandrakasan, and Robert W. Brodersen. "Predictive System Shutdown and Other Architectural Techniques for Energy Efficient Programmable Computation". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 4.1 (1996), pp. 42–55.

[229] Bo Su et al. "Implementing a Leading Loads Performance Predictor on Commodity Processors". In: *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC'14)*. 2014.

[230] M. Aater Suleman et al. "Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures". In: *ACM SIGARCH Computer Architecture News* 37.1 (2009), pp. 253–264.

[231] Simon M. Tam et al. "SkyLake-SP: A 14nm 28-Core Xeon® Processor". In: *2018 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE. 2018, pp. 34–36.

[232] Rizwan Ali Tau Leng et al. "An Empirical Study of Hyper-Threading in High Performance Computing Clusters". In: *Linux HPC Revolution* 45 (2002).

[233] Michael B. Taylor. "A Landscape of the New Dark Silicon Design Regime". In: *IEEE Micro* 33.5 (2013), pp. 8–19.

[234] 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*. IEEE. 2012, pp. 1131–1136.

[235] *The /proc filesystem*. `https://www.kernel.org/doc/Documentation/filesystems/proc.txt`.

[236] Praveen Kumar Tiwari et al. *Accelerating x265 with Intel® Advanced Vector Extensions 512*. Tech. rep. Intel, May 2018.

[237] Nathan Tuck and Dean M. Tullsen. "Initial Observations of the Simultaneous Multithreading Pentium 4 Processor". In: *2003 12th International Conference on Parallel Architectures and Compilation Techniques*. IEEE. 2003, pp. 26–34.

[238] *Turbo Boost Max Technology 3.0 (TBMT) - Intel*. `https://en.wikichip.org/wiki/intel/turbo_boost_max_technology`.

[239] Paul Turner, Bharata B. Rao, and Nikhil Rao. "CPU bandwidth control for CFS". In: *Linux Symposium*. 2010, p. 245.

[240] Kimiyoshi Usami et al. "Automated Low-Power Technique Exploiting Multiple Supply Voltages Applied to a Media Processor". In: *IEEE Journal of Solid-State Circuits* 33.3 (1998), pp. 463–472.

[241] M Michael Vai. *VLSI Design*. CRC Press, 2000.

[242] Ashish Venkat and Dean M. Tullsen. "Harnessing ISA Diversity: Design of a Heterogeneous-ISA Chip Multiprocessor". In: *ACM SIGARCH Computer Architecture News* 42.3 (2014), pp. 121–132.

[243] Ganesh Venkatesh et al. "Conservation Cores: Reducing the Energy of Mature Computations". In: *ACM Sigplan Notices* 45.3 (2010), pp. 205–218.

[244] Ganesh Venkatesh et al. "QsCores: Trading Dark Silicon for Scalable Energy Efficiency with Quasi-Specific Cores". In: *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. IEEE. 2011, pp. 163–174.

[245] Nicolas Viennot. *GitHub – twosigma/libvirtcpuid: libvirtcpuid provides transparent CPUID virtualization, all in userspace*. `https://github.com/twosigma/libvirtcpuid`. Two Sigma.

[246] Jons-Tobias Wamhoff et al. "The TURBO Diaries: Application-controlled Frequency Scaling Explained". In: *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association. 2014, pp. 193–204.

[247] Zhenghong Wang and Ruby B. Lee. "Covert and Side Channels due to Processor Architecture". In: *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC'06)*. IEEE. 2006, pp. 473–482.

[248] Hiroshi Watanabe and Koh M. Nakagawa. "SIMD vectorization for the Lennard-Jones potential with AVX2 and AVX-512 instructions". In: *Computer Physics Communications* 237 (2019), pp. 1–7.

[249] Ralph O. Weber. "Information Technology–ATA/ATAPI Command Set-3 (ACS-3)". In: *Working Draft Project American National Standard* 13 (Oct. 2013).

[250] Mark Weiser et al. "Scheduling for Reduced CPU Energy". In: *Mobile Computing*. Springer, 1994, pp. 449–471.

[251] Andreas Weissel and Frank Bellosa. "Process Cruise Control: Event-Driven Clock Scaling for Dynamic Power Management". In: *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. 2002, pp. 238–246.

[252] Chengjian Wen et al. "PCFS: A Power Credit based Fair Scheduler under DVFS for Multi-core Virtualization Platform". In: *2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*. IEEE. 2010, pp. 163–170.

[253] Alberto Wiltgen et al. "Power Consumption Analysis in Static CMOS Gates". In: *2013 26th Symposium on Integrated Circuits and Systems Design (SBCCI)*. IEEE. 2013, pp. 1–6.

[254] *wrk – a HTTP benchmarking tool*. https://github.com/wg/wrk.

[255] Qiang Wu et al. “A Dynamic Compilation Framework for Controlling Microprocessor Energy and Performance”. In: *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO’05)*. IEEE. 2005, 12–pp.

[256] *x265, the free H.265/HEVC encoder*. https://www.videolan.org/developers/x265.html.

[257] Fen Xie, Margaret Martonosi, and Sharad Malik. “Compile-Time Dynamic Voltage Scaling Settings: Opportunities and Limits”. In: *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*. 2003, pp. 49–62.

