Karlsruher Institut für Technologie

Constructing a Library for Mitigating AVX-Induced Performance Degradation

Masterarbeit
von

cand. inform. Ioannis Papamanoglou

an der Fakultät für Informatik

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

Bearbeitungszeit: 25. September 2018 – 24. März 2019

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 24. März 2019

Abstract

AVX-512 is a recent x86 instruction set extension that aims to accelerate vectorizable workloads by increasing the vector size further. The on-CPU SIMD units that make efficient operation on very wide vectors possible, take up a lot of space on the chip and have high power requirements, as a result they are part of the dark silicon of a CPU. The dark silicon of a chip is circuitry that has to be turned off during normal operation, due to high power requirements. To maintain a reasonable TDP, current CPUs reduce the core frequency when those units are active. A lot of workloads benefit from vectorization despite the frequency reduction. However, the high power consumption of wide vector instructions cause additional side effects. Most workloads do not consist of only vectorizable code. To achieve good performance for non-vectorized (scalar) code that is run on the same core, the frequency needs to be increased as soon as possible. Increasing the frequency is inflicted with delays, causing non-vectorized code to run at unnecessarily low frequencies during the changing period. Slowing down the non-vectorized code on a core can result in worse overall performance, making the feasibility of AVX-512 and alike very unpredictable.

In this thesis we create a framework that supports application developers in mitigating the overall system performance degradation induced by AVX-512. We build upon an existing approach that uses core specialization to isolate instructions leading to performance degradation onto a small set of cores, letting scalar parts of the system workload run unrestrained. Application developers can use our library to mark code that potentially executes AVX-512 instructions. We designed a dynamic policy that decides during runtime whether to offload marked code regions onto a set of dedicated cores to prevent them from slowing down subsequent scalar code. While our framework is focused on AVX-512, our design and theories apply to general high power instructions that require a frequency reduction.

We show that our framework can reduce the performance degradation in a realistic web server benchmark from 17% to 9%. When a scalar version of the benchmark is run, our framework does not introduce significant performance penalties, making the framework a useful tool to reduce the risk of unexpected performance degradation caused by AVX-512.

Contents

Chapter 1

Introduction

Accelerating workloads with vector instructions can yield large benefits. Numerous workloads benefit from being executed on SIMD capable GPUs, outweighing the large offloading overhead [KDK+11, KES+09]. Vector instruction set extensions can accelerate many applications for which the GPU offloading overhead are too high [LKC+10]. Historically, the feasibility of vector extensions for single components of applications could be inspected in isolation since it was not dependent on other workloads in a system. Vectorized code interferes on modern CPUs with other code on the system, due to power requirements. Dennard scaling is at its end and with each processor generation more parts of the chip become part of the dark silicon [EBSA+11]. Big power dense execution units, such as the vector ALUs that make AVX-512 possible, require a core frequency reduction during execution to stay within realistic TDP limits [wika]. The power dissipation of a CPU is directly proportional to the frequency and thus the power dissipation can be reduced by reducing the frequency [int04]. We refer to instructions that cause a temporarily reduced core frequency as *high power wide vectorized instructions* (HPWVIs). AVX-512 and AVX2 are good examples for HPWVIs and are the focus of this work. Using HPWVIs can result in low overall system performance even when the accelerated components are running faster [Kra17]. The performance penalty is due to frequency change delays. Changing back to a higher frequency as soon as non-HPWVIs are executed takes about 2ms. This delay causes non-HPWVIs that follow HPWVIs to run at lower frequencies than normally possible [int18a]. Only in some cases can the acceleration of HPWVIs outweigh the slowdown they impose on subsequent code. This is mostly dependent on the ratio of HPWVIs and scalar code and how much they are intertwined. The feasibility of HPWVIs is therefore very dependent on the system workload and thus unpredictable. Using HPWVIs becomes unfeasible if the slowed down non-HPWVIs are critical for good system performance.

In this thesis we construct a framework that supports application developers in mitigating an overall system performance degradation caused by AVX-512. We extend an existing approach that executes critical AVX-512 parts of a workload on a small set of CPU cores. This approach is based on core specialization and mitigates the slowdown effect by isolating AVX-512 code from scalar code [GB18]. The framework we built lets application developers mark code regions that potentially execute AVX-512. Not all marked code regions will always be worth offloading, because the feasibility of offloading always is very dependent on the system and the workload running on there. The dynamic policy we designed maximizes system performance during runtime by choosing marked code regions that shall be offloaded. Our framework offloads code onto dedicated cores and chooses among marked regions of all cooperating applications. If the amount of scalar code on the cores running AVX-512 is minimized and the offloading mechanism is efficient, then the slowdown effect can be nearly completely avoided in theory. In practice some performance is lost due to insufficient system utilization caused by dedicating cores.

In the following chapters we elaborate on our design decisions for a dynamic policy that maximizes overall system performance by choosing code regions with HPWVIs that are worth offloading. The overall thesis is applicable for any high power vector instructions that require a frequency reduction. We further describe our framework supports developers in mitigating the performance degradation effect caused by the recent HPWVIs AVX-512. We inspect the limits of AVX-512 in general and investigate the side effects caused by using it and how efficient our approach is in mitigating them. Our evaluation on a realistic web server setup shows that using our framework results in less performance variability and mitigates the slowdown caused by AVX-512 by more than 50%.

The thesis is structured as follows: In Chapter 2 we provide background to AVX-induced performance degradation, introduce a possible solution to mitigate it and discuss related work. We present vector instructions in general, talk about dark silicon and why some vector instruction execution units are part of it. Further into the chapter we explain why vector instructions cause unpredictable performance degradation and introduce core specialization as a potential solution. We analyse of the feasibility and side effects of AVX-512 and core specialization as a solution in Chapter 3. The design chapter (Chapter 4) explains our framework and policy in detail. We elaborate on our design decisions and challenges to construct a viable HPWVI-induced performance degradation mitigation mechanism. In Chapter 5 (Implementation) we detail specifics of our platform and show a concrete example of how applications mark code for our framework. Chapter 6 contains the evaluation of our framework for realistic workloads. Chapter 7 concludes our insights and proposes future work.

Chapter 2

Background and Related Work

A lot of workloads that operate on larger data words can benefit from using vectorization. Using on-CPU SIMD instructions can accelerate certain workloads through vectorization without the huge overhead involved when using GPUs (Section 2.1). The most recent vector instruction set extensions can operate on words so wide that supplying all transistors with enough power becomes difficult. To reduce power dissipation, this circuitry is switched off when not needed. Such parts of the chip are called dark silicon (Section 2.2). To stay within reasonable TDP boundaries when parts of this circuitry are needed, CPU cores reduce their power requirements by reducing the frequency they run at. While slowed down vectorized instructions outperform most of their scalar counterparts, frequency change delays slow down subsequent code that does not need to run at lower frequencies. The slowdown of scalar code makes wide vector instructions unfeasible in systems where the workload is predominantly scalar (Section 2.3). To mitigate this performance degradation we extend an approach that uses core specialization for isolating the effect onto a small amount of cores (Section 2.4). We supply the mechanism with a policy that finds code worth offloading and construct a framework that helps application developers mitigate the effect in their applications. For generating minimal additional overhead we have to choose an efficient code offloading solution that uses pool threads and closures. Therefore, we describe existing means of code offloading in Section 2.5.

2.1 Vectorized Instructions

Certain workloads require the consecutive execution of the same instructions multiple times on different data. In a classical single instruction single data (SISD) architecture such as x86, this requires letting instructions traverse the complete pipeline every time from the beginning. Modern CPUs include single instruction multiple data (SIMD) instruction set extensions to improve the performance of these kinds of workloads. Those so-called vectorized instructions can be used transparently in between normal scalar instructions. To execute the vectorized instructions, modern CPU cores contain specialized units for vectorized execution next to normal ALUs in the execution engine. For example, such a unit might be a vectorized integer ALU that operates on up to 512 bit wide words instead of only 64 bit words. Those units support interpreting very wide words as a vector consisting of multiple smaller sized words. For example, instead of using 8 times the add instruction to add 8 64 bit integers, a 512 bit vector ALU needs only one instruction to add two 512 bit vectors consisting each of 8 of the integers. There are other execution units in modern CPU cores next to the int ALU and int vector ALU. Figure 2.1 shows the ones in the Intel Skylake-SP architecture.

