Karlsruher Institut für Technologie

Core Specialization for AVX-512 Using Fault-and-Migrate

Masterarbeit
von

cand. inform. Peter Brantsch

an der Fakultät für Informatik

Erstgutachter:	Prof. Dr. Frank Bellosa
Zweitgutachter:	Prof. Dr. Wolfgang Karl
Betreuender Mitarbeiter:	Mathias Gottschlag, M.Sc.

Bearbeitungszeit: 07. Januar 2019 – 08. Juli 2019

KIT – Die Forschungsuniversität in der Helmholtz-Gemeinschaft

www.kit.edu

Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

Karlsruhe, den 08. Juli 2019

Abstract

The Advanced Vector Extensions 512 (AVX-512) are modern Single Instruction Multiple Data (SIMD) extensions to the x86 instruction set using 512-bit wide registers, enabling substantial acceleration of numeric workloads, for example processing eight sets of 64-bit operands in parallel. Because of the high power consumption of the corresponding functional units, a CPU core executing AVX-512 instructions has to temporarily reduce its clock frequency to maintain thermal and electrical limits. This clock frequency reduction can slow down the scalar part of mixed workloads because it persists substantially beyond the last AVX-512 instruction.

To mitigate this performance impediment, core specialization can be used, which is the preferred use of certain cores for specific kinds of computation. By running AVX-512 and scalar code on disjoint sets of CPU cores, throttling of cores executing scalar code can be avoided.

The Operating Systems Group at the Karlsruhe Institute of Technology has already demonstrated that core specialization can be effectively employed against the aforementioned performance reduction by implementing it in Linux. A new system call is introduced to mark the beginning and end of AVX-512 phases of a task, such that the scheduler can migrate it to a specialized core. However, the existing implementation is neither transparent nor automatic, but instead requires the application to be modified.

This thesis presents an extension of the existing core specialization implementation, making it transparent and automatic by efficiently virtualizing AVX-512 to intercept the instructions and subsequently trigger migration. Our extension determines the necessary number of AVX-512 cores at runtime based on CPU time consumed. Because there is no trivial way of detecting the end of an AVX-512 phase, we compare different heuristics for re-migration.

We evaluate our prototype in a web server scenario with nginx and OpenSSL using ChaCha20-Poly1305 encryption and brotli compression, using AVX-512 to accelerate the combination of cipher and message authentication code. The benchmarks show that the performance degradation caused by AVX-512-induced frequency reductions can be almost completely mitigated, without having to modify the application.

Contents

1 Introduction

Until around 2007, semiconductor scaling worked approximately like Dennard et al. [7] described it in 1974, with each new technology generation bringing smaller, faster transistors at a constant power density. Then, Dennard scaling ended [2], as leakage currents through the ever thinner gate oxide increased, becoming a noticeable part of chip power consumption. The power density could no longer be kept constant and clock frequencies stopped increasing, though transistor miniaturization continued, and the number of transistors per chip increased. To still increase the computing power of chips, even though single-thread performance scaling is hampered by stagnating clock frequencies, contemporary designs employ parallelism of both code[1] and data[2].

Because of the high power density and parallelism of processors, an increasing share of die area is under-utilized, either because running all of the chip at its maximum clock rate would exceed thermal or electrical limits, or because programs do not efficiently exploit the processor's parallelism [8]. This share is called *dark silicon*.

A recent example are the Advanced Vector Extensions 512 (AVX-512) [21, 22], because their functional units can raise the power consumption of a CPU core up to the point that it has to reduce its clock frequency, depending on the type[3] and rate of Advanced Vector Extensions 2 (AVX 2) and AVX-512 instructions, and the number of similarly loaded cores. However, these SIMD additions to the x86 instruction set operating on 512-bit wide registers can significantly accelerate computation such as cryptography (see the microbenchmark in Section 5.3), offsetting the otherwise detrimental effect of the frequency reduction on performance.

If this frequency reduction only affected the SIMD instructions, it would not pose a problem, but before the core can return to higher clock frequencies, a timer of approximately 2ms first has to expire without any events again necessitating frequency changes [18, section 17.26]. Mixed workloads, in which only small parts of computation are accelerated, can therefore suffer an overall slowdown, because the reduced clock frequency persists beyond the last AVX-512 instruction, affecting the following scalar instructions. This slowdown was demonstrated by Krasnov [28] in a web server using AVX-512-accelerated cryptography functions. Chapter 2 will describe the frequency behavior of Intel CPUs in regard to AVX-512 in more detail.

To avoid any possible frequency reduction because of wide SIMD instructions, vectorization could simply be disabled, though only at the cost of all its advantages. No general decision whether to enable or disable vectorization for all mixed applications can be made, though, because the net benefit for even a single application depends on the scenario.

[1]Multiple Instruction streams, Multiple Data streams (MIMD), i.e. multi-core systems
[2]SIMD
[3]floating point or integer, multiplication or simpler operation

For the same application, vectorization can be advantageous in some circumstances, and detrimental in others. Instead, a method of maintaining predictable system performance and still reaping the benefits of vectorization is needed. Ideally, to save developers from laboriously instrumenting their applications, and users from unwelcome surprises in the form of library updates bringing performance regressions, such a method would be transparent and automatic.

Gottschlag and Bellosa [13, 14] have shown that core specialization can successfully be used to mitigate the performance impediments faced by mixed workloads by restricting the execution of AVX-512 code to certain cores, such that only these cores are affected by frequency reductions. This thesis builds upon their previous work, and makes the following contributions:

Efficient Virtualization of AVX-512 The CPU is configured to *fault* on AVX-512 instructions, hence the name "fault-and-migrate", such that we can intercept these instructions and accordingly migrate the task. This is equivalent to efficient virtualization as per the criteria of Popek and Goldberg [36].

Transparency The existing implementation introduces a new system call to inform the scheduler about the beginning and end of AVX-512 use, requiring modification of the application code. Using the efficient virtualization described above, the scheduler can detect the beginning of AVX-512 use transparently.

Automation To require the least amount of tuning by the user and easily adapt to a variety of mixed workloads, the system determines the number of AVX-512 cores automatically.

Evaluation using a web server scenario derived from that of Krasnov [28] shows that core specialization using fault-and-migrate can mitigate the performance reduction of mixed workloads caused by AVX-512-induced frequency reductions. Our mechanism for automatically determining the number of AVX-512 cores is not yet optimal, tendentially choosing too many cores and causing greatly variable performance. However, using a static set of cores for AVX-512, automatic migration and re-migration are working completely satisfactorily. The web server scenario, originally suffering from an 11.6% performance reduction with AVX-512, reaches 99% of its non-AVX-512 throughput using our prototype.

The remainder of this thesis is structured as follows: Chapter 2 gives detailed background information and an overview of related work, Chapter 3 analyses the problem and prerequisites and discusses possible approaches, followed by Chapter 4, which presents a solution and explains the design and implementation decisions. A comprehensive evaluation is performed in Chapter 5. Concludingly, Chapter 6 summarizes the lessons learned and outlines possible future work.

2 Background and Related Work

Heavy use of AVX-512 causes a CPU core to reduce its clock frequency as the core would otherwise exceed thermal and/or electrical limits. The core maintains the reduced frequency until a 2ms timer expires without any event that would cause a frequency change, so that some scalar code in mixed workloads is slowed down, degrading overall performance.

For a better understanding of this frequency reduction, this chapter first describes AVX-512 and the clock frequency behavior of Intel CPUs. We then proceed with an overview of related research into mixed-workload performance. Some related work suggests *core specialization* to mitigate the performance degradation, that is the segregation of AVX-512 and scalar code onto disjoint sets of CPU cores. The Operating Systems Group at KIT has researched and implemented core specialization both as an application-level library and in the operating system kernel, successfully reducing the impact of AVX-512-induced throttling on performance. We will review these approaches, especially the kernel-level implementation, because this thesis extends it to be transparent and automatic. Concluding this chapter, we look at similar topics of research, namely staged computation and cohort scheduling, which are related to this thesis because of their concept of a program's distinct phases of computation, and reconfigurable systems, which change their functionality more substantially than the systems considered by this thesis.

2.1 AVX-512

The Advanced Vector Extensions 512 (AVX-512) [20, Volume 1, Section 5.19, 22] are a family of 512-bit SIMD extensions for Intel x86, consisting of a large set of foundation instructions, essentially 512-bit versions of Advanced Vector Extensions (AVX) and AVX 2 instructions using a new instruction encoding, and more specialized classes of instructions, e.g. for acceleration of AES. AVX-512 provides 32 512-bit `ZMM` registers, and 8 opmask registers (for conditional processing).

The defining feature of AVX-512 is its ability to operate on 512-bit wide registers, enabling parallel processing of 8 64-bit operations or 16 32-bit operations. Generally, wide SIMD instructions can be categorized in *light* and *heavy* instructions, light instructions being all those which do not use floating point (FP) or perform any multiplication or fused-multiply-add (FMA), and heavy instructions being all those which do. Light instructions do not consume as much power as heavy ones, but even light instructions can cause the CPU to reduce its clock frequency.

The following section describes the processors clock frequency behaviour in more detail.

Table 2.1: Power licenses and corresponding instruction types [18, Section 17.26]

Level	Category	Frequency Level	Maximum Frequency	Instruction Types
0	AVX 2 *light* instructions	highest	max.	scalar, 128-bit AVX, SSE, AVX 2 without: FP, integer multiplication or FMA
1	AVX 2 *heavy* + AVX-512 *light* instructions	medium	max. AVX 2	AVX 2 FP and integer multiplication/FMA, AVX-512 without: FP, any multiplication or FMA
2	AVX-512 *heavy* instructions	lowest	max. AVX-512	AVX-512 with FP, integer multiplication and FMA

2.2 Processor Clock Frequency Behavior

For the rest of this thesis, we need to understand how the processor determines the clock frequencies of its cores, especially how frequency reductions are handled.

Intel Turbo Boost is a mechanism for opportunistic extraction of additional performance by means of dynamic frequency changes of individual cores. As long as thermal and electrical constraints allow, it increases the clock frequency of cores on demand up to a certain limit [20, Volume 3, Section 14.3.3, 23]. For short periods in time, the CPU may even exceed its thermal design power (TDP). However, that limit may fall below the regular operating frequency, if required to maintain thermal or electrical constraints. When executing AVX-512 or heavy AVX 2 instructions, CPU cores can not run at their maximum clock frequency, because the functional units used would otherwise consume too much power and produce excessive heat. The actual maximum frequency is determined by the core's current *power license* (see Table 2.1), which depends on the type and rate of SIMD instructions, as well as the number of other cores with similar characteristics [18, Section 17.29]. After being relegated to a lower power license, the core will maintain it until an approximately 2 ms timer expires without any event again requiring a frequency reduction, and only then return to a higher license. Also, the power controller unit (PCU) takes up to 500 µs to grant the core a new license. Figure 2.1 shows how different cores have different frequency ranges depending on their instruction mix.

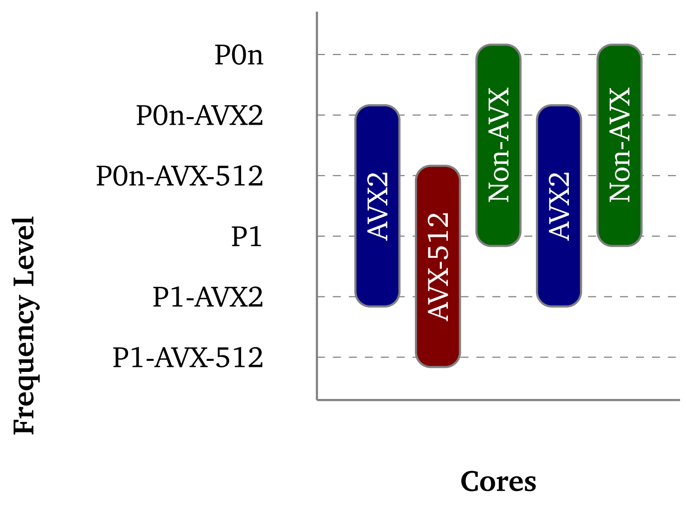


Figure 2.1: Frequency levels in mixed workloads [18, Figure 15-17]: P0n is the respective maximum performance level, and P1 is a reduced level.

2.3 Research in Mixed AVX-512 Workloads

Krasnov [28] examines the impact of AVX-512 on mixed workloads, using the example of the *nginx* [39] web server and the *OpenSSL* [33] and *BoringSSL* [3] (a fork of OpenSSL) cryptography libraries. The web server is configured to serve a simple HTML file via HTTPS using *brotli* compression. Both different ciphers and implementations of the same cipher using different instruction set extensions for acceleration are compared. The throughput is measured under full load. Nginx handles 10% fewer requests per second when using OpenSSL's AVX-512 variant of ChaCha20-Poly1305 compared to BoringSSL's AVX 2 variant or either library's AES-128-GCM implementation, even though *perf* [43, `tools/perf`] reveals that the AVX-512 workload only uses 2.5% of CPU time. Throughput is also reduced if only a small share (20%/10%) of HTTP requests uses the AVX-512-accelerated ciphers, though less severely (7%/5.5%), which Krasnov explains with the CPU reducing its clock frequency irrespectively whenever AVX-512 is used on all cores. This thesis uses a very similar scenario to analyze the problem and evaluate the proposed solution.

