Karlsruhe Institute of Technology

Analysis and Optimization of Dynamic Voltage and Frequency Scaling for AVX Workloads Using a Software-Based Reimplementation

Bachelor's Thesis
submitted by

cand. inform. Yussuf Khalil

to the KIT Department of Informatics

Reviewer:	Prof. Dr. Frank Bellosa
Second Reviewer:	Prof. Dr. Wolfgang Karl
Advisor:	Mathias Gottschlag, M.Sc.

May 03 – September 02, 2019

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any source or auxiliary means other than these referenced. This thesis was carried out in accordance with the Rules for Safeguarding Good Scientific Practice at Karlsruhe Institute of Technology (KIT).

Karlsruhe, September 2, 2019

Abstract

While using the Advanced Vector Extensions (AVX) on current Intel x86 processors allows for great performance improvements in programs that can be parallelized by using vectorization, many heterogeneous workloads that use both vector and scalar instructions expose degraded throughput when making use of AVX2 or AVX-512. This effect is caused by processor frequency reductions that are required to maintain system stability while executing AVX code. Due to the delays incurred by frequency switches, reduced clock speeds are attained for some additional time after the last demanding instruction has retired, causing code in scalar phases directly following AVX phases to be executed at a slower rate than theoretically possible.

We present an analysis of the precise frequency switching behavior of an Intel *Syklake (Server)* CPU when AVX instructions are used. Based on the obtained results, we propose AVXFREQ, a software reimplementation of the AVX frequency selection mechanism. AVXFREQ is designed to enable us to devise and evaluate alternative algorithms that govern the processor's frequency smarter with regard to workloads that mix both AVX and scalar instructions. Using an oracle mechanism built atop AVXFREQ, we show that the performance of scalar phases in heterogeneous workloads can theoretically be improved by up to 15 % with more intelligent reclocking mechanisms.

Contents

Chapter 1

Introduction

Vector instruction sets enable software developers to execute the same operation on multiple data in parallel and allow for fast and energy-efficient implementations in microprocessors as fewer clock cycles and fewer instructions are required to achieve equal results compared to a purely scalar instruction set where each operation is only applied to a single datum. Traditionally, 3D graphics processing is a typical use case for vector instructions – large amounts of polygons need to be processed and transformed using the very same calculations, therefore it makes sense to design instruction sets that are inherently parallel for this purpose. This is why graphics processors (GPUs) were invented in the first place: these chips are specifically designed for vector processing and are thus also categorized as *Single Instruction, Multiple Data* (SIMD) processors. However, other use cases exist where vector instructions seem useful but do not justify the use of a specialized processor. For example, some cryptographic algorithms or multi-media codecs may be accelerated through vector execution but are typically not run long enough or do not expose the massive parallelism of applications like 3D graphics, hence, to some extent, it is desirable to include vector processing functionality in general-purpose processors.

With the advent of the Multi Media Extension (MMX) in the *Pentium MMX* processor in 1997, Intel was the first vendor to introduce an SIMD extension to processors based on the x86 instruction set architecture [21]. However, due to rapidly increasing computational demands, MMX was quickly superseded by the Streaming SIMD Extensions (SSE) that added dedicated 128-bit vector registers and debuted with the *Intel Pentium III* processor in 1999 [22]. Many iterations of SSE led up to SSE4, which finally evolved into the Advanced Vector Extensions (AVX) that introduced 256-bit vectors and that constitute the topic of this thesis. After the first release of AVX in Intel's *Sandy Bridge* microarchitecture in 2011 [20], Intel quickly published several improvements in the years to come: AVX2 with many new instructions was introduced with

the *Haswell* microarchitecture in 2013 [16], and finally, Intel's *Skylake (Server)* microarchitecture in 2017 came with support for AVX-512, which added 512-bit-wide vector registers and several new instructions [27]. Notably, there is also a client variant of the *Skylake* microarchitecture that does not feature AVX-512. However, AVX-512 is set to be made available to the broad consumer market with the arrival of the *Icelake* microarchitecture towards the end of 2019 [3].

For many decades, *Moore's Law* [37] and *Dennard's Law* [12] have guided the development of microprocessors: the first states that, with evolving process technology, the integration density (i.e., the amount of transistors per area) doubles about every two years. The latter declares that, as transistors become smaller, their voltage and current requirements scale proportionally with the reduction in size, and thus the power density stays constant over time. In the recent years, however, development has slowed down as we have nearly reached the physical limitations of what is theoretically possible with silicon. In turn, both of the scaling laws that used to be vital for processor evolution have nearly ended and power density is rising in current generations. This has the effect that it is generally not possible anymore for all transistors within a single chip to switch at full frequency due to arising electrical and thermal issues [41]. The amount of transistors that may run at their maximum frequency is, of course, not only limited for a complete chip but also for any given area within it. As such, the new capabilities of current AVX implementations come at a cost: since an arithmetic unit designed for AVX needs to have many adjacent data paths within a comparatively small area, processors supporting AVX2 and AVX-512 commonly need to reduce their clock frequency when executing such instructions in order to maintain stability.

Reducing the frequency while executing AVX code, however, also means that other instructions executed on the same core will run with a lower frequency, too. Further, since frequency switches need some time and therefore come at a slight cost, they should not be done more often than necessary. Consequently, current Intel processors will operate at a reduced frequency for a while even after the last AVX instruction has been executed. For heterogeneous workloads that consist of both vector and scalar parts, it was shown that this behavior can have a severe negative impact on a program's overall performance. Using nginx, a wide-spread web server software, with a vectorized implementation of the ChaCha20 algorithm used for encrypting network traffic, Gottschlag et al. [15] have measured an 11.2 % reduction in throughput when using AVX-512 compared to an implementation with SSE4. To mitigate the negative effects, they propose *core specialization* as a technique that migrates programs to specific cores when executing AVX code and back again when they are in a phase with purely scalar instructions. This way, only a few cores

of a processor are affected by AVX-induced frequency reduction which, in the case of nginx, yields a vast improvement but does not fully alleviate the performance drawback.

In this thesis, we want to look at a different optimization approach: instead of moving workloads between cores, we want to explore possibilities of improving the reclocking algorithm itself. However, Intel only provides a very vague description of what precisely they have implemented in their chips. For this reason, our work began with conducting an analysis of the reclocking behavior of a current-generation Intel processor. Using the Performance Monitoring Unit (PMU) of such a processor, we measured the amount of instructions required to trigger frequency reductions as well as the delays incurred by them, and how long it takes until the clock speed is raised again. We prove that Intel's claims are not only vague, but also wrong in terms of declared delays and we will provide a more thorough model of the algorithm. Further, as we are unable to modify the hardware itself, we will describe AVXFREQ, a reimplementation of Intel's AVX reclocking behavior through software means, again by using the PMU to generate interrupts upon events that indicate a situation where frequency changes may be required. AVXFREQ is designed to be able to act as a foundation for devising, implementing and evaluating alternative algorithms that potentially improve performance through more intelligent reclocking that allows scalar phases in a program to be executed at a higher clock frequency compared to Intel's implementation. We show that AVXFREQ is capable of reflecting a simplified model of the hardware algorithm with a performance decrease of about 1 %. To evaluate the potential of our approach, we will present a reclocking strategy based on AVXFREQ where we leverage the user-space program's knowledge about what it is going to do next to select the frequency to apply to the processor. For this implementation, we show that it can yield improved performance for heterogeneous workloads by achieving up to 15 % higher throughput during a purely scalar phase immediately following an AVX phase. Another, more automatic optimization idea that may be implemented atop AVXFREQ is to have the operating system measure vector and scalar execution phases of a process and provide predictions of how long a scalar phase will last. This may be done as future work.

We start by introducing the original problem as well as previous work on the topic in a deeper and more thorough fashion in Chapter 2. As our analysis of Intel's implementation is a required prerequisite for a software reimplementation, Chapter 3 describes the design and implementation of the analysis framework we built and presents the results we obtained. Guided by the insights from our analysis, we will detail the design of AVXFREQ and the user-space-driven reclocking approach outlined above in Chapter 4. Afterwards,

in Chapter 5, we evaluate how well AVXFREQ fulfills its purpose and what improvements are achieved when user-space can exercise control over AVX reclocking. We conclude this work and propose ideas for further research based on our findings in Chapter 6.

Chapter 2

Background & Related Work

In this thesis, we present an analysis of the reclocking behavior of a current-generation Intel processor when executing AVX instructions. We then use the information obtained from the results of this analysis to build a software prototype that tries to accurately reimplement a subset of the algorithm employed by Intel. This reimplementation is designed to constitute a foundation for exploring possible optimizations to the algorithm that yield a better performance in heterogeneous workloads. In this chapter, we will provide an extensive motivation for our work, describe the technologies we built upon, and talk about previous and related work.

2.1 Dark Silicon and AVX

In 1975, Gordon Moore, one of the co-founders of Intel Corporation, predicted that the transistor density of integrated circuits made of silicon would double roughly every two years [37]. Known today as *Moore's Law*, his prediction held true for several decades with surprising accuracy. However, like every technology, the advancement of silicon-based microprocessors is confined to physical limitations. About thirty years after his famous prophecy, in 2007, Moore projected his law to only have about 10 to 15 years left [13]. And again, he was right: the 14 nm process technology Intel used in 2014 yielded an integration density of about 37.5 million transistors per mm^2, whereas Intel's current 10 nm fabrication node from 2018 allows for around 100.8 million transistors within the same area. This means that, over four years, the density has increased only by a factor of 2.688, whereas – according to Moore's Law – it should have been 4 times [10].

The second large scaling law of microelectronics is known as *Dennard's Law* or *Dennardian Scaling* and is named after Robert H. Dennard who is one

of the authors of a historically important paper from 1974 on the design of very small MOSFETs [12]. In their work, Dennard et al. showed that when a silicon MOSFET is scaled down by a factor κ, its voltage and current requirements decline proportionally with κ. In turn, this means that as smaller MOSFETs can be manufactured with evolving chemical process technology, their power density stays constant, or in other words: as manufacturing possibilities advance, chips of the same total area can be produced with smaller and more transistors and no increases in power density. Similar to Moore's Law, Dennard's Law has come to an end in recent years: in the 1970s, current leakage only had very small and negligible impact on a chip's power consumption and was therefore not considered as a component in Dennard's formulas [6]. However, today a point is reached in the scaling of MOSFETs where switching current and threshold voltage are low enough that leakage has become a major source of energy usage in microprocessors, causing increasing power densities and the breakdown of Dennardian scaling.

These scaling laws used to be essential for the advancement of microprocessors as they guaranteed the ability to build more complex and at the same time energy-saving designs over the years. Nowadays, however, doubling the transistors per area also means about doubled power usage if all transistors are run at their full frequency [41]. If power consumption were to stay constant, that would mean that only half of the transistors (in practice, likely half of the available cores) could be active at the same time, whereas the rest would need to stay turned off – *dark silicon* was coined as a term for this issue. Given the unpleasant outlook, creativity is required from engineers to come up with new ways of improving both power consumption and performance of processors. One approach that has been proposed is the use of *dim silicon* [18]: instead of giving up area to dark silicon, the higher the amount of transistors that are active at the same time, the lower are their frequencies – either all have their clock speed reduced by the same offset or at least some run slower and thus, these transistors are *dimmed*.

In current Intel x86 processors, we can find a dim silicon approach in their implementation of the AVX instruction set extension: as these instructions cause higher energy consumption than previous vector instruction sets or scalar instructions, they may not be executed at full frequency or otherwise system stability would be at risk due to exceeded electrical and thermal limits [23]. In turn, whenever a core of these processors is fed with demanding AVX instructions, it will reduce its clock frequency. From this moment on, everything executed on that specific core runs slower, allowing AVX instructions to run, but at the cost of reduced performance for other operations. However, frequency changes need some time: Mazouz et al. [34] have shown them to take between

25 µs and 52 µs on an Intel Core i7-3770 processor from the *Ivy Bridge* generation. To avoid wasting too much time with frequency switches, a core will keep running at a reduced frequency for a while after the last AVX instruction has been executed [23].

A practical example where vectorization on the CPU can be beneficial for performance is *ChaCha20* [5], a stream cipher algorithm for symmetric encryption presented by Daniel J. Bernstein in 2008 that, in the recent years, quickly gained traction as various implementations have been developed and employed in wide-spread commercial products. Google has added support for this algorithm in their Android operating system as well as the Chrome web browser in 2014 [9] and, as of June 2016, a cipher suite based on ChaCha20 has become a proposed standard for use in the Transport Layer Security (TLS) protocol that is commonly used to encrypt internet traffic [28]. Given the rising adoption of Bernstein's algorithm, it became desirable to create implementations optimized for speed and energy efficiency. Goll et al. [14] have presented an AVX2 implementation of ChaCha20 using 256-bit vectors that provides about doubled performance on an Intel *Haswell* processor compared to an implementation using SSE with 128-bit vectors. However, in practice, engineers from Cloudflare [26] have benchmarked the nginx web server with a version of the OpenSSL cryptographic library that contains an implementation of ChaCha20 with support for AVX2 and AVX-512 on an Intel Xeon Silver 4116 processor and found that nginx's throughput is reduced by 10.6 % compared to when it runs with a variant of OpenSSL without AVX. These findings were confirmed by Gottschlag et al. [15] who measured a 11.4 % decrease in throughput on an Intel Xeon Gold 6130 processor when comparing nginx with an AVX-512-capable implementation of OpenSSL versus an SSE4 one.

While at first glance it may seem strange that increased vectorization with AVX reduces performance, this behavior is easily explained by the description of Intel's AVX implementation above: nginx in combination with an AVX-capable build of the OpenSSL library is an example of what we call a *heterogeneous workload* – a program where only certain parts benefit from vector execution, whereas the rest solely uses scalar instructions. Encryption only takes up a fraction of a web server's total processing time, so only this particular fraction is accelerated through AVX. Now, the performance reduction observed when enabling AVX originates from the fact that the other part of the program is slowed down due to the attained frequency reduction after AVX instructions were executed. This poses a problem for the use of AVX in real-world software: only programs that can use vectorization for large parts or nearly all of their code may see benefits, whereas there is a large hazard for developers of software with heterogeneous workloads to cause harm to their program's performance.

2.2 Dynamic Voltage and Frequency Scaling

Mittal [35] defines *Dynamic Voltage and Frequency Scaling* (DVFS) as a technique "for altering the voltage and/or frequency of a computing system based on performance and power requirements." The required electrical power P for switching a CMOS gatter can be characterized as $P = \frac{1}{2} \times C \times U^2 \times f$, where C is the circuit's electrical capacitance, U is the voltage, and f is the switching frequency [17]. Looking at the formula where power increases linear in frequency and quadratic in voltage, it becomes clear that the power consumption of a processor can be regulated by controlling these two parameters. Therefore, the idea behind DVFS is to have a microprocessor run at high frequencies when system load demands for high performance and at lower frequencies when the system is idling to save energy – the voltage is set according to what is required to maintain stable execution (i.e., all transistors properly switch within a single clock cycle) at the respective frequencies.

Modern Intel processors have a DVFS implementation called *Enhanced Intel SpeedStep Technology* (EIST) [40]. Using EIST, the operating system may select a P-state for each core – these *performance states* govern a core's frequency and its voltage. On current-generation processors, P-states simply represent integer multipliers of the chip's bus clock.

Further, the *Skylake* microarchitecture generation introduced in 2015 was the first to support *Intel Speed Shift Technology* [25], an implementation of the *Collaborative Processor Performance Control* (CPPC) interface defined by the ACPI standard [2]. Outside of public marketing, Speed Shift is also called *Hardware-Controlled P-States* (HWP) in Intel's technical documents [40]. As the name implies, when using HWP, control over the processor's P-states is transferred from the operating system to the hardware itself. Precisely, with HWP, a *Power Control Unit* (PCU) in the processor constantly monitors load on different parts of the chip as well as power consumption and accordingly assigns voltages and frequencies to the different units within the chip. Contrary to previous possibilities with EIST, the operating system is merely left with the ability to hint the processor about the desired minimum and maximum performance as well as the user's preferences for energy efficiency.

Notably, starting with the *Haswell* microarchitecture (a predecessor of *Skylake*), Intel implemented physical measurement of the power consumption in their processors, whereas previous generations only relied on statistical models. Schuchart et al. [38] have shown these measurements to be very precise and in turn, the PCU of these chips is capable of limiting a processor's power consumption nearly exactly at its specified Thermal Design Power (TDP). However, as the authors have shown, this introduces deviations in the performance of different chips of the same model up to 5 % compared to the

average due to production fluctuations where some chips require more or less power than others. It is plausible that this variance may be carried over to the results of our analysis when executed on different chips.

In this thesis, we will conduct the AVX reclocking analysis with HWP enabled as this would be the usual case for real-world systems equipped with current Intel processors. For our software reimplementation, however, we are going to disable HWP and make use of the legacy EIST software interface as we need to be able to control the processor's clock speed via means of the operating system.

2.3 Power Management

The issue that motivates our work is, in its essence, related to problems found in the power management of hardware devices [33]. Here, the overall goal is to fulfill a computing system's tasks with minimal power consumption. For example, a hard disk drive (HDD) in a system is only required to be active when there are pending read or write requests, otherwise, hard disks only waste electricity without serving a useful purpose. Thus, vast amounts of energy may be saved by turning them off when they are not needed for a period of time. However, simply spinning disks down as soon as no requests are pending and up again when they arrive is not a feasible strategy as hard disks generally need some time for both operations. In addition, that would potentially waste more power than not shutting them down at all: during spin-up, the motor needs to accelerate the disks from zero to thousands of rotations per minute, which consumes considerable more energy than keeping them running for a period of time when the motor is already at speed.

To estimate whether energy can be saved by turning a hard disk off for a time period, we need to calculate the break-even time t_{be}, i.e., the amount of time after which the energy savings from being shut down exceed the energy drawn during the spin-down and spin-up phases. We can simply formulate the equation to find t_{be} as follows:

$$\underbrace{P_a \times t_{be}}_{\text{energy consumed when disk stays active}} = \underbrace{E_t + P_s \times (t_{be} - t_t)}_{\text{energy consumed when disk is turned off}}$$

Here, P_a is the electrical power consumed while a disk is active, P_s is the power consumption while in standby (note that this is not 0 as some parts of the electronics need to stay active even while the motor is off), E_t is the energy needed during spin-down and spin-up and t_t is the time needed for

these transitions. Thus, $t_{be} - t_t$ is the actual time spent in the standby state. Transforming the formula yields

$$t_{be} = \frac{E_t - P_s \times t_t}{P_a - P_s}$$

as a way of directly calculating t_{be}.