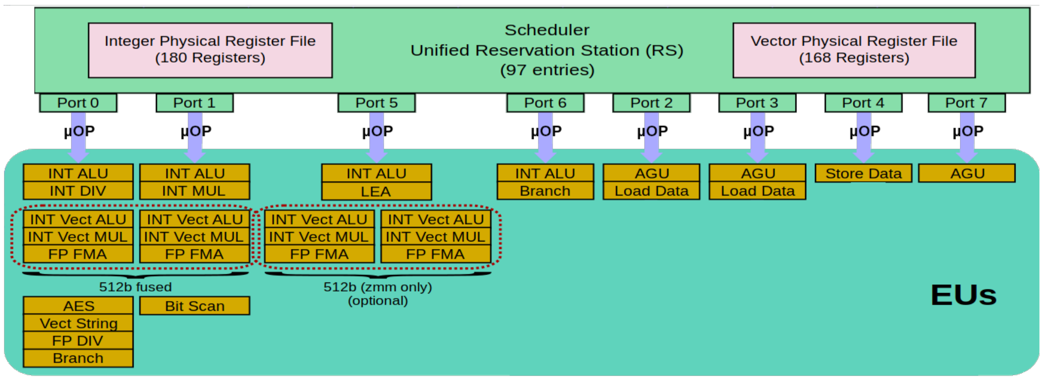


Figure 2.1: Intel Skylake-SP execution units [wikc]

2.2 Dark Silicon

Working with very wide words results in a big chip footprint and many transistors being active at the same time, which ultimately results in high thermal dissipation and power requirements. How much space the unit that executes the vector instruction set extension AVX-512 needs in Intel Skylake-SP chips is shown in Figure 2.2. Processors are required to stay under a certain TDP (thermal design power). Making use of the massively parallel, power-hungry vectorized units would result in exceeding a reasonable TDP. The parts of the circuitry that can not be used at normal operating constraints without exceeding the given TDP are referred to as dark silicon [EBSA$^+$11]. A lot of vectorized units are part of the dark silicon. When parts of the dark silicon are switched on, processors need to decrease their power draw to stay within TDP limits. For vectorized code this is

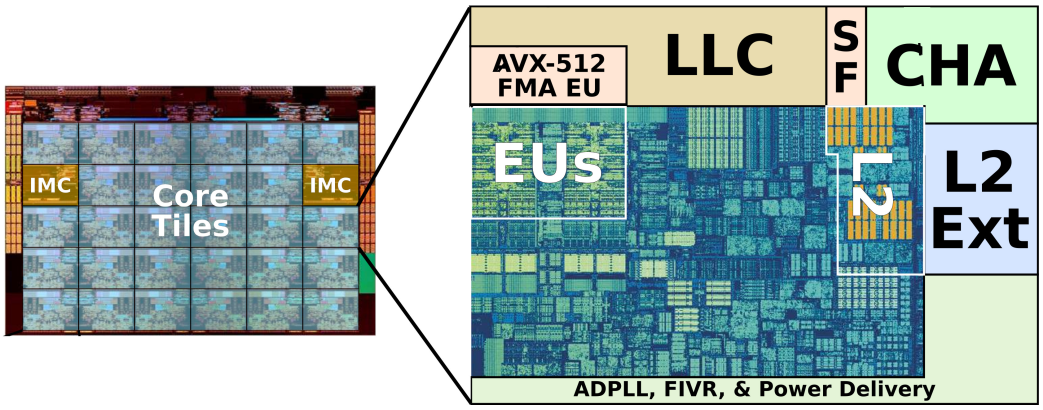


Figure 2.2: On the left side is a 28 core Intel Skylake-SP die. The right side is a bigger picture of a single core. It is visible that a lot of space is taken by caches. Those are not massively parallel in use, so their power requirements are not as high as that of the execution units. The AVX-512 FMA (Fused Multiply Accumulation) EU (Execution Unit) alone takes already half the space as all other EUs together. Additional AVX-512 units are part of the EUs. [wikc]

commonly done by reducing the frequency of cores that are executing high-power-drawing vectorized instructions. For example, Intel added for their Skylake-SP processors two additional frequency bands for AVX-512 and AVX2 like shown in Figure 2.3. In the scope of this work we refer to instructions,that require a power level adaptation as *high power wide vectorized instructions* (HPWVIs).

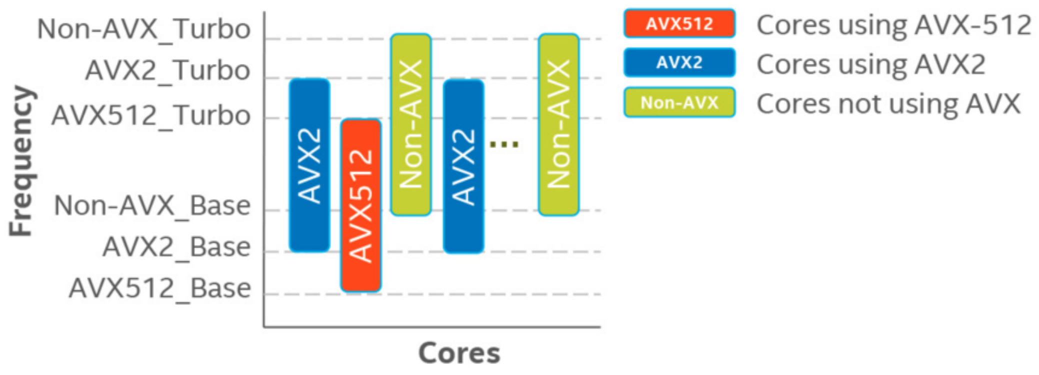


Figure 2.3: Intel Skylake-SP processors have different (lower) frequency ranges for the vector instruction set extensions AVX2 and AVX-512 [wika].

2.3 Performance Degradation

Vectorized code that is run at lower frequencies is in general still faster than a scalar version of that code. For example running twice as wide instructions might only require a 25% drop in core frequency while being up to twice as fast. Whether vectorization is despite the lower frequency beneficial, can be estimated quite well with benchmarks during development. An example can be found in Section 3.1. Aside from having to run vectorized code at lower frequencies, there is another slowdown effect which is more problematic. Most workloads consist of a mix of vectorized and non-vectorized instructions, or more precisely a mix of HPWVIs and non-HPWVIs. When changing between those instructions, the core frequency has to be adjusted. When changing from non-HPWVI to HPWVI code, the frequency is decreased to stay within TDP limits. When changing from HPWVI to non-HPWVI code, the frequency is increased to achieve optimal performance for the non-HPWVI code. Increasing the frequency after the execution of HPWVI code takes a while. While the frequency is being increased, non-HPWVI code that could be normally run at higher frequencies is executed at the lower frequency, slowing it down. The non-HPWVI slowdown problem is illustrated in Figure 2.4 while Figure 2.5 illustrates how using vector instructions should look optimally. In practice not all HPWVIs require the same amount of frequency

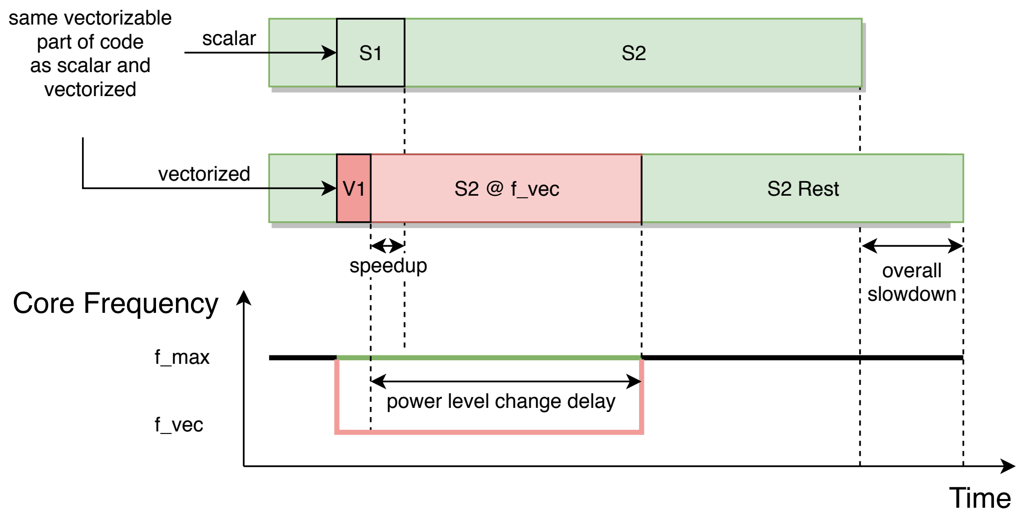


Figure 2.4: Running a vectorized version of a scalar code part might accelerate the vectorized part while slowing down overall performance due to power level change delays.

reduction. HPWVIs are categorized into how much frequency reduction they require. Section 5.1 contains an illustration of the slowdown problem when there