Tiwari et al. [42] improve performance of the open-source x265 video encoder using AVX-512. They find that only select kernels of the encoder benefit from AVX-512, namely those in which the lower cycle count outweighs the frequency reduction. Similar to the approach taken in this thesis, they suggest limiting the throttling to certain cores, but by modification of the encoder, such that only certain types of worker threads execute AVX-512. However, modifying the application to only execute AVX-512 on a specific set of cores creates additional work for application developers, especially without any support from the operating system. In contrast, the approach presented in this thesis will not require any action of the application developer, being completely transparent and automatic.

2.4 Core Specialization

This thesis uses the same definition as Saez et al. [37], who define *core specialization* as *"preferring certain types of cores for certain types of computation"*.

Core specialization can be used to isolate AVX-512 code from the remainder of the code by only executing it on a subset of the available CPU cores, such that all other cores and the code they run are not affected by frequency reductions.

Chakraborty, Wells, and Sohi [4] present *computation spreading*, which is core specialization for the different user and operating system (OS) parts of threads using hardware-assisted thread migration. Two policies are presented, either only separating OS code from user code, or additionally specializing cores for individual system calls. Both successfully reduce destructive interference in architectural state such as caches and branch predictors for server applications.

Our approach will, as a side effect, also spread computation across cores. One of our re-migration policies migrates tasks away from AVX-512 cores before the actual system call code runs, thereby separating OS and user code. However, the criteria (type of instruction) and objective (limiting frequency reduction to AVX-512 code) of core specialization in this thesis are different from that of Chakraborty, Wells, and Sohi.

Papamanoglou [35] implemented a core specialization library targeted at AVX-512. Using it, applications can submit work (in the form of closures) to be executed on the specialized cores. To manage the specialized cores across different processes, they use a system-global policy server that each library instance communicates with, which decides which cores should be specialized cores and which closures to offload. As their approach modifies the application, they have the advantage of precisely determined code regions to potentially be offloaded. However, the library approach requires access to the application source code and potentially laborious modification thereof, whereas the approach presented in this thesis does not.

Gottschlag and Bellosa [12, 14] implement core specialization for AVX-512 in Linux. Instead of modifying the Completely Fair Scheduler (CFS), Linux's default scheduler, which is refined, though highly complex, a simpler out-of-tree scheduler, the Multi-Queue Skiplist Scheduler (MuQSS), is used instead, because it is easier to extend. A task wishing to use core specialization has to mark and unmark itself using a newly introduced system call. For this purpose, a type attribute is added to the task data structure to discern three kinds of tasks:

avx Tasks which have marked themselves as executing AVX-512.

scalar Tasks which had previously been marked as executing AVX-512.

default The initial type of tasks. All tasks which never mark themselves will keep this type.

MuQSS uses per-CPU runqueues sorted by virtual deadline, which are replicated in the modified scheduler to separate tasks of different types. Core specialization is then implemented by having scalar and AVX-512 cores treat task types differently.

Scalar cores run only tasks from the default and scalar runqueues, and AVX cores run tasks from the default and AVX queues, giving precedence to AVX tasks by adding a large penalty to the virtual deadlines of default tasks. This way, starvation of default tasks (e.g. system tasks) pinned to AVX cores is avoided.

MuQSS does not have an explicit load balancing mechanism or migration thread. Instead, each core choosing the next task to run checks the earliest deadline of both its own runqueue and all others, and takes the task with the earliest virtual deadline, even if it is from another core. A task changing its type is enqueued in the corresponding local runqueue, and migration happens implicitly, because whichever core is allowed to run the task will pick it eventually.

Evaluation shows that core specialization for AVX-512 successfully reduces the performance degradation caused by AVX-512-induced frequency reduction in a web server scenario similar to that of Krasnov [28].

In Gottschlag and Bellosa's implementation, the set of specialized cores is fixed, and applications have to be manually instrumented for core specialization. This thesis will extend it to choose the set of AVX-512 cores dynamically and detecting AVX-512 usage at runtime without instrumentation, thereby making it fully automatic and transparent.

2.5 Scheduling for Heterogeneous Systems

Executing AVX-512 code only on a subset of CPU cores means treating the homogeneous multi-core system, in which all cores have the same performance and instruction set (respectively features), as if it were a heterogeneous system in which all cores share the largest part of the instruction set, but are not equal in terms of performance and features.

Such performance- and/or feature-asymmetric systems actually exist, and can be advantageous compared to fully symmetric multiprocessor systems in terms of performance and energy efficiency, achieving greater computational power at the same area and/or using less power.

Li et al. [32] research scheduling for heterogeneous multi-core systems with overlapping instruction set architectures (ISAs), which exhibit asymmetry of both performance and functionality. Specifically, these systems are composed of big cores, which have larger caches and higher clock frequencies, and small cores, which have smaller caches and run at lower frequencies. All cores share almost the complete ISA, except for instructions set extensions present only on a subset of cores. This functional asymmetry poses one of the problems that we also have to solve, namely handling tasks attempting to execute instructions unsupported on their current core, which Li et al. solve in the same way as this thesis. Their fault-and-migrate implementation in Linux on Intel is very similar to ours, because it also uses the invalid opcode trap and performs migration by changing the task's CPU affinity and triggering the existing migration mechanism. The problem of determining the cause of an invalid opcode trap is also solved the same way as in our implementation. Lacking hardware support for distinguishing whether the trapping instruction is invalid or part of an instruction set extension that is unavailable on the current core, both implementations assume the latter on cores without the instruction set extension.

On a fault, they migrate the task, and if the instruction faults again, even though on a core with the instruction set extension, it must have been actually invalid. Li et al. achieve promising results, especially demonstrating low overhead of fault-and-migrate.

However, there are important differences, most notably the problem: Li et al. optimize scheduling to extract more performance from statically heterogeneous systems, whereas we make homogeneous systems dynamically heterogeneous at runtime to alleviate performance impediments. The systems targeted by this thesis have no performance asymmetry other than reduced clock frequencies on specialized cores executing AVX-512, whereas those considered by Li et al. are asymmetric from the start. Li et al. handle performance asymmetry by means of faster-first scheduling and dynamically weighted round-robin (DWRR) with scaled CPU time. The latter is the product of CPU time and a speed rating of the respective core, which can be established by a benchmark run at system startup or later changed at runtime. Faster-first scheduling means that, when the number of threads does not exceed the number of fast cores, cores with higher speed ratings will be preferred. When the system is more heavily utilized, DWRR with scaled CPU time maintains scheduling fairness. Also, on our systems, all functional asymmetry can be changed at runtime, because we can dynamically activate/deactivate AVX-512 on each core (see Subsection 4.1.1). In contrast, Li et al. have to detect the static feature asymmetry of the system.

Saez et al. [37] present a Comprehensive Scheduler for Asymmetric Multicore Systems (CAMP), that maximizes performance on asymmetric multiprocessor systems, which consist of specialized cores sharing the same ISA but differing in performance. Comprehensive Scheduler for Asymmetric Multicore Systems (CAMP) uses metrics or models to allocate tasks to fast or slow cores to maximize both energy efficiency and thread-level parallelism.

Such metrics-based approaches to scheduling are not applicable to a system using core specialization for AVX-512, unless it already has performance performance asymmetry beforehand, because in an otherwise homogeneous system, all practical heterogeneity stems from core specialization, and actual performance asymmetry will vary at runtime, depending on the exact instruction mix executed on the specialized cores, which determines their maximum clock frequency.

2.6 Staged Computation and Cohort Scheduling

Core specialization for AVX-512 considers two kinds of phases in a task's execution, those that use AVX-512, and those that do not. More generally, the execution of any task can be partitioned in any number of distinct phases. *Staged computation* [29] is a programming model designed around this notion, describing a program as a set of *stages*, in which related operations and data are grouped. Larus and Parkes [29] describe cohort scheduling, i.e. scheduling separate invocations of the same stage together. Cache efficiency is improved, because locality is increased. Schwarz [38] and Gottschlag et al. [15] spread stages over CPU cores, so that each core repeatedly executes the same set of stages. Again, cache efficiency is improved, because the working set of each core is now no longer the sum of the working sets of all stages, overwhelming the cache, but only of a much smaller set.

Instead of isolating scalar code from AVX-512 code by running these two types of code on disjoint sets of cores, they could be separated in time by applying a form of cohort scheduling, such that longer phases of scalar and AVX-512 execution would alternate. Section 3.1 explains why a cohort scheduling-like approach was ruled out for this thesis.

2.7 Reconfigurable Systems

Chapter 4 will later show that AVX-512 can actually be disabled and enabled per CPU core at runtime, in a sense reconfiguring the core. This alone does not make the CPU an actual reconfigurable system, as that would require changing the *implemented* functionality, instead of enabling/disabling it. Reconfigurable adaptive systems change their functionality much more profoundly, e.g. by means of programmable logic. The Rotating Instruction Set Processing Platform (RISPP) [1] is such a system, implementing a core ISA using fixed hardware and special instructions using a reconfigurable fabric. In a way very similar to the fault-and-migrate mechanism implemented for this thesis, RISPP uses trap-and-emulate for its special instructions to avoid overly frequent reconfiguration and costly pipeline stalls while waiting for reconfiguration to finish. When an instruction occurs that is not currently available either because it is not (yet) configured or currently being configured, or because hardware resources are better used for other instructions, the runtime system emulates the faulting instruction using the core ISA and may trigger reconfiguration. There are, however, important differences between the RISPP concept and core specialization for AVX-512 using fault-and-migrate. RISPP implements custom special instructions using programmable logic, whereas reconfiguration in our implementation is limited to enabling/disabling existing instruction set extensions. RISPP's trap handling uses emulation, whereas fault-and-migrate does not. Also, existing RISPP implementations target different CPU architectures (SPARCv8 and DLX).

However, the possibility of more powerful reconfiguration of Intel processors is hinted at in papers written at Intel, which mention the use of proprietary tools to change cache sizes, clock frequencies, and instruction execution width [32, Section 5]. Moreover, actual reconfiguration at runtime could be possible using the microcode loading mechanism [20, Volume 3, Section 9.11], but lacking the documentation, toolchain and encryption keys necessary to produce microcode updates, this possibility can not be researched, except by sophisticated reverse engineering as done by Koppe et al. [27].

3 Analysis

When a CPU core executes AVX-512 instructions, it temporarily reduces its clock frequency, especially for heavy instructions (e.g. multiplication), to avoid exceeding thermal or electrical limits. The overall performance reduction of mixed workloads is caused by scalar code being subject to the reduced clock frequency of heavy AVX-512, which is maintained beyond the last AVX-512 instruction, until the expiration of a timer.

Related work has demonstrated that core specialization can mitigate the problem. This chapter affirms the decision to use core specialization instead of staged computation, and analyses approaches at making core specialization transparent and automatic.

3.1 Core Specialization

It should be possible to alleviate the aforementioned decrease in performance by separating scalar code and AVX-512 in the spatial or temporal domain. Temporal isolation would alter the scheduling sequence, such that AVX-512 and scalar code blocks of tasks would not be interspersed as they are now, but executed in bursts/batches, at the cost of increased worst-case latency, because each of the two classes of code would have to wait while the other is being executed. Such scheduling methods, which group tasks by their *stage* of execution, are also referred to as *staged computation* [29, 15, 38]. Schwarz [38] successfully implements stage-aware scheduling without increasing request latency. Cohort scheduling, as researched by Larus and Parkes [29], successfully uses staged computation to increase locality for better cache efficiency, but causes increased latency under light load.

Even though request latency is not necessarily impacted by staged computation, manually identifying stages is prohibitively laborious for large applications, and even when identification of stages is automated instead, the application still has to be recompiled from instrumented source code. Therefore, we can not use staged computation, because we want our approach to be transparent and automatic.

Spatial isolation leverages the multiprocessor system to achieve separation by running scalar and AVX-512 code on disjoint sets of CPU cores. By limiting AVX-512 to certain cores, possible frequency reductions are also limited to these cores, such that the scalar code running on other cores is not impacted. Using a symmetric multiprocessor (SMP) system as if it were heterogeneous/asymmetric, by executing certain classes of instructions only on a subset of the available CPU cores, matches the definition of *core specialization*. Related work (see Chapter 2) has already demonstrated that core specialization is a viable approach to mitigating the performance impediments of mixed AVX-512 workloads. This thesis uses core specialization, building on the work of Gottschlag and Bellosa [14] (see also Section 2.4 for a more detailed description of their implementation).

3.2 Fault and Migrate

To make core specialization automatic, a way of identifying AVX-512 sections in tasks is needed. The following options were considered:

Static Source Code Analysis As application source code might not always be available, it has to work without access to the application source. In addition, SIMD instructions are impossible to preclude given only the source code, because they can be generated by automatically vectorizing compilers, or be contained in library functions, or even generated at runtime by a just-in-Time Compiler (JIT).

Static Binary Analysis Disassembling x86 binaries is difficult and potentially inaccurate [34], so no (offline) disassembly and analysis shall be performed, ruling out binary instrumentation. Also, offline analysis is inapplicable to code loaded or generated at runtime (e.g. JIT).

Emulation Full emulation would trivially allow different handling of AVX-512 instructions, but also have to solve the problem of decoding x86 instructions, and by its very nature (and the overheads it incurs) defeat the purpose.

Efficient Virtualization In contrast, *efficient* virtualization according to the criteria of Popek and Goldberg [36] would work, as it causes no overhead for the vast majority of instructions. For an architecture to be efficiently virtualizable, *sensitive instructions*[1] are required to be *privileged instructions*[2], so that, when executed in the user mode virtual machine, they trap to the virtual machine monitor (VMM), which in turn emulates the effect of the instructions. Thus, to efficiently virtualize AVX-512, a mechanism to transparently intercept AVX-512 instructions is needed. Henceforth, said mechanism will be referred to as *fault and migrate*, as this thesis investigates how to configure an x86 CPU to make AVX-512 instructions trap, so that the respective task can subsequently be migrated to a specialized core.