An optimal solution to the original problem, however, would require an oracle that foresees the future – only then, we can know for what period of time a hardware device is going to remain unused, and whether this time is longer than t_{be} for this specific device. So far, no scientifically proven way of precisely predicting the future is known to mankind. Therefore, several approaches using profiling and stochastic models have been proposed in the past and are implemented in modern operating systems. As we do not want to dive deeper into this topic here, see Lu et al. [33] for a more thorough overview of strategies that have previously been studied.

How is this related to the problem statement in this thesis? We do not want to save power, but rather improve performance. Nevertheless, the same thoughts as above also apply to our case. For hard disks, energy can be saved by turning them off in phases where they are not needed – in our case, performance can potentially be improved by raising the frequency during phases where AVX units are not needed. Raising and lowering clock frequencies requires some time, similar to how spin-downs and spin-ups of hard disks take some time. Thus, for our needs, a break-even time like described above exists, too. Further, an oracle would be required for optimal decisions in our case similar to the way it would be required for optimal power management. This also hints that strategies used in power management techniques based on profiling or modeling could perhaps also be applicable to the problem we are investigating.

However, unlike power management researchers, we can not simply test different approaches and algorithms. Power management is generally considered to be a task of an operating system, whereas – as described in the previous section – current Intel processors manage frequency scaling (especially AVX reclocking) solely in hardware. This motivates our reimplementation: without it, we could only theorize about what could be possible or not when employing alternative strategies for reclocking. Further, using this reimplementation allows us to implement what is basically an oracle, albeit not applicable for practical applications: when the operating system is handed control over frequency reductions needed for AVX, we may also have user-space applications tell us when they are going to stop using AVX for a while – thus, we achieve foresighted knowledge about scalar phases.

2.4 Core Specialization

The idea behind the approach we follow in this thesis is to optimize the reclocking behavior itself for heterogeneous workloads. Gottschlag et al. [15] have proposed an alternative method of mitigating the negative effects of frequency reduction during and after AVX execution by using *core specialization*. In their work, they categorize both threads and cores into "AVX" and "non-AVX" ones. Only AVX cores may then execute AVX threads, whereas AVX code is kept away from the other cores. In turn, only these specific cores experience AVX-induced frequency reduction while other cores can keep running at their maximum speed. The authors have built a proof-of-concept implementation using the Multiple Queue Skiplist Scheduler (MuQSS) where they added specific runqueues for each task type to the scheduler and a system call that allows a thread to communicate to the kernel when it enters and exits AVX phases. The modified scheduler implementation then migrates tasks between cores depending on their type.

To evaluate the benefits of this approach, they compared the average frequency of all cores as well as the throughput of an nginx web server with a vectorized implementation of the ChaCha20 algorithm (as previously described in Section 2.1) on an Intel Xeon Gold 6130 once when run with an unmodified system and once with enabled core specialization. While unmodified scheduling saw a reduction of 11.2 % in throughput and 11.4 % in average frequency compared to nginx without vectorized ChaCha20, their core specialization approach largely alleviated the reductions to a mere 3.2 % and 4.0 %, respectively.

Building upon this work, Brantsch [8] reached a further improvement of the core specialization approach. Instead of relying on user-space to communicate execution phases to the kernel via system calls, he implemented a mechanism called *fault-and-migrate*: it is possible for an operating system to selectively disable AVX instructions per core. Then, when a program tries to execute such an instruction, the processor will generate an exception and trap into the kernel. This way, the kernel is notified whenever AVX instructions are to be executed and can then migrate the task to a core designated for AVX purposes. However, there is no proper notification mechanism available for when a thread has stopped using AVX, so different ways of estimating this were implemented. The most promising one turned out to be an approach where a thread is considered to have returned to a scalar phase whenever it does a system call, based on the idea that AVX is solely used during phases of numeric computation where there is no need to call the operating system. In case of the nginx benchmark outlined above, this decreased the performance degradation compared to running without AVX to just 1 %. This is a drastic improvement

against the 11.2 % that were originally measured without core specialization, however, performance is still not on par with the variant without AVX-512.

Contrary to Brantsch' approach that defines system calls as the end of AVX phases, in this work, we will make estimates by using periodic timer interrupts and counting the executed AVX instructions during each period. Whenever the counter is zero, we know that AVX was not used in the previous period. This gives us an approximation that is off by one period length in the worst case.

Chapter 3

Analysis

In order to be able to evaluate potential means of improving Intel's AVX reclocking algorithm, we first need to obtain thorough knowledge of the algorithm as it is implemented in current Intel x86 CPUs. We can then use this knowledge for the software-based reimplementation presented in Chapter 4 and to understand the hardware-induced constraints Intel needs to keep within, which is in turn necessary for designing a feasible and implementable improved reclocking algorithm.

Intel regularly publishes optimization manuals [23] intended for compiler developers and software engineers which contain a vague description of the mechanism used for deciding when to lower or raise the processor's frequency upon execution of AVX instructions. Precisely, Intel defines three *turbo license levels*, which designate frequency offsets for different instruction mix scenarios:

- Level 0: only non-demanding (i.e., scalar, SSE, AVX1 or light AVX2) instructions are being executed; a core may run at its maximum turbo frequency. This is the default state.
- Level 1: active during the execution of heavy AVX2 and/or light AVX-512 instructions. The maximum frequency is lowered to a SKU-specific value.
- Level 2: used for the execution of heavy AVX-512 instructions. The maximum frequency is lowered to a SKU-specific value that is further below the frequency used in level 1.

Here, "heavy" instructions are defined to be floating-point, integer multiplication or integer fused multiply-add (FMA) operations. Given these license levels, Intel states that it may take up to 500 µs until the new frequency is applied and about 2 ms until a core reverts to level 0 after executing the last "heavy" instruction. Before the frequency is lowered, a core operates at "a lower peak

capability," however, Intel does not further specify what that exactly means. Intel hints that the license decisions are not solely bound to the instruction types as given in the level descriptions, but rather depend on the mix of instructions executed within a certain time window.

In this chapter we will describe the design of a framework that allows us to analyze the actual behavior of an x86 processor during the execution of AVX instructions. Afterwards, we will present and evaluate the results generated when executed on a system equipped with a modern Intel CPU, and point out deviations between the actual behavior and what Intel maintains in their specification.

3.1 Methodology

For our reimplementation, our goal is to create a model of the reclocking behavior of an AVX-512-capable CPU that is as complete as possible and reflects the decisions made by the hardware with high accuracy. Therefore, by conducting this analysis, we want to answer the following aspects:

- When exactly does a CPU core decide to reduce or raise its frequency during and after AVX execution?
- How much time do turbo license level switches need?
- Do the CPUs switch directly from level 0 to level 2 in case of heavy AVX-512 instructions or is there a step to level 1 in between?
- What does Intel mean by "lower peak capability" while lowering the clock?
- How complete is Intel's description of the reclocking algorithm?

In order to create a precise model we want to analyze these questions in different scenarios, i.e., for different instruction types, for different global load situations as well as with and without enabled turbo frequencies. To reach our goal, we run our analysis framework with synthetic code snippets that are designed to trigger the behavior to be analyzed.

3.2 Design

Our analysis framework consists of a module for the Linux kernel as well as a user-space component which interact with each other and make use of the PMU,

a unit commonly found in modern microprocessors that enables software to measure performance and bottlenecks on the hardware level. In the following sections, we will present the design and features of these components and describe how they contribute to our analysis purposes.

3.2.1 Performance Monitoring Unit (PMU)

Modern x86 CPUs commonly feature a Performance Monitoring Unit (PMU) which exposes a set of *performance counters* that may be configured to count assertions of a large set of *performance events* [40].

Precisely, we use version 3 of the x86 *Architectural Performance Monitoring* facility, which features three *fixed counters* per logical core that count retired instructions, cycles during which the core is not in a halt state, and TSC cycles in unhalted state, respectively. The time-stamp counter (TSC) is a simple counter found in current x86 CPUs that increments steadily with a fixed frequency and independent of the core clock, thus making it suitable for measuring wall-clock time. In addition to the fixed counters, eight freely configurable counters are available per physical core (four per logical core when Simultaneous Multi-Threading (SMT) is enabled). These counters may be set to count any of the performance events available for a specific microarchitecture, e.g., most architectures define events for cache hits/misses, execution stalls or load on specific execution units.

Each counter is represented via a model-specific register (MSR) and also configured through one. More specifically, software may configure the event to count (non-fixed counters only) and when to count (i.e., in user mode (ring ≥ 1) and/or kernel mode (ring 0)). Additionally, the counter can be configured to trigger an interrupt when it overflows. By setting the counter to its maximum value less an offset, this can be used as a mechanism to generate notifications when a certain amount of events of a specific type has occurred. The interrupt vector used for delivery can be configured in the core's APIC's Local Vector Table (LVT). Optionally, the PMU may be instructed to freeze all counters at their current values as soon as an interrupt is triggered.

3.2.2 Overview

The analysis tool presented here is made up of a kernel and a user-space component where the former provides the latter with means to configure the PMU and efficient handling for interrupts generated by performance counter overflows.

As depicted in a simplified way in Figure 3.1, the user-space component spawns $n \in \mathbb{N}$ *execution threads* and $w \in \{1, n\}$ *wait threads*, each corresponding

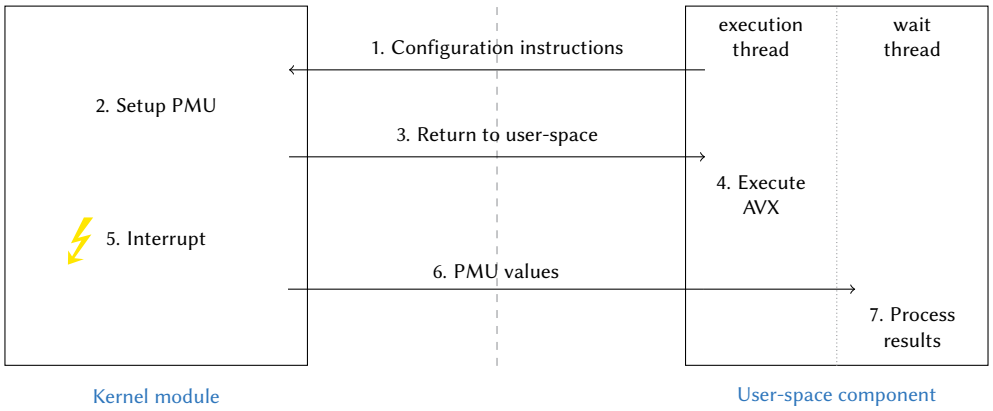


Figure 3.1: Simplified analysis framework architecture. The kernel module enables the user-space component to configure the PMU and handles interrupts.

to exactly one execution thread. The idea behind having multiple execution threads is to be able to make measurements on multiple cores simultaneously, thereby simulating parallel workloads. In *pre-throttling mode*, only one execution thread has an associated wait thread ($w = 1$): the other execution threads are started earlier to simulate an environment with pre-existing load, where global load is merely increased by utilizing one more core. Without pre-throttling, there is a wait thread for each execution thread ($w = n$).

Upon startup, each execution thread generates a PMU configuration designed to produce the desired measurements, which is then applied by the kernel module. Now, the kernel module jumps back into user-space to an address previously defined by the execution thread. Then, the thread will execute AVX instructions until preempted by an overflow interrupt generated by the PMU according to its configuration (as described in Section 3.2.1). Each wait thread is initially suspended until an interrupt is triggered on its corresponding execution thread, at which point it is resumed by the kernel component and provided with the raw performance counter values.

3.2.3 Kernel Component

Our kernel component is not supposed to conduct any analysis tasks by itself, but is designed to aid the user-space component described later in Section 3.2.4. We chose to implement it as a module for version 5.1 of the Linux kernel, which implies that it is written in the C programming language. Existence and design of this kernel module are motivated by our user-space component's needs to configure the PMU in order to conduct the measurements required for our analysis. This can only be done from kernel-space.

During module startup, the PMU is reset to a default state and the performance counter overflow interrupt vector is set in the LVT of each core's APIC. Notably, this degrades the functionality of Linux's `perf` subsystem as `perf` partially relies on using the PMU.

The module interfaces with user-space by defining a custom device class and then providing a virtual device of the previously defined class, exposed via `/dev/reclocking_analysis` in the virtual file system. User-space may then `open()` the provided device file and interact with the module by using several offered `ioctl()` calls.

Execution threads, on the one hand, initiate their execution by using the `SETUP ioctl()` call. A C `struct` must be passed that contains a set of MSRs to be written by the kernel module – these are used to configure the PMU. In order to increase the precision of our measurements, it is desirable to cut time spent in user-space without actually executing the specific code to be measured. Therefore, a value for the instruction pointer (IP) must also be passed that will be set in the thread's context before returning to user-space. Thus, the thread will not directly return at the previous position in the libc's `ioctl()` wrapper but rather be redirected to another location in memory (i.e., where the AVX code for our measurements lies). Optionally, the `r12` x86 architectural register may also be set so that the code executed in user-space upon returning is able to access data structures in an easy manner without needing to use the stack. As described further below, the interrupt action must also be defined beforehand by the execution thread. After applying the configuration, the `ioctl()` handler saves the current TSC value (to be able to measure elapsed time later on) and returns to user-space.

Wait threads, one the other hand, start with the `WAIT_FOR_INTERRUPT ioctl()` call, which takes two arguments: first, a pointer to a `interrupt_result` structure in the user-space component where the resulting performance counter values shall be stored later and second, the numeric identifier of a CPU core where an interrupt is expected to occur – this is what effectively binds a wait thread to its corresponding execution thread. The calling thread is suspended by setting it into `TASK_UNINTERRUPTIBLE` state. This state in Linux's task state machine allows a thread to be woken up only by the kernel itself and not via any user-space mechanisms (e.g., UNIX signals) [31]. Consequently, this way we ensure the execution flow is not interrupted unexpectedly.

It is expected that all execution threads that have an associated wait thread trigger a performance counter overflow interrupt some time after setup. The interrupt handler will then proceed with one of multiple actions as instructed by the `SETUP` call:

- `WAKE_WAIT_THREAD`: this action reads all performance counters and writes them along with the current TSC value and the recorded TSC value at `SETUP` to the `interrupt_result` structure of the corresponding wake thread. Then, the wait thread is woken up, so that it may process and print the results. The execution thread that triggered the interrupt is returned to its previous instruction pointer (from before the `SETUP` call). Notably, from user-space's view, the original `SETUP` call returns only now.
- `SET_MSRS`: this is used for analysis tasks consisting of two consecutive steps, e.g., when two different AVX phases are to be executed and we are only interested in performance events from the latter. The TSC value is recorded and another set of MSRs is configured on the thread's core. For the next interrupt, the action is unconditionally set to `WAKE_WAIT_THREAD`.
- `GOTO`: exactly like `SET_MSRS`, but also sets a new instruction pointer on the execution thread.

Each of these actions concludes with resetting the PMU's overflow bit and the APIC's state in order to be ready for further interrupts.

A practical software engineering issue arises from the fact that wait threads are suspended in an uninterruptible state after startup: they may easily get stuck due to programming errors that cause a lack of interrupts. For these cases, a third `ioctl()` call was implemented: `RESET_WAIT_THREADS`, which simply wakes all suspended wait threads and makes their pending `ioctl()` calls return with an error status.

Note that our implementation has a limitation: it does not work properly on processors with SMT enabled. There are two reasons for this: first, as there is only one PMU per physical core, only four performance counters are available per logical core with twofold SMT. However, not all performance events we use are on a per-thread level, some are only per-core (e.g., all events that correspond to license levels), and thus it would not be required to measure them on both threads of a physical core. Second, as we will explain in Section 3.2.5, all events we use to generate interrupts are on a per-core level, thus our interrupt handling would need to know that an interrupt generated on one thread also affects the other thread on the same physical core (if there are two execution threads running on one core). Nevertheless, as we do not have implemented any such SMT awareness, this remains future work.

3.2.4 User-Space Component

The user-space component of our analysis framework is the one that implements and performs the actual analysis tasks and is aided by the previously

described kernel module by instructing it to configure the PMU and handle performance interrupts. Akin to the kernel module, our user-space program is written in C with some additional helper tools implemented in the PHP scripting language for invocation and monitoring tasks and to generate spreadsheets containing the results. AVX instructions included in the program are directly written in x86 assembly.

As briefly described in Section 3.2.2, the user-space process spawns $n \in \mathbb{N}$ execution threads and $w \in \{1, n\}$ wait threads. The execution threads create the PMU configurations to be applied by the kernel and then run the code to be measured. Wait threads are each associated with an execution thread and receive and process the raw performance counter values when interrupts are triggered on their corresponding execution threads. The amount of execution threads (n) is specified as a command line argument, whereas either all of them or only just one have an accompanying wait thread. This depends on whether *pre-throttling mode* is enabled: the idea of this mode is to create an artificial, pre-existing global load situation across several cores where load is already high and further increased by utilizing an additional core. Therefore, the startup of one specific execution thread is delayed by a moment, and we also only want to collect results obtained from this thread, thus just one wait thread is required. In contrast, without pre-throttling, all execution threads start at precisely the same time and results are gathered from all of them. The code run on the threads used for pre-throttling may either be purely scalar or can use AVX, too. This way we can test whether the load type (i.e., scalar or AVX) on other cores makes a difference to AVX reclocking on one core. Execution threads are bound to CPU cores 1 to n, respectively; the wait threads to the following cores. This also implies that at maximum $\lfloor \frac{C}{2} \rfloor$ execution threads may be run, where C is the number of CPU cores installed in the system, and a minimum of two cores must be available. With pre-throttling enabled, it is theoretically possible to have $C - 1$ execution threads running. Note that pre-throttling mode does not have any effect when only one execution thread is used.