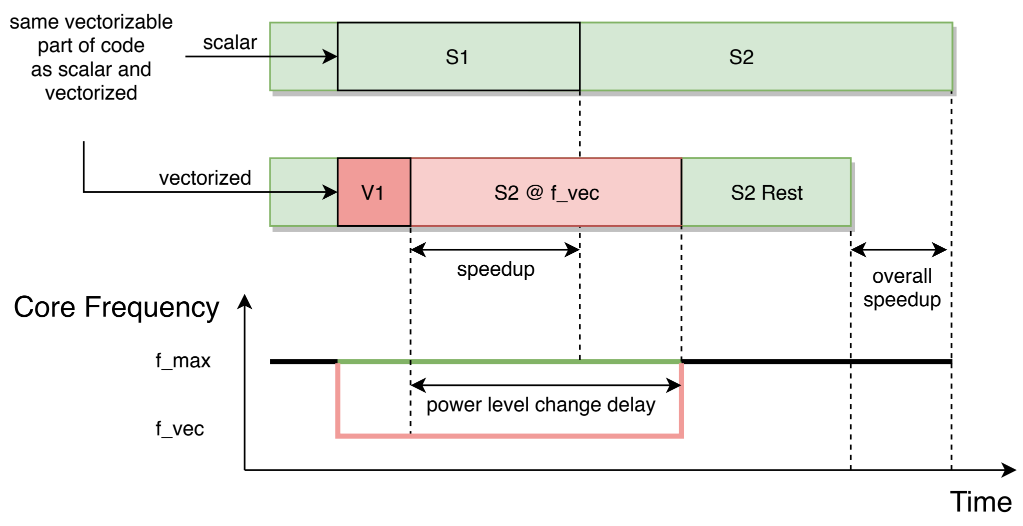


Figure 2.5: Under the right circumstances the benefit of vectorized code can outweigh the slowdown problem.

are multiple power levels.

How much performance is lost by slowing down subsequent code can not be easily inspected during development since it is largely dependent on the system and the workload. In [Kra17] performance (requests/sec) drops up to 10% were observed when using AVX-512 to accelerate OpenSSL in NGINX in comparison to using BoringSSL which does not use AVX-512. This big performance degradation is mainly due to the vectorized portion of the workload being very little. Having only small bursts of HPWVIs results in regular slowdown of the surrounding non-HPWVI code while not providing large enough performance benefits to counteract the performance degradation caused by this slowdown.

2.4 Core Specialization

We chose an approach to mitigate the performance degradation caused by HPWVIs that consists of identifying problematic HPWVI code regions and grouping them together on CPU cores that we dedicate to execute migrated HPWVI core regions [GB18]. This approach is able to bring the 13.4% performance drop when using AVX-512 in the NGINX experiment [Kra17] down to 3.7%. The idea is to accumulate all slowdown-triggering code onto a small pool of cores, while freeing other cores from the slowdown (Figure 2.6). We extend the approach by fully dedicating cores and designing a policy that analyzes when and which code regions should be migrated to the dedicated cores. Finally, we implement a framework that uses those mechanisms to support application developers in mitigating the slowdown in their applications.

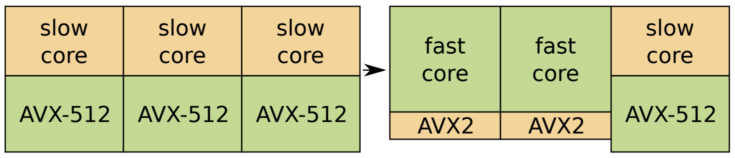


Figure 2.6: Specializing cores like described in [GB18] can reduce the impact of the slowdown caused by vector instructions onto a few cores.

Dedicating cores and effectively remodeling a general purpose multicore system as a reconfigurable asymmetric system to deal with heterogeneous workloads is an approach that is more widely applicable. On heterogeneous systems, moving workloads onto cores that better suit the workload can improve performance in non-AVX-512-related cases, too [BC06]. Migrating work onto other cores can also draw other benefits than performance such as better power efficiency [SPFB10]. FlexSC uses core specialization similarly to our approach to reduce the performance impact of system calls [SS10]. The idea of FlexSC is to increase spacial and temporal locality by grouping system calls on specialized cores.

Server applications that use thread and event-driven models often lack data and instruction locality which results in poor performance due to a lack of proper hardware utilization. Cohort scheduling identifies similar computations in applications and executes them consecutively on a processor in an effort to increase locality [LP02]. To complement cohort scheduling, staged computation is introduced as an alternative programming model to threads and events. In staged computation, applications consist of multiple asynchronously executed stages that are groups of operations that share logic and state and are perfect candidates for cohort scheduling. Our approach to use HPWVI code regions and group them to make better use of processor resources is similar to the idea of stages.

Aside from frequency reductions, inefficient mixes of parallelizable and sequential code lead to pessimal utilization of system parallelism. Compilers can reorder code sequences and divide unrelated instructions to make better use of hardware resources by increasing instruction level parallelism. Region scheduling is an approach that uses data and control dependencies to divide code into regions that are reordered in a way that enables parallel execution of the instructions to better utilize hardware parallelism [GS90].

2.5 Efficient CPU Offloading

Making use of core specialization paired with fine granular offloading requires an efficient mechanism to offload work from one CPU core to another. Efficiently offloading work from the CPU onto other specialized units is a recurring problem. Using GPUs as fast SIMD execution units is common, but is impaired by large latency and synchronization overhead [LM13]. Due to the large overhead in GPU offloading, only very specific workloads can profit from GPU offloading. [AK13] accelerate GPU offloading by using semantically aware caching to minimize transactions and [LM13] reduce latency of large transactions by enabling partial synchronization. Coprocessor accelerator cards such as Intel Xeon Phi can synchronize faster than GPUs and accelerate certain workloads with the vector instruction set extension AVX-512 that GPUs can not accelerate [TKK$^+$13]. Vector instruction set extensions in a core specialization context share the same problem as GPU offloading, but on a different scale. On-CPU SIMD is practically slower than GPU SIMD, but causes less overhead, placing vector instructions somewhere between scalar execution on general-purpose CPUs and using external GPU accelerators. Just like in GPU offloading, the main single application performance bottleneck in core specialization is caused by synchronization delays. In terms of whole system performance, core specialization suffers from lower maximum system utilization, which is described and rationalized in Section 4.2.1.

There are two methods to move work from one CPU core to another one: Either by migrating the thread temporarily onto the other core or by having a thread on the other core that executes a closure. The Linux thread migration mechanism is not optimized for fast and frequent migration [SMM$^+$09], requiring either writing a better performing migration mechanism or introducing the constraint of only being able to migrate closures. Having to use closures is not necessarily a strong restriction, since threads running in the same address space share all memory implicitly. Instead of spawning new threads for each closure, [HPS97] showed that having pool threads amortizes thread creation costs and reduces latency. Executing a closure on a thread on another core requires an efficient IPC and synchronization mechanism. The mean latency of transferring messages $\leq 1\,\text{kB}$ with semaphores and shared memory is under 3us [VJ15], making them a feasible mechanism.

Chapter 3

Analysis

Making use of HPWVIs results not only in a potential performance improvement in comparison to scalar equivalents, but also in side effects regarding performance. We constructed a series of microbenchmarks to observe those effects and find out how to deal with them in isolation. Our exemplary HPWVIs are AVX-512 FMA instructions. Those are the most power intensive HPWVIs in current CPUs. Our machine is described in Section 6.1.

In the first benchmark (Section 3.1) we will prove that using HPWVIs can increase the performance, otherwise all the efforts to mitigate the negative side effects would lose value. We demonstrate the acceleration of FMAs (fused multiply-additions) when using HPWVIs. Further we show that HPWVIs result in CPU frequency reduction.

For the rest of the microbenchmarks we wrote a primesieve calculator that is periodically injected with other code during execution. The primesieve benchmark is described in more detail in Section 3.2. In Section 3.2 we show that using HPWVIs as injected code can result in a poor primesieve performance, which illustrates the performance degradation effect. That experiment also demonstrates that the feasibility of using HPWVIs can not be predicted before runtime.

Our work is based on the idea of using core specialization to isolate the performance degradation onto one core. Section 3.3 shows that offloading HPWVIs onto other cores prevents HPWVI-induced performance degradation, doing so achieves near baseline performance when executing the injected code on another core.

Offloading code onto another core requires a potentially inefficient offloading mechanism. We compare multiple methods in Section 3.4 and find that using a thread pool is the most efficient one.

In conclusion, we show with the microbenchmarks that using HPWVIs can be beneficial but the effect on performance is hard to predict. HPWVIs slow down subsequent scalar code. The effect can be minimized by using core specialization to isolate the effect onto a small set of cores. Offloading is best done by having a thread pool.