Of the above approaches, *efficient virtualization* appears the most promising, because it incurs no overhead for the vast majority of instructions, obviates any need for application source code or cumbersome binary analysis, and is completely transparent.

[1]A sensitive instruction is *"any instruction that would affect the allocation of resources"* [36, p. 419]. According to this notion of sensitive instructions, AVX-512 instructions can even be considered sensitive, as a task executing them can (accidentally) reduce the amount of computing power available to other tasks, because heavy use of AVX-512 causes the CPU to temporarily decrease its clock frequency.

[2]A privileged instruction traps when executed in user mode, but not in supervisor mode [36, p. 415].

4 Design and Implementation

AVX-512 instructions can cause the CPU core executing them to reduce its clock rate, lest it should consume too much power or generate excessive heat. The reduced clock rate is maintained for a short time after the last AVX-512 instruction, slowing down the scalar part of mixed workloads, and thereby potentially degrading performance of the application. Related work (see Chapter 2) has shown that *core specialization* can be used to mitigate this effect. This thesis extends the KIT Operating Systems Group's existing implementation of core specialization for AVX-512 for Linux, making it transparent and automatic. The analysis shown in the previous chapter affirms the decision to use core specialization rather than other scheduling strategies, and motivates our decision to implement fault-and-migrate. Other strategies for identifying a task's phases of AVX-512 were ruled out because they either required access to the source code, which is not always available, would fail for code loaded or generated at runtime, or even defeat the purpose.

This chapter explains the design and implementation of transparent automatic core specialization for AVX-512 using fault-and-migrate, and motivates the underlying decisions. We make AVX-512 instructions trap by disabling the corresponding XSTATE components in the `XCR0` control register during FPU context switching, after the old state has been saved, and before the new one is loaded. Then, AVX-512 instructions raise invalid opcode traps. The trap handler then triggers migration of the offending task to a specialized core using the existing CPU affinity mechanism. Multiple heuristics for re-migration are tried: a timeout after the last detectable use of AVX-512, re-migration on any context switch, and re-migration on system call. For automatically determining the number of AVX-512 cores, moving averages of consumed total and AVX-512 CPU time are used, assuming that the ratio of AVX-512 CPU time to total CPU time is equal to the ratio of AVX-512 cores to total cores.

4.1 Fault-and-Migrate

As Section 3.2 has established, it has to be investigated how to make the CPU raise an exception for the kernel to handle in case a task attempts to execute AVX-512. In turn, the exception handler can then trigger migration.

4.1.1 Making AVX-512 Trap

It is possible, using the performance counters, to count the number of cycles a core has executed in one of the throttled power licenses (see Subsection 5.1.3). Counters can be set up to overflow instantly on the first event by presetting them to their maximum value.

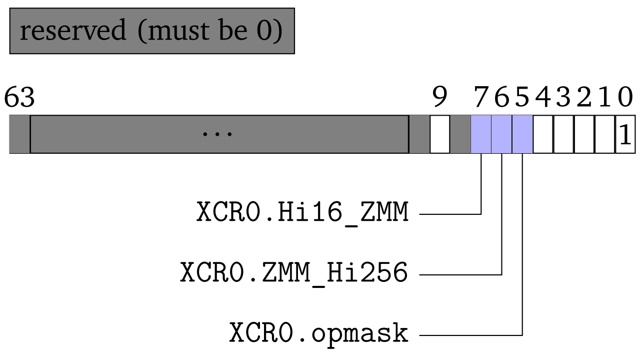


Figure 4.1: AVX-512-related bits of the `XCR0` control register [20, Figure 2-8]

Also, they can be configured to generate an interrupt on overflow [20, Volume 3, Chapter 18], such that the overflow interrupt handler mark the corresponding task. However, using throttling events to detect AVX-512 usage would have the disadvantage that by the time AVX-512 usage is detected, the core would already reduce its clock frequency for at least 2 ms. Therefore, another mechanism is necessary, which detects AVX-512 before throttling occurs.

Current x86 CPUs support the `XSAVE`/`XRSTOR` feature set [20, Chapter 13] for saving and restoring processor state to or from specific data structures in memory. `XSAVE` is advantageous for operating systems, because it can save large parts of CPU state, including the state of SIMD extensions, in an optimized way[1], with only a single instruction. The operating system can select which parts of state to save by setting bits in the CPU-core-specific `XCR0` register, as shown in Figure 4.1. Disabling the state components for AVX-512 in `XCR0` not only prevents these parts of CPU state from being saved/restored, but also renders AVX-512 instructions inoperable on the respective CPU core. [20, Figure 2-8]. While the respective bits in `XCR0` are 0 instead of 1, AVX-512 instructions will cause an `#UD` (invalid/undefined opcode) trap.

To avoid accidentally destroying userspace floating-point unit (FPU) state by not saving it, we can only disable AVX-512 during context switches after the old FPU state has been saved. Switching the userspace FPU context during the task switch is a two-part process [43, `arch/x86/include/asm/fpu/internal.h`]. The `switch_fpu_prepare()` function saves the old context, and `switch_fpu_finish()` restores the new context. We modify `switch_fpu_finish()` such that AVX-512 is enabled/disabled after saving the old context and before restoring the new one. Our implementation disables AVX-512 during each task switch on a scalar core to ensure that AVX-512 is always disabled before returning to userspace, as in-kernel traps because of unavailable AVX-512 are handled without migration by enabling AVX-512 on the respective core (see also Subsection 4.1.3).

[1]Some forms of `XSAVE` only save parts of CPU state which are either not in their initial configuration or have been modified since the last execution of `XRSTOR` [20, Chapter 13.6].

4.1.2 Triggering Migration

When a task attempts to execute an AVX-512 instruction on a CPU core on which AVX-512 is disabled as described in the previous subsection, the core raises an invalid opcode trap (`#UD`), which we want to use to detect AVX-512 usage to trigger migration. However, not all such traps are caused by AVX-512 instructions. The `#UD` handler has to determine whether the trap was caused by an AVX-512 instruction on a scalar core, or an actual illegal opcode or the `BUG()` macro in the kernel [43, `arch/x86/include/asm/bug.h`], which is implemented using the special `UD2` instruction [20, instruction set reference, m-u], that is explicitly defined to raise an invalid opcode exception. To simplify the handler and avoid accessing user memory to inspect the offending instruction, the handler assumes that an invalid opcode trap (not caused by an invocation of the `BUG()` macro) on a core without active AVX-512 is caused by an AVX-512 instruction. Then, it sets the type of the task accordingly, marks it for re-scheduling so that the migration mechanism of MuQSS can come into action, and returns without changing the task's program counter register. On the interrupt return path, the kernel invokes the scheduler, which then selects another non-AVX-512 task to run on the scalar CPU and moves the previous task to an AVX-512 core. Because the program counter register is not changed, execution of the previous task resumes at the previously trapping AVX-512 instruction.

4.1.3 Handling AVX-512 in The Kernel

User space tasks are easy to migrate between CPU cores, whereas migration of kernel code is more difficult. We do not yet implement fault-and-migrate for kernel code, because AVX-512 generally is not used in the kernel, apart from rare exceptions such as RAID6 acceleration [43, `lib/raid6/avx512.c`]. The target scenario for this thesis does not exercise any kernel code paths making use of AVX-512. Instead, the trap handler enables AVX-512 when an in-kernel invalid opcode trap happens because of AVX-512 code.

4.2 Core Specialization

The existing implementation of core specialization (as described in Section 2.4), which this thesis extends, modifies the CPU affinity mechanism of MuQSS, which restricts execution of a task to the set C of cores specified in its CPU affinity bitmap. Wherever the scheduler would originally have used the CPU affinity attribute of the task data structure, the newly introduced `allowed_cpus()` function is called instead. Depending on the task type, it returns either the task's original set C of CPUs or $C \cap C_{\text{AVX-512}}$, the intersection of C and the set of AVX-512 cores. When a task's type is changed to make it an AVX-512 task and it is rescheduled, the scheduler checks on which CPUs the task is allowed to run, and migrates it to an AVX-512 core, because `allowed_cpus()` returns $C \cap C_{\text{AVX-512}}$.

We modify the existing implementation to make $C_{\text{AVX-512}}$ dynamic. When an AVX-512 instruction traps on a core, and the number of AVX-512 cores is either zero or lower than an upper limit calculated heuristically, the respective core is added to $C_{\text{AVX-512}}$.

Periodically, the scheduler checks for each AVX-512 core whether it can be removed from the set again. That is the case when there are no AVX-512 tasks running on it anymore, or there are no AVX-512 tasks in the system anymore.

Core specialization for AVX-512 will cause performance asymmetry in the form of reduced clock frequency on the AVX-512 cores, which needs to be addressed when scheduling scalar tasks which can run on either set of cores. Scalar tasks should be scheduled on scalar cores instead of AVX-512 cores, where they would suffer from reduced performance because of the lower clock frequency. The implementation we extend accomplishes that by applying a penalty to the deadlines of scalar tasks on AVX-512 cores, such that MuQSS preferentially schedules scalar tasks on non-AVX-512 cores.

4.3 Determining the Number of AVX-512 Cores

The previous section described the dynamic set of AVX-512 cores, to which cores can be added until an upper limit for its size is reached, leaving open the question of how to calculate this limit. The smaller the number of specialized cores, the fewer cores can be affected by frequency reductions, and the more CPU time on cores running at their regular clock frequency is available to the scalar parts of tasks. It is thus important that the number of specialized cores be minimized, while also minimizing latency introduced by AVX-512 tasks waiting for execution, lest scalar cores would unnecessarily be idle waiting for tasks to migrate back from specialized cores. In effect, no set of cores must be a bottleneck for the other.

To require the least amount of tuning by the user, and because workloads differ in their ratio of scalar and AVX-512 code, the number of specialized cores has to be determined dynamically. Assuming that the ratio of specialized cores needed to total available cores corresponds to the ratio of CPU time for AVX-512 to total CPU time, the approximate number of specialized cores can be directly calculated by maintaining moving averages of the used total (t_{total}) and AVX-512 ($t_{\text{AVX-512}}$) CPU time:

$$n_{\text{specialized}} = \frac{t_{\text{AVX-512}}}{t_{\text{total}}} \cdot n_{\text{total}}$$

Moving averages are necessary to react quickly to changing workloads, as the regular CPU time accounting would look at an overly long period. We maintain exponentially weighted moving averages (EWMAs) [17] per CPU core, as then they are easy to update in the CPU time accounting of the scheduler. For calculation of the global target number of AVX-512 cores, the sum of per-CPU averages is used. On every scheduler clock tick, the EWMA t of CPU time is updated with the time t_{elapsed} which has passed since the last tick, such that for the n-th scheduler tick, t_n is calculated as follows:

$$t_n = \alpha \cdot t_{\text{elapsed},n} + (1-\alpha) \cdot t_{n-1} = \alpha \cdot \sum_{i=0}^{n} (1-\alpha)^i \cdot t_{\text{elapsed},n-i}$$

The α factor determines the rate at which the weights of old values exponentially decrease. A value $t_{\text{elapsed},n-k}$ used in the calculation k iterations ago only has a weight of $\alpha \cdot (1-\alpha)^k$.

If the respective CPU has been idle since the last tick, t_{elapsed} is zero, such that the moving average decays. Similarly, in the calculation of $t_{\text{AVX-512}}$, t_{elapsed} is zero in case the CPU has not been executing an AVX-512 task, such that $t_{\text{AVX-512}}$ decays during execution of scalar code.

4.4 Re-Migration Heuristics

Determining when to migrate a task to a specialized core is trivial. It has to move as soon as it attempts to execute AVX-512 instructions on a non-specialized core, lest the scheduler would risk having the core affected by reduced clock frequency. In contrast, it is not trivial to decide when to migrate a task back from a specialized core, because unlike the onset of a task's AVX-512 use, it's end is not directly detectable. To determine experimentally which re-migration heuristic works best, all of the following were implemented. Also, because all these heuristics make assumptions about the task's behavior, there might not be one ideal heuristic for all possible workloads.

Timeout Assuming that mixed workloads have alternating phases of AVX-512-accelerated and non-accelerated computation, a timeout mechanism can be used to detect transitions from accelerated back to non-accelerated computation. Figure 4.2 illustrates this re-migration heuristic. As soon as the timeout expires without the task having used AVX-512, it is migrated away from the specialized core. This approach creates the problem of detecting whether the task has used AVX-512 since it has last been scheduled. The previously mentioned `XSAVE` mechanism tracks which state components have been used, allowing the scheduler to determine whether the previously running task has used AVX-512 while switching tasks. This method is based on proposed kernel patches by Li [31], which add an approximate timestamp of last AVX-512 to tasks. When switching tasks, the scheduler tests whether the tasks timestamp of AVX-512 use is sufficiently far in the past, and resets the task type accordingly.

On Context Switch Again assuming the aforementioned alternating phases of computation, migrating tasks away upon the first context switch (e.g. caused by timeslice expiration) is a simpler heuristic than a timeout. The scheduler unconditionally resets the task type when switching tasks, instead of checking for timeout expiry. This unconditional heuristic as well as the next are illustrated in Figure 4.5. We implement a separate configurable timeslice duration for AVX-512 tasks, such that scheduling on AVX-512 cores may be very fine-grained, avoiding execution of scalar code on AVX-512 cores at the cost of more migrations.