We want to be able to run tests with different instruction types. Therefore, an arbitrary number of ELF sections containing AVX instructions may be included in the component's compiled binary executable. The address of one of these sections and its length must be passed as arguments to the program (these values are easily obtainable using tools like `objdump`). On startup, one or more executable memory areas, each consisting of four pages, are mapped and filled with the content of the passed section, repeated until the area is full, or alternatively, filled only with a precisely set amount of repeated instructions. This allows our measurement modes to investigate the CPU's behavior when only a very specific amount of AVX instructions is executed. For example, this

can be used to determine how many instructions exactly are required to trigger a frequency reduction, whereas infinite loops are useful to measure the time taken to switch frequencies. In order to ensure the instruction flow does not run outside the allocated area, one of two different loop modes is used at the end of each memory area, depending on the measurement mode:

- `LOOP_AVX`: a jump instruction to the beginning of the area is added in order to make the constructed code loop – this allows for infinite AVX execution until the executing thread is interrupted.
- `LOOP_R12_CMP`: a spinlock-style loop is inserted that constantly compares the value referenced by the pointer stored in the `r12` register to 0 and returns as soon as it isn't equal to 0 anymore:

  ```
  loop:
  cmp 0x0, (%r12)
  je loop
  ret
  ```

  This way, after a specific amount of AVX instructions was executed, the executor may spin using only scalar instructions until it is instructed to return from outside when another thread updates the value underneath the pointer in `r12`. Note that our AVX memory area does not have its own stack frame in any way, so, assuming that the executing thread jumped into the memory area by using the `SETUP ioctl()` described in Section 3.2.3, we actually return the `ioctl()`'s stack frame here. This is a rather fragile and non-portable approach and may not work as desired with every libc implementation.

Further, some measurement modes need to execute AVX instructions until interrupted and then want to execute purely scalar code to wait for another event (e.g., until the clock speed is raised again). For this purpose, we also allow mapping memory areas that solely consist of an empty loop.

The number of four pages per area was not chosen arbitrarily: on an x86 processor running in 64-bit mode [40], pages have a default size of 4 KiB, thus four pages equate a total size of 16 KiB. We originally used an area size of 2 MiB (512 pages), based on the idea of achieving a purely homogeneous workload that does not contain any jumps to increase the precision of our measurements. However, tests showed that the code would become approximately 20 % faster when run on multiple cores in parallel. We believe this behavior to be caused by instruction cache misses – although modern CPUs commonly feature an instruction prefetcher, there is a caveat: it does not load instructions across page

boundaries, and thus every 4 KiB of instructions we would see a cache miss and a costly pipeline stall until the next instructions arrive from memory. By using parallel execution on multiple cores, this effect is mitigated as the fastest core would already have loaded the instructions into its L1 and L2 caches[1]. Thus, when other cores try to fetch the instructions from memory, the requests are instead served by a core that already has them via the cache coherency protocol, thereby dramatically reducing the latency. We could theoretically verify this theory by using performance events for measuring cache hits and misses, however, this is not too interesting for our purposes. It could be worthwhile to look into huge pages as an alternative solution, though that remains future work.

In order to avoid inaccuracies caused by preemption, all execution threads use the `SCHED_RR` scheduling policy offered by Linux's Completely Fair Scheduler (CFS) [39] which is designed for near-real-time execution and selects real-time threads ordered by their priority; threads of equal priority are executed in a round-robin fashion. CFS exposes additional configuration settings [44] to control the fraction of time that may be consumed by real-time processes, namely `sched_rt_period_us` and `sched_rt_runtime_us`. The former sets a time window (1 s per default) and the latter contains the absolute amount of time within that window that is available to real-time threads (950 ms per default). In theory, we could configure these to allow for infinite real-time execution, however, practical tests have shown this leads to unbearable system hangs that would require further work on our implementation in order to fix. We settled for a value of 990 ms for `sched_rt_runtime_us` as compromise.

In all measurement modes, we configure the following performance events (as documented by Intel in [36]):

- `CORE_POWER.LVL0_TURBO_LICENSE`: counts core cycles spent in turbo license level 0.
- `CORE_POWER.LVL1_TURBO_LICENSE`: counts core cycles spent in turbo license level 1.
- `CORE_POWER.LVL2_TURBO_LICENSE`: counts core cycles spent in turbo license level 2.
- `CORE_POWER.THROTTLE`: counts core cycles during which the out-of-order (OoO) engine is throttled.

[1] Note that on *Skylake (Server)* processors the L2 caches are inclusive, whereas L3 is a victim cache [27].

- `INT_MISC.CLEAR_RESTEER_CYCLES`: counts core cycles while the execution engine is stalled waiting for instructions to be delivered. This is used to estimate the time spent before the actual execution when switching from kernel-space to user-space in execution threads.
- `FP_ARITH_INST_RETIRED.PACKED`: counts retired packed floating-point vector instructions. Several variants for 128-bit, 256-bit and 512-bit vectors and single- and double-precision instructions are available which we select according to the instruction type used in the AVX code section passed at startup.
- `UOPS_DISPATCHED_PORT.PORT_0`: counts micro-instructions dispatched by the processor's scheduler at execution port 0. The use of this performance event is motivated by the *Skylake (Server)* microarchitecture on which the CPU we used for our analysis is based. These processors have an AVX-512 unit fused from two 256-bit units at execution ports 0 and 1 [27]. For other microarchitectures, other performance events may be appropriate.
- `UOPS_DISPATCHED_PORT.PORT_5`: counts micro-instructions dispatched by the processor's scheduler at execution port 5. The motivation here is the same as for the performance event counting micro-instructions at port 0, however, only some specific *Skylake (Server)* CPUs have an additional, dedicated (i.e., non-fused) AVX-512 unit at port 5.

At startup, wait threads simply set their core affinity and then block at a synchronization barrier. Execution threads, in contrast, need to set their core affinity, their scheduling policy and build up the configuration to pass to the `SETUP ioctl()` call provided by our kernel module. Afterwards, they block at the same synchronization barrier as the wait threads. As soon as all threads have reached the barrier, in order to ensure their respective cores are ramped up to their maximum turbo frequency before starting the test run, the execution threads will enter a 150 ms busy-wait loop before calling the `SETUP ioctl()`. The wait threads directly jump into the `WAIT_FOR_INTERRUPT ioctl()`. Execution threads used for the previously described pre-throttling mode are an exception here, as they do not synchronize with the others but rather start executing right away, thus giving them a head start of about 150 ms. This reflects the desired behavior as, again, pre-throttling mode is designed to create pre-existing load conditions where load is further increased, so that we can measure the impact the type of load on other cores has on AVX reclocking.

After execution has completed, there is not much to do for the execution threads: their `SETUP ioctl()` returns and then they simply exit. Wait threads,

on the other side, will again synchronize at a barrier and then output the results as provided by the kernel module one after another before they exit, too. As soon as all threads have completed, the program quits.

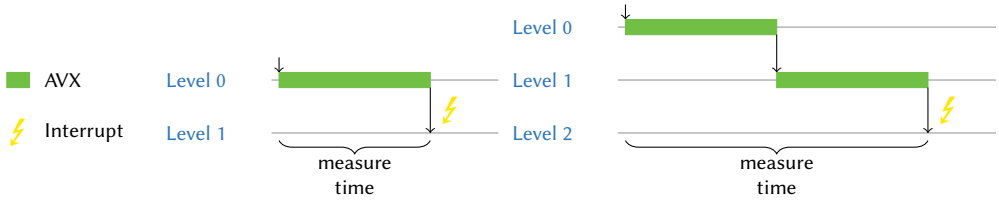
3.2.5 Measurement Modes

In order to answer the questions named in Section 3.1, we implemented several different measurement modes in our user-space component, which are to be presented hereafter.

DOWNCLOCK

Our first measurement mode is designed to measure the downclocking behavior – i.e., how long it takes for a CPU to reduce its frequency and whether there is a step to turbo license level 1 before switching to level 2 for instructions that target level 2.

In this mode, we simply map a single memory area with AVX code to be run by all execution threads and configure the PMU to trigger an interrupt and freeze the performance counters as soon as one cycle is spent in either level 1 or level 2, depending on the target license level passed as command line argument to the program. We use the `WAKE_WAIT_THREAD` interrupt action provided by the kernel component. Thereby, we can measure the time taken for the frequency reduction. When running a test case using this measurement mode with level 2 as target, we will also see whether any cycles were spent in level 1 from the respective performance counter. Thus, this measurement mode indeed answers the aforementioned questions.

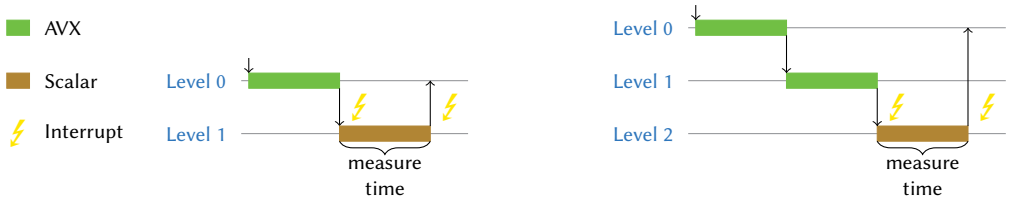


(a) Level 1 (b) Level 2

Figure 3.2: Illustration of the `DOWNCLOCK` measurement mode. This mode measures the time until the requested target license level is reached.

UPCLOCK

After analyzing the downclocking times, the next logical step is to look at the reverse process: the upclocking. Here, we are mainly interested in the time the



(a) Level 1 (b) Level 2

Figure 3.3: The `UPCLOCK` mode measures how long it takes a core to return to its level 0 frequency after an AVX-induced reduction.

CPU takes before returning back to its non-throttled frequency after the last AVX instruction has retired.

Like in the `DOWNCLOCK` mode, we map an AVX memory area into which the execution threads jump after startup, however, we also map an additional page with an infinite loop. In the first step, we configure an interrupt to be fired after switching to either level 1 or level 2, depending on the input, in order to be able to measure upclocking from both throttle levels. Then, using the `GOTO` interrupt action, we move the execution thread to the infinite loop page, reset our performance counters and configure the PMU to trigger an interrupt for when level 0 is reached again and to freeze the performance counters at this point. It is important to not simply leave the core in a completely idle state, as the kernel would then eventually run the `MWAIT` [19] instruction on the core, causing it to enter a halt state, and thus our measurements would be useless given that it would not reflect real-world heterogeneous applications, and because the clock is disabled when the core is halted (i.e., there is no frequency anymore whatsoever).

Using the described procedure, we measure only the time spent after reaching a turbo license level with reduced frequency until returning to nominal frequency, which is exactly what we are interested in. Notably, in this mode, we instruct the PMU to also count cycles while running in kernel-space (i.e., ring 0) as this is also time spent without executing AVX instructions, and thus must be measured to retrieve precise results.

PRE_THROTTLE_TIME

As cited at the beginning of this chapter, Intel talks in their optimization manual [23] about the CPUs operating at a “lower peak capability” before the switch to a turbo license level with lower frequency is completed. Early experimentation showed this state is seemingly represented by the `CORE_POWER.THROTTLE` performance event which is described to count cycles where the OoO engine is

throttled [40]. We want to find out when exactly this throttle state is activated and what instruction throughput the CPU achieves before throttling to get an idea of the theoretically possible performance if the frequency reduction did not exist.

For this purpose, the `PRE_THROTTLE_TIME` mode conceptually works very much the same way as the `DOWNCLOCK` mode: we configure an interrupt that fires and freezes the performance counters as soon as the first cycle was spent in throttled mode and run our AVX code using the `WAIT_FOR_INTERRUPT` interrupt action. Thereby, we obtain the desired information about the behavior during the time between starting execution and OoO engine throttling.

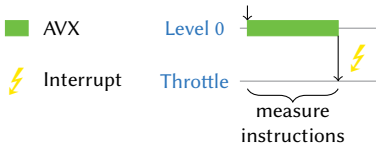


Figure 3.4: After starting to execute demanding AVX instructions, a core enters a state where the OoO engine is throttled. The `PRE_THROTTLE_TIME` mode measures the time it takes until the throttling takes place.

REQUIRED_INSTRUCTIONS

To obtain a model of the reclocking algorithm that is as complete as possible, we are not only interested in the time it takes for a CPU to switch turbo license levels, but we also want to know how many instructions are precisely required to eventually trigger a frequency reduction.

The implementation of this measurement mode is more complex compared to the other modes and also partially depends on the license level transition to be examined. For this mode, the idea is to run multiple iterations where the amount of executed AVX instructions is incremented in every iteration until executing the generated code triggers an interrupt on each execution thread due to license level switches after some time.

In every case, we map an AVX memory area in `LOOP_R12_CMP` mode which initially contains only just one copy of the AVX code in the selected ELF section. As a reminder, this loop mode executes all AVX instructions and then spins in purely scalar code until the `r12` register is set to a value other than zero. When license level 2 was chosen as target, we additionally map an area in `LOOP_AVX` mode that is executed until level 1 is reached, as in this case, we are interested in the amount of instructions required to cause a switch from level 1 to level 2. Unlike `LOOP_R12_CMP`, `LOOP_AVX` keeps executing AVX code forever until interrupted. In AVX pre-throttling mode, another area is created

(a) Level 1

(b) Level 2

Figure 3.5: A certain amount of AVX instructions is required to actually trigger frequency switches. The `REQUIRED_INSTRUCTIONS` mode measures precisely how many are needed.

in `LOOP_AVX` mode to be run by all execution threads used for pre-throttling. This allows them to execute AVX code infinitely as required to fulfill their purpose of creating an artificial, pre-existing load.

In case level 1 is targeted, all (non-pre-throttling) execution threads directly jump into the `LOOP_R12_CMP` area and use the `WAKE_WAIT_THREAD` interrupt action mode. For level 2 as target, we select the `GOTO` interrupt action and first jump into the aforementioned `LOOP_AVX` area and configure an interrupt to be triggered as soon as one cycle in level 1 was completed. Afterwards, the execution threads are also moved to the `LOOP_R12_CMP` area, hence we used the `GOTO` action.

As we only want to see whether the number of repeated instructions in the `LOOP_R12_CMP` area is sufficient to eventually cause a frequency reduction, execution threads instruct the kernel module to fill the `r12` register with a pointer to a global variable that is set to a non-zero value by the main thread after a delay of 1 ms, thereby making all (again, non-pre-throttling) execution threads return after this time – generally assuming that 1 ms is enough time for a turbo license level change to happen. Then, the main thread checks whether an interrupt was triggered on all expected cores. If yes, the test run has completed and the program quits. Otherwise, we remap the `LOOP_R12_CMP` memory with one more copy of the AVX code than in the previous iteration. Wait threads that completed because an interrupt was triggered on their corresponding execution thread are respawned, afterwards we reset the variable used for the pointer in `r12` and start all execution threads again. Unlike in the first iteration, where execution threads generally spin for 150 ms to ramp up the core frequency (as described in Section 3.2.4), we only have them spin for 3 ms in further iterations as their respective cores are already running at their maximum frequency but possibly need to return from an attained turbo license level. Leaving them spinning again for 150 ms would incur an unnecessary prolongation of the test run. This procedure repeats until we have enough AVX

code in our `LOOP_R12_CMP` area to trigger interrupts on all desired cores. Note that execution threads used for pre-throttling are started once at the beginning and keep running without disruptions until the program exits.

At the end, the number of copies of the AVX code in the `LOOP_R12_CMP` memory reflects the amount of instructions needed to cause a frequency transition.

3.3 Results

We use the described analysis framework to conduct measurements with several different AVX instructions using all available combinations of modes, i.e., with and without pre-throttling, different target turbo license levels, with turbo frequencies enabled and disabled, and with 1, 2, 3, and 4 execution threads. All measurements are executed 1000 times in order to obtain statistical certainty. In this section, we present our system setup, describe the precise instructions we used for testing, and present as well as discuss the results.

3.3.1 System Setup

We performed our analysis on an Intel Core i9-7940X processor which features 14 physical cores with twofold SMT, running at a nominal base frequency[2] of 3.1 GHz with a maximum turbo frequency of 4.3 GHz [24]. Additionally, the chip supports *Intel Turbo Boost Max Technology 3.0*, essentially meaning that two specific cores may operate at a higher turbo frequency, in this case 4.4 GHz. These cores are selected based on their electrical and thermal properties during the manufacturing process [27] – this technique is otherwise also known as *speed binning* [32]. The chip's nominal TDP is specified at 165 W, which is the maximum power consumption the chip will sustain over long time periods. Note that, as our analysis framework does not currently support running with SMT enabled (as explained in Section 3.2.3), we have disabled SMT in our system, and thus each physical core exposes only one logical core. Further, whereas our analysis tool theoretically would allow running with 7 execution threads on this CPU (as described in Section 3.2.4), we only tested with a maximum of 4 due to the limited time that was available. Nevertheless, we believe this is enough to generate representative results.

This CPU is based on the *Skylake (Server)* microarchitecture, the first x86 implementation featuring support for the AVX-512 extension [27], making it one of the oldest processors that expose the AVX reclocking issue for heterogeneous workloads. However, not all CPUs built with this microarchitecture feature the

[2]Note that the base frequency equals the TSC's frequency [40].

same number of 512-bit vector execution units: some have two, others only one. The i9-7940X used here has two.

The processor was installed on an ASUS TUF X299 MARK 2 motherboard along with 32 GiB of DDR4 system memory operating at a frequency of 2666 MHz and a NVMe solid-state drive. The motherboard was not chosen arbitrarily: being designed for the needs of the overclocking community, it – unlike most other motherboards for this platform – allows to customize the frequency targets for AVX-induced reclocking in its UEFI's configuration menu. For this analysis, the frequency offsets were configured to 3 and 7 for turbo license levels 1 and 2, respectively, resulting in target frequencies of 3.4 GHz and 2.8 GHz.

We opted to use Fedora 29 (Server Edition) as the operating system with a custom-built Linux 5.1.0 kernel and glibc version 2.28-33. The kernel and all of our own code were compiled using GCC 8.3.1-2 with the default `-O2` optimization level. To minimize overhead and latencies caused by context switches from user-space to kernel-space and vice versa, we disabled all mitigations provided by the Linux kernel for hardware vulnerabilities found in recent CPUs (e.g., Spectre and Meltdown) as well as Kernel Address Space Layout Randomization (KASLR). This improves the quality of our results as we only want to measure hardware behavior, and therefore want to avoid software overhead as much as possible.

3.3.2 Tested Instructions

Different instructions cause different switching activity and therefore need different amounts of energy. In order to create a precise model of the AVX reclocking algorithm as it was implemented by Intel, we want to conduct our measurements with different kinds of AVX instructions to find possible differences in the behavior – especially with regard to what is documented. Note that we only tested homogeneous loads and did not run any tests with heterogeneous mixtures of different instruction classes. Characterizing the frequency scaling behavior for these remains future work.