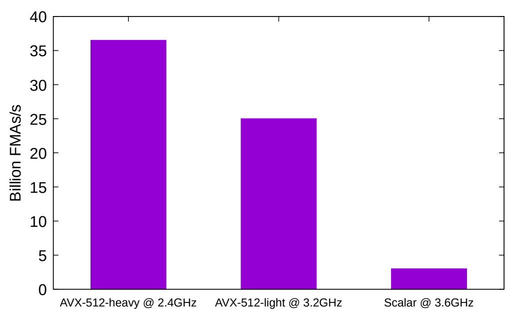
3.1 Acceleration Through Vectorization

In this work we try to mitigate the performance degradation that HPWVIs impose on subsequent scalar code. This benchmark shall show that using vector instructions can result in performance benefits for the right workloads. We measure the performance by calculating the fused multiply-add $\vec{a} \leftarrow \vec{a} \odot \vec{a} + \vec{b}$ for two vectors ($\vec{a}$ and $\vec{b}$) consisting each of eight doubles. We compare how many of those operations can be done per second and per cycle with AVX-512-heavy, AVX-512-light and scalar code. In AVX-512-heavy code the FMA can be performed with a single instruction *vfmadd132pd* on 512 bit registers (zmm) eight times in parallel. For AVX-512-light we choose the same instruction but on 256 bit registers (ymm). Since we need to use twice as many registers, we can only execute four FMAs in parallel. Our scalar implementation uses *mulsd* and *addsd* on 64 bit registers (xmm). Since we need all 16 registers to hold the $2 \times 8 \times 64bit$ doubles, the scalar version can not calculate multiple vectors in parallel. We also execute the same tests with versions that write the results back to memory. The memory addresses remain static in between FMAs to avoid cache misses.

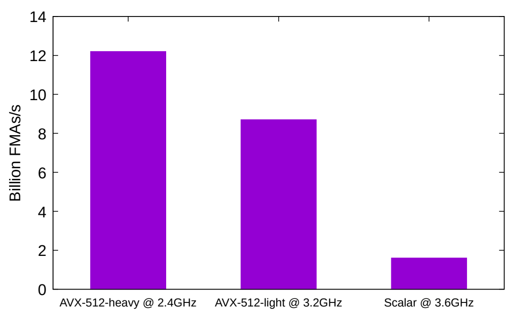
Figure 3.1 (c) shows that the scalar version can execute about 1 FMA/cycle. This accounts to one *mulsd* and one *addsd* per cycle. AVX-512-heavy and -light can execute $vector\_width * avx\_execution\_units$ FMAs/cycle which amounts to $8 * 2 = 16$ for AVX-512-heavy and $4 * 2$ for AVX-512-light. In Figure 3.1 (a) we account for the frequency drop by measuring the throughput (FMA/s) instead. This experiment shows well that certain workloads can largely benefit even after the slowdown.

3.2 AVX-512 Induced Performance Degradation

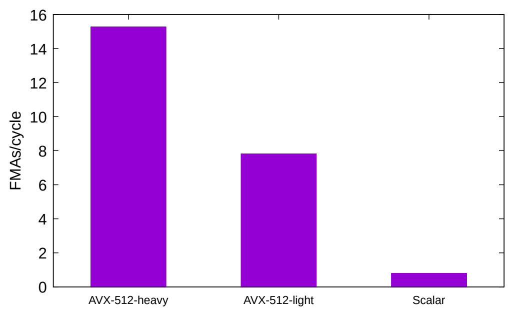
In Section 3.1 we showed that vector instructions can outperform their scalar counterparts. We mentioned in Section 2.3 that frequency reduction does not impact vectorized code too much, but can have a negative impact on subsequent scalar code. We will inspect in this subsection that slowing down scalar code can make it unfeasible to use vector instructions without modifications. We built an experiment that tries to replicate the effect shown in Figure 2.4. An example workload that generates this effect consists of a large scalar part and some vector code that is periodically executed inbetween. In such workloads most of the performance is dependent on the scalar code, but at the same time the vector code periodically slows the scalar code down. Because the vector code is so small the



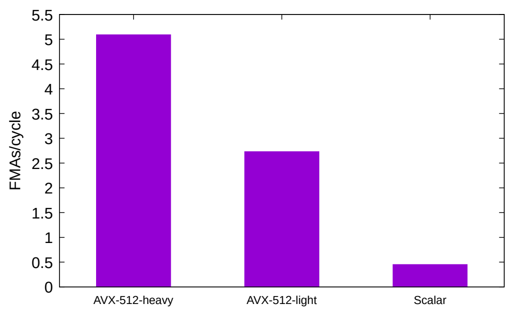
(a) FMA/s



(b) FMA/s with memory access



(c) FMA/cycle



(d) FMA/cycle with memory access

Figure 3.1: Acceleration of FMAs with Vector Instructions: One full 8-element FMA is counted as 8 FMAs.

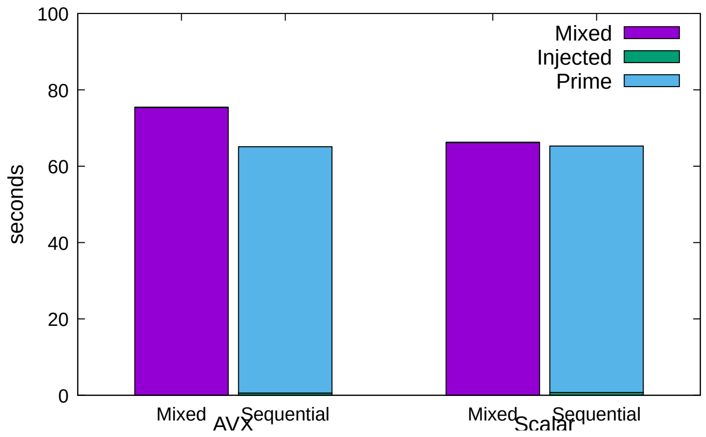
acceleration benefits can not outweigh the caused slowdown. Furthermore, we want to show that the feasibility of vector instructions becomes unpredictable by also replicating the effect in Figure 2.5 (performance increase through vectorization).

We constructed two experiments, whose results are shown in Figure 3.2. In the first experiment (a) we made the vector part small (1:100) to illustrate the peformance degradation effect like in Figure 2.4. In the second (b) we put the workloads into a 50:50 relation to show, that AVX-512 can improve the system performance like in Figure 2.5 and is thus strongly dependent on the system's workload. Each experiment is run one time with injected code (=mixed), one time without injected code (=prime) and one time without the scalar workload (=injected). To prove that vectorization is the cause of the slowdown, we also run the experiments with scalar code injected. So each of the experiments (a and b) is run once with vector code injected (left) and once with scalar code injected (right).

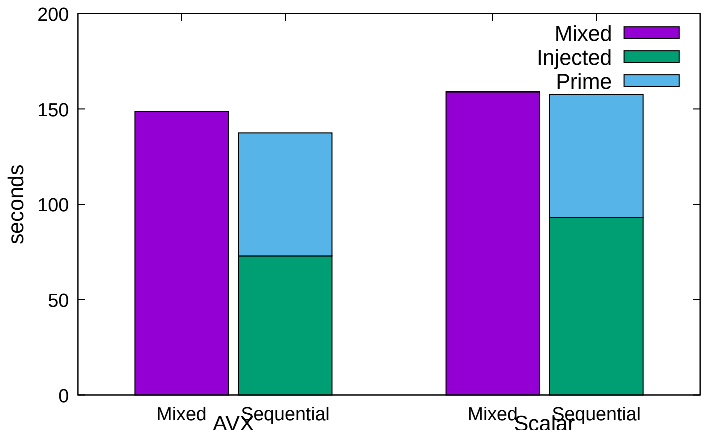
For the scalar workload we chose a scalar primesieve that calculates primes up to $repetitions * 2^{pri\_pow} * 64$. The injected vector code repeats 2^{avx_pow-1} 16 parallel 512 bit FMAs (=2^{avx_pow+6} 64 bit FMAs) on doubles. The injected scalar code does $2^{avx\_pow-4} * 16 * 8 = 2^{avx\_pow+3}$ double additions, which takes about as long as the 2^{avx_pow+6} 64 bit FMAs (see the injected part in Figure 3.2). We execute $iteration$ times the scalar workload after the injected code and measure the total execution time.

We chose for the experiments pri_pow = 9, 4 repetitions and 40000 iterations which accounts to about 64.54 s runtime. This means each iteration of the scalar workload needs about 1.61 ms at the highest frequency.

The figures (a) and (b) in Figure 3.2 show the results of the two experiments, each on the left side with injected AVX-512 and on the right side with scalar code. The prime workload always takes up the same time in all experiments when run alone, since it does not differ in the setups. The injected vector code is in both experiments a bit faster than the injected scalar code, which is realistic as we have seen in Section 3.1. (a) shows worse results when the experiment is run with injected AVX-512 code, which proves that using vector code can result in net performance loss. The (b) experiment however has better results with AVX-512, showing that the feasibility is very dependent on the mix ratio and in general on the system workload and thus very hard to predict. A dynamic policy is therefore required to achieve optimal performance.