On System Call System calls cause kernel entries, and kernel code generally does not use AVX-512. Thus, this heuristic migrates the task away as it attempts to invoke a system call, such that the system call and subsequent execution of the task itself will happen on a scalar core. The generic x86 system call wrapper is changed to reset the task's type and trigger a re-scheduling, causing the task to be migrated away from the AVX core before the system call is executed, as shown in Figure 4.3. We also test a variant which does not re-schedule the task before the actual system call runs, shown in Figure 4.4.

4.5 Orthogonality of Approaches

The solution presented in this thesis strives for versatility and applicability to diverse workloads. To keep it as general as possible, and maintain compatibility with the solution it extends [14], instrumentation and fault-and-migrate need to be combined. Moreover, that would create prerequisites for future automatic instrumentation (see Section 6.1).

A mechanism for core specialization requires at least the set of cores to specialize and information when to execute which task on which set of cores, and works irrespective of the information source. Therefore, fault-and-migrate can be made orthogonal to the existing instrumentation-based approach, by implementing it such that it only supplies information to the underlying mechanism.

4.6 Debugging and Configuration

To evaluate the effect of different policies or parameter values without having to laboriously and time-consumingly re-compile the kernel and wait for the system to reboot, we need a way to configure policies and scheduler parameters such as the set of AVX-512 cores at runtime. SystemTap [40] is a versatile instrumentation tool that was extensively used in the development process for fault finding. Even though it can change kernel data, it is only able to do so in the so-called *guru mode*, forgoing safety measures otherwise preventing write access. Moreover, using SystemTap in evaluation scripts (see Chapter 5) would be more difficult than writing and reading files.

Therefore, the necessary scheduler data and parameters are exposed in Linux's *debugfs* [43, `Documentation/filesytems/debugfs.txt`] instead, which is a special file system allowing developers to easily make kernel data available to userspace in the form of a directory tree. Unlike other virtual filesystems of the Linux kernel such as procfs or sysfs, the layout and contents of debugfs are not subject to any rules and may be changed at any time, as it is not meant to be part of the Application Binary Interface (ABI).

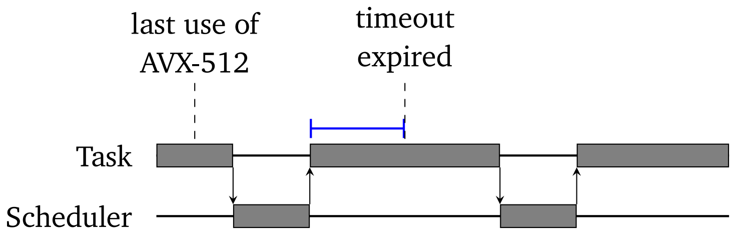


Figure 4.2: Timeout-based re-migration

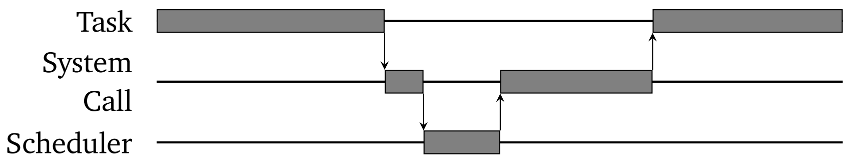


Figure 4.3: Re-migration on system call entry

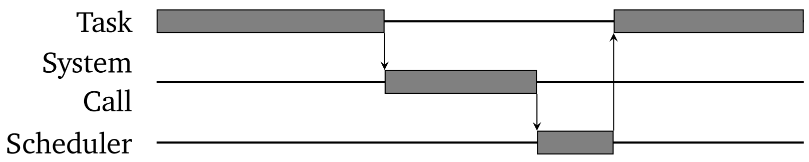


Figure 4.4: Re-migration on system call return

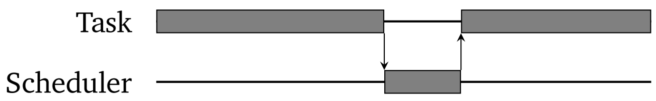


Figure 4.5: Re-migration on timeslice expiry or context switch

5 Evaluation

AVX-512 can cause a temporary reduction in clock frequency, which persists beyond the last AVX-512 instruction, degrading the performance of mixed workloads which have only a small fraction of AVX-512-accelerated code by slowing down the scalar part. Previous research has shown that isolating AVX-512 code from non-AVX-512 code using core specialization can mitigate this effect. This thesis extends the KIT Operating Systems Group's existing implementation of core specialization for AVX-512, which required applications to be instrumented. To make it fully transparent and automatic, we augment the existing implementation with a fault-and-migrate mechanism which makes AVX-512 instructions trap on non-AVX-512 cores such that the trapping task can subsequently be migrated to an AVX-512 core, and later migrated back again according to a heuristic.
In this chapter, we evaluate our solution regarding the following questions:

How well does core specialization using fault-and-migrate mitigate the AVX-512-induced performance reduction of mixed workloads?
We use a web server scenario derived from the one Krasnov [28] described, running nginx to serve a small static file via HTTPS (ChaCha20-Poly1305 and brotli compression), optionally accelerating encryption and message authentication code (MAC) with AVX-512. The wrk2 load generator is used for throughput and latency measurements. We compare different re-migration heuristics against (non-)AVX-512 baselines and each other and investigate the influence of other parameters, especially the number of AVX-512 cores.

We find that our mechanism for determining the best number of AVX-512 cores does not yet work well, resulting in variable and unsatisfactory performance. However, using a static set of AVX-512 cores, the AVX-512-induced performance impediment is almost completely mitigated and the AVX-512 case achieves 99% of the scalar base case's throughput.

How does our solution compare to the implementation it extends?
We run the same web server benchmark with the web server instrumented for explicit migration and compare it to our solution under identical conditions, in particular the same static set of AVX-512 cores. Our best-performing configurations achieve higher throughput (99% resp. 97% of non-AVX-512 baseline).

Why does our solution work?
We assume that core specialization can mitigate the effect of AVX-512-induced throttling on mixed-workload performance by isolating non-AVX-512 code from AVX-512 code. To verify this assumption, we use hardware performance counters and find that our solution indeed limits throttling to AVX-512 cores.

What is the theoretical limit of speedup?
Using an OpenSSL microbenchmark, we measure the speedup of the cipher and MAC in isolation and calculate a theoretical upper limit (analogous to Amdahl's Law) for the combination of nginx and OpenSSL. As the AVX-512-accelerated part of our mixed workload is very small, the speedup because of AVX-512 would likely not be noticeable even if there was no throttling.

Why does our solution not reach it? What causes the remaining overhead?
We run microbenchmarks to measure the average trap latency ($\overline{t_0} \approx 370ns$), and the number of kernel instructions a fault and subsequent task migration take on average (approximately 11000). We compare MuQSS (with and without our modifications) against CFS and rule out that the scheduler itself causes the overhead. To measure the the overhead of fault-and-migrate, we configure the BIOS of our test machine to disable throttling of AVX-512 and run the application scenario again, finding that fault-and-migrate causes approximately 5% overhead over AVX-512 without migration.

What is the effect of fault-and-migrate on latency?
Real-world web servers are not operated at full CPU utilization, so for more realistic latency measurements, we also run the web server benchmark at much lower CPU utilization. We find that fault-and-migrate has slightly lower latency than the AVX-512 baseline.

The rest of this chapter is structured as follows: First, Section 5.1 describes the machine used for our experiments and our methods, i.e. patching feature detection to force applications to use or not to use AVX-512, ensuring repeatability through automation of benchmarks, and recording hardware and software performance counters. Then, Sections 5.2, 5.3, 5.4 and 5.5 answer the research questions listed above. Concluding this chapter is Section 5.7 in which we discuss the evaluation results and assess the overall quality of our approach.

5.1 Setup and Methods

The experiments run on a machine equipped with an Intel® Core™ i9-7940X CPU [6], 32 GB of DDR4 memory, and 250GB of NVMe SSD storage. See Table 5.1 for more detailed specifications of the CPU.

Table 5.1: Intel Core i9-7940X specifications [6]

Cores	14
Threads	28
Base Frequency	3.10 GHz
Max. Turbo Frequency	4.30 GHz
Cache	19.25 MiB

To prevent the operating system from changing the clock frequency of CPU cores, thereby interfering in the experiments, all cores use Linux's `performance` CPU frequency governor. Measurements are taken on an otherwise unloaded machine.

5.1.1 Patching CPU Feature Detection

Some software such as OpenSSL, which is used in our evaluation, detects present and enabled CPU features to select the best code paths for the given machine. To compare scalar against AVX-512 code paths, we need to influence this selection, for example by modifying the feature detection logic.

By patching OpenSSL's CPU feature detection logic, two variants of `libcrypto` are created, one of which always takes the AVX-512 code paths, and the other never takes them. Applications which dynamically link against `libcrypto` can be made to use either variant of it by means of the `LD_LIBRARY_PATH` environment variable [30].

5.1.2 Repeatability

All experiments are automated using scripts, ensuring repeatable setup and execution. Benchmark results are serialized to disk for subsequent analysis and visualization. Parameterized experiments allow exploration of variables, that is determining the influence of a variable by running a series of benchmarks with different values for it. Before each benchmark run, all our modifiable scheduler parameters (see Section 4.6) are either set to their default value, or the programmed value for the specific benchmark.

We will measure the directly observable application performance, such as throughput or latency, and also collect hardware and operating system performance counters using *perf* [43, `tools/perf`]. The next subsection describes the hardware counters in more detail.

5.1.3 Performance Counters

To ascertain that we can prevent throttling of non-AVX-512 cores, we need to measure throttling on each core. Counting the cycles executed under each power license is possible using the *configurable performance counters*. Table 5.2 lists the performance monitoring events corresponding to the aforementioned power licenses. Intel processors feature configurable performance counters for each logical CPU [20, Volume 3, Chapter 18], which count specific *performance monitoring events* selected using model-specific registers. A complete list of the numerous events is available in the Software Developer's Manual [20, Volume 3, Chapter 19]. Only a limited number of counters is available (depending on the processor model), so when wishing to count more events than there are counters, tools such as `perf` [43, `tools/perf`] have to multiplex, trading off accuracy[1]for a more comprehensive set of metrics.

[1]However, the Software Developer's Manual warns in numerous places [20, Volume 3, Chapter 18] about the counters being only approximate even if no multiplexing is used. Also, because of interrupt processing delays, instruction retirement, and compulsory pipeline flushing on interrupt, events may *skid*, that is, be inaccurate relative to the program counter [25].

Table 5.2: Performance monitoring events for frequency levels: For each of these events, the cycles the core has spent in the respective power level are counted [18, Section 17.26].

Level	Event Name	Description
0	`CORE_POWER.LVL0_TURBO_LICENSE`	core executes at maximum frequency
1	`CORE_POWER.LVL1_TURBO_LICENSE`	core executes at reduced frequency (maximum frequency for AVX-2)
2	`CORE_POWER.LVL2_TURBO_LICENSE`	core executes at further reduced frequency (maximum frequency for AVX-512)

5.1.4 The “Perf” Tools

This thesis uses perf extensively to obtain performance counter values during development and for evaluation. *Perf* [43, `tools/perf`] is a comprehensive suite of performance analysis tools for Linux, allowing, among other functions, recording and analysis of both hardware- and software-based performance counters, such as those mentioned in the previous subsection. Counter values can be obtained for the whole system or only a select application, filtered by user or kernel mode. The collection can be restricted to a set of CPU cores, and values can be reported in an aggregated manner or per CPU core. However, perf may incur some overhead. Therefore, some parts of the evaluation will additionally report benchmark results obtained while not collecting perf metrics.

5.2 Mitigation of Frequency Reduction Effects

To measure how well core specialization actually mitigates the performance degradation, we use an application scenario modeled after the mixed workloads considered by this thesis. Based on the CloudFlare example [28], it uses the `nginx` web server [39], serving a small (approximately 72 KiB) static file via HTTPS. Using the `LD_LIBRARY_PATH` environment variable, one of the two OpenSSL variants is selected to choose whether to run with or without AVX-512 (see Subsection 5.1.1). We test web server performance using the `wrk2` benchmark [41], which measures both throughput and latency. To prevent the web server and benchmark client from interfering with each other’s scheduling, they are pinned to disjoint sets of CPU cores. Also, the benchmark client uses a non-AVX-512 variant of the OpenSSL library to prevent it from interfering with the throttling of the AVX-512 cores, because the maximum turbo boost frequency is also determined by the characteristics of other cores (see Section 2.2). Additional metrics are collected using perf, if need be.

In contrast to actual datacenter use, CPU utilization in this application benchmark is near 100%. On the test machine, it causes almost complete utilization of the cores `nginx` runs on, whereas in the datacenter, CPU utilization will be lower [10]. However, at any constant utilization, reduced overhead allows for increased throughput, such that mitigating AVX-512-induced throttling is still worthwhile.

First, we determine which of our re-migration policies work best for the application scenario, and then we investigate how the number of AVX-512 cores and the α parameter of the CPU time EWMA influence throughput.

5.2.1 Re-Migration Heuristics

Figure 5.1 shows the heuristics presented in Section 4.4 in comparison for different AVX-512 task timeslice lengths. To study the effect of the policies in isolation, the mechanism for dynamically choosing AVX-512 cores was not used, but the set of AVX-512 cores was statically configured to contain one core. Using the timeout-based re-migration heuristic (with a timeout of 0 scheduler ticks, so it should re-migrate immediately when scheduling an AVX-512 task which has stopped using AVX-512), only very little throughput is achieved. Monitoring of CPU utilization and task types during benchmark execution reveals that, when using the timeout mechanism, only the AVX-512 core is used, because the timeout mechanism fails to detect the end of AVX-512 usage, which we assume happens because of a defect in our implementation. Re-migrating upon system calls, in contrast, results in throughput close to the non-AVX-512 baseline, almost completely mitigating the performance impediment. Re-migrating upon context switches achieves almost as much throughput, but only for very short AVX-512 task timeslice lengths. Therefore, re-migration upon system call appears to be the best heuristic.