We tried to select both floating-point and integer operations that reflect the "heavy" and "light" instruction types as defined in Intel's optimization manual [23] as well as instructions we guess to be implemented differently in the hardware's execution units. Consequently, we chose the following subset of AVX instructions for our measurements (as obtained from Intel's manual for software developers [19]):

- `vfmaddsub132pd` (double-precision) and `vfmaddsub132ps` (single-precision) are floating-point FMA instructions that alternatingly add and subtract

the values from a third vector after multiplying the values from two other vectors. I.e., for input vectors a, b and c, they calculate the result vector r according to the following rule:

$$\begin{pmatrix} r_1 \\ r_2 \\ r_3 \\ \dots \end{pmatrix} := \begin{pmatrix} a_1 \times b_1 + c_1 \\ a_2 \times b_2 - c_2 \\ a_3 \times b_3 + c_3 \\ \dots \end{pmatrix}$$

- `vmulpd` (double-precision) and `vmulps` (single-precision) simply calculate the products of all corresponding floating-point members from two input vectors.
- `vpmullq` multiplies corresponding 64-bit integers from two input vectors into 128-bit intermediate results and stores the lower 64 bits of every intermediate result in the target result vector.
- `vpackssdw` merges two vectors with signed 32-bit integers into one vector consisting of signed 16-bit integers by handling overflow conditions via saturation arithmetic, i.e., for values larger than 32767 ($= 2^{15} - 1$) or smaller than -32768 ($= -2^{15}$), the conversion results in these extreme values. In mathematical terms, the operation may be described as follows for input vectors a, b and result vector r:

  $$\forall i \in \{1, \dots, |a|\} \cap \mathbb{N} \colon r_i := saturate(a_i), r_{|a|+i} := saturate(b_i)$$

  where $saturate$ is defined as

  $$saturate \colon \begin{cases} \{-2^{31}, \dots, 2^{31} - 1\} \cap \mathbb{Z} \longrightarrow \{-2^{15}, \dots, 2^{15} - 1\} \cap \mathbb{Z}, \\ x \mapsto min(2^{15} - 1, max(x, -2^{15})). \end{cases}$$

  Note that $|a| = |b|$ and $|r| = |a| + |b|$.
- `vpaddsw` adds signed 16-bit integers from two input vectors using saturation arithmetic as described above.
- `vpmaddwd` is an FMA-style operation that first multiplies corresponding signed 16-bit integers from two input vectors, thereby creating an equal amount of 32-bit temporary results. Afterwards, the adjacent results are added together to generate the result vector. For input vectors a and b, this is the operation executed to obtain the result vector r:

  $$\begin{pmatrix} r_1 \\ r_2 \\ \dots \end{pmatrix} := \begin{pmatrix} (a_1 \times b_1) + (a_2 \times b_2) \\ (a_3 \times b_3) + (a_4 \times b_4) \\ \dots \end{pmatrix}$$

We wrote ELF sections for our user-space component (as described in Section 3.2.4) containing assembly code for all of these instructions in two variants with 256-bit `YMM` and 512-bit `ZMM` registers, respectively. Additionally, for each variant, there are two versions: an "unrolled" one and another non-"unrolled" version. The non-unrolled ones simply contain a single instruction using the first three registers, e.g.:

```
vfmaddsub132pd %zmm0, %zmm1, %zmm2
```

By constantly executing the same instruction with the same operands, we create some artificial register pressure that prevents a core's scheduler from maximizing utilization of the two 512-bit vector units available in the execution engine. The unrolled versions, on the other side, alleviate this pressure by repeating the same instruction, but with different register operands:

```
vfmaddsub132pd %zmm0, %zmm0, %zmm1
vfmaddsub132pd %zmm0, %zmm0, %zmm2
vfmaddsub132pd %zmm0, %zmm0, %zmm3
...
```

Every unrolled section contains the same instruction repeated 31 times, always using `%zmm0`/`%ymm0` for the first two operands and `%zmm{1-31}`/`%ymm{1-31}` as last operand. Thereby, we exhaustively make use of all 32 `ZMM`/`YMM` architectural registers available on AVX-512-capable processors [4].

3.3.3 Downclocking

For our model of Intel's AVX reclocking algorithm, the downclocking behavior – i.e., the process of frequency reduction when executing AVX instructions – is an important puzzle piece. Here, we want to answer questions such as "How long does it take a CPU to switch to its reduced frequency?" and "When is a frequency reduction triggered?". We obtained the results to be presented in this section using the `DOWNCLOCK`, `PRE_THROTTLE_TIME` and `REQUIRED_INSTRUCTIONS` modes provided by our measurement system as described in Section 3.2.

Coarsely, we found the CPU to generally run through the following steps for its frequency reduction:

1. Throttle the out-of-order engine
2. Switch to turbo license level 1 and alleviate OoO engine throttling
3. Switch to turbo license level 2 (for AVX-512 "heavy" instructions)

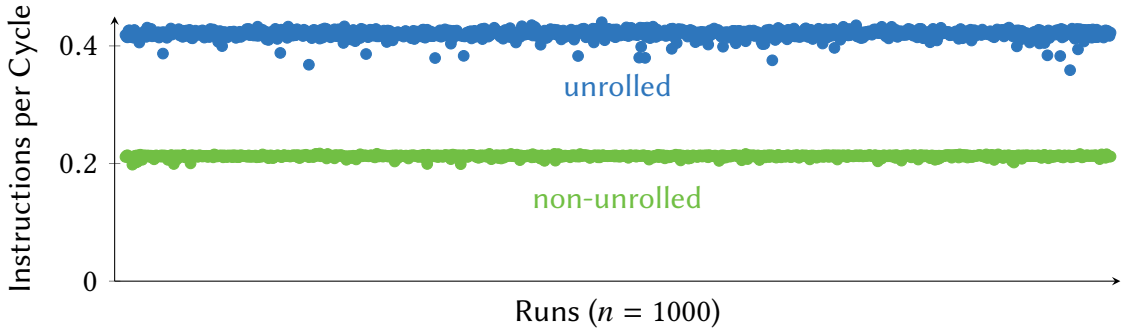


Figure 3.6: Throughput of the `vfmaddsub132pd` instruction before switching from level 0 to level 1 is doubled when unrolled.

This already contains our first insight: even for AVX-512 instructions that Intel defines to be "heavy", the processor will first switch to license level 1 and spend some time in that mode before performing another frequency shift to level 2 – Intel does not mention this in their optimization manual [23]. This information is easily obtained by executing the `DOWNCLOCK` mode twice with both levels as targets – if the CPUs did not make this intermediate step, the test would simply hang when executed with level 1 as target as this level would never be reached.

Similarly, we can confirm an observation previously made by Lemire et al. [29]: even AVX-512 heavy instructions do not always trigger a switch to level 2. We observed this behavior with the `vfmaddsub132pd` instruction, for which only the unrolled version (i.e., the one without register pressure) will ever reach level 2. The very same behavior exists with the 256-bit version of this instruction, too: the core only switches to level 1 when unrolled. We can imagine two different factors that could potentially influence this decision: the load on the core's AVX units and the register utilization itself.

Before the first frequency reduction from level 0 to level 1 happens, we find that the instruction throughput (instructions per cycle, IPC) of the 512-bit variant is precisely doubled from 0.21 to 0.42 on average with the unrolled version compared to the non-unrolled implementation, as depicted in Figure 3.6.

This is expected: as described in Section 3.3.2, the CPU we used for our tests features two AVX-512 units per core, and as such, when no register pressure prevents a core from parallelizing consecutive instructions, it can make full utilization of both units. However, we also found that the cores always run roughly equal amounts of instructions through both units, even in the non-unrolled case. Most likely, Intel's scheduler uses a simple round-robin algorithm to assign micro-instructions to the units. This is a sign that the load on the

units is not the determining factor: in the non-unrolled case the units take turns and each one stays unloaded only for a few cycles at a time.

In addition, the theory of the register pressure being the culprit here is supported by a patent on local power gating in processors published by Intel [7]: here, Intel describes a technique to dynamically cut and restore power to vector units as well as vector registers upon demand to save energy, which our system's processor likely implements as described. Notably, execution units and registers are controlled independent of each other and it is also noted that the `vzeroupper` instruction directly impacts the power gating behavior. This instruction zeroes the upper 384 bits of each architectural 512-bit vector register [19]. Indeed, if we explicitly set the 512-bit register `ZMM0` to a non-zero value before starting execution, we find that heavy 256-bit (i.e., AVX2) vector instructions – which would normally only cause switches to level 1 – suddenly trigger frequency switches to level 2, too, but only if we do not additionally execute `vzeroupper` after setting `ZMM0`. This is not documented in Intel's description of the reclocking algorithm, but hints that register usage directly impacts the turbo license level selection in addition to the types of the executed instructions.

Apart from this discrepancy, we found that Intel's description of the instruction types and their associated turbo license levels holds true for the instructions we selected for testing. Now that we know the steps for frequency reductions, the next logical step is to find out when each of them occurs, what is required to trigger them and how much time a core spends in each state.

In Section 3.2.5, we described the implementation of the `PRE_THROTTLE_TIME` measurement mode, designed to find out how many AVX instructions may be executed before the out-of-order engine is throttled. Additionally, we wanted to use it to measure what throughput could theoretically be achieved if there was no frequency reduction at all.

The answer here, however, is very simple: in all tested cases, the throttling occurs immediately after execution of the first instruction has completed – no matter what instruction is tested, how many cores are used or whether pre-throttling and turbo frequencies are enabled. This also means that we are unable to measure the theoretically achievable throughput here: with a duration of just one instruction, no reliable numbers may be obtained.

Next, we are interested in the amount of instructions required to eventually trigger a switch to level 1. This is what the `REQUIRED_INSTRUCTIONS` measurement mode was built for: it incrementally builds and executes AVX code with more instructions until a license level switch can be observed within a time window after an execution iteration.

Here, we indeed found interesting differences between different instruction

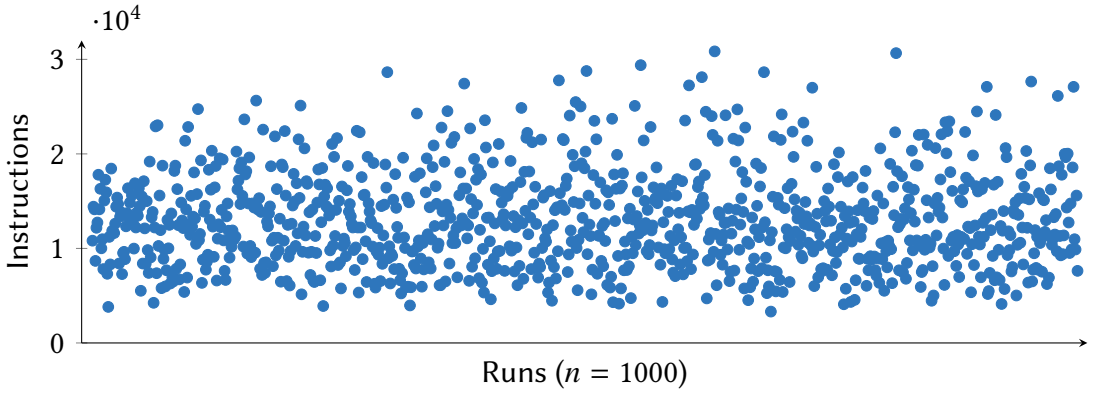


Figure 3.7: Required 256-bit `vfmaddsub132pd` instructions to trigger a turbo license switch to level 1. Unlike with 512-bit instructions, the amount varies a lot.

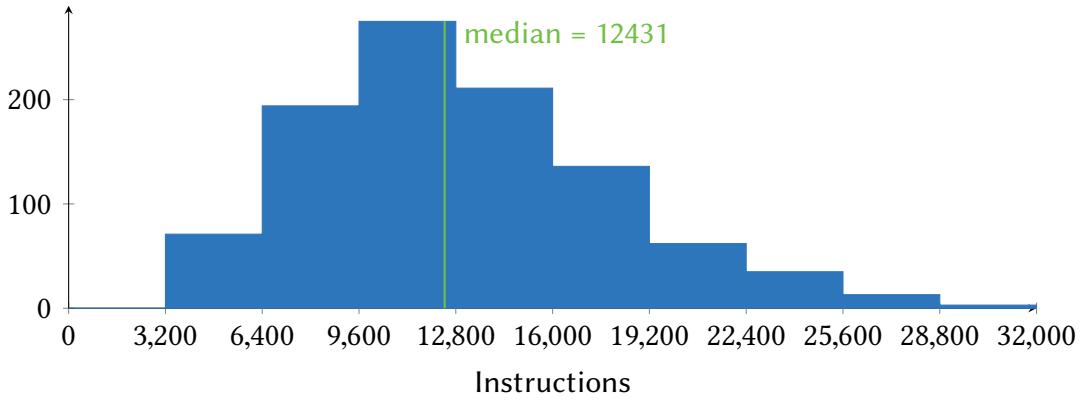


Figure 3.8: Histogram of the data depicted in Figure 3.7. The distance between median and maximum is much larger than between median and minimum.

types: all 512-bit instructions trigger a switch to level 1 after exactly one executed instruction. For 256-bit instructions, however, a quite different picture emerges: Figure 3.7 depicts the required amount of instructions exemplary for the 256-bit `vfmaddsub132pd` unrolled case, without pre-throttling and with one core executing AVX instructions. The result varies between a minimum of 3317 and a maximum of 30845 instructions while the average and the median are set rather near each other, at 12982.8 and 12431, respectively. Figure 3.8 shows the same data, plotted as a histogram. It becomes clearly visible that the median is far nearer to the minimum than it is to the maximum.

We find that average and median values rise when executing the test with more cores. While this may seem surprising at first glance, it is expected: when

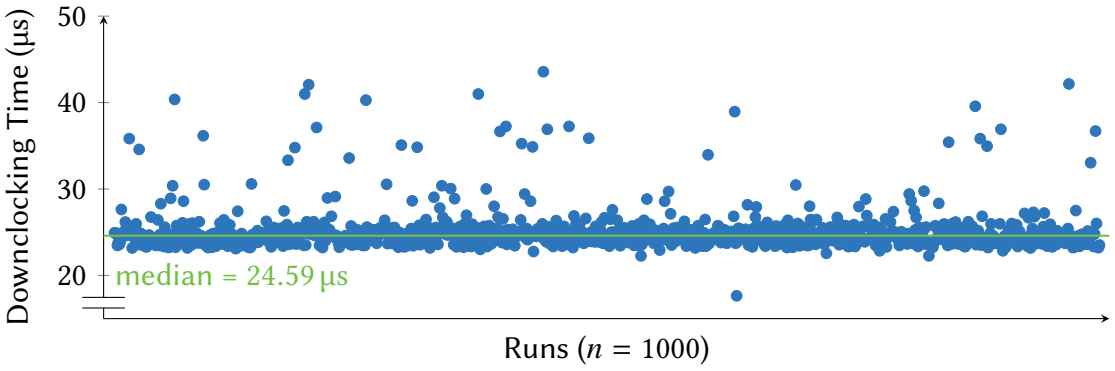


Figure 3.9: Downclocking time to level 1 for the 512-bit `vfmaddsub132pd` instruction. The results are very homogeneous around a median of 24.59 µs.

run with multiple cores, the test only ends if a frequency switch is triggered on all cores. Therefore, given the high variance of the results, it becomes less likely that all cores reach a frequency switch with less instructions.

We implemented pre-throttling mode to be able to test what happens when additional load is created when a further core is utilized in an environment where other cores are already loaded, contrary to the default mode where the AVX workload is started on all cores at the same time. However, pre-throttling does not seem to have any effect that can not be attributed to statistical noise. Nevertheless, disabling turbo frequencies does have one: we can still observe a high variance within the results, but all statistic measures are *lower*: with a single core, we find a minimum of 527, a maximum of 21793, 8642.49 on average, and a median of 8137.5. This is intriguing as one would expect a frequency drop to be less necessary when starting off a lower base frequency. We can not be sure of an explanation for this, however, this observation may hint that voltage stability is a crucial factor here: in the previously cited patent [7], Intel notes that a voltage drop may occur due to falling electrical resistance (and in turn, rising current) upon powering the vector units and that a detector for this situation is in place. Given that, at a lower frequency, the core also runs at a lower voltage, it seems plausible that the voltage drops below the critical threshold earlier as the subtracted resistance is the same regardless of voltage and frequency.

As with these results we have fully established all conditions required for AVX-induced frequency reductions to level 1, we are now interested in the time required for the actual switch (i.e., the time between starting execution and the moment the new frequency is applied). Again, we find 512-bit and 256-bit instructions to have noticeable differences. For example, with 512-bit `vfmaddsub132pd` instructions executed on a single core, all results are very

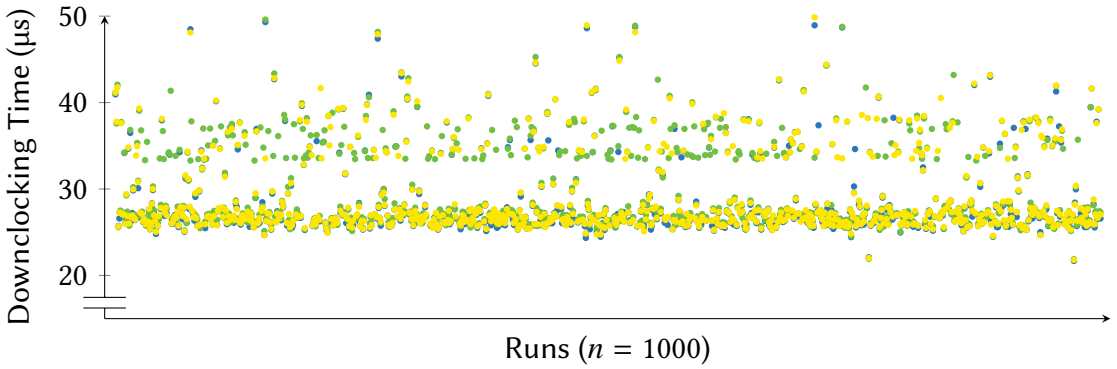


Figure 3.10: Downclocking time to level 1 for the 512-bit `vfmaddsub132pd` instruction when executed on three cores (each color represents a specific core). Compared to Figure 3.9, median and standard deviation are higher.

homogeneously distributed around a median of 24.59 µs with only a few outliers, as depicted in Figure 3.9. There is no statistically relevant difference with other 512-bit instructions. However, median and deviation both rise with multiple cores: for example, with three cores (Figure 3.10), we find the average of the median across all three cores to be at 27.39 µs – an increase of nearly three microseconds. Whereas the standard deviation with only one core is very low at 0.0025 µs, it quintuples with three cores and amounts to an average of 0.013 µs. Notably, this increase only happens with pre-throttling mode disabled, i.e., when only one core switches its license level, whereas the others are already running in a level with reduced frequency. This is interesting because it tells us that the frequency is not the determining factor here: the maximum turbo frequency of a single core depends on the available electrical power budget as well as on how many cores are under load, and thus with three cores each core runs at a lower frequency, compared to when only one core is active. If, however, the increase is not visible with pre-throttling – i.e., when two cores are already at level 1 – the lower frequency can not be at fault for the increased latency. A simple and plausible explanation could be that the PCU requires more time to make its decision when more license requests are pending.