(a) In this experiment is avx_pow = 13 ($\sim 15us$). It makes up less than 1% of the workload. The most important result is, that the mixed workload with AVX-512 is a lot slower than when it is run sequentially. This confirms the assumption that AVX-512 code slows down subsequent scalar code.



(b) In this experiment is avx_pow = 20 ($\sim 1ms$). It makes up about 50% of the workload. While the slowdown is still visible when comparing the mixed and sequential column in the AVX experiment, the performance benefit of vector instructions outweigh the slowdown. This is shown in the Mixed AVX workload taking less time than the Mixed Scalar workload.

Figure 3.2: Unmodified primesieve benchmark: Lower is better; Relation injected code to scalar primesieve: (a) 1:100, (b) 50:50

3.3 Offloading AVX-512 Code to Mitigate Degradation

In Section 2.4 core specialization is introduced as a possible solution for mitigating the performance degradation induced by HPWVIs. To demonstrate that core specialization is a feasible approach, we adapted our primesieve benchmark to offload the injected workload onto a dedicated core. We set the injected code ratio to 1:100 like in Figure 3.2 (a). We spawn a pool thread on a dedicated core and prevent any other threads from on that core. The injected code is modified to be always executed on that pool thread. This results in all AVX-512 code being executed on a separate core and leaving the scalar primesieve run at the maximum frequency. Figure 3.3 shows that moving the AVX-512 code away can achieve a performance close to the baseline. The 0.8% extra time in comparison to the baseline stems from offloading overhead, which we further inspect in Section 3.4.

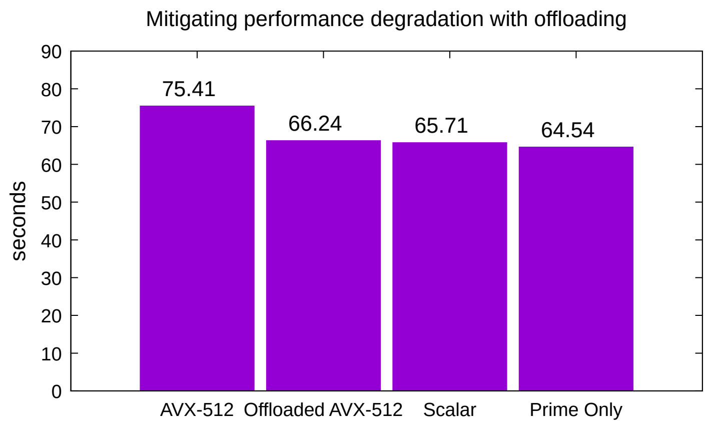


Figure 3.3: Offloading AVX-512 code onto another core nearly completely eliminates the slowdown problem.

3.4 Offloading Performance

In Section 2.5 we enumerate the different possibilities to offload work from one CPU core to another. Our framework supports offloading with Linux thread migration, spawning new threads or using threads of a thread pool combined with lock-free queues to execute the closures on. We use pthreads and migrate or pin all threads with "sched_setaffinity" out of the Linux "sched.h". We used the same primesieve benchmark to measure the difference of the offloading mechanisms. Figure 3.4 shows the results.

Our tests yieled following results: Linux thread migration is not suited for fast frequent migration, even when using the MuQSS scheduler [Kol]. The threadpool is the most efficient solution. Pool threads are only marginally better than spawning new threads because a pool thread needs to do synchronization. New threads can be spawned and then joined, while a pool thread needs an event loop that polls events out of a queue. Our queue is lock-free (Section 4.3.2) which results on nearly no additional delay on the core where the application runs on. This means in theory that during the synchronization delays other work can run on the source core.

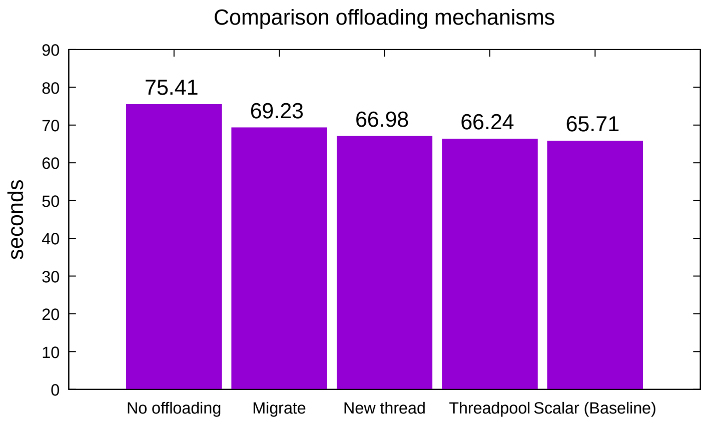


Figure 3.4: Comparison of offloading mechanisms, not offloading at all and using scalar injected code.

Chapter 4

Design

It has been shown that using high-power-requiring wide vectorized CPU instructions (HPWVIs) can result in degradation of overall system performance. While our work and approach are applicable for HPWVIs in general we worked mostly with the most prominent examples AVX2 and AVX-512, whose specifics are further explained in Section 5.1. The HPWVI-induced slowdown is caused by scalar code running at low CPU frequencies. Running a certain amount of HPWVIs requires reducing the CPU frequencies to stay within power design limits. Increasing the frequency again after returning to scalar code is delayed and thus scalar code runs for a short while at those lower frequencies. The frequency change delays can be avoided by executing the frequency-reducing HPWVI code on separate CPU cores where scalar code is only rarely executed. The goal of this work lies in building a framework that mitigates HPWVI-induced performance degradation by automatically offloading HPWVI-intensive parts of the corresponding workloads onto other cores and thus maximizing overall system performance. This is done with the support of the developer who marks code in his application that might execute HPWVIs.

4.1 Overview

How the developer marks his code is explained in Section 4.3.1. Not all marked code is worth being offloaded, thus we constantly need to decide whether to offload a given part of the workload or not. A detailed description of how we do the offloading decision is in Section 4.2.2. To achieve the best possible global system performance we need to have a global view and make global decisions. In general there are multiple applications active on the system that might use our framework. The applications need to cooperate with each other to attain a global system view. We do this by managing all decisions globally in a central policy server that is further described in Section 4.3.3. To enable applications to communicate with and benefit from our framework we need a bridge between the central policy server and the applications. For that purpose we constructed a library that makes up the other part of the framework and is detailed in Section 4.3.2. The library is linked against the target applications and the instances communicate with the policy server to share data and communicate decisions.

The policy in the policy server mainly decides on which CPU core to execute a marked code segment. This can be the CPU core the scheduler originally determined or another CPU core. Offloading HPWVI code onto cores that execute scalar code might result in recreating the same slowdown problem as on the original core and makes measuring slowdown very hard. We circumvent this problem by dedicating a dynamic set of CPU cores to run only offloaded work. Dedicating cores naturally results in lower system utilization. We minimize the system utilization degradation by saturating one core after another and thus keeping the amount of not fully utilized dedicated cores low. Our dedicated core approach is described in Section 4.2.1.

Determining which regions are worth offloading means we need to find out what parts are causing a slowdown. Since we focus mainly on HPWVI-caused slowdown, we use a heuristic in the policy that estimates the slowdown by measuring HPWVI usage. Adequate parameters for the heuristic differ a lot for different system loads. In order to find good parameters we constantly adapt them during runtime and observe the overall system performance (see Section 4.2.4) in a feedback loop (see Section 4.2.3).

4.2 Policy

The goal of the framework is to maximize overall system performance by counteracting the performance degradation caused by using HPWVIs (with a focus on AVX-512). Maximizing overall performance means that it does not matter if single applications get slowed down as long as the overall system performance improves. Our approach to minimize performance degradation is to offload HPWVI-intensive work onto other cores. There are three core decisions that have to be made by the policy.

1. The policy has to decide whether to offload the part of the workload given to the framework onto another core.

2. The policy has to decide how many cores to consider for offloading work onto.

3. If the policy decides to offload, it has to decide which core to offload onto.

4.2.1 Dedicated Cores

When offloading code onto other cores that are running scalar code, the slowdown problem is recreated on the cores we offload the work to. Also, choosing cores for offloading becomes very hard because the policy would have to account for the effect on those cores. We avoid both those problems by introducing the notion of HPWVI cores. HPWVI cores are CPU cores which the framework considers offloading work onto. We forbid the scheduler to run any other work which is not explicitly demanded by the framework on these cores. Hence we establish dedicated CPU cores that serve the framework for running (primarily) HPWVI parts of the workloads that trigger slowdown. This approach stems from heterogeneous computing and implements the principle of core specialization. Through core specialization decision 3 becomes effectively synonymous with “How many CPU cores are dedicated HPWVI cores?”. Dedicating cores that are not fully saturated reduces the maximum achievable system utilization. Hence, we need to keep the amount of dedicated cores as low as possible and maximize their individual saturation. This requirement directly results in very simple policies for decisions 2 and 3. We minimize the amount of HPWVI cores by fully saturating them and only after that dedicating more HPWVI cores. By fully saturating all cores before spawning a new one we can achieve a maximum utilization of $\frac{core\_count-1}{core\_count}$ in the worst case. The current trend of increasing amounts of cores in CPUs suggests that this effect will be negligible in the future. For example, the coming AMD Rome CPUs will have up to 64 cores [wikb].