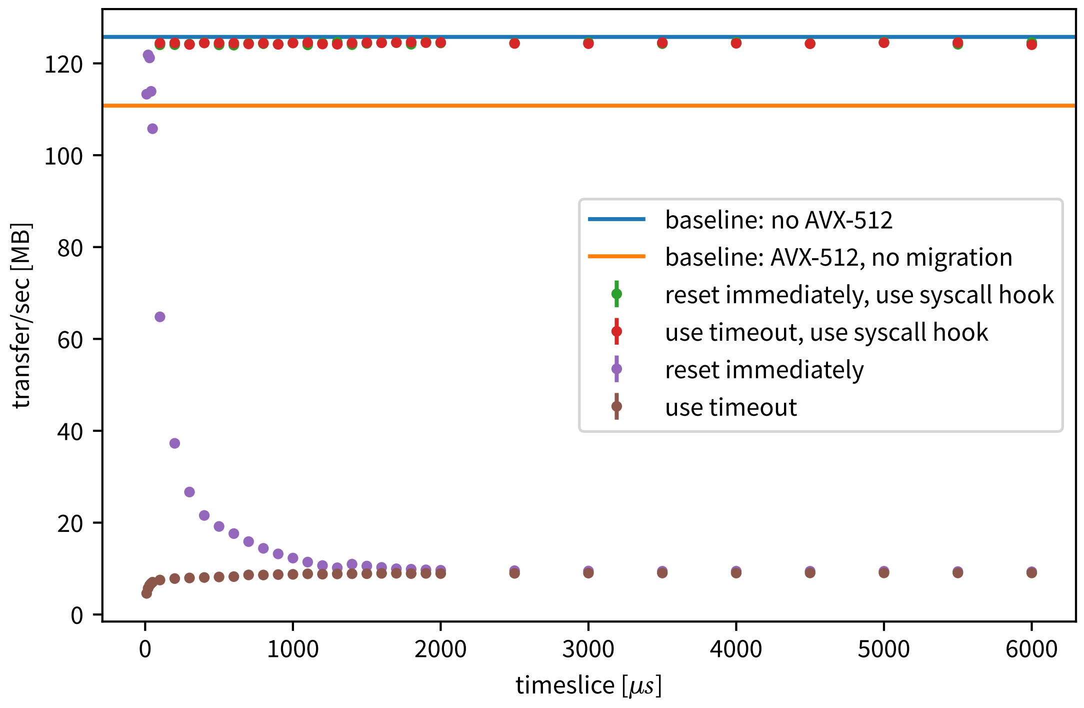


Figure 5.1: Web server throughput: Comparison of different re-migration heuristics and AVX-512 task timeslice length.

5.2.2 Selection of AVX-512 Cores

Figure 5.2 shows how the throughput of nginx with OpenSSL varies depending on the re-migration policy and the selection of AVX-512 cores, comparing static selection of increasing numbers of cores against dynamic selection with increasing upper limits on the number of AVX-512 cores. To maximize performance and establish a meaningful benchmark for automatic core selection, the static selection chooses them in a topology-aware manner, simultaneous multithreading (SMT) siblings[2] first, such that in all configurations with even numbers of AVX-512 cores no logical thread of our two-way SMT system is used for scalar tasks while the other logical thread causes the shared physical core to reduce its clock frequency.

For each of these experiments, Table 5.3 shows the mean throughput and parameters for the best-performing configurations (i.e. number of AVX-512 cores, static/dynamic selection, re-migration heuristic). We make the following observations:

Performance reduction is almost completely mitigated Using a static $C_{\text{AVX-512}}$, the average throughput is much better, consistently at least as good as the existing solution this thesis extends (marked in the chart with *explicit migration*). The best throughput with fault-and-migrate, reaching 99% of the non-AVX-512 baseline, is achieved with a static set of two AVX-512 cores which are SMT siblings.

Re-Migration on syscall entry is disadvantageous We had assumed that re-scheduling on syscall entry would improve throughput by running system call code on scalar cores. However, contrary to our expectations, the measured average throughput (marked in the chart with *re-schedule*) was lower than with regular scheduler invocation on the system call return path. Re-scheduling on syscall entry presumably results in extra scheduler invocations, such that the task is re-scheduled on syscall entry and return, causing the overhead.

Dynamic selection of AVX-512 cores is inadequate Throughput is unsatisfactory and extremely variable when using our method for automatically determining $C_{\text{AVX-512}}$. Monitoring during benchmark execution shows that the method tends to choose too many cores as well as add and remove cores overly frequently.

[2]logical threads sharing a physical thread

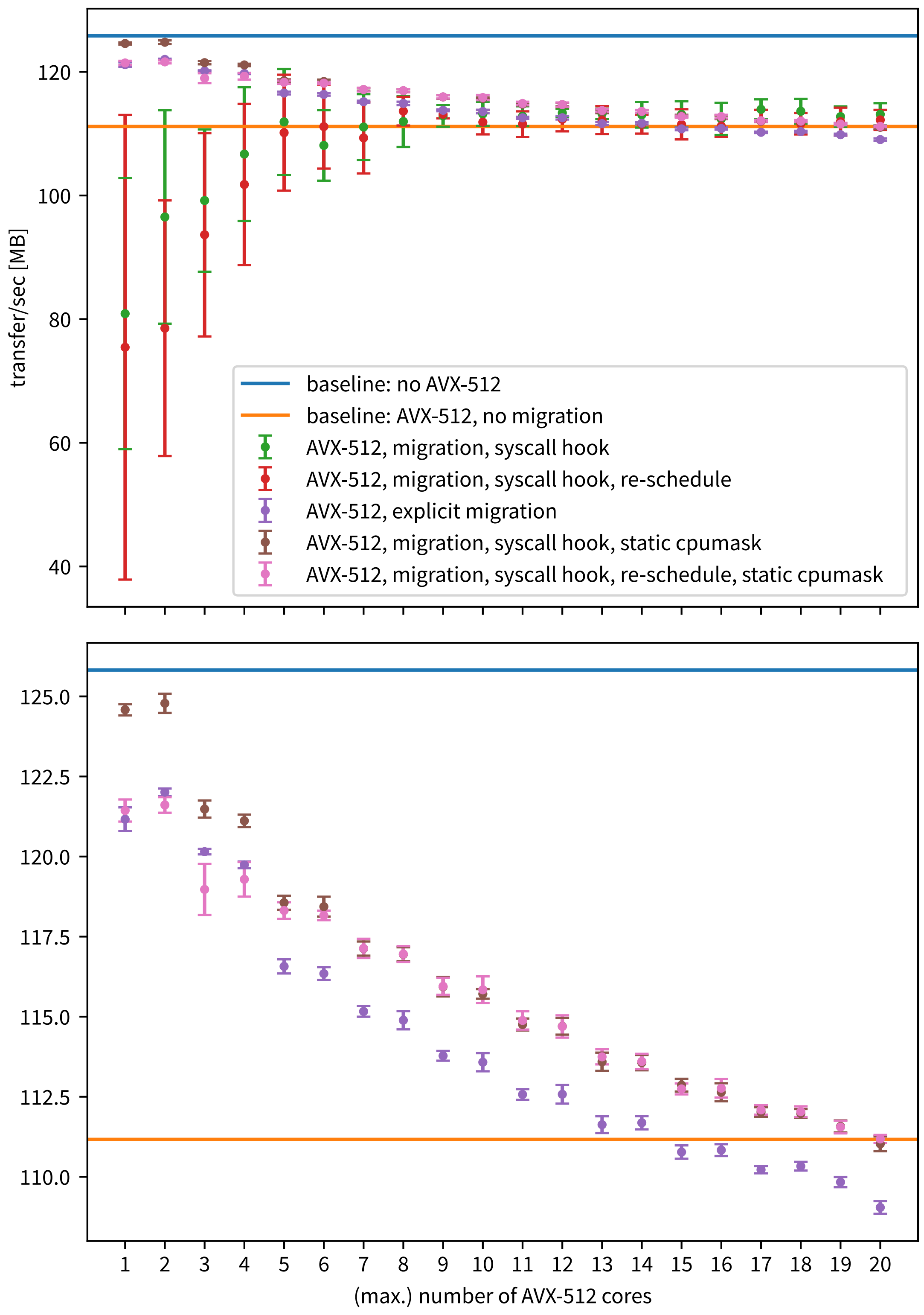


Figure 5.2: Web server throughput: Influence of the number of (maximally allowed or statically allocated) AVX-512 cores on throughput. The bottom plot shows the measurements with static $C_{\text{AVX-512}}$ in more detail, as some of their differences are small.

Table 5.3: Throughput and benchmark parameters for best-performing configurations in Figure 5.2

	baseline: no AVX-512	AVX-512, migration, syscall hook, static cpumask	AVX-512, explicit migration	AVX-512, migration, syscall hook, re-schedule, static cpumask	AVX-512, migration, syscall hook	AVX-512, migration, syscall hook, re-schedule	baseline: AVX-512
Throughput [MB/s] (mean)	125.82	124.78	122.01	121.61	113.93	113.64	111.17
Throughput [MB/s] (stddev.)	0.17	0.30	0.12	0.24	1.62	2.32	0.15
AVX-512	no	yes	yes	yes	yes	yes	yes
(maximum allowed) number of AVX-512 cores	–	2	2	2	17	8	–
syscall hook	no	yes	no	yes	yes	yes	no
reschedul on syscall entry	no	no	no	yes	no	yes	no

5.2.3 Influence of the α Parameter

The previous experiment has shown that automatic selection of AVX-512 cores results in low and overly variable performance. To explore the influence of the α parameter used in calculation of the exponentially moving average (see Section 4.3) of AVX-512 and total CPU time on performance, we re-run the application scenario with AVX-512, automatic core selection and a wide range of values for α. Figure 5.3 shows the results. Smaller values, which mean a lesser influence of the most recent CPU time consumption, perform better, but no α value resulted in throughput on par with static core selection.

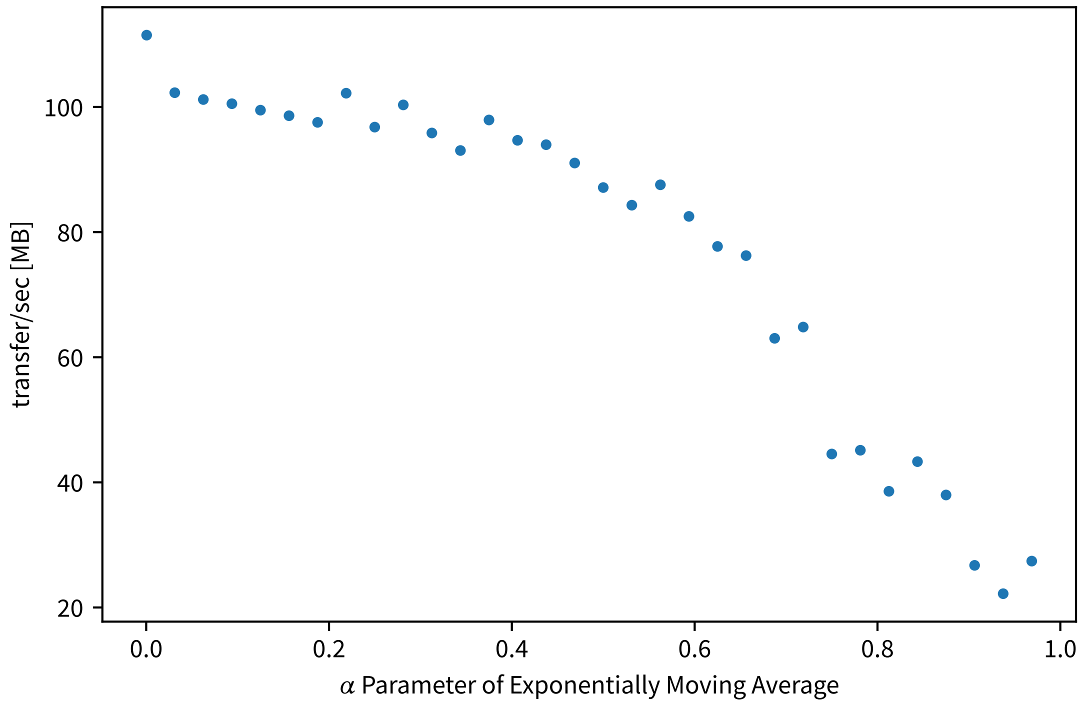


Figure 5.3: Influence of exponentially moving average parameter

5.2.4 Isolation of Throttling

Core specialization using fault-and-migrate successfully mitigates the performance impediment because it concentrates throttling to AVX-512 cores. Table 5.4 shows the per-core numbers of `core_power.lvl2_turbo_license` events during three executions of our application scenario. Running the variant without AVX-512 produces no throttling events on any core. The AVX-512 variant, when executed without automatic migration, exhibits exhibiting throttling on all cores used for the web server. Executing the AVX-512 variant with automatic migration, throttling events only occur on the single core (CPU04) used for AVX-512 in this experiment and its SMT sibling (CPU18), demonstrating that our solution can successfully limit throttling to AVX-512 cores.