Looking at 256-bit instructions, we find that the downclocking time to level 1 is still very homogeneous across all runs, albeit a lot higher than in the 512-bit case. As shown in Figure 3.11, with 256-bit `vfmaddsub132pd` instructions executed on one core, the median is at 51.52 µs – more than doubled compared to the 24.59 µs of the 512-bit variant. The results do not correlate with the amount of instructions required to cause the frequency reduction previously depicted in Figures 3.7 and 3.8, which makes it seem likely that the difference in timing is induced by an algorithmic difference in the implementation.

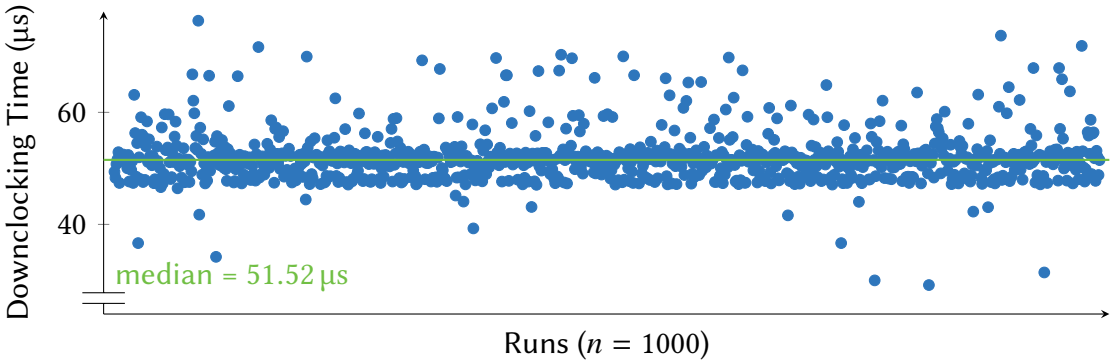


Figure 3.11: Downclocking time to level 1 for the 256-bit `vfmaddsub132pd` instruction. This takes a lot longer than with the 512-bit version.

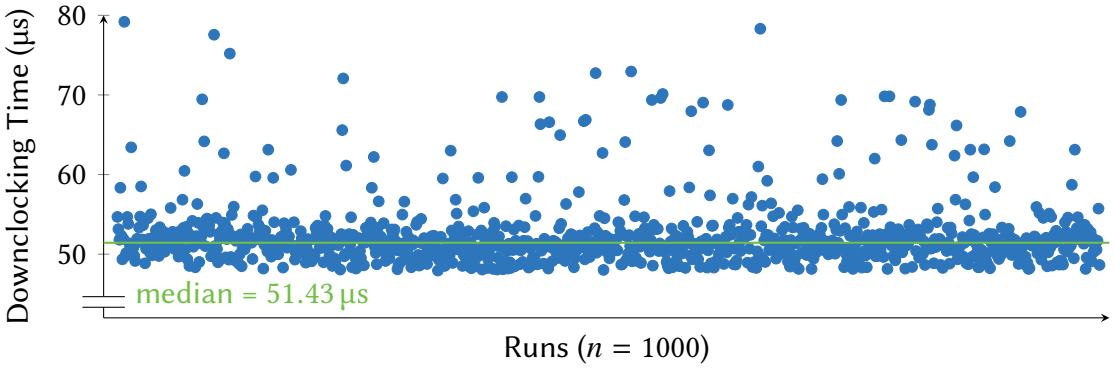


Figure 3.12: Time taken from level 0 to level 2 with the `vfmaddsub132pd` instruction. Again, the results are very homogeneous.

So far we have described the frequency reduction process until level 1 is reached. For cases that target this level (i.e., everything apart from AVX-512 heavy instructions with the notable exceptions outlined above), nothing further happens until the load that induced the license level switch ceases and the frequency is brought back to its previous level again, as described later in Section 3.3.4.

Our findings about the second turbo license switch from level 1 to level 2 (where applicable) did not yield any surprises: in general, the behavior is similar to what happens during the switch from level 0 to level 1.

Figure 3.12 shows the time needed to reach level 2 using the unrolled `vfmaddsub132pd` instruction as an example. We find the data to be homogeneous with only a few outliers, similar to previous results. The median is located at 51.43 µs, however, note that this also includes the time taken from level 0 to

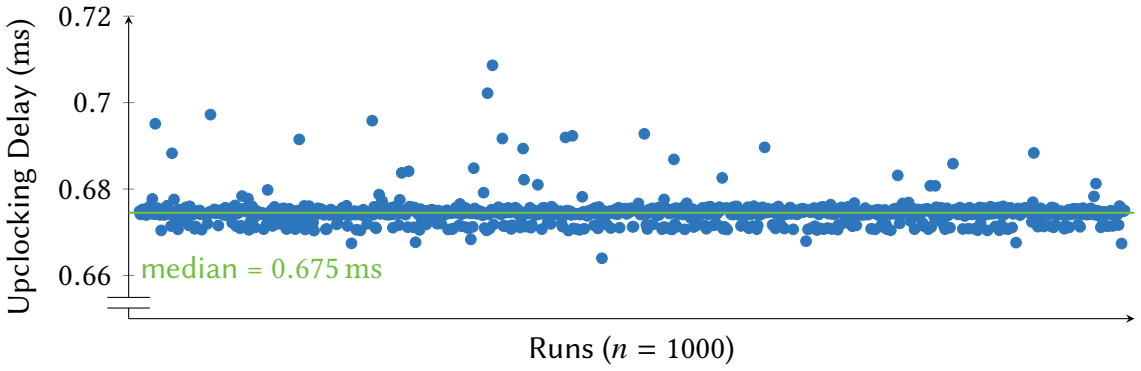


Figure 3.13: Upclocking times after executing 256-bit `vfmaddsub132pd` instructions until level 1 is reached. Results show the upclocking delay to uniformly be around $\frac{2}{3}$ ms.

level 1. By subtracting the median of the transition to level 1 (24.59 µs), we can deduce it takes 26.9 µs from level 1 to level 2, which is only slightly longer. For multiple cores and pre-throttling mode, the general behavior and the increases with multiple cores are about the same. This fits our theory that the PCU takes longer to make decisions with multiple license transition requests pending.

3.3.4 Upclocking

After a core's clock is reduced due to a license level transition, it runs at the lower frequency until no more heavy instructions are being executed. However, the frequency can not be raised immediately in order to avoid wasting time with too many frequency switches, and thus the core keeps executing further instructions at a lower speed for a while – this is what essentially causes the performance issue for heterogeneous workloads that motivated this work. Further, the upclocking part of the reclocking algorithm is the one where it is most likely to find room for possible optimizations. Therefore, the process of raising the frequency (actually, reverting the reduction) deserves particular attention. According to Intel, as cited in this chapter's introduction, the processor generally delays increasing the frequency again by about 2 ms. To verify this claim, we used our framework's `UPCLOCK` measurement mode, which executes AVX instructions until a license level transition occurs on a given core and then keeps the core spinning in a scalar loop until it switches back to level 0.

Again using the results from the `vfmaddsub132pd` instruction as example, we find that the upclocking behavior differs between several test configurations. For the 256-bit unrolled version executed on a single core with pre-throttling

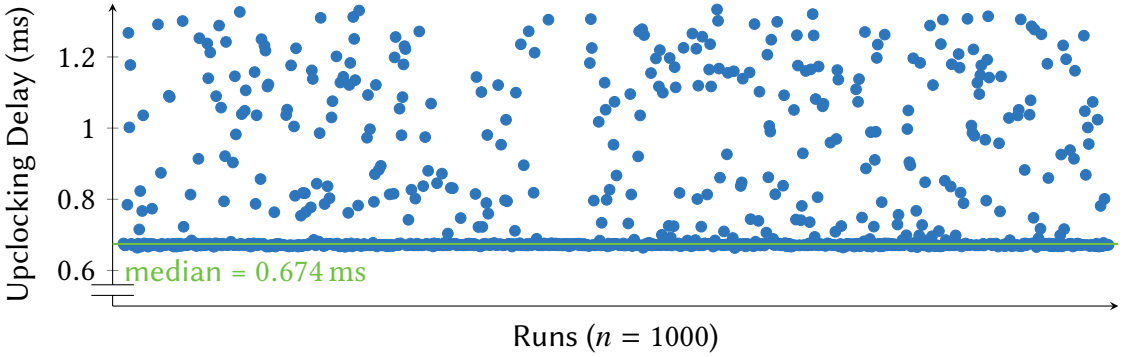


Figure 3.14: Upclocking times after executing 512-bit `vfmaddsub132pd` instructions until level 1 is reached. While most runs still yield a time of $\frac{2}{3}$ ms, some are scattered within a range up to $\frac{4}{3}$ ms.

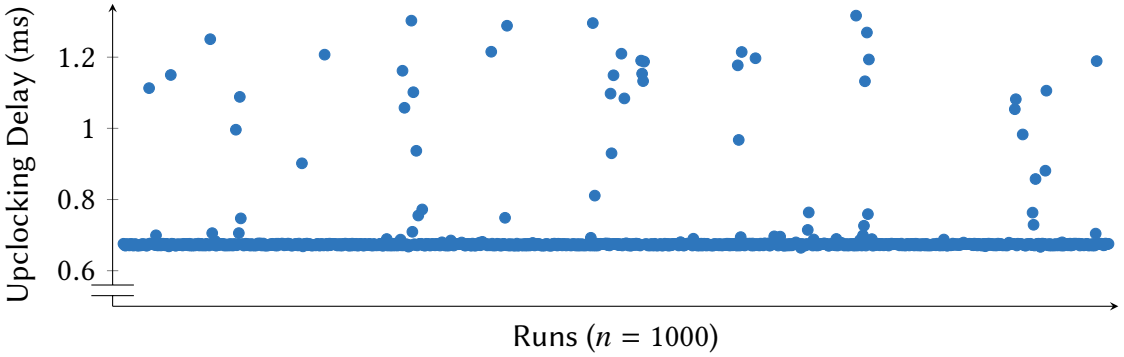


Figure 3.15: Upclocking times after executing 512-bit instructions until level 1 is reached with two cores in AVX pre-throttling mode. Compared to Figure 3.14, where only one core was active, the variance is a lot smaller.

disabled and targeting level 1, we get the results depicted in Figure 3.13. These are very uniformly distributed around a median of 0.675 ms. Notably, this is suspiciously near to $\frac{2}{3}$ ms. Most likely this is the value Intel tried to approximate.

Looking at the very same instruction in its 512-bit variant under the same test conditions in Figure 3.14, a different picture emerges: while the median is still nearly the same at 0.674 ms, the maximum is at 1.333 ms – about $\frac{4}{3}$ ms. However, when going to level 2, all runs are again homogeneously distributed around $\frac{2}{3}$ ms. The results are mostly the same when executed with multiple cores, save the notable exception of pre-throttling mode: in Figure 3.14, 69.4 % of the results are below 0.7 ms. With two cores and AVX pre-throttling enabled, as graphed in Figure 3.15, this applies to 94.7 % of the runs. Similar results are obtained with more cores.

Chapter 4

Design

The ultimate goal of this work is to find optimizations to the reclocking algorithm implemented in Intel CPUs for AVX-induced frequency reductions. In Chapter 3 we have conducted a thorough analysis of the implementation in order to be able to conceive possible optimizations. Concretely, ideas for such algorithms may be derived from what is done in power management: we have described how the problems and possible gains of AVX reclocking resemble the ones of power management in Section 2.3. The strategies employed there, such as profiling and the use of statistic models, and previous research on these approaches may be useful to develop a AVX reclocking mechanism that can improve performance for heterogeneous workloads. Any ideas, however, would remain purely theoretical unless we are able to implement and experimentally test them. In this chapter, we present AVXFREQ, a modified version of the Linux kernel's DVFS driver for Intel processors that tries to mimic Intel's hardware-based reclocking behavior. We can then use this *reimplementation* to explore, implement and evaluate optimization approaches.

4.1 Reimplementation

To be able to implement alternative AVX algorithms, we want to reproduce Intel's AVX reclocking mechanism in software. This means that we want to take over control over AVX-induced frequency reductions from the hardware. To achieve this goal, we have built AVXFREQ, a reimplementation based on a model obtained through the results of the analysis we performed in Chapter 3. AVXFREQ counts retired AVX instructions over time periods to govern the frequency, therefore, akin to our analysis framework, AVXFREQ makes use of the processor's PMU, the features of which we outlined in Section 3.2.1. This section describes the pre-existing kernel driver our reimplementation is based

on, the modifications we made to it and how exactly we use the PMU to reach our goals.

4.1.1 The intel_pstate Driver

AVXFREQ was implemented by modifying Linux's `intel_pstate` module as it is found in version 5.1.0 of the kernel. This driver is part of the *`cpufreq`* subsystem which contains all of the kernel's DVFS drivers and provides generic policies that govern the frequency selection based on system load and the user's wishes [42].

On modern Intel CPUs with support for Hardware-Controlled Performance States (HWP) (as described in Section 2.2), the `intel_pstate` driver usually does not do much: it enables HWP using the processor's `IA32_PM_ENABLE` MSR, reads the `IA32_HWP_CAPABILITIES` MSR to determine the available P-states on each core and then only manages the `IA32_HWP_REQUEST` MSR to hint the hardware about changes of the user's preferences on, for example, minimum and maximum performance [40] [43]. When HWP is enabled, the operating system loses its ability to precisely manage the processor's frequency. However, using HWP is not a strict requirement – as long as HWP is not explicitly activated by the computer's UEFI or the operating system, traditional software-based performance scaling remains entirely possible. For users desiring to retain the pre-HWP behavior, the `intel_pstate=no_hwp` flag exists. When this flag is passed to the Linux kernel's command line during startup, `intel_pstate` will not enable HWP, and thus is still able to exercise complete control over the processor's speed. This is an important corner stone for AVXFREQ: we need to be able to fully govern P-state selection in order to switch frequencies by ourselves in response to changing AVX load conditions.

4.1.2 AVXFreq

To obtain a minimal working prototype that allows us to evaluate the general feasibility of a software-based approach to AVX reclocking optimization, we chose to implement a simplified version of Intel's algorithm that we derived from the CPU's behavior when executing 512-bit `vfmaddsub132pd` instructions on a single core, as obtained from the analysis in Chapter 3. For this instruction type, we found that a single instruction is sufficient to trigger a frequency reduction. After approximately 25 µs, level 1 is reached, and – in the case of the unrolled version – about 27 µs later, the core runs at level 2. As soon as no heavy AVX instructions have been executed anymore for $\frac{2}{3}$ ms, the processor reverts the frequency reduction and returns to level 0. The instruction type we want to support was not chosen arbitrarily: AVXFREQ relies on using performance

events for measuring executed AVX instructions and Intel processors currently only offer such events exclusively for floating-point vector instructions [36].

For reasons of practical convenience, AVXFREQ was implemented to be opt-in: the `intel_pstate=avxfreq` flag must be passed on Linux's command line, otherwise `intel_pstate` behaves just like it normally would in its unmodified version. Note that this flag also implies `intel_pstate=no_hwp`, i.e., it also disables HWP for the reasons explained above.

To manage license levels, we added a per-core state variable that contains the virtual license level and adapted the method that communicates the selected P-state to the processor to offset the P-state according to the frequency reduction that would be induced by the hardware in this license level. This way, the driver's frequency selection mechanism stays unmodified and fulfills its tasks unaware of and unaffected by our AVX reclocking. To ensure that changes to the virtual license level immediately take effect, we invoke a recalculation every time, and thereby an update of the target P-state.

During the boot process, Linux's `perf` subsystem configures performance interrupts to be delivered as non-maskable interrupts (NMIs) [30]. In our analysis framework, we wanted to minimize overhead caused by interrupt handling to keep our workload as free from non-AVX code as possible, and therefore implemented a custom interrupt handler. Here, however, we consider the latency and overhead induced by Linux's NMI handling process negligible, hence we opted to keep the NMI configuration here, but override `perf`'s handler and register our own during module initialization. This implies that, in the same way as when using our analysis framework, `perf`'s functionality is degraded when AVXFREQ is enabled.

Further, during initialization we configure a performance counter on each of the CPU's cores to count `FP_ARITH_INST_RETIRED.512B_PACKED_DOUBLE` events in both user-mode and kernel-mode and to trigger an interrupt as soon as a single assertion has occurred. These events are asserted every time a 512-bit packed double-precision vector micro-instruction has completed execution [40].

When our NMI handler is called, and thus as soon as a vector instruction as selected above was executed, the handler sets the virtual license to level 1 and reconfigures the PMU to count further assertions of the previously selected event without triggering more interrupts. Secondly, a 100 µs timer is configured using Linux's `hrtimer` subsystem which provides precise timers with nanosecond resolution for kernel-internal purposes. After the timer has elapsed, we read the previously configured performance counter to check whether any of the instructions we are looking for have been executed since the timer was started. If yes, and if the average throughput equals at least 1 operation per cycle, we

switch to virtual license level 2 – this is to approximate the differing behaviors of the unrolled and non-unrolled versions as described in Section 3.3.3. The performance counter is then reset and the timer is restarted to trigger again after another 100 µs. Otherwise, as long as none of the monitored 512-bit vector instructions were executed, we start counting the total elapsed time without them, again in 100 µs steps, and we reset the counter as soon as the performance counter rises above zero. Like in the first case, we reset the timer for another 100 µs unless 600 µs have already been elapsed – then, we only wait for 66 µs to resemble the upclocking results from our analysis. As soon as this last timer is over we revert back to level 0 and reset to our initial state as configured during system boot so that we can run through the same procedure again when heavy instructions are being executed once more.