Dedicating cores not only simplifies the last two decisions and avoids creating unavoidable slowdown effects on cores that run HPWVI, it also results in more homogeneous workloads (in terms of vectorization) on a single core. Having homogeneous workloads on cores plays a major role in the detection of HPWVI heavy parts of workloads (see Section 5.2).

4.2.2 Offloading Decision

The dedicated HPWVI cores are our target for offloading work onto. We have to decide what exactly we want to offload onto those cores. We only want to offload something if offloading increases the overall system performance. The framework focuses on performance degradation caused by HPWVI usage and gets informed of candidate code regions by the application developer, only such code regions are elligible to offloading.

Some marked code regions might not execute any HPWVIs at all. This might be due to developer errors or compilers not vectorizing certain code regions. Scalar code will not slow down other scalar code by frequency reductions, so offloading it results generally in no performance gains and in practice in performance losses. The performance losses are created by offloading overhead and the offloaded scalar code being slowed down. There are also parts of workloads that use HPWVIs and cause frequency reductions but are still not worth offloading, for example because of oversaturated HPWVI cores. Thus, offloading all marked regions is in general not a very good policy.

The main part of the policy (decision 1) is deciding whether marked code is worth being offloaded. We achieve better system performance by avoiding scalar code to be executed at lower frequencies. This means we are looking for code regions that execute enough HPVWIs to trigger a slowdown and are followed by scalar code that gets slowed down. HPWVIs trigger slowdown when they exceed a certain instruction mix density. Thus, we estimate the amount of slowdown by measuring how many HPWVIs were executed on average in a time window. We use a heuristic that assumes that offloading becomes feasible as soon as the rate of HPWVIs exceeds a certain threshold. This threshold is not the same for every type of system load. To perform well for all types of loads we need to adjust the threshold dynamically during runtime. Since our goal is to maximize overall system performance, we observe the correlation between the threshold and system performance during runtime. That correlation is used to adjust the threshold in a way that the system achieves maximum performance. This process of measuring performance and adjusting the threshold creates a feedback loop.

4.2.3 Feedback Loop

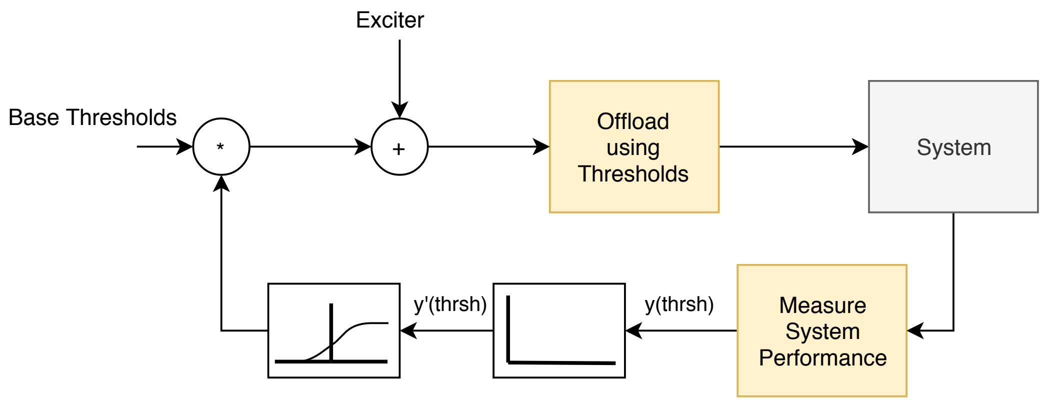


Figure 4.1: A feedback loop is used to find a correlation between thresholds of the heuristic, which decides whether a part of a workload is offloaded, and overall system performance. y is a function mapping thresholds to performance. The exciter keeps the feedback loop moving by preventing static thesholds.

Depending on the current system workload offloading the same part might be sometimes beneficial and sometimes detrimental to the system performance. Our framework needs to be able to dynamically decide whether it should offload or not. We make our policy flexible and dynamic by introducing a feedback loop in our offloading decision that aids in maximizing system performance.

In the offloading decision, the heuristic for determining whether a part of a workload is worth offloading uses a threshold on a HPWVI density metric. The threshold gives us control over the amount of offloading and a selection of which types of routines to offload. The goal of the feedback loop is to find adequate values for the threshold parameter of our heuristic during runtime.

The feedback loop depicted in Figure 4.1 is split into three logical parts: Using a threshold to control offloading is explained in Section 4.2.2. Measuring system performance is explained in Section 4.2.4. In the following paragraphs we focus on the threshold adjustment part of the loop.

We get a performance metric from the "measure system performance" block and need to supply a threshold to the "offload using thresholds" block. Since we know the threshold that was used, we can correlate the threshold to the metric creating the function $y : threshold \rightarrow performance$. This function is only valid as long as the system load does not change dramatically, thus only within a certain time window ($y_t : threshold \times timewindow \rightarrow performance$). This means we have to regularly reset our function y. We derive $\frac{dy}{dthreshold} =: y' : threshold \rightarrow performance\_trend$. Using our current threshold thr_{cur} in y' yields the current performance trend $y'(thr_{cur}) =: trend_{cur}$. This value $trend_{cur}$ is then put into the sigmoid function to get $thresh\_factor \in [0, bound], bound \in [1, \infty)$. Depending on the performance trend we get different cases:

- $trend_{cur} \rightarrow -\infty \implies thresh\_factor \rightarrow 0$
  Performance improves with lowering the threshold.
- $trend_{cur} \rightarrow \infty \implies thresh\_factor \rightarrow bound$
  Performance improves with raising the threshold.
- $trend_{cur} \in [-\epsilon, \epsilon] \implies thresh\_factor \rightarrow\sim \frac{1}{2}bound$
  Performance is close to a local minimum.

We use $thresh\_factor$ to direct our threshold by multiplying it to the base threshold. If the threshold to performance correlation function y is accurate, then performance should increase in the next step. This approach is similar to a gradient descent. Gradient descent is a first-order iterative optimization algorithm that finds local minima of a function [Wik19]. This means that the global minimum is potentially not reached when using an unmodified gradient descent. Even worse, the factor will converge to a certain number. Having a meaningful y' requires an y that is valid for big parts of the input domain (many recent different

$thresholds \rightarrow performance$ statistics). When the factor converges because of gradient descent, large parts of y become invalid with time. To circumvent this problem we introduce an exciter. The exciter adds a distortion value to the threshold before it is passed to the offloading heuristic. Hence we keep the threshold value moving and avoid the problems that are caused by gradient descent.

Implementing the feedback loop exactly the way it is designed introduces some complications such as the need for an explicit update of the correlation function y. Instead, the feedback loop is implemented by having a direction $growth\_direction$ in which we grow the thresholds and a rate $growth\_rate$ by how much it is grown as additional context. By combining those with a timewindow over the performance metric and observing the performance trend, we achieve a similar effect, but have the advantage of having an implicitly up-to-date function. We use the previously defined function $y_t : threshold \rightarrow performance$ and derive it over time instead of threshold $\frac{dy_t}{dt} =: trend'_{cur}$ to get a performance trend over time in our window. We still need a correlation between threshold and performance trend. Thus, the windows are cut as soon as the threshold growth direction changes, making the threshold growth direction static within a window. With this assertion $trend'_{cur}$ correlates to $trend_{cur}$, and we get following cases:

- $trend'_{cur} < 0 \implies growth\_direction := -growth\_direction$ $\land\ growth\_rate := growth\_rate * \frac{1}{factor}, factor > 1$
- $trend'_{cur} > 0 \implies growth\_rate := growth\_rate * factor, factor > 1$

The exciter is implemented by timing how long the amount of offloaded clients has not changed and increasing the growth rate if the timer exceeds a certain threshold.

Implementing a feedbackloop on such a system is linked to some key challenges. The system contains a lot of delay and we can only use sampling to model the system.

4.2.4 Measuring System Performance

Implementing a feedback loop requires us to measure the overall system performance that we want to maximize. Before we can measure the system performance we need to define the scope of the system and construct a metric. Regarding applications that are not part of our framework or not on the same machine would complicate meaningful measurement. Thus, we define the system as the collection of all applications that use our framework on the same machine.