Table 5.4: Per-CPU counts of `core_power.lvl2_turbo_license` events during application scenario: The set of AVX-512 cores is fixed (only one core, CPU04) such that the effect of core specialization on throttling can be observed. Other parameters (e.g. the set of CPUs for nginx) are the same as in the rest of this section.

	no AVX-512	**AVX-512, no migration**	**AVX-512, fixed $C_{\text{AVX-512}}$, automatic migration**
CPU04	0	$15.33{\cdot}10^9$	$64.80{\cdot}10^9$
CPU05	0	$16.08{\cdot}10^9$	0
CPU06	0	$15.31{\cdot}10^9$	0
CPU07	0	$15.93{\cdot}10^9$	0
CPU08	0	$15.54{\cdot}10^9$	0
CPU09	0	$14.58{\cdot}10^9$	0
CPU10	0	$14.71{\cdot}10^9$	0
CPU11	0	$11.70{\cdot}10^9$	0
CPU12	0	$15.19{\cdot}10^9$	0
CPU13	0	$14.01{\cdot}10^9$	0
CPU18	0	$15.32{\cdot}10^9$	$64.74{\cdot}10^9$
CPU19	0	$16.07{\cdot}10^9$	0
CPU20	0	$15.30{\cdot}10^9$	0
CPU21	0	$15.92{\cdot}10^9$	0
CPU22	0	$15.53{\cdot}10^9$	0
CPU23	0	$14.57{\cdot}10^9$	0
CPU24	0	$14.70{\cdot}10^9$	0
CPU25	0	$11.69{\cdot}10^9$	0
CPU26	0	$15.18{\cdot}10^9$	0
CPU27	0	$14.01{\cdot}10^9$	0

5.3 Theoretical Limit of Speedup

Benchmarking the cipher and MAC used in the application scenario in isolation shows an 1.589-times speedup of the AVX-512 variant, suggesting that the application scenario should (without throttling) run faster with AVX-512 than without it, and leading to the question of a theoretical upper limit on its speedup.

Analogous to Amdahl's Law, we can establish an upper limit for the theoretical speedup of a mixed workload using AVX-512 without any throttling by considering the fraction of computation that uses AVX-512 and the speedup of that part. In the workloads considered in this thesis, only a potentially very small part of the workload is accelerated using AVX-512.

Let $q \in [0,1]$ be the part of the computation which is AVX-512-accelerated and a the factor of acceleration, then s is the overall speedup.

$$s = \frac{1}{(1-q)+\frac{q}{a}}$$

Using perf, we establish that ChaCha20-Poly1305 is present in $q \approx 1.5\%$ of call stack samples in the application scenario. To determine a, we use a microbenchmark that examines acceleration of computation by AVX-512. It compares scalar and AVX-512 implementations of operations used in the application scenario, namely the *ChaCha20* cipher. OpenSSL 1.1.1a [33] provides the `speed` subcommand, which measures the amount of data that can be encrypted using the given cipher and number of parallel threads in a set time. Table 5.5 and Figure 5.4 show benchmark results and CPU frequency during benchmark execution for runs of `openssl speed -evp chacha20-poly1305 -multi 28` (28 threads, one for each CPU thread), comparing the variants with/without AVX-512 (see Subsection 5.1.1). These results show that AVX-512 can accelerate computation despite the frequency reduction it causes.

Table 5.5: Throughput measured in OpenSSL microbenchmark

Block Size	**Throughput** GiB/s	
	No AVX-512	**AVX-512**
16	4.50	4.08
64	10.72	10.04
256	17.09	19.98
1024	18.63	28.29
8192	19.08	30.20
16384	18.89	30.10

The OpenSSL microbenchmark shows that ChaCha20-Poly1305 is accelerated by $a \approx 1.589$. Applying the above formula, the maximum speedup then is $s \approx 1.0056 = 0.56\%$. This means that a speedup of the AVX-512 variant of the application scenario with our solution would not have been expectable, because even if the AVX-512 part of our application scenario were not subject to clock frequency reductions, we would not have observed a significant speedup.

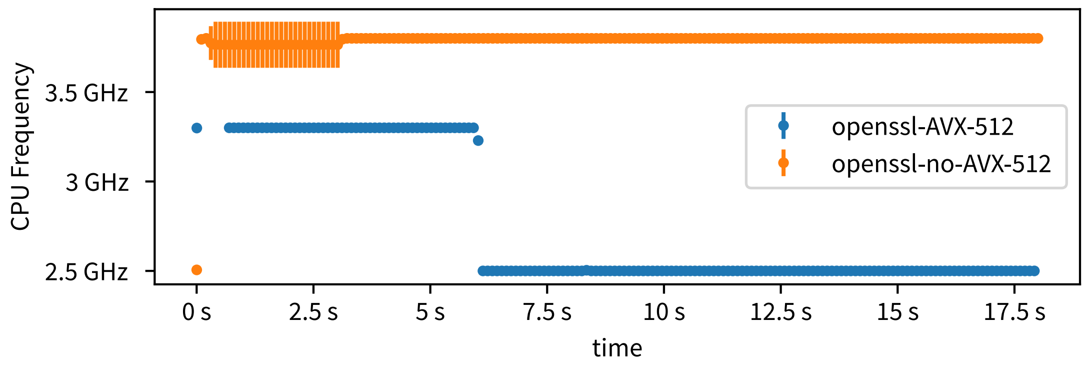


Figure 5.4: CPU frequency during OpenSSL microbenchmark (with throttling)

5.4 Remaining Overhead

Even in the best configurations, the measured throughput falls slightly short of the theoretical maximum speedup calculated in the previous section. As the theoretical limit of performance has not been reached, we have to determine the cause(s). Possibly the performance impediment could not be completely mitigated, and/or new overhead has been introduced because of direct and indirect costs of trap handling and task migration.

5.4.1 Trap Handling

The hardware and operating system take some time to raise a trap, handle it (even if the actual handler function does nothing), and return to the task that has caused the trap. To measure the average trap and handling latency, a simple benchmark is devised. We change the kernel trap handler to do the bare minimum necessary to allow a dummy workload to progress, i.e. skip the offending instruction. The dummy workload consists of a tight loop executing an AVX-512 instruction for a known number n of iterations. To avoid inflating the execution time of the non-trapping base case, we replace the AVX-512 instruction with a no-operation (NOP) of equivalent length. To calculate the average overhead of a single invalid opcode trap $\overline{t_o} = \frac{t_2 - t_1}{n}$, the difference in execution times is divided by n. The costs of program startup and the loop, which we assume to be invariant, are included in both measurements, and therefore cancel each other out by subtracting one execution time from the other, so that only the overhead remains. For $n = 8 \cdot 10^6$ we measure $t_1 = 2.13ms$ in the non-trapping case and $t_2 = 2958.84ms$ in the trapping case using `perf stat`, so the cost of a single trap is $\overline{t_o} \approx 370\,\text{ns}$. That would mean, for example, only $\frac{50000 \cdot 370\,\text{ns}}{1\,\text{s}} = 0.0185 = 1.85\%$ direct overhead even at 50000 exceptions per second.

This measurement includes indirect costs of traps (e.g. induced instruction cache misses). However, they are not representative for the real trap handler, because the dummy trap handler has a much smaller cache footprint.

5.4.2 Fault-and-Migrate

Even though the direct cost of traps is moderate, the subsequent migration could be expensive. To measure the cost of fault-and-migrate, we need a scenario in which no frequency reductions occur, such that the effects of fault-and-migrate can be studied in isolation. Allegedly, that should be possible by configuring the UEFI BIOS of our test machine to disable the frequency reduction.[3] Measurements reveal that even if there is no frequency reduction for AVX-512 configured, the OpenSSL microbenchmark still runs at lower frequencies when using AVX-512. Figure 5.5 compares the different clock frequency behaviors.

[3]However, such configurations are unsuitable for production use, and we assume the CPU might become unstable because we expect it to exceed thermal and/or electrical limits.

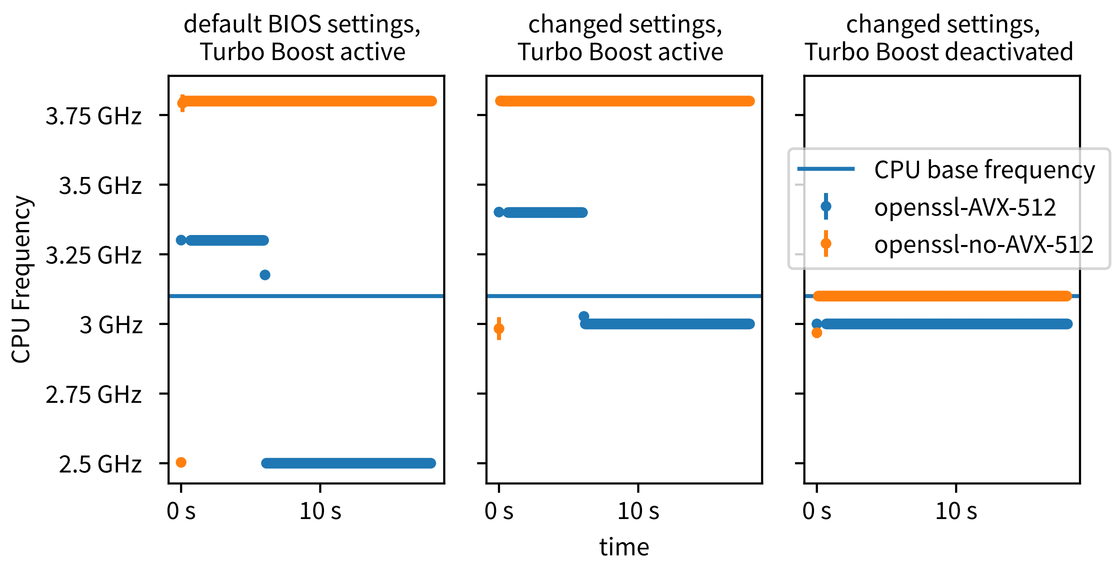


Figure 5.5: Clock frequency during execution of OpenSSL microbenchmark: The plots show the average clock frequency of all CPU cores during execution of the same microbenchmark as in Section 5.3. Note that the standard deviation for most samples is so small that the error bars lie within the points.

The frequency drop after a few seconds when using AVX-512 is most likely explained by the CPU exceeding its TDP in the beginning, which it may temporarily do for additional performance (see Section 2.2), and then adhering to the TDP again, which requires lower frequencies. When deactivating Turbo Boost[4] too, the frequency drop disappears, and the difference in clock frequencies disappears mostly (though still not completely).

To measure the cost of fault-and-migrate and compare it against the cost of explicit migration, we run a simple microbenchmark. We divide the number of additional executed kernel instructions (compared to a base case without migration) by the number of iterations to obtain the average number of kernel instructions per migration, finding that fault-and-migrate needs approximately $\frac{65.25 \cdot 10^9 - 20.48 \cdot 10^9}{4.08 \cdot 10^6 - 2.16 \cdot 10^3} = 11 \cdot 10^3$ kernel instructions, about the same as explicit migration, which needs $\frac{66.67 \cdot 10^9 - 20.48 \cdot 10^9}{4.10 \cdot 10^6 - 2.16 \cdot 10^3} = 11 \cdot 10^3$. Table 5.6 shows the detailed results.

In the microbenchmark, each of n threads executes an AVX-512 instruction and then relinquishes the CPU using `sched_yield()` in a tight loop for i iterations, such that (in the migration cases) each iteration results in two migrations, for $2 \cdot n \cdot i$ migrations in total. The AVX-512 instruction triggers the fault-and-migrate mechanism. Later, the task is migrated away from the specialized core again when it yields the CPU. For comparison, we also run a version of the benchmark which uses Gottschlag and Bellosa's [12, 14] system call instead of fault-and-migrate.

[4]Using Linux's Intel P-State driver, Turbo Boost can be deactivated by writing 1 to `/sys/devices/system/cpu/intel_pstate/no_turbo`.

In the base case, all cores are configured as AVX-512 cores such that the fault-and-migrate mechanism will not be triggered. In the migration cases, there is only one AVX-512 core to force the maximum number of task migrations.

Between "No Migration" and "Fault-and-Migrate", the number of executed user mode instructions differs (approximately) by the $n \cdot i$ AVX-512 instructions which are counted twice because their first execution traps. For $n = 256$ threads and $i = 8000$ iterations, we expect $4.10 \cdot 10^6$ migrations. Perf counts $4.08 \cdot 10^6$, only very slightly less, presumably because some scalar tasks were scheduled on AVX cores where they could not trigger another migration.

This simple synthetic microbenchmark is not representative of realistic workloads, because the indirect costs resulting from migrations will vary. However, it shows qualitatively what to expect. Fault-and-migrate requires the execution of more kernel code and induces additional instruction cache misses. Therefore, to obtain an estimate for real-world applications, we also measure the overhead of fault-and-migrate in our application scenario below.

Running the same web server scenario as in Section 5.2 with AVX-512 but no throttling and with/without fault-and-migrate, we find that fault-and-migrate causes a 5% reduction in throughput. Table 5.7 shows the throughput and metrics measured using perf during the web server benchmark execution on our test machine. Even though the machine was configured not to throttle CPU cores executing AVX-512, and turbo boost was deactivated, the AVX-512 cores still ran at lower frequencies (3.00 GHz instead of 3.10 GHz). This 3.23 % clock frequency reduction coincides with the slightly lower (2.29 %) throughput reduction of AVX-512 without migration compared to no AVX-512.