This way, AVXFREQ is capable of fulfilling what we outlined at the beginning of this section: we switch to frequencies equivalent to turbo license levels 1 and 2 according to the load and revert back to level 0 as soon as the load situation was alleviated for a continuous time period of about $\frac{2}{3}$ ms. Note that we did not implement any artificial delays during the downclocking as we assume the delays we observed in our analysis to equal the time required by the hardware to complete the frequency reduction. Furthermore, the switch from level 1 to level 2 only occurs after a delay of 100 µs and not immediately as implemented in the hardware. This is because we use the throughput to decide whether to transition into level 2 – and that requires some time for a meaningful measurement. We probably could achieve an improved approximation by using shorter delays at the expense of system performance. A possible trade-off could be to only make the first delay shorter. However, that remains future work for now.

In general, it should not be forgotten that this implementation only tries to reflect a subset of the results from our analysis in Chapter 3. Some parts are not possible to reimplement, e.g., because there are no performance events available specifically for integer vector instructions [40]. For other parts, we deliberately decided not to implement them for now in order to reduce complexity – what we built should be enough for a basic evaluation of our approach.

Note that for AVXFREQ to work properly, disabling or at least reducing AVX reclocking in hardware is a necessary prerequisite as the processor will forcibly reduce the frequency when executing demanding AVX instructions even when HWP is not enabled. Certain motherboards allow controlling the frequency offsets used for the different license levels via their UEFI configuration, and thus the use of such a motherboard is a strict requirement for AVXFREQ.

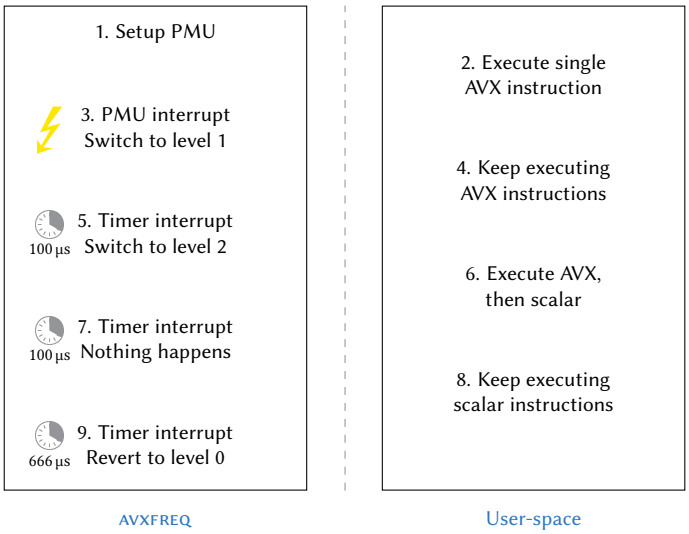


Figure 4.1: Simplified exemplary license level switching process with AVXFREQ and a user-space process that executes AVX instructions for a while before continuing with purely scalar instructions.

4.2 User-Space-Driven Decisions

In the previous section we have described AVXFREQ, our reimplementation that implements a subset of the AVX reclocking algorithm employed by Intel CPUs. This work is intended to lay a foundation that may be used for building and evaluating possible optimizations of the DVFS mechanism without needing to modify the hardware itself. In this section, we present the implementation of a simple approach where user-space is given control over the license levels, based on the idea that a program with a heterogeneous workload knows best when it is going to execute AVX instructions and when it is in a scalar-only stage. This approach is especially useful as it is presumably capable of providing us with a theoretically optimal baseline for AVX reclocking that can be used to evaluate the quality of other algorithms. For this purpose, we have extended AVXFREQ with a user-space interface consisting of multiple system calls:

- `int avxfreq_is_enabled(void)`
  Returns 1 if AVXFREQ was enabled during system boot, 0 otherwise. Only when AVXFREQ is enabled, user-space may use the other system calls for managing software-based reclocking and license levels.

- `int avxfreq_set_reclocking(bool reclock)`
  Can be used to enable and disable AVXFREQ's own reclocking mechanism.

When disabled, user-space must take caution to always reduce frequency when required. Otherwise, system stability may be at risk. Note that it is enabled by default when AVXFREQ is enabled.

- `int avxfreq_set_license(unsigned char license)`
  Invokes a license level transition to the level passed in the `license` parameter (either 0, 1, or 2).

While this interface is very simple, it allows to do exactly what we wanted to: a user-space program may use these methods to change the applied license level at will, and thus choose appropriate license levels when it knows what instructions it is going to use over periods of time.

Chapter 5

Evaluation

We have presented the design of AVXFREQ, a software-based reimplementation of a partial model of Intel's AVX reclocking algorithm, in the previous chapter. Further, we have described how AVXFREQ can be leveraged to allow user-space programs to choose the applied turbo license levels themselves. In this chapter, we will evaluate these implementations and measure how well our prototype reflects Intel's hardware implementation. For this purpose, we want to leverage the analysis framework we presented in Chapter 3. However, both AVXFREQ and our analysis framework need to make use of the PMU and can not do so concurrently. Further, to evaluate AVXFREQ, the analysis system needs to obtain the virtual license levels from AVXFREQ instead of reading the license levels via the PMU. For this reason, we will present modifications to AVXFREQ itself as well as the analysis tool that allow us to use the latter in order to compare AVXFREQ's behavior to what Intel does and to the simplified algorithm we wanted to reflect. For the user-space-driven decisions we will describe the design of a simple program that simulates heterogeneous workloads with both scalar and AVX instructions. Finally, we will present and discuss the results obtained from executing tests with these tools.

5.1 Methodology and Design

One of the aims of this chapter is to show how well AVXFREQ implements the model it is supposed to reflect. First, we want to verify whether frequency reductions are triggered in the right situations, i.e., a switch to license level 1 happens when a single 512-bit floating-point vector instruction was executed, and a second switch to level 2 is triggered when, on average, at least one of these instructions is executed over a period of 100 µs. Second, as soon as none of the monitored vector instructions were executed over period of $\frac{2}{3}$ ms,

AVXFREQ should revert to license level 0. As described in Section 3.2.5, the `DOWNCLOCK`, `UPCLOCK`, and `REQUIRED_INSTRUCTIONS` measurement modes of our analysis framework are capable of measuring these values. For this reason, we will modify our analysis framework and AVXFREQ to talk to each other, so that, in turn, we can leverage the analysis system.

The second aim of this evaluation is to measure the performance penalty incurred by AVXFREQ compared to Intel's hardware and implementation and to explore whether an improvement for heterogeneous workloads is generally possible with alternative AVX reclocking governors. For this reason, we presented an interface in Section 4.2 that allows for optimal reclocking by handing control over license levels to user-space. To conduct our evaluation, we will build a tool that alternates between phases of configurable time periods consisting of either AVX or scalar code. Using this tool, we will measure the throughput in an AVX phase as well as an immediately following scalar phase with a length of 666 µs – this is approximately the delay applied before switching from license levels 1 or 2 to 0 as shown in our analysis, hence this should generate worst-case results with hardware reclocking. We then compare results obtained with the hardware implementation, with AVXFREQ reclocking, and with user-space-governed license level switches.

5.1.1 AVXFreq

We want to compare performance counter values using our analysis framework from Chapter 3 – in the best case, running it with AVXFREQ should deliver similar results to what is seen with hardware reclocking. However, our framework relies on using the processor's PMU (as described in Section 3.2.1) just like AVXFREQ does, and due to the limited amount of available performance counters they may not use the PMU concurrently. Further, hardware-side performance events that count cycles a processor's core spends in a specific turbo license level do not make sense anymore when AVXFREQ is enabled as the license levels are now emulated by software.

To circumvent these obstacles, we adapted AVXFREQ and our analysis framework to work together to emulate the most important performance events in software: the idea is that AVXFREQ already uses some of the performance events that the analysis system would need anyway and that it is perfectly capable of counting cycles spent in different states. Therefore, we let AVXFREQ do the bookkeeping and provide our framework's kernel component with software-generated events instead of interrupts in a manner that is completely transparent to the user-space part.

AVXFREQ in itself only needs one programmable performance counter to count retired 512-bit floating-point double-precision packed vector operations

in addition to a fixed counter to count core cycles. For this purpose, AVXFREQ keeps raw counters in software that are updated using the values provided by the PMU whenever any event triggers AVXFREQ code to be run. As AVXFREQ exclusively manages these counters, it needs to provide them to the analysis framework's kernel module – other counters may be managed by the module itself. To enable interaction between the analysis framework's kernel module and AVXFREQ, we defined and implemented the following kernel-internal interface:

- `bool avxfreq_is_enabled(void)`
  Returns `true` if AVXFREQ was enabled during system boot, `false` otherwise. Only if AVXFREQ is enabled, the analysis framework needs to behave differently than in Chapter 3.

- `avxfreq_counters *avxfreq_get_counters(int cpu)`
  `avxfreq_counters` is a C language `struct` that contains the raw counters for the values defined above. One instance is defined for every core in the system. Given a core number, this method returns a pointer to the instance for the respective core.

- `void avxfreq_reset_cycle_counter(void)`
  This method resets the current core's cycle fixed counter.

- `void avxfreq_set_license_transition_listener (void (*listener)(u8 from, u8 to))`
  Using this method, the analysis kernel module can hook into AVXFREQ and receive notifications whenever the applied virtual license level changes. The argument provided is a pointer to a function that takes two arguments: the previous virtual license level and the new one.

Using this interface, we modify our kernel component as follows: when loaded, it calls `avxfreq_is_enabled()` to check whether AVXFREQ is active. If so, it will not reconfigure the APIC to take over handling of PMU interrupts, but rather call `avxfreq_set_license_transition_listener()` to receive updates from AVXFREQ about license level switches and `avxfreq_get_counters()` once for every CPU core in the system to locally store pointers to all counters provided by AVXFREQ. In addition to the raw counters, we added a local structure that contains derived counters, e.g., AVXFREQ only counts core cycles since they were last reset, but we also need to be able to distinguish cycles spent in the different license levels.

Whenever the module needs to write to PMU MSRs after being instructed to do so by the user-space component (e.g., during the `SETUP ioctl()`), it stores

updated performance counter configurations in an array, so that it can later map them to software counters from AVXFREQ. Direct writes to performance counters are remapped to the corresponding software counters, unless there is no software counter available for them – in this case, they are routed to the MSRs just like when AVXFREQ is disabled. Similar action is taken when trying to read performance counters. This way, PMU configuration remains fully transparent to user-space and hardware counters are automatically remapped to software counters when required without need for further intervention.

Interrupt handling is, as previously mentioned, disabled in case AVXFREQ is enabled. Instead, when the `avxfreq_license_transition_listener` is called, it will update all local software counters with the values currently set in AVXFREQ's counters and trigger the interrupt handler if the license level transition reflects one for which user-space requested an interrupt. Again, using this mechanism, we substitute hardware interrupts with software-based events in a manner that is transparent to user-space.

The interrupt handler itself (which now does not necessarily handle *interrupts*) only needs a very small modification to work in this scenario: it must not reset the interrupt state in hardware, both for the PMU and the APIC. Apart from that, all interrupt actions we defined in Section 3.2.3 work the same way as before.

As shown, these modifications allow us to fully employ our analysis framework with all its features and without drawbacks in combination with AVXFREQ enabled.

5.1.2 Multi-Phase Execution

In this section, to create worst-case scenarios that we can use to measure the maximum negative impact of AVX reclocking on heterogeneous workloads, we will present a tool that allows to arbitrarily simulate such workloads. Further, we can use it to evaluate any improvements achieved with modified reclocking algorithms, e.g., the user-space-driven decisions we presented in Section 4.2 – note that the worst case scenario for hardware reclocking would equal the best case for user-space-governed reclocking in terms of theoretically possible improvements. In addition, the tool may be employed to measure the impact of AVXFREQ on performance in comparison with the hardware implementation in Intel processors. We can not use real-life workloads as they would most certainly make use of instruction types that are not considered by AVXFREQ's reclocking mechanism, and second, because likely none of them would expose worst-case behavior.

The idea is to executes multiple *phases* of arbitrary duration, where each phase represents a different load type: scalar, AVX instructions targeting turbo

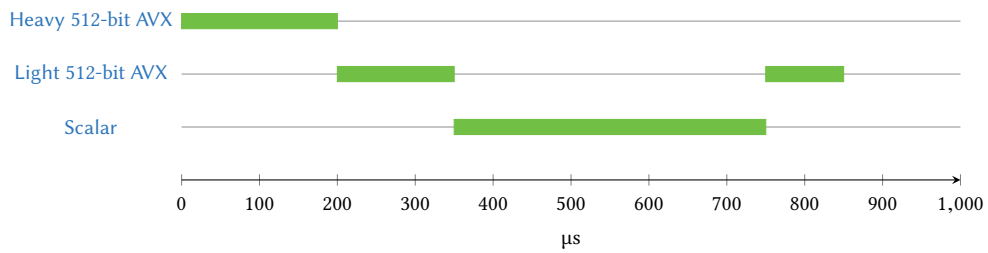


Figure 5.1: Exemplary run of our multi-phase execution tool when called with `200 150 400 0 100` as command line. Heavy 512-bit AVX instructions are executed for a duration of 200 µs, then light ones for 150 µs, then purely scalar instructions for 400 µs, and finally a last phase with heavy 512-bit AVX again for another 100 µs.

license level 1, and AVX targeting level 2. We use 512-bit FMA instructions (`vfmaddsub132pd`, precisely) for the two latter phase types, where the level 1 variant uses code that exhibits register pressure to limit throughput, whereas the level 2 variant is designed to achieve full utilization of the available AVX execution units. Scalar phase only use simple increment and jump instructions. This was not an arbitrary choice: AVXFREQ, by design, only tries to resemble the hardware's implementation for 512-bit FMA instructions.

The user may pass an arbitrary amount of arguments to the program, each representing a phase and how long it should be executed in µs. For example, the first argument represents a run of a scalar phase, the second one a level 1 phase, the third one a level 2 phase, the fourth one a scalar phase again, and so on. Our implementation then runs each of the passed phases one after another for the given duration. During startup, the program spawns a separate thread and binds it to a specific core to ensure the process scheduler does not migrate the thread between cores and enables the `SCHED_RR` scheduling policy to engange real-time execution – just like our analysis framework's user-space component (see Section 3.2.4). This thread then executes each phase by running a loop where each iteration contains a single instruction corresponding to the phase type as described above. We enforce the configured time windows for each phase by having the main thread sleep for this duration and notify the executing thread after it has passed. Performance is then measured by counting how many iterations of the loop are executed within the phase's time window.

Additionally, in order to implement the user-space-driven decisions as mentioned above, we have added a command line option which instructs the tool to make use of the interface we presented in Section 4.2 to set license levels according to the phase that is to be executed.

5.2 Results

For our evaluation we used the same system that we previously described in Section 3.3.1 with an Intel Core i9-7940X processor installed on an ASUS TUF X299 MARK 2 motherboard. However, to make AVXFREQ work correctly, we needed to modify some settings in the system's UEFI configuration: first, we had to disable Intel HWP as the firmware enables it by default, whereas it is necessary to have HWP disabled for AVXFREQ to allow it to exercise full control over the processor's P-states as outlined in Section 4.1.1. Second, while it is not possible to disable AVX-induced reclocking entirely, we have set the reclocking offsets to 1 for both license levels to minimize the impact by the hardware's implementation – as pointed out in Section 4.1.2, this is necessary for AVXFREQ to work properly and this is the reason we chose that specific motherboard. This, though, also means that with AVXFREQ enabled, there are two reclocking responses to the execution of AVX code: by the processor itself and by AVXFREQ. While the resulting final frequencies still equal the ones reached with unmodified hardware reclocking (i.e., 3.4 GHz for level 1, and 2.8 GHz for level 2), frequency changes potentially happen twice.

In this section, we will present the results obtained by executing tests using the modified AVXFREQ and analysis framework implementations as well as the multi-phase execution tool we presented in the previous section.

5.2.1 AVXFreq

As discussed in Section 3.2.5, our analysis framework provides multiple measurement modes: `DOWNCLOCK` to measure the time taken until a frequency reduction is completed, `UPCLOCK` for the reverse process, `PRE_THROTTLE_TIME` to determine the time before a core is throttled after AVX execution has begun, and `REQUIRED_INSTRUCTIONS` to find out how many instructions exactly are required to trigger the reclocking. To verify how well AVXFREQ implements the simplified reclocking algorithm we wanted to reflect, we executed tests using all these modes with AVXFREQ and the above-mentioned system configuration, with the exception of the `PRE_THROTTLE_TIME` mode: as the throttling is purely controlled by the hardware itself and not affected by any system configuration or AVXFREQ, it makes no sense to run this mode again here. Further, AVXFREQ is designed to only resemble the CPU's behavior when executing 512-bit FMA instructions, with turbo frequencies enabled, and with only one active core, and as such, we only ran tests under these specific conditions.

In the case of transitions from level 0 to level 1 measured using the `DOWNCLOCK` and `REQUIRED_INSTRUCTIONS` modes, there are no surprises: AVXFREQ was designed to immediately switch to level 1 as soon as a single 512-bit double-

precision floating point vector instruction is retired and we find this to essentially hold true. The amount of completed operations in the `DOWNCLOCK` test for the unrolled case (i.e., when both AVX-512 units of the core are utilized in parallel) varies between 18 and 34 (equaling 9 and 17 instructions, respectively, as a FMA instruction consists of two operations) before AVXFREQ invokes and reports the transition to level 1. Given that the hardware implementation reacts precisely after the first instruction there is a small deviation here, however, we believe this is caused by PMU interrupts being slightly delayed and consider it to be negligible. This assumption is supported by the fact that the `REQUIRED_INSTRUCTIONS` test generally shows one single instruction to be sufficient to trigger a level transition.

Transitions to level 2 examined using the same measurement modes, however, show a larger deviation between the hardware's behavior and the our results: the median of the time needed here lies at 103.8 µs, whereas Intel's implementation will immediately start a switch to level 2 after the first heavy instruction was executed in level 1. Looking at how AVXFREQ was implemented, though, this is not surprising: after having moved to level 1, AVXFREQ will trigger interrupts every 100 µs to check whether any of the measured instructions were executed in meantime and then either triggers a switch to level 2 if the throughput is high enough or counts towards the upclocking timer if no instructions were executed. Thus, a transition from level 1 to level 2 may only occur at least 100 µs after the transition from level 0 to level 1. Delays caused through context switches and kernel code could provide a plausible explanation for the remaining ~4 µs.