Measuring the performance of the applications can be done with two methods: Either heuristics have to estimate the performance implicitly or the application itself has to explicitly and regularly provide a performance metric. The explicit metric needs to be normalized to work seamlessly with other applications. The implicit metric is calculated for every application with the same method and thus does not have to be normalized. We decided to only use implicit metrics in our prototype to require minimal information from application developers. Explicit metrics are promising candidates for better performance heuristics and should be further investigated (see Section 7.1).

Implicit performance detection is difficult. We implemented and evaluated (Section 6) two approaches. Our first approach measures only marked code regions. We can assume some marked code regions executing mainly scalar code. If this was not the case the thresholds would converge against 0 because of the feedback loop. The performance of HPWVIs is assumed to be invariant under offloading. Hence we can resort to only measure performance of marked code, since it correlates to overall scalar code performance in the application. We use average time of execution of the marked code regions as a heuristic for the implicit metric. Our other approach tries to estimate the performance of the whole application by measuring the call frequency. For applications that have only a small percentage of code marked this heuristic better matches the application performance.

4.3 Framework

Transparent detection of problematic HPWVI executing code regions is difficult. We went with a slightly intransparent approach which cooperates with application developers to mitigate the performance degradation caused by HPWVIs. The application developer marks certain code regions that might execute HPWVIs. Our framework then only has to decide for those code regions as a whole what to do. The framework should be as transparent as possible and provide an easy-to-use interface. Sections might be wrongly marked, this means marked sections are only candidates for actions.

Which candidates get offloaded has to be decided by a central policy that has a global system view. The framework is split into a policy server that contains the policy and a library that applications interact with directly. The policy server manages all library instances on the same host. It further contains the complete offloading policy and makes decisions for all clients. The library instances communicate with the policy server for exchange of monitoring data and decisions. Every instance contains a decision cache that gets regularly updated by the server. This cache is used for determining whether to offload at all and where to offload to. During offloading or direct execution of the given code, the library collects data about the code execution. The server contains a statistics storage for each client where the monitoring data is stored after the corresponding library instance sends them to the server.

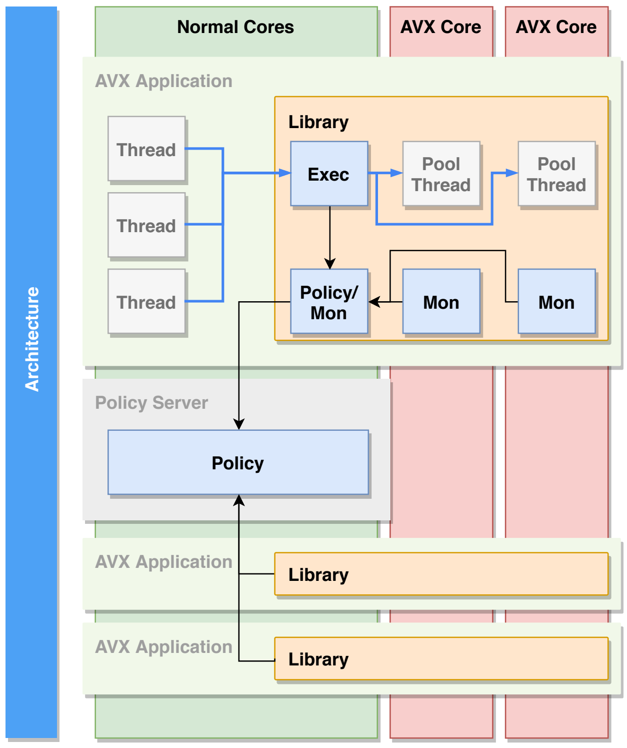


Figure 4.2: The framework architecture defines two components: The policy server contains the policy and coordinates all library instances. The library is the interface between applications and our framework. Black arrows symbolize communication and blue arrows the passing of parts of workloads to execute. Everything runs on normal CPU cores, just the pool threads that execute the closures run on the dedicated HPWVI cores.

4.3.1 Marking Code

Marked code has to be able to be moved onto other cores. The library moves marked sections as a whole. Therefore, it would be optimal if sections each have a common entry and exit point. Moving whole sections facilitates moving code between cores. The most intuitive candidates for this type of code sections are closures. Application developers have to convert code sections that need to be marked into closures. Marking closures is best done by passing them to the framework to be executed. Hence our framework can easily analyze them and move them around. The process of passing our framework a closure for analysis and execution must be as performant as possible to avoid slowdown. Minimal overhead is achievable by separating our framework into a part that contains the global policy and a part that sits close to the application for maximum performance. We created a dynamically linked shared library that applications link against and then

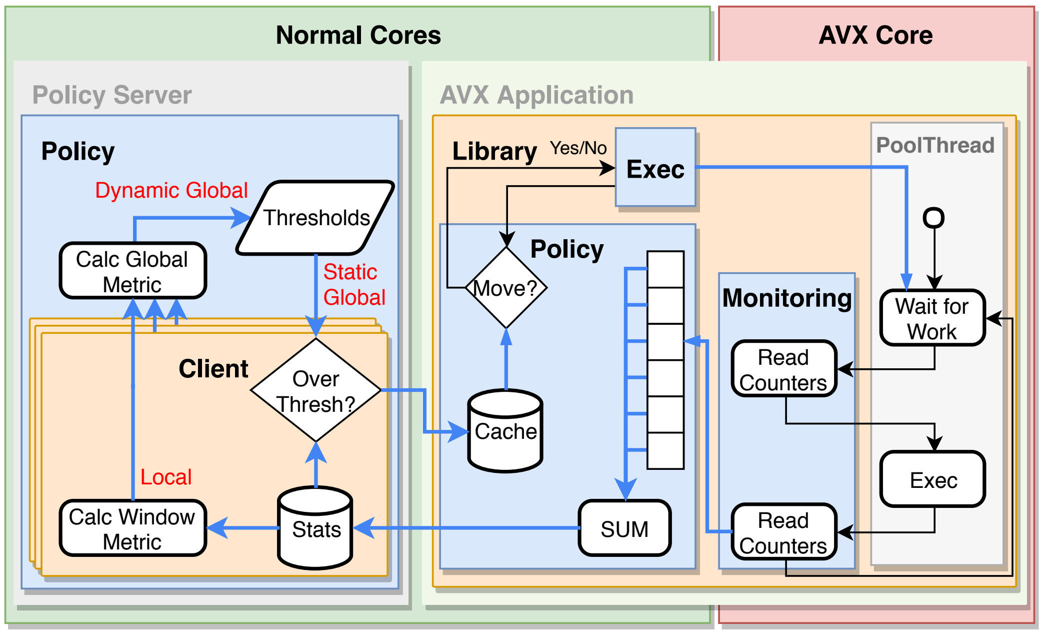


Figure 4.3: Overview of the implementation of the framework. Blue arrows symbolize data flow and black ones control flow. The policy server and library are split. The library executes closures on pool threads that run on HPWVI cores. The library uses performance counters to gather statistics and sends the compactified results to the policy server. The policy server runs the policy for all clients and regularly communicates the decision back to the library instances. Those store the decision until the next update.

can pass marked code in form of closures to. With a linked library our code is executed in the same address space as the code we want to potentially offload. This approach not only results in good performance but also makes the mechanism of passing functions and converting code into closures easy. The following listing demonstrates in pseudo code how marking a closure is done:

```
//closure to be marked
Calculate(param1, param2) {
    //do calculations, potentially using HPWVIs
}

//without library
Calculate(param1, param2)

//passing to library
AVXPERF_EXEC(Calculate, {param1, param2})

//optional: passing explicit performance metric
```

```
AVXPERF_METRIC(performanceOfLastCall)
```

The function “Calculate” is the closure we want to mark. Normally it would just be executed, but to mark it, it is passed to our library for execution. If explicit performance metrics would be implemented, this is how they could be passed to the framework (more details in Section 4.2.4). The rest happens within our framework. For a concrete language specific code marking implementation see Section 5.3.

4.3.2 Library

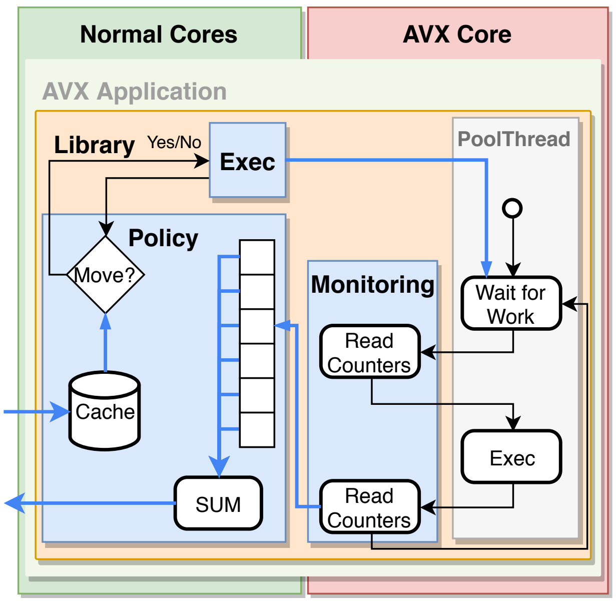


Figure 4.4: The library consists of following components: execution, monitoring and policy. The monitoring component reads performance counters and gives the statistics to the policy where the statistics are compactified into a sum over a time window before they are sent out. The policy component also contains the decision cache that is filled by the server. This cache is used by the execution component to decide onto which pool thread to offload.