[258] *Xiph.org Video Test Media [derf’s collection]*. https://media.xiph.org/video/derf/.

[259] Ahmad Yasin. “A Top-Down Method for Performance Analysis and Counters Architecture”. In: *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2014, pp. 35–44.

[260] Yi-Ping You, Chingren Lee, and Jenq Kuen Lee. “Compiler Analysis and Supports for Leakage Power Reduction on Microprocessors”. In: *15th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2002)*. Springer. 2002, pp. 45–60.

[261] Heng Zeng et al. “Currentcy: Unifying Policies for Resource Management”. In: *Proceedings of the 2003 USENIX Annual Technical Conference (USENIX ATC’03)*. USENIX Association. 2003.

[262] Heng Zeng et al. “ECOSystem: Managing Energy as a First Class Operating System Resource”. In: *ACM SIGOPS Operating Systems Review* 36.5 (2002), pp. 123–132.

[263] Gangyi Zhu, Peng Jiang, and Gagan Agrawal. “A Methodology for Characterizing Sparse Datasets and Its Application to SIMD Performance Prediction”. In: *Proceedings of the 28th International Conference on Parallel Architectures and Compilation Techniques (PACT’19)*. IEEE. 2019, pp. 445–456.

List of Figures

1.1 Remote AVX overhead 2

2.1 Synchronous sequential logic 10
2.2 Main reasons for pipeline stalls and countermeasures 11
2.3 Out-of-order processor structure 13
2.4 CMOS logic 16
2.5 Skylake-SP frequency/voltage domains 18
2.6 Voltage droop 19
2.7 Voltage droop mitigation techniques 20

3.1 Throughput, energy, and power of a single core at 3.5 GHz 35
3.2 All-core turbo frequency, IPC, and voltage during transitions to more power-intensive code 40
3.3 Single-core turbo frequency, IPC, and voltage during transitions to more power-intensive code 41
3.4 Frequency, IPC, and voltage during transitions to less power-intensive code 43
3.5 Performance of OneDNN and OpenSSL ChaCha20-Poly1305 on one core and on all cores 45
3.6 Remote frequency reduction 47
3.7 Average CPU frequencies while running OneDNN and x265 50
3.8 Remote AVX overhead caused by AVX2 and AVX-512 51
3.9 Frequency levels while running OneDNN and x265 54

4.1 Measurement of unnecessary frequency reduction 61
4.2 Impact of turbo level changes during profiler-induced pauses 62
4.3 Determining the source of remote AVX overhead 63
4.4 Impact of memory stalls on performance 65
4.5 Leading-loads performance prediction model 66
4.6 Stall cycle counting 67
4.7 Prediction accuracy of our DVFS performance model 69
4.8 Comparison between profiler output and direct measurements 72
4.9 Overhead caused by our profiler 74
4.10 Impact of profiler overhead on barrier synchronization 75
4.11 Impact of our profiler on masstree latency 75
4.12 Accuracy of overhead classification 76
4.13 Hardware-software interface to detect remote frequency reduction . . . 79

5.1 Comparison between dynamic power management and DVFS 85
5.2 Experiment to measure frequency reduction overhead 88

5.3 Overhead when reducing or increasing the frequency 89
5.4 Experiment to measure the overhead of frequency boosts 90
5.5 Break-even times of AVX2 and AVX-512 frequency level transitions . . . 91
5.6 Design of our DVFS policy simulator 93
5.7 DVFS policy simulation 94
5.8 Overhead for 2-competitive and oracle DVFS policies 95
5.9 Overhead for 2-competitive and oracle DVFS policies without hyper-threading 96

6.1 Core scheduling for AVX-512 applications 102
6.2 Core specialization for AVX-512 applications 103
6.3 Design for AVX-512 core specialization 104
6.4 XCR0 bits and their associated registers 108
6.5 Heuristics to detect the end of AVX-512 phases 111
6.6 Replication of run queues 113
6.7 Overhead for a very memory-heavy program 116
6.8 Remote AVX overhead with MuQSS, core scheduling, and core specialization 118
6.9 Impact of core specialization on frequency and IPC 119
6.10 Overhead caused by core specialization 120
6.11 Impact of detection of short non-AVX-512 phases 122
6.12 Power interrupts 125

7.1 Performance impact of AVX2 and AVX-512 on benchmarks executed alongside x265 130
7.2 Performance impact of AVX2 and AVX-512 on benchmarks executed alongside a synthetic program 131
7.3 Modified virtual runtime accounting 132
7.4 Scheduling based on overall remote performance impact 133
7.5 Calculation of the ideal frequency 137
7.6 Divergence of virtual runtimes on different CPUs 140
7.7 Performance isolation in two-application benchmarks 142
7.8 Performance isolation for benchmarks restricted to two cores 144
7.9 Overhead introduced by frequency reduction compensation 144
7.10 Performance comparison between CFS and our baseline scheduler 146

List of Tables

2.1 Dennard scaling and leakage-limited scaling 23

3.1 Intel Xeon Gold 6130 Frequencies 37
3.2 Frequency level performance events 53

Listings

6.1 Synthetic workload to trigger task type changes 121