Tests using the `UPCLOCK` mode show that AVXFREQ behaves as designed: the average and median times taken to revert a frequency reduction lie at about 665 µs, which is slightly below the 666 µs we were aiming at. All results are distributed very homogeneously around this value. Again, we attribute the small deviation to time taken by context switches and code executed within the kernel.

5.2.2 Overhead and Reclocking Optimization Potential

We designed our multi-phase execution tool with the goal of being able to determine a baseline for what optimized reclocking algorithms could achieve in the best case by using user-space-driven decisions as we described in Section 4.2. Further, this tool allows us additional comparisons between AVXFREQ and Intel's hardware implementation – in the previous section, we essentially measured how well our reimplementation reflects the model we wanted to achieve, whereas here, we can verify that AVXFREQ provides similar performance to the hardware implementation.

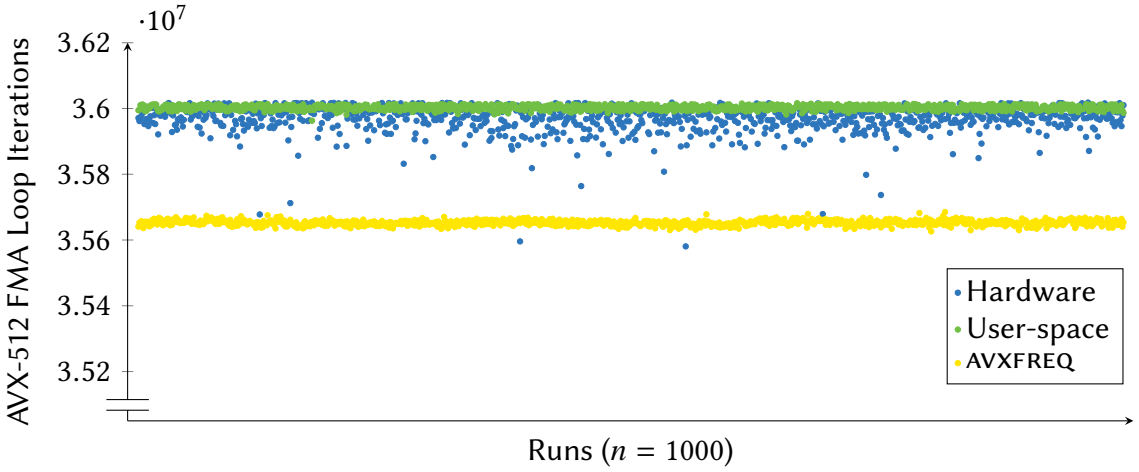


Figure 5.2: Completed AVX-512 FMA loop iterations within 200 ms when core frequency is controlled by the hardware, by the user-space program itself, and by our reimplementation AVXFREQ. Hardware as well as user-space reclocking yield the best performance, while AVXFREQ is only slightly slower with a difference of ~1 %.

To obtain the following results, we ran our multi-phase execution tool using `0 0 2000000 200000 0 666` as command line. This means that it first runs a scalar loop for 2 s, then an AVX-512 FMA loop for 200 ms, and finally another scalar loop for 666 µs. The last scalar loop's length is motivated by the fact that the hardware reclocking algorithm usually takes $\frac{2}{3}$ ms to return a core to its level 0 frequency as we found out through the analysis we conducted in Chapter 3, thus, this should expose worst-case behavior when using hardware reclocking. The other durations were selected mostly arbitrarily – we only wanted to ensure that there is enough time for a core to reach its maximum frequency within the first scalar loop. We ran our tool with three configurations, each executed 1000 times: with hardware reclocking, with AVXFREQ reclocking, and with user-space-driven manual reclocking through AVXFREQ.

The first scalar loop achieves approximately similar median performance with all three configurations: 1261.4×10^6 iterations with hardware reclocking, 1265.3×10^6 with AVXFREQ (approx. 0.31 % faster), and 1262.4×10^6 with user-space-driven reclocking (approx. 0.08 % faster). These deviations seem negligible and we consider them to be statistical noise as AVXFREQ is only waiting for an interrupt and does not actively do anything.

For the following heavy AVX-512 loop, we find more interesting differences: as shown in Figure 5.2, AVXFREQ performs about 0.9 % worse than hardware reclocking, yet it does so in a very consistent manner across all runs. User-

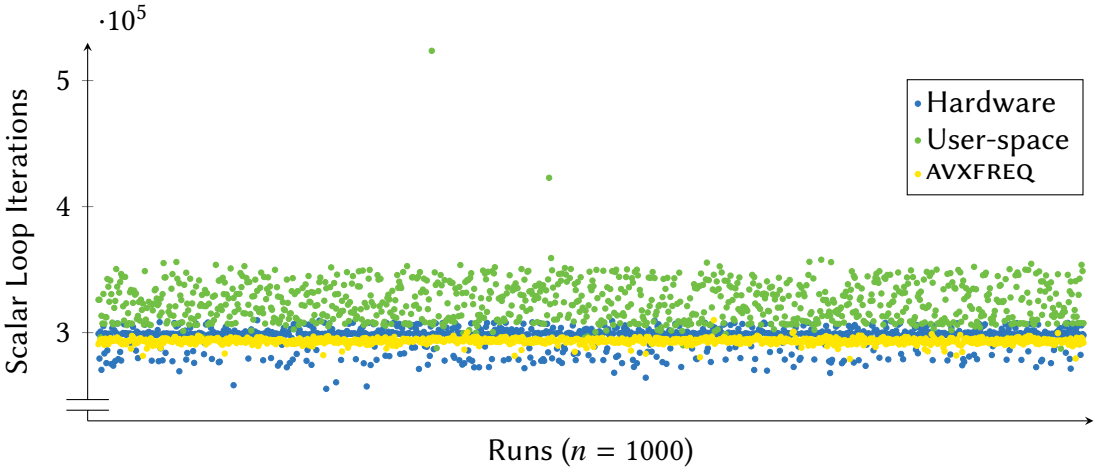


Figure 5.3: Completed scalar loop iterations within 666 µs after previous AVX-512 FMA execution when core frequency is controlled by the hardware, by the user-space program itself, and by our reimplementation AVXFREQ. Again, hardware is faster than AVXFREQ, but in the median case user-space reclocking outperforms both, with the caveat of exhibiting high variance.

space-driven decisions yield a performance increase of around 1 % compared to AVXFREQ and reach similar performance as hardware reclocking – surprisingly, with lower variance.

The overhead incurred by AVXFREQ is easily explained: unlike the hardware algorithm, AVXFREQ requires two interrupts for conducting a switch from turbo license level 0 to level 2. Further, since we can not disable hardware reclocking entirely on our test system, the hardware will still induce a single additional frequency switch (but only one, since the offsets for both levels are the same). User-space decisions being faster than AVXFREQ is not surprising, given that they do not require any interrupts, but only a single system call that directly switches from level 0 to level 2 without a step to level 1 in between.

Figure 5.3 finally depicts the completed iterations of the last scalar loop for the three different setups. The hardware implementation reaches a median of 299004 iterations, AVXFREQ is again about 1.8 % slower with a median of 293518.5. Compared to these results, user-space manual reclocking shows extremely varying performance. However, in most runs it outperforms hardware reclocking with a median of 323083.5, equaling an 8 % performance increase. Looking at the 99th percentile throughput of both cases, the improvement is even larger with about 14.9 %. We are unsure about the precise source of the high variance, but it is intriguing that we only experience this variance when upclocking, but not when downclocking. However, looking at it from the PCU's

perspective might yield an explanation: downclocking can only reduce the chip's power consumption, whereas upclocking will most certainly increase it. Thus, it is likely that a requested frequency reduction may be granted instantaneously, whereas a frequency increase requires the PCU to reevaluate the global load situation as it must ensure at all times that the processor's power consumption never exceeds its TDP. This issue would not occur with hardware reclocking as the hardware merely applies a negative offset to a frequency that was previously deemed to be in line with the electrical limitations, so it is clear that upon reverting a frequency reduction, the raised frequency can not incur a problem.

5.3 Discussion

In this thesis, our primary goal is to evaluate whether the idea of a software-based reimplementation of Intel's AVX reclocking algorithm is a feasible and useful approach to finding ways of optimizing the frequency scaling behavior to achieve improved performance for heterogeneous workloads consisting of both scalar and vector parts. For this purpose, we have conducted a thorough analysis of the frequency scaling behavior of a current Intel processor while executing AVX instructions in Chapter 3 and then used the obtained information in Chapter 4 to build AVXFREQ, our Linux-based reimplementation, on top of which we implemented a way for user-space programs to use their knowledge of what they are going to do next in order to control AVX reclocking themselves as a simple optimization approach. We then compared AVXFREQ's behavior to the hardware's and tested the performance of user-space-based reclocking. Putting it all together, our answer to the question "Is reimplementation in software a sensible approach for optimizing DVFS?" is "maybe."

The achievable quality of a reimplementation is limited by several factors: first, it can neither determine nor control the power gating status of vector registers and vector execution units which, on the one hand, seems to be a relevant factor in the downclocking behavior and, on the other hand, would be necessary to do manual upclocking without incurring an unnecessary waste of energy as the units remain powered even after raising the frequency through software means. Second, it is impossible to start measuring time as soon as the processor's pipeline is free of AVX instructions, albeit that would be necessary for a precise replication of the observed upclocking behavior. Third, by using the available performance events, we can only measure executed vector floating-point instructions but not integer ones. And finally, with the hardware in our test system, we were unable to completely disable the processor's own AVX-induced reclocking, but rather had it reduced to a minimum.

However, some of these issues do not seem too bad: for the purpose of testing alternative algorithms that solely impact when and how the frequency is changed, it is not important to control power gating as long as execution does not become unstable and the chip's total power consumption is below its TDP – otherwise, unnecessarily powered units may have an impact on the achievable frequency. Nevertheless, this issue can be circumvented by not utilizing all cores or by raising the applied TDP (on modern Intel CPUs, the TDP can be configured through an MSR [40]). Further, while missing hardware support for measuring executed integer vector instructions hinders us from being able to test real-world workloads, it is still possible to conduct tests with synthetic programs. It is conceivable that, with some work, a synthetic workload may reflect a real-world one very accurately with respect to all factors relevant for AVX reclocking (i.e., when are AVX instructions executed and when not).

The other two remaining issues are the biggest ones in our view: any conceivable automatic (i.e., when decisions are not simply made by user-space) reclocking algorithm would need precise information about when a vector execution streak has ended. While approximations as used in AVXFREQ are certainly possible, they incur a trade-off between accuracy and performance due to the need for short timer interrupts and remain a major source of inaccuracy. However, an improvement may be achievable here using the upcoming *Icelake* processor generation from Intel: these chips will feature an extended Processor Event-Based Sampling (PEBS) interface compared to *Skylake* [40]. Using PEBS, it is already possible on current generations to collect samples containing register states, TSC values and other data every n assertions of a performance event. However, on pre-*Icelake* processors, PEBS is only supported for a few specific performance events – the ones required for counting vector instructions not among them. Starting with *Icelake*, PEBS will be made available for all defined events. Then, it would be possible to configure PEBS to store the TSC value every time a certain number of floating-point vector instructions has retired. This way, we could create approximations that are off by only up to a few hundred nanoseconds without incurring further overhead.

Further, while it is not possible to disable the hardware's AVX reclocking in our current test setup, it could be worthwhile to put some reverse engineering effort into how the motherboard configures the processor's reclocking offsets. It seems plausible that this happens via an undocumented MSR or a register of the PCIe host bridge. Intel has not published a datasheet containing descriptions of the host bridge's registers for the processor we used in our system, however, such datasheets exist for other Intel processors (e.g., [11]). Although these do not seem to contain a register that controls AVX reclocking, again, it may be there but not publicly documented. By modifying the UEFI's settings, then

dumping undocumented MSR and host bridge register addresses and looking for differences it may be possible to find out how the UEFI applies the configuration to the CPU. In turn, it is conceivable there is a way to actually disable reclocking in hardware entirely. However, as long as we do not manage to achieve that, hardware reclocking even with minimal offsets remains another large obstacle for accurate evaluations of alternative algorithms.

Looking at our evaluation results from the previous section in this chapter, however, we have proven that improved performance for heterogeneous workloads by using better reclocking is indeed possible. Using a configuration where we exposed the worst-case behavior of the hardware (which is, in turn, the best case for user-space reclocking), we have reached a performance increase of nearly 15 % in the 99th percentile (8 % median) for a scalar workload directly following an AVX-512 FMA one. Unfortunately, we have also seen that software-controlled upclocking causes a high variance in performance, presumably because the PCU can not grant requests for frequency raises immediately. Assuming that this theory is correct though, the 99th percentile improvement likely represents the performance that would be reached if implemented in hardware as this should reflect the case where upclocking had the smallest delay.

We therefore believe that, despite the aforementioned issues, it still seems plausible that an automatic reclocking algorithm for AVX workloads with better performance than what is currently achieved using hardware-controlled reclocking may be implemented using the software approach we presented in this work. However, to prove this, we would need to actually build and evaluate such an implementation, but for now, this remains future work.

Chapter 6

Conclusion

Contemporary Intel processors need to reduce their frequency when executing AVX code in order to maintain stability. This has the effect that the performance of heterogeneous workloads with only some specific vectorizable code paths can suffer from these frequency reductions when using AVX on these chips. In this work, we wanted to explore the possibility of optimizing Intel's frequency scaling behavior by designing a software-based reimplementation that is supposed to allow us to test alternate algorithms without needing to modify the hardware itself. To be able to do this, we first needed to perform an analysis of the reclocking implementation found in the hardware as there is only very little documentation by Intel available.

In said analysis we found that, for most demanding AVX instructions, downclocking is triggered as soon as a single operation was executed and takes about $25 - 50\,\mu s$, depending on the heaviness of the executed instructions. As soon as the last instruction has retired, a processor will wait for around $\frac{2}{3}$ ms before reverting an AVX-induced frequency reduction. We used a simplified model based on these results to construct our reimplementation AVXFREQ, which proved to reflect the model reasonably well. We also were able to show that exploiting an application's knowledge about what it is going to do next can improve performance in a heterogeneous program by 15 % during scalar phases. However, we also experienced several drawbacks with our reimplementation approach. These include unpredictable delays when raising the frequency, the lack of ways to fully disable the reclocking done by the hardware itself and the impossibility to precisely measure time starting from the point when the last AVX instruction in a consecutive streak was executed. Some of these issues may be alleviated or circumvented with further work. As of now, we can not provide a conclusive answer to the feasibility of our idea, as more exploration would be required through implementations of alternative reclocking governors.

6.1 Future Work

Although we were able to show that improved performance in heterogeneous workloads through better reclocking is theoretically possible, research on this topic is nowhere complete. In this section, we will present several ideas and open issues that may pose interesting topics for future work.

Alternative Reclocking Algorithms

We designed our reimplementation AVXFREQ with the goal of being able to test alternative reclocking algorithms. So far we implemented a simple design which uses oracle-style foresight to switch frequencies by having user-space programs tell the kernel whenever a phase with AVX-512 heavy instructions starts or ends. This approach allowed us to estimate what performance improvement may be possible in theoretical best case. However, of course this is not feasible for real-world application as one of the main tasks of an operating system is to hide hardware implementation details such as the need for reclocking from software. We have outlined an approach where the operating system may profile durations of different execution phases in user-space threads to make predictions about the future length of AVX and scalar phases that may be used to estimate whether immediately raising the processor's frequency after an AVX phase would have a positive impact on overall performance.

Analysis on Other Processors

Our analysis in Chapter 3 was conducted using an Intel Core i9-7940X processor from the *Skylake (Server)* generation. However, even though older chips from the *Haswell* and *Broadwell* generations only supported AVX2 and no AVX-512 [1] [27], they still reduced their frequency when executing demanding AVX2 instructions, albeit to a smaller extent. Given that we found several discrepancies in the reclocking behavior between what Intel claims in their manuals and what we saw in our analysis results, it may be plausible that Intel's implementation has changed between *Haswell* and *Skylake* and their documentation is simply outdated. Running our tests with a *Haswell* processor could therefore yield interesting insights on why Intel has chosen to implement reclocking the way they did in *Skylake*.

Further, not all *Skylake (Server)* cores are equally equipped with the same amount of AVX-512 execution units. Our processor had two units per core, whereas other processors from this generation only have one [27]. As more units incur larger power consumption and higher energy density, it is possible that chips which only have one unit per core may behave differently. For the

sake of completeness in our analysis and because of potentially different requirements to alternative reclocking algorithms, we should run our framework again on such a processor.

Finally, towards the end of 2019, *Icelake* processors will be the first in the consumer desktop and mobile markets to have AVX-512 [3]. As Intel makes this instruction set available to a broader range of customers and given that *Icelake* will be the first large microarchitectural overhaul since *Skylake* along with a leap in process technology, they possibly have made changes to their reclocking algorithm. Again, it may be interesting to test our analysis framework on an *Icelake* chip to find potential differences that may have an impact on our and other approaches to optimizing the performance of heterogeneous workloads.

SMT Support in Our Analysis Framework

As described in Section 3.2.3, our analysis framework does not support running on processors with Simultaneous Multi-Threading (SMT) enabled. Although we can not conceive of a reason why a processor should behave differently with respect to reclocking when SMT is enabled, it is still possible that deviations exist. Given that in nearly all real-world scenarios SMT would be enabled, we should implement SMT support and verify that there are indeed no differences.

Huge Pages in the Analysis User-Space Component

In Section 3.2.4 we described how cache coherency effects impacted the results of our analysis framework when run on multiple cores with many pages, each 4 KiB in size. We alleviated the impact by using less pages and instead having a jump at the end to make them loop. However, this theoretically introduces small inaccuracies to the results as the instruction stream becomes slightly heterogeneous. Modern x86 processors alternatively also support pages that are 2 MiB large [40], which may be a viable alternative to improve the accuracy of our results.

Instruction Mixtures

For our analysis, we ran each test using just one specific instruction, repeated many times. However, as Intel hints in their optimization manual [23], the reclocking is actually triggered depending on the instruction mix executed over a window of several clock cycles. Using data science and machine learning techniques, it may be possible to create a model that can very precisely predict what mixtures will cause a frequency reduction.