Table 5.6: Metrics for fault-and-migrate microbenchmark: Averages and standard deviations across 100 runs measured using perf

Metric	No Migration Avg.	No Migration Stddev.	Fault-and-Migrate Avg.	Fault-and-Migrate Stddev.	Explicit Migration Avg.	Explicit Migration Stddev.
migrations	$2.16 \cdot 10^{3}$	1.24%	$4.08 \cdot 10^{6}$	0.01%	$4.10 \cdot 10^{6}$	0.00%
instructions (kernel)	$20.48 \cdot 10^{9}$	0.01%	$65.25 \cdot 10^{9}$	0.07%	$66.67 \cdot 10^{9}$	0.01%
instructions (user)	$21.10 \cdot 10^{6}$	0.00%	$23.13 \cdot 10^{6}$	0.00%	$72.30 \cdot 10^{6}$	0.00%
i-cache misses (kernel)	$11.98 \cdot 10^{6}$	0.39%	$315.77 \cdot 10^{6}$	0.29%	$313.34 \cdot 10^{6}$	0.89%
i-cache misses (user)	$1.23 \cdot 10^{6}$	0.94%	$5.20 \cdot 10^{6}$	1.60%	$13.07 \cdot 10^{6}$	5.18%

Table 5.7: Perf metrics, throughput and latency for fault-and-migrate overhead in the application scenario: Perf metrics are normalized per HTTP request. The *Fault-and-Migrate Overhead* column compares fault-and-migrate against AVX-512 without migration. Because perf has some overhead of its own, we also show throughput without perf.

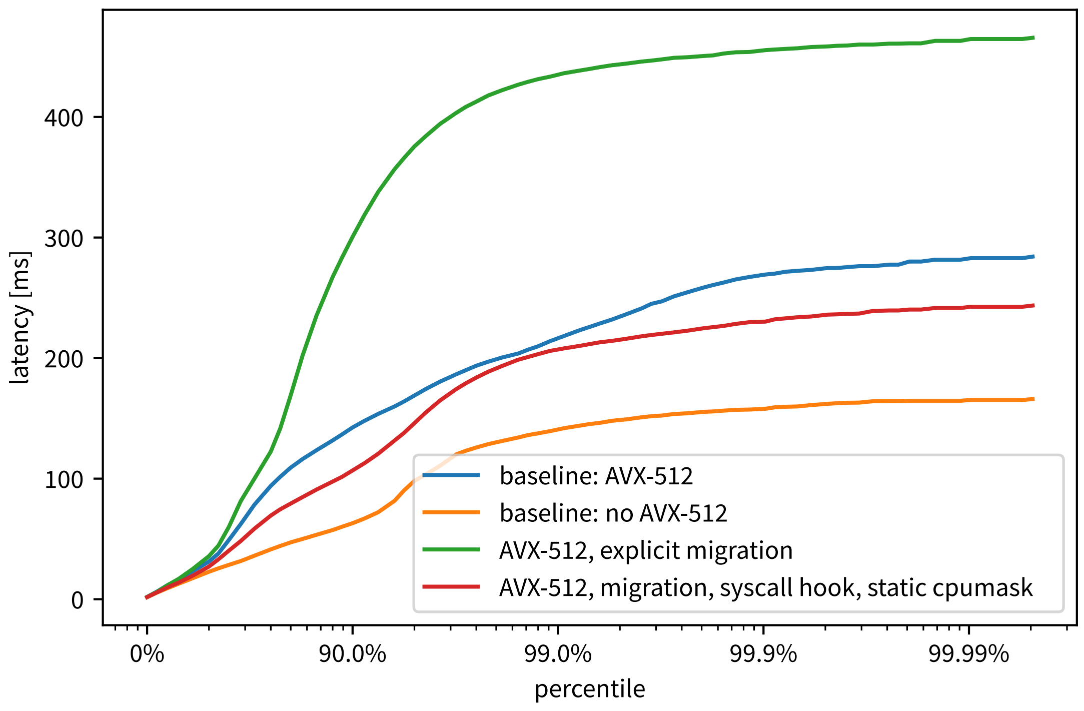
Metric	No AVX-512	AVX-512, No Migration	AVX-512, Fault-and-Migrate	Fault-and-Migrate Overhead
i-cache misses	$14.91 \cdot 10^{3}$	$15.30 \cdot 10^{3}$	$18.02 \cdot 10^{3}$	17.78%
last-level cache load misses	990.32	992.88	$1.14 \cdot 10^{3}$	14.46%
last-level cache store misses	$5.12 \cdot 10^{3}$	$5.07 \cdot 10^{3}$	$5.96 \cdot 10^{3}$	17.46%
migrations	$889.39 \cdot 10^{-3}$	$906.53 \cdot 10^{-3}$	3.49	285.47%
instructions (kernel)	$307.28 \cdot 10^{3}$	$309.77 \cdot 10^{3}$	$348.10 \cdot 10^{3}$	12.37%
instructions (user)	$10.28 \cdot 10^{6}$	$10.15 \cdot 10^{6}$	$10.15 \cdot 10^{6}$	−0.00%
traps because of AVX-512	0.00	0.00	2.20	
reference cycles (kernel)	$467.66 \cdot 10^{3}$	$486.53 \cdot 10^{3}$	$714.40 \cdot 10^{3}$	46.83%
reference cycles (user)	$8.93 \cdot 10^{6}$	$9.09 \cdot 10^{6}$	$8.67 \cdot 10^{6}$	−4.60%
transfer/sec	103.87	101.36	95.03	6.25%
transfer/sec without perf	104.66	102.26	97.02	5.12%

5.5 Latency

In practical applications of network services, request latency influences user experience. As our evaluation is based primarily on an HTTP server, the effect of fault-and-migrate on latency is interesting. Figure 5.6 illustrates latency distribution at low numbers of concurrent requests per second for the best-performing configurations established in Section 5.2. The chart shows that our solution achieves a lower maximum latency than the implementation we extend (shown in the chart as “explicit migration”), and a latency distribution very similar to the AVX-512 baseline, even though it still exhibits higher latency than the non-AVX-512 baseline.

To obtain realistic latency measurements, we run the same web server benchmark as described in Section 5.2 at significantly lower load because web servers in practical use are not run at full CPU utilization [10]. Throughput (requests per second and data transfer rate) during benchmark execution was the same for all measurements, because the number of concurrent requests the wrk2 load generator should perform was chosen sufficiently low for the web server not to be limited by CPU utilization.

Figure 5.6: Latency by percentile distribution



5.6 The Multi-Queue Skiplist Scheduler

The previous benchmarks all used MuQSS. For completeness and to rule out MuQSS as a cause of remaining overhead, we compare it against the CFS. Table 5.8 shows throughput and latency for both schedulers in comparison. We run the same application scenario as above, without AVX-512, and without tuning any parameters for either scheduler. The comparison shows that MuQSS alone, both with and without our modifications, does not cause the remaining overhead.

Technically, the methods presented in this thesis could also be applied to the CFS, because its thread migration mechanism, even though it works differently from that of MuQSS, could be used in the same way to implement core specialization by migrating tasks. Facilitating the porting of the remaining logic necessary for automatic and transparent core specialization is the fact that many functions in MuQSS and CFS have identical names and purposes.

Table 5.8: Web server throughput and latency using MuQSS and CFS. Extreme latency values are caused by the high system utilization this benchmark induces.

	CFS	**MuQSS**	**MuQSS (modified)**
maximum latency [s]	9.99	7.58	7.75
mean latency [s]	4.79	4.14	4.17
latency stddev. [s]	1.55	1.31	1.31
requests [1/s]	$7.62 \cdot 10^3$	$7.94 \cdot 10^3$	$7.93 \cdot 10^3$
transfer [MB/s]	121.03	126.16	125.93

5.7 Discussion

This chapter has so far shown that our approach can successfully mitigate AVX-512-induced performance reduction in mixed workloads, achieving 99% of the throughput of the non-AVX-512 variant. However, our prototype has the following limitations:

- Our mechanism for determining the optimum number of AVX-512 cores does not perform satisfactorily and chooses too many cores, impairing performance.
- The evaluation presented in this chapter has shown that our approach works, but it has only demonstrated the viability for one particular workload. Further evaluation using diverse workloads is necessary to validate our solution or reveal further potential for improvement.
- Generally, intercepting all AVX-512 instructions might be an overly extreme measure, lacking the insight into the prospective effect of the instruction mix on the clock frequency that advanced hardware feedback, such as a notification mechanism for impending frequency reduction, could provide.
- We do not achieve complete transparency, because our manipulation of the `XCR0` control register in enabling/disabling AVX-512 breaks certain feature detection mechanisms which read the aforementioned control register. For this thesis, we simply patched the feature detection of our evaluation workload, because we needed to do so anyway to force our evaluation workload to either always or never use its AVX-512 code paths. Future research might solve this problem, for example through virtualization of `XCR0`.

6 Conclusion

Increasing fractions of a contemporary CPU's silicon area can not efficiently be used either because of insufficient software parallelism, or because the chip has to operate at a reduced frequency if they are used. This share of silicon area is called *dark silicon*. A recent example of dark silicon is AVX-512, a family of modern SIMD extensions for Intel x86, enabling great acceleration of vectorizable code, but consuming much power, requiring CPU cores using it to reduce their clock frequency not to violate thermal and electrical constraints. This frequency reduction is maintained until a timer expires without any event requiring another frequency change, such that it persists beyond the last AVX-512 instruction. Mixed workloads can suffer an overall slowdown because of this behavior, as some scalar code following AVX-512 code runs at the reduced clock frequency.

Related work has shown that core specialization, that is the preferred use of certain cores for certain types of computation [37], can mitigate this performance reduction by isolating AVX-512 and scalar code from each other, such that much less scalar code is affected by the frequency reduction. The Operating Systems Group at the KIT has implemented core specialization in Linux by modifying the Multi-Queue Skiplist Scheduler (MuQSS) and introducing a new system call by which programs inform the scheduler about the beginning or end of their AVX-512 use, such that the scheduler can migrate the task to a subset of cores or away again.

In this thesis, we extended the KIT Operating Systems Group's existing implementation of core specialization for AVX-512 to be fully transparent and automatic. This we achieved by adding a fault-and-migrate mechanism which automatically triggers migration of tasks attempting to use AVX-512 on a scalar core, heuristics for re-migration of tasks away from AVX-512 cores, and a heuristic for calculating the number of necessary AVX-512 cores. Our fault-and-migrate mechanism efficiently virtualizes AVX-512 instructions to intercept them on scalar cores such that they raise an invalid opcode trap. The corresponding trap handler detects the trap's cause, marks the task as an AVX-512 task, and flags it for rescheduling, during which it is migrated. Of the heuristics we tested, re-migration upon system call worked best, which changes the type of an AVX-512 task to scalar as soon as it performs a system call on an AVX-512 core and marks it for rescheduling, such that it is moved to a scalar core.

Because the existing implementation uses a newly introduced system call, applications need to be modified to benefit from it, whereas our solution does not. Also, our solution aims to require very little tuning for specific workloads, ideally none at all, especially automatically determining the number of AVX-512 cores to allocate, even though the mechanism tends to overestimate the number of cores needed for AVX-512. Future work should find out why this happens or develop a better mechanism.

We evaluated our implementation using custom microbenchmarks and a web server scenario derived from the one publicised by Krasnov [28]. The microbenchmarks affirm that AVX-512 can substantially accelerate computation on our test system (ChaCha20-Poly1305 from OpenSSL experiences a speedup of approximately 1.6), and that the average overall latency of traps (the trap itself, handling it, and returning to user space) is not prohibitively large, at approximately 370ns. Using the web server scenario, we compared our different re-migration heuristics against each other, and our solution against the existing core specialization implementation it extends. We found that using our mechanism for automatically determining the set of AVX-512 cores $C_{\text{AVX-512}}$ results in low and overly variable performance, and discovered that the mechanism tends to choose too many cores and change them too frequently. However, when configuring $C_{\text{AVX-512}}$ statically, we found that our solution performs better than the previous implementation, almost completely mitigating the performance reduction in the web server scenario. The system call-based re-migration heuristic, which migrates tasks away from AVX-512 cores, worked best, confirming our assumption that system calls delimit phases of program execution.

6.1 Future Work

In light of the deficiencies the evaluation of our implementation has revealed, we identify the following opportunities for further research.

Improvement of Self-Tuning Capabilities First and foremost, the evaluation shows the importance of using the optimum number of cores for AVX-512, and the deficiency of our approach at determining it. A better method of automatically determining the number of AVX-512 cores needs to be found.

Also, this thesis has by far not exhausted the possibilities of control theory. For example, new feedback variables might be devised to control the parameters of our modified scheduler, such as a metric from the user application, which thereby informs the scheduler of its performance.

Fault-and-Migrate for Kernel Code Our prototype does not yet implement fault-and-migrate for kernel code (see Subsection 4.1.3). However, there is some kernel code which might use it, for example for filesystem encryption or RAID acceleration.

Correct Handling of Feature Detection We implemented a mechanism for efficient virtualization of AVX-512 instructions. For the correctness and transparency of the virtualization mechanism, it is necessary that the CPU core appear unchanged, indicating AVX-512 as present and enabled. However, as the mechanism disables the state components of AVX-512 in the `XCR0` register, applications which use the `xgetbv` instruction to detect supported features will correctly read from the register that AVX-512 is disabled and subsequently use other code paths, thereby defeating our intention of using AVX-512. Thus, to make feature detection logic act as if AVX-512 were enabled, it either has to be patched (as for our evaluation), or `xgetbv` needs to be virtualized. The latter is possible by disabling XSAVE/XRSTOR (setting the `OSXSAVE` bit in the `CR4` control register to zero), making all XSTATE-related instructions, including `xgetbv`, trap with a `#UD` fault. which could be handled to virtualize `xgetbv` and other XSTATE operations. Doing so also sets the `OSXSAVE` bit in the output of `CPUID` to zero, such that feature detection by `CPUID` would again perceive features as disabled which we only intend to virtualize. However, the Linux kernel already has an abstraction [16] for the `CPUID` faulting mechanism [24, Section 2.3.2] of modern Intel CPUs which could be used to virtualize `CPUID` as well.