Compiler Optimizations Using Analysis Results

Due to the dangers of possible performance degradation for heterogeneous workloads, most compilers do not generate AVX-512 code unless explicitly instructed to do so [29]. Even Intel's own C and C++ compiler that previously used to generate AVX-512 instructions aggressively whenever possible has become very conservative in this regard. With a deeper understanding about when and under what circumstances AVX-induced reclocking happens, it may be possible for compilers to make use of AVX more often without needing to worry about negative impacts. Thanks to the analysis we have done, much of the required knowledge is now available, while the rest may be achieved through further work as outlined in this chapter.

Reverse Engineering of AVX Frequency Offsets

In our discussion in Section 5.3 we have identified the impossibility to completely disable AVX reclocking in hardware as a major drawback for our reimplementation approach. However, it is unknown how our motherboard's UEFI configures AVX frequency offsets in the processor. It may be possible to find out through some reverse engineering efforts though, and this could theoretically reveal a way to achieve full suppression of the processor's own reclocking mechanism.

Hardware Limitations

On Intel *Skylake* CPUs it is possible to measure executed floating-point vector instructions, however, no similar performance events exist for other vector instructions. Although this is not within our scope of control, we would like to see event types for all possible types of vector instructions in future processors. It might even make sense to have events that count executed instructions per instruction set. Being able to count all instruction types would be necessary for building a complete reimplementation that can be tested with real-world programs.

Another obstacle preventing a precise software reimplementation of Intel's reclocking algorithm is that we are not able to start measuring time immediately after the last AVX instruction has retired. As outlined in Section 5.3, a vast improvement here is likely possible with future *Icelake* processors by using improved capabilities of the PEBS facility. However, for our purposes it would be optimal if the PMU would simply record the TSC time-stamp on every assertion of a performance event.

Further, although this is not too much of a problem for us, we are unable to control the power gating of vector registers and vector execution units. In

Section 5.3 we have discussed that the worst conceivable impact are slightly reduced frequencies due to exceeded power budgets. However, it may be interesting to measure the implications of alternative reclocking algorithms for power consumption, which is only inaccurately possible without being able to disable power supply to the gated units upon reverting a frequency reduction.

Glossary

Advanced Configuration and Power Interface (ACPI) ACPI is a standard that defines means for discovery and configuration of hardware devices as well as system power management. ACPI is commonly implemented by all modern UEFIs and operating systems. 10

Advanced Programmable Interrupt Controller (APIC) One of the interrupt controllers found in modern x86 CPUs. 17, 19, 20, 49, 50, 67

Advanced Vector Extensions (AVX) An instruction set extension for x86 CPUs which adds complex vector instructions. v, 3–9, 11–16, 18–22, 24–30, 32–36, 39–45, 47, 48, 50–52, 56–60, 62, 65

AVX-512 The third iteration of the AVX instruction set with support for 512-bit-wide vectors. v, 4, 9, 14–16, 24, 29, 32, 33, 38, 53–55, 58, 60–62

AVX1 The initial version of AVX with 256-bit-wide vectors. 15, 65

AVX2 AVX version 2, which extended the previous AVX1 with new instructions. v, 3, 4, 9, 15, 34, 60

Basic Input/Output System (BIOS) The first software loaded upon starting a legacy x86-based computer system, which conducts early initialization work before booting the operating system. Today, however, BIOSes are considered deprecated and have mostly been replaced with UEFIs. 70

Central Processing Unit (CPU) The main execution unit (processor) of a computer. v, 9, 15–17, 19, 21, 22, 24–27, 29, 30, 32, 33, 41–43, 45, 49, 52, 57, 58, 62, 65–67, 69

Complementary Metal-Oxide-Semiconductor (CMOS) CMOS technology implements digital logic gatters by using pairs of positively charged (pMOS) and negatively charged (nMOS) MOSFETs and is commonly used to construct integrated circuits. 10

Completely Fair Scheduler (CFS) The default scheduler used in current Linux kernels. Designed to offer both low latency for interactive systems and high throughput for servers. 23

Dennard's Law Dennard's Law was devised by Robert Dennard and states that the power density of transistors stays constant as they become smaller. 4, 7, 8

Dynamic Voltage and Frequency Scaling (DVFS) A generic term for techniques that seek to maximize both energy efficiency and performance by dynamically selecting frequencies along with appropriate voltages for different components of a microprocessor. 10, 41, 42, 45, 56

Executable and Linkable Format (ELF) The standard file format for executable binary files commonly used by many UNIX systems, including Linux. 21, 27, 32

fused multiply-add (FMA) An instruction that multiplies two values and adds (or subtracts) a third one, i.e., $a * b + c$. 15, 30, 31, 51–55, 58

Graphics Processing Unit (GPU) A kind of processor that is specifically designed for 3D graphics purposes. Nowadays, GPUs are often also employed for scientific computing and artificial intelligence. 3

hard disk drive (HDD) Hard disks are mass storage devices that use magnetization to store data on rotating disks. Nowadays, HDDs increasingly vanish from all kinds of computers and are more commonly replaced by SSDs. 11, 69

Hardware-Controlled Performance States (HWP) A feature of modern Intel CPUs where a PCU in the chip completely takes over P-state management. 10, 11, 42–44, 52

instruction pointer (IP) The instruction pointer of a thread contains the memory address of the instruction that is currently being executed. 19, 20

instruction prefetcher A unit in a microprocessor that predictively loads instructions from system memory before they are executed in order to prevent pipeline stalls. 22

instructions per cycle (IPC) Average amount of instructions executed by a processor per cycle for a given workload. 33

Kernel Address Space Layout Randomization (KASLR) In general, ASLR is a technique that tries to prevent malicious code from being able to guess the memory locations of symbols or data in a process by randomizing their positions. KASLR implements this for the Linux kernel. 30

L1 cache Level 1 caches are the fastest caches in modern processors. Commonly, each core has its own. L1 caches are often organized in Harvard architecture, i.e., there are two separate stores for data and instructions. 23, 67

L2 cache Like L1 caches, level 2 caches are private to each core in most current CPUs, albeit a little slower, but larger in return. 23

L3 cache Level 3 cache, usually the last cache in the hierarchy on most modern CPUs. Typically several Megabytes large and shared between all cores. 23

libc libc is the name of the standard library of the C programming language. 19, 22

Linux Linux is an operating system kernel originally published by Linus Torvalds in 1991. Today it dominates many markets, including smartphones, servers, and high-performance computing and has become the world's most widely used kernel. It even runs on cars, airplanes, and trains. 16, 18, 19, 23, 30, 42, 43, 56, 66–68

Local Vector Table (LVT) An APIC's LVT stores the interrupt vectors to use for certain purposes, e.g., interrupts from the PMU. 17, 19

macro-instruction A macro-instruction is an instruction as defined by the instruction set architecture and may be split up into several micro-instructions for execution by a specific implementation (i.e., a processor). 68

Metal-Oxide-Semiconductor Field-Effect Transistor (MOSFET) MOSFETs are an important type of semiconductor transistors that are capable of changing their conductivity depending on the voltage applied to their gate. Silicon MOSFETs are commonly used in micro- and nanoprocessors. 8, 65

micro-instruction Micro-instructions are the instructions passed into the execution engine of a processor after being split up from the macro-instructions that make up the program that is being executed. 24, 33, 43, 67

model-specific register (MSR) A special register, usually for configuration or measurement purposes, that isn't defined by the instruction set architecture but is specific to a microarchitecture or SKU. 17, 19, 20, 42, 49, 50, 57, 58

Moore's Law Moore's Law is named after Gordon Moore, one of the co-founders of Intel. It states that the integration density of integrated circuits doubles every two years. 4, 7, 8

Multi Media Extension (MMX) A very early vector instruction set extension to x86, introduced by Intel in 1997. 3

Multiple Queue Skiplist Scheduler (MuQSS) MuQSS is an alternative scheduler developed by Con Kolivas for the Linux kernel that is not part of the official Linux packages. One of its primary aims is to provide strong latency guarantees for interactive workloads. 13

non-maskable interrupt (NMI) An interrupt that can not be disabled and is always immediately delivered. 43

Non-Volatile Memory Express (NVMe) NVMe is a modern device interface specification designed specifically for SSDs attached via PCIe, crafted to reach lower latencies and higher throughput compared to earlier interfaces. 30

out-of-order (OoO) A processor featuring out-of-order support is capable of executing instructions in a different order than given in the program while maintaining correct and consistent results in order to achieve higher utilization of a superscalar pipeline. 23, 26, 27, 32, 34

Performance Monitoring Unit (PMU) A unit found in many modern microprocessors that enables software to measure performance and bottlenecks on the hardware level. 5, 16–21, 25, 26, 41–43, 47–50, 53, 62, 67, 69

performance state (P-state) A performance state describes at the very least a specific frequency. Depending on the specific hardware, it may also include a voltage corresponding to the frequency. 10, 42, 43, 52, 66

Peripheral Component Interconnect Express (PCIe) PCIe is a high-speed bus for connecting peripheral devices (e.g., storage, graphics processors, …) with the rest of a computer system. 57, 68

Power Control Unit (PCU) A unit in current Intel processors that monitors load and power consumption of different components within the chip and dynamically assigns frequencies and voltages as needed. 10, 37, 39, 55, 56, 58, 66

Processor Event-Based Sampling (PEBS) PEBS is a feature of current Intel processors that builds atop the PMU and allows to record precise data including register states and time-stamps upon the assertion of performance events. 57, 62

Simultaneous Multi-Threading (SMT) SMT-capable processors feed the execution pipeline of a single physical core simultaneously with instruction streams from multiple threads to achieve better execution unit utilization by presenting the core in the form of multiple *logical cores*. x86 CPUs often support twofold SMT. 17, 20, 29, 61

Single Instruction, Multiple Data (SIMD) See vector instruction. 3

solid-state drive (SSD) A solid state drive uses flash memory for persistent data storage. Modern SSDs commonly beat the performance of HDDs by several orders of magnitude with only a fraction of their energy consumption. 30, 66, 68

stock-keeping unit (SKU) A specific processor model brought to market. 15, 68

Streaming SIMD Extensions (SSE) Various early vector instruction sets for the x86 architecture, which are still commonly supported by modern CPUs but do not require frequency reduction for stable execution. 3, 4, 9, 15

superscalar pipeline Processors featuring a superscalar pipeline are capable of executing multiple instruction of the same thread in parallel by having duplicates of all necessary ressources (e.g., decode and execution units). 68

Thermal Design Power (TDP) Different manufacturers have used different definitions over time. For current-generation Intel processors, this value is a hard cap for the chip's power consumption. 10, 29, 56, 57

time-stamp counter (TSC) A simple counter on current x86 processors that increments steadily with a fixed frequency and that can thus be used to measure wall-clock time. 17, 19, 20, 29, 57, 62

Transport Layer Security (TLS) TLS is a wide-spread network protocol used for the secure encryption and authentication of internet traffic. 9

Unified Extensible Firmware Interface (UEFI) The modern successor of the classical BIOS with features like more thorough graphics and networking support and a cryptographically secured operating system boot process. 30, 42, 44, 52, 57, 58, 62, 65

UNIX signal Signals defined by the UNIX operating system specification that may be sent to a process either by another process or by the kernel itself. Most signals terminate a process by default, unless the process opted to use custom handling. 19

vector instruction A vector instruction is an instruction for a microprocessor which executes an operation not on only just one value, but on a vector consisting of several values. 24, 65, 68, 69

x86 x86 is an instruction set architecture introduced by Intel in 1978. Today, x86-based processors are commonly found in workstations, servers, and laptops. v, 3, 8, 17, 19, 21, 22, 29, 61

Bibliography

[1] *4th Generation Intel® Core™ Ushers New Wave of 2-in-1 Devices*. Intel Corporation. URL: https://newsroom.intel.com/news-releases/4th-generation-intel-core-ushers-new-wave-of-2-in-1-devices/.

[2] *Advanced Configuration and Power Interface (ACPI) Specification*. Jan. 2019. URL: https://uefi.org/sites/default/files/resources/ACPI_6_3_final_Jan30.pdf.

[3] P. Alcorn. *Intel Unveils 10th-Gen Core Chips, 10nm Ice Lake, 18% IPC Improvement, Sunny Cove Cores, Gen11 Graphics, Thunderbolt 3*. URL: https://www.tomshardware.com/news/intel-10th-generation-core-10nm-ice-lake-gen11-graphics-sunny-cove-thunderbolt-3-usb-c,39477.html.

[4] *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Vol. 1: *Basic Architecture*. May 2019.

[5] D. J. Bernstein. "ChaCha, a variant of Salsa20." In: *Workshop Record of SASC*. Vol. 8. 2008, pp. 3–5.

[6] M. Bohr. "A 30 year retrospective on Dennard's MOSFET scaling paper." In: *IEEE Solid-State Circuits Society Newsletter* 12.1 (2007), pp. 11–13.

[7] N. Bonen et al. *Performing local power gating in a processor*. US Patent 9,229,524. Jan. 2016.

[8] P. Brantsch. *Core Specialization for AVX-512 Using Fault-and-Migrate*. Master Thesis. July 2019.

[9] E. Bursztein. *Speeding up and strengthening HTTPS connections for Chrome on Android*. URL: https://security.googleblog.com/2014/04/speeding-up-and-strengthening-https.html.

[10] R. Courtland. "Intel now packs 100 million transistors in each square millimeter." In: *IEEE Spectrum* 30 (2017).

[11] *Datasheet, Vol. 2: 7th Gen Intel® Processor Family for S Platforms and Intel® Core™ X-Series Processor Family*. Intel Corporation. URL: https://www.intel.de/content/www/de/de/processors/core/7th-gen-core-family-desktop-s-processor-lines-datasheet-vol-2.html.

[12] R. 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.

[13] P. Glaskowsky. *IDF Fall 2007, part 3 – Gordon Moore interview*. Sept. 2007. URL: https://www.cnet.com/news/idf-fall-2007-part-3-gordon-moore-interview/.

[14] M. Goll and S. Gueron. "Vectorization on ChaCha stream cipher." In: *2014 11th International Conference on Information Technology: New Generations*. IEEE. 2014, pp. 612–615.

[15] M. Gottschlag and F. Bellosa. "Reducing AVX-Induced Frequency Variation With Core Specialization." In: *The 9th Workshop on Systems for Multi-core and Heterogeneous Architectures*. Dresden, Germany, Mar. 2019.

[16] P. Hammarlund et al. "Haswell: The fourth-generation Intel Core processor." In: *IEEE Micro* 34.2 (2014), pp. 6–20.

[17] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. 6th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017. ISBN: 0128119055, 9780128119051.

[18] W. Huang et al. "Scaling with design constraints: Predicting the future of big chips." In: *IEEE Micro* 31.4 (2011), pp. 16–29.

[19] *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Vol. 2: *Instruction Set Reference*. May 2019.

[20] *Intel Details 2011 Processor Features, Offers Stunning Visuals Built-in*. Intel Corporation. URL: https://newsroom.intel.com/news-releases/intel-details-2011-processor-features-offers-stunning-visuals-built-in/.

[21] *Intel Introduces The Pentium® Processor With MMX™ Technology*. Intel Corporation. URL: https://www.intel.com/pressroom/archive/releases/1997/dp010897.htm.

[22] *Intel Launches the Pentium® III Processor*. Intel Corporation. URL: https://www.intel.com/pressroom/archive/releases/1999/dp022699.htm.

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

[24] *Intel® Core™ i9-7940X X-series Processor Product Specifications*. Intel Corporation. URL: https://ark.intel.com/content/www/us/en/ark/products/126695/intel-core-i9-7940x-x-series-processor-19-25m-cache-up-to-4-30-ghz.html.

[25] *Introducing 6th Generation Intel® Core™, Intel's Best Processor Ever*. Intel Corporation. Sept. 2015.

[26] V. Krasnov. *On the dangers of Intel's frequency scaling*. 2017. URL: https://blog.cloudflare.com/on-the-dangers-of-intels-frequency-scaling/.

[27] A. Kumar and M. Trivedi. *Intel® Xeon® Scalable Processor Architecture Deep Dive*. URL: https://en.wikichip.org/w/images/0/0d/intel_xeon_scalable_processor_architecture_deep_dive.pdf.

[28] A. Langley et al. *ChaCha20-Poly1305 Cipher Suites for Transport Layer Security (TLS)*. RFC 7905. June 2016. DOI: 10.17487/RFC7905. URL: https://rfc-editor.org/rfc/rfc7905.txt.

[29] D. Lemire and T. Downs. *AVX-512: when and how to use these new instructions*. Sept. 2018. URL: https://lemire.me/blog/2018/09/07/avx-512-when-and-how-to-use-these-new-instructions/.

[30] *Linux kernel source code: arch/x86/events/core.c*. Version 5.1. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/x86/events/core.c?h=v5.1.

[31] *Linux kernel source code: include/linux/sched.h*. Version 5.1. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/linux/sched.h?h=v5.1.

[32] D. D. Lopata. *Speed binning for dynamic and adaptive power control*. US Patent 8,234,511. July 2012.

[33] Y.-H. Lu and G. De Micheli. "Comparing system level power management policies." In: *IEEE Design & test of Computers* 18.2 (2001), pp. 10–19.

[34] A. Mazouz et al. "Evaluation of CPU frequency transition latency." In: *Computer Science-Research and Development* 29.3-4 (2014), pp. 187–195.

[35] S. Mittal. "A survey of techniques for improving energy efficiency in embedded computing systems." In: *arXiv preprint arXiv:1401.0765* (2014).

[36] *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Vol. 4: *Model-Specific Registers*. May 2019.

[37] G. E. Moore et al. "Progress in Digital Integrated Electronics." In: *Electron Devices Meeting*. Vol. 21. 1975, pp. 11–13.

[38] J. 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.

[39] C. Scordino and I. Molnár. *CFS Scheduler*. URL: `https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html`.

[40] *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Vol. 3: *System Programming Guide*. May 2019.

[41] M. B. Taylor. "Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse." In: *DAC Design Automation Conference 2012*. IEEE. 2012, pp. 1131–1136.

[42] R. J. Wysocki. *CPU Performance Scaling*. URL: `https://www.kernel.org/doc/html/latest/admin-guide/pm/cpufreq.html`.

[43] R. J. Wysocki. *`intel_pstate` CPU Performance Scaling Driver*. URL: `https://www.kernel.org/doc/html/latest/admin-guide/pm/intel_pstate.html`.

[44] P. Ziljstra, V. Radnai, and I. Molnár. *Real-time group scheduling*. URL: `https://www.kernel.org/doc/html/latest/scheduler/sched-rt-group.html`.