Hardware Feedback As Papamanoglou [35] writes, the scheduler would benefit from any more accurate mechanism of establishing the share of AVX-512 in the used CPU time, such as a performance counter which actually counts AVX-512 instructions, or a hardware event signalling impending throttling. Future CPUs might exhibit different clock frequency behavior, e.g. throttle down for a subset of AVX-512 instructions or not reduce their frequency immediately. Also, a hardware feedback mechanism would enable a more generic implementation of core specialization, as the scheduler would not need to be aware of the instruction mix a core is running, but only of impending throttling.

Improving Thread Migration Our approach causes numerous thread migrations, which could benefit from a faster mechanism. Currently, thread migration (as our approach uses it) in MuQSS works by having the destination CPU inspect the runqueue of the original CPU, requiring that the thread to be migrated first be inserted in the source runqueue. A more efficient mechanism might directly insert the thread in the destination runqueue once it stops running on the source CPU, for example due to a trap.

Other Re-Migration Triggers It might be worthwhile to investigate more possible triggers for re-migration, in search of one that is applicable to the greatest number of possible workloads. For our web server scenario, re-migration on system call worked well. For other scenarios, that might not be the case, as their SIMD-accelerated operations might not be as conveniently followed by system calls.

Porting to Another Scheduler The Multi-Queue Skiplist Scheduler (MuQSS) is not a very widespread scheduler, and originally targets desktop systems instead of servers. To ease adoption of the solution presented in this thesis, it could be ported to a more popular scheduler such as the CFS (Linux's default scheduler).

Combination with Other Forms of Heterogeneity Machines will exist that combine AVX-512 with heterogeneity and/or performance asymmetry, such as different base clock frequencies for CPU cores. The latter has already been implemented by Intel in their *Speed Select Technology* [19], which allows the base frequency of certain cores to be increased. Therefore, it would be advantageous to combine core specialization for AVX-512 with existing scheduling approaches for heterogeneous and/or asymmetric parallel systems, especially that of Li et al. [32], which extends an SMP-fair scheduling algorithm to performance-asymmetric systems.

NUMA Awareness To avoid causing unnecessary overhead, a core-specializing scheduler needs to consider the system's topology. The system used in this thesis made core specialization decisions simple, because it had only one socket, and all processor cores shared a last-level cache and address space. However, there are machines with non-uniform memory access (NUMA), consisting of multiple nodes with local memory, each of which can also access remote memory (memory of other nodes), but at reduced bandwidth and increased latency. The automatic core specialization mechanism presented in this thesis could be extended to be NUMA-aware, e.g. spreading specialized cores over non-uniform memory access (NUMA) nodes to avoid task migration across nodes.

Automatic Instrumentation As Gottschlag and Bellosa [14] and Papamanoglou [35] have demonstrated, instrumenting the application is also a viable approach to core specialization. This thesis has shown that both approaches are orthogonal (see also Section 4.5) by further extending Gottschlag's scheduler such that the kernel has both a system call to mark tasks and the ability to automatically mark and unmark them. We have thus seen *manual instrumentation* of application code and *transparent automatic migration* (completely agnostic of application code), leaving out the possibility of *automatic instrumentation*, that we explicitly decided not to pursue when designing our application. Analogous to existing work on automatic staged computation [15], which splits applications at points identified using simulation and executes the parts on separate cores to increase cache locality, an execution trace from the real system or a simulator could be used to identify possible migration points in the application. Such an execution trace could be obtained by extending fault-and-migrate to feed into one of the existing tracing mechanisms of the Linux kernel. Also, popular open-source compilers [11, 5] support plugins, or could otherwise be extended to perform instrumentation at translation, possibly additionally guided by the aforementioned trace information.

Bibliography

[1] Lars Bauer. "RISPP: A Run-time Adaptive Reconfigurable Embedded Processor." PhD thesis. Universität Fridericiana zu Karlsruhe (TH), 2009.

[2] M. Bohr. "A 30 Year Retrospective on Dennard's MOSFET Scaling Paper." In: *IEEE Solid-State Circuits Society Newsletter* 12.1 (Winter 2007), pp. 11–13. ISSN: 1098-4232. DOI: 10.1109/N-SSC.2007.4785534.

[3] *BoringSSL*. Google. URL: https://boringssl.googlesource.com/boringssl/.

[4] Koushik Chakraborty, Philip M. Wells, and Gurindar S. Sohi. "Computation Spreading: Employing Hardware Migration to Specialize CMP Cores On-the-fly." In: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XII. San Jose, California, USA: ACM, 2006, pp. 283–292. ISBN: 1-59593-451-0. DOI: 10.1145/1168857.1168893. URL: http://doi.acm.org/10.1145/1168857.1168893.

[5] *Clang Plugins – Clang 9 documentation*. URL: https://clang.llvm.org/docs/ClangPlugins.html.

[6] Intel Corporation. *Intel® Core™ i9-7940X X-series Processor*. 2017. URL: https://ark.intel.com/products/126695/.

[7] 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 (Oct. 1974), pp. 256–268. ISSN: 0018-9200. DOI: 10.1109/JSSC.1974.1050511.

[8] H. Esmaeilzadeh et al. "Dark silicon and the end of multicore scaling." In: *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. June 2011, pp. 365–376.

[9] M. J. Flynn. "Some Computer Organizations and Their Effectiveness." In: *IEEE Transactions on Computers* C-21.9 (Sept. 1972), pp. 948–960. ISSN: 0018-9340. DOI: 10.1109/TC.1972.5009071.

[10] P. Garraghan, P. Townend, and J. Xu. "An Analysis of the Server Characteristics and Resource Utilization in Google Cloud." In: *2013 IEEE International Conference on Cloud Engineering (IC2E)*. Mar. 2013, pp. 124–131. DOI: 10.1109/IC2E.2013.40.

[11] *GNU Compiler Collection (GCC) Internals: Plugins*. URL: https://gcc.gnu.org/onlinedocs/gccint/Plugins.html.

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

[13] Mathias Gottschlag and Frank Bellosa. “Mitigating AVX-Induced Performance Variability with Core Specialization.” In: Operating Systems Group, Karlsruhe Institute of Technology (KIT). 2018.

[14] Mathias Gottschlag and Frank Bellosa. “Reducing AVX-Induced Frequency Variation With Core Specialization.” In: *The 9th Workshop on Systems for Multi-core and Heterogeneous Architectures*. Dresden, Germany, Mar. 2019.

[15] Mathias Gottschlag et al. “Towards Fully Automatic Staged Computation.” In: *The 8th Workshop on Systems for Multi-core and Heterogeneous Architectures*. Porto, Portogal, Apr. 2018.

[16] Kyle Huey. *[PATCH v16 08/10] x86/arch_prctl: Add ARCH_[GET|SET]_CPUID*. Mar. 2017. URL: https://lore.kernel.org/kvm/20170320081628.18952-9-khuey@kylehuey.com/.

[17] J. Stuart Hunter. “The Exponentially Weighted Moving Average.” In: *Journal of Quality Technology* 18.4 (1986), pp. 203–210.

[18] Intel Corporation. *Intel™ 64 and IA-32 Architectures Optimization Reference Manual*. Apr. 2019. URL: https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf.

[19] Intel Corporation. *Intel® Speed Select Technology – Base Frequency - Enhancing Performance*. Apr. 2019.

[20] *Intel® 64 and IA-32 Architectures Software Developer’s Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4*. Intel Corporation. May 2018. URL: https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf.

[21] *Intel® Advanced Vector Extensions 512 (Intel® AVX-512)*. Intel Corporation. URL: https://www.intel.com/content/www/us/en/architecture-and-technology/avx-512-overview.html.

[22] *Intel® Architecture Instruction Set Extensions and Future Features Programming Reference*. Programming Reference. May 2018. URL: https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf.

[23] *Intel® Turbo Boost Technology 2.0*. Intel Corporation. URL: https://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html.

[24] *Intel® Virtualization Technology FlexMigration Application Note*. Intel Corporation. Oct. 2012. URL: https://www.intel.com/content/dam/www/public/us/en/documents/application-notes/virtualization-technology-flexmigration-application-note.pdf.

[25] *Intel® VTune™ Amplifier 2019 User Guide – Hardware Event Skid*. Intel Corporation. Feb. 2018. URL: https://software.intel.com/en-us/vtune-amplifier-help-hardware-event-skid.

[26] Con Kolivas. *MuQSS – The Multiple Queue Skiplist Scheduler*. URL: http://ck.kolivas.org/patches/muqss/sched-MuQSS.txt.

[27] Philipp Koppe et al. "Reverse Engineering x86 Processor Microcode." In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 1163–1180. ISBN: 978-1-931971-40-9. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/koppe.

[28] Vlad Krasnov. *On the dangers of Intel's frequency scaling*. Cloudflare, Inc. Nov. 2017. URL: https://blog.cloudflare.com/on-the-dangers-of-intels-frequency-scaling/.

[29] James R Larus and Michael Parkes. "Using cohort scheduling to enhance server performance." In: *LCTES/OM*. 2001, pp. 182–187.

[30] *LD.SO(8)*. Mar. 2017.

[31] Aubrey Li. *[PATCH v9 2/3] x86,/proc/pid/status: Add AVX-512 usage elapsed time*. Feb. 2019. URL: https://lore.kernel.org/lkml/20190211185931.4386-2-aubrey.li@intel.com/.

[32] Tong Li et al. *Operating System Support for Overlapping-ISA Heterogeneous Multi-core Architectures*. Tech. rep. Intel Corporation, 2009.

[33] *OpenSSL Cryptography and SSL/TLS Toolkit*. Nov. 2018. URL: https://www.openssl.org/.

[34] Roberto Paleari et al. "N-version disassembly: differential testing of x86 disassemblers." In: *Proceedings of the 19th international symposium on Software testing and analysis*. ACM. 2010, pp. 265–274.

[35] Ioannis Papamanoglou. "Constructing a Library for Mitigating AVX-Induced Performance Degradation." MA thesis. Karlsruhe Institute of Technology, 2019.

[36] Gerald J. Popek and Robert P. Goldberg. "Formal Requirements for Virtualizable Third Generation Architectures." In: *Commun. ACM* 17.7 (July 1974), pp. 412–421. ISSN: 0001-0782. DOI: 10.1145/361011.361073. URL: http://doi.acm.org/10.1145/361011.361073.

[37] Juan Carlos Saez et al. "Leveraging Core Specialization via OS Scheduling to Improve Performance on Asymmetric Multicore Systems." In: *ACM Trans. Comput. Syst.* 30.2 (Apr. 2012), 6:1–6:38. ISSN: 0734-2071. DOI: 10.1145/2166879.2166880. URL: http://doi.acm.org/10.1145/2166879.2166880.

[38] Christian Schwarz. "Stage-Aware Scheduling in a Library OS." Bachelor Thesis. Operating Systems Group, Karlsruhe Institute of Technology (KIT), Germany, Mar. 2018.

[39] Igor Sysoev. *nginx*. URL: https://nginx.org.

[40] *SystemTap*. URL: https://sourceware.org/systemtap/.

[41] Gil Tene. *wrk2 – a HTTP benchmarking tool based mostly on wrk*. Mar. 2018. URL: https://github.com/giltene/wrk2.

[42] Praveen Kumar Tiwari et al. *Accelerating x265 with Intel® Advanced Vector Extensions 512*. May 2018. URL: https://software.intel.com/sites/default/files/managed/d5/f6/mcw-intel-x265-avx512.pdf.

[43] Linus Torvalds, ed. *Linux 4.17*. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/.

Glossary

ABI Application Binary Interface. 20

AVX Advanced Vector Extensions. 5, 20, 36

AVX-512 Advanced Vector Extensions 512. v, 1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 14, 15, 16, 17, 18, 19, 20, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 49

AVX 2 Advanced Vector Extensions 2, 256-bit predecessor to AVX-512. 3, 5, 6, 7

CAMP Comprehensive Scheduler for Asymmetric Multicore Systems. 10

CFS Completely Fair Scheduler. 8, 24, 39, 43

DWRR dynamically weighted round-robin. 10

EWMA exponentially weighted moving average. 18, 27

FMA fused-multiply-add. 5, 6

FP floating point. 5, 6

FPU floating-point unit. 16

ISA instruction set architecture. 9, 10, 11

JIT just-in-Time Compiler. 14

MAC message authentication code. 23, 24, 32

MIMD Multiple Instruction streams, Multiple Data streams: classical multi-processors [9]. 3

MuQSS Multi-Queue Skiplist Scheduler by Kolivas [26]. 8, 9, 17, 18, 24, 39, 41, 43

NOP no-operation. 34

NUMA non-uniform memory access. 44

OS operating system. 8

PCU power controller unit. 6

RISPP Rotating Instruction Set Processing Platform. 11

SIMD Single Instruction stream, Multiple Data streams: each instruction processes more than one set of operands, for example whole arrays [9]. v, 3, 5, 6, 14, 16, 41

SMP symmetric multiprocessor. 13, 44

SMT simultaneous multithreading. 28, 31

SSE Streaming SIMD Extensions, 128-bit SIMD Extension to x86. 6

TDP thermal design power. 6, 35

VMM virtual machine monitor. 14