The dynamically linked library is the frontend of the framework. Applications that shall make use of our framework need to link against this library.

Frontend To let the library consider a part of an application for offloading, the code region has to be put into a closure and passed to the function *AVX-PERF_EXEC*. The library then does the rest of the work without further interacting with the application. A possible exception is passing a metric to the library in addition to the closure to yield better results in terms of performance. How code marking works is explained in Section 4.3.1 and Section 5.3.

Offloading Mechanism Internally the library first has to decide for a given closure where to execute it on. The decision whether a library instance shall offload is done by the policy server and stored locally in a decision cache. The library itself only has to execute the closure on the correct core. We mentioned different offloading mechanisms in Section 2.5. Pool threads cause the lowest offloading overhead as measured in Section 3.4. Since we have dedicated cores for executing parts of workloads we can prepare pool threads on those cores. A pool thread for each core is spawned as soon as it transitions into a dedicated HPWVI core. The pool thread is part of the library and thus also part of the application address space. Offloading a closure is thus as simple as passing it to the pool thread via the implicit shared memory for execution. The pool thread contains a lock-free single consumer queue into which closures are put. The queue resides in the implicitly shared memory between threads.

Since each application has their own pool threads the pool threads compete amongst each other for time on the HPWVI cores and thus should not use spinning for synchronization. Instead, a futex is used to wake a pool thread as soon as work exists in its queue.

Monitoring The library is responsible for executing marked code, thus it is also responsible for monitoring. Performance counters are read before and after executing the closure. The amount of active performance counters is limited and the selection of them is thus part of the policy. Further the choice of performance counters needs to be done globally. For those two reasons, the performance counter choice is done in the policy server. Besides performance counters, time is also measured. Only the events caused by the closure are interesting thus for each counter the difference between the values before and after execution has to be calculated.

Communicating too often to the policy server is detrimental to performance (Section 6.2.3). Thus the values are summed up into running sums and sent regularly to the policy server. Those values are used for determining performance (Section 4.2.4) and detecting HPWVIs (Section 5.2).

Server Communication The library communicates statistics after every decision cache update period to the server and requests a decision cache update. Each library instance owns a local decision cache that contains information about whether and where to offload and which HPWVI cores are active. The decision cache is updated every x executions of closures. Each library instance is the client in a client-server-relationship with the policy server. The communication between the clients and the server has to be reasonably fast, but does not need to be as fast as the queue in the pool threads. We use explicitly shared memory wherein a locking multi producer queue resides for clients contacting the server, each client has its own single producer queue for responses.

4.3.3 Policy Server

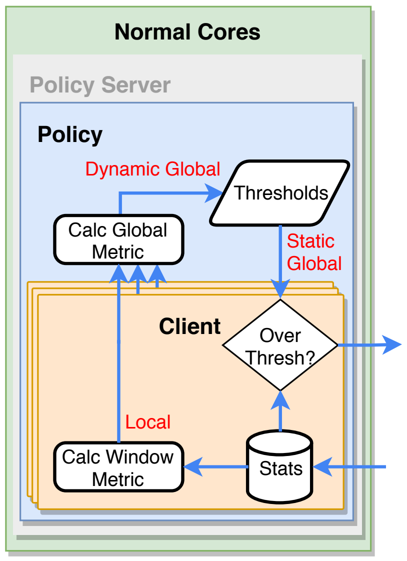


Figure 4.5: The policy server has a statistics storage for each client that is used to calculate the local window metric. This local window metric is then used to calculate the global metric which then is compared against the other global metrics to determine adequate thresholds. Those thresholds get used to determine for each client whether it should offload.

We implemented a central policy server which contains the whole offloading decision policy. The idea behind a central server is motivated by the need to have a complete system view. Doing that with a central server is in this case simpler and more efficient than letting the clients interact directly with each other.

Feedback Loop The policy server implements the feedback loop depicted in Figure 4.1 and described in Section 4.2.3. From the clients, the server gets a sum for each configured performance counter and a time counter over a certain time window. Those values are used in the "measure system performance" block shown in Figure 4.1 and also for calculating values that are compared against the thresholds (Section 5.2).

We need to process the raw per-client values spread over time into a single value $trend'_{cur}$. The value $trend'_{cur}$ is just the derivative of our global performance metric over time. So we first need to determine a per-client performance metric as described in Section 4.2.4, which is done in “calc window metric” (in Figure 4.5). This yields $y_t^c := threshold \times timewindow \times client \rightarrow performance$ which is a collection of y_t, one for each client. We determine y_t from y_t^c by setting $y_t := \sum_{c \in Clients} y_t^c$. With y_t we can get $trend'_{cur}$ and thus determine the values for $growth\_direction$ and $growth\_rate$. Note that, as described in Section 4.2.3, if we notice performance growth over the last windows, we keep the direction and increase the growthrate. If the performance gets worse, we invert the direction and decrease the growth rate. Determining the growth rate and direction is done in “calc global metric” (Figure 4.5). To obtain the threshold for offloading we use the growth rate, growth direction and potentially apply the exciter. Finally, the threshold is stored and used until recalculation.

As soon as a client then requests a decision update, the threshold is applied to the client’s heuristic and the resulting decision is sent to the client to be written into its local decision cache. This decision whether a client should offload or not is static for a whole decision cache update period. The update period can be configured. We found in Section 6.2.3 that updating the decision cache every 100 executed closures provides enough resolution while not causing any significant overhead.

The exciter becomes active when the amount of offloaded clients has not changed within a certain period. Determining the amount of offloaded clients is trivial in the policy server.

HPWVI cores The amount of active HPWVI cores is also determined by the policy server. The policy server is responsible for setting the cores up. This includes enabling monitoring, migrating all tasks away and preventing the scheduler from putting there any tasks by using `cpusets`. A list of active HPWVI cores is put into the decision cache of the clients, so they know where to spawn the pool threads on.

Having too many HPWVI cores reduces the system utilization and thus the overall system performance. To profit from an additional core being put into HPWVI mode, all other HPWVI cores should already be utilized to a high degree. If the average HPWVI core utilization falls under a certain threshold, HPWVI cores are removed. However, if the average HPWVI core utilization reaches a certain threshold another HPWVI core can be added. To avoid thrashing those two thresholds should be different. Due to time constraints our prototype does not automatically add more than one HPWVI core.

Chapter 5

Implementation

Executing *high power wide vectorized instructions* (HPWVIs) on the same cores as scalar parts of a workload results in unwanted slowdown of the scalar code. HPWVIs can not run at the same CPU frequencies as scalar code because it would exceed the designed power limitations. Increasing the CPU frequency after executing HPWVIs takes time and thus subsequent scalar code runs at lower frequencies than normally possible. We designed a framework that cooperates with HPWVI application developers to mitigate this effect by offloading HPWVI parts of workloads onto cores dedicated for this type of workload.

Our framework focuses on current generation Intel processors (Intel Skylake-SP) running GNU/Linux and is written in C. The main concepts of this work should apply to any HPWVIs on any platform. In this chapter we break down specifics of our approach for the platform we focused on.

Our policy relies heavily on the detection of HPWVIs in between scalar code. One of the main problems of this platform regarding our solution is the lack of instrumentation to count executed HPWVIs. This means we need to rely on heuristics and some form of feedback mechanism to detect HPWVIs. Our framework cooperates with the application developers to find candidates and thus reduce the searching scope drastically.

5.1 HPVWIs in Current Generation Intel Processors

Intel Skylake-SP processors support several instruction set extensions that support vectorized execution. Only AVX2 and AVX-512 are HPWVIs. Each core has 3 power levels with different frequency bands. Level 0 corresponds to the normal state, while level 1 and 2 are frequency-reduced states. If the instruction mix within a small window has too many level-1-triggering instructions, the core will switch into level 1 power mode. The same applies for level 2 instructions and the level 2 power mode. Level 1 instructions are AVX2 instructions that use FP or INT MUL/FMA and AVX-512 instructions that do not use INT MUL/FMA or FP. Level 2 instructions are AVX-512 FP and AVX-512 instructions that use INT MUL/FMA. [int18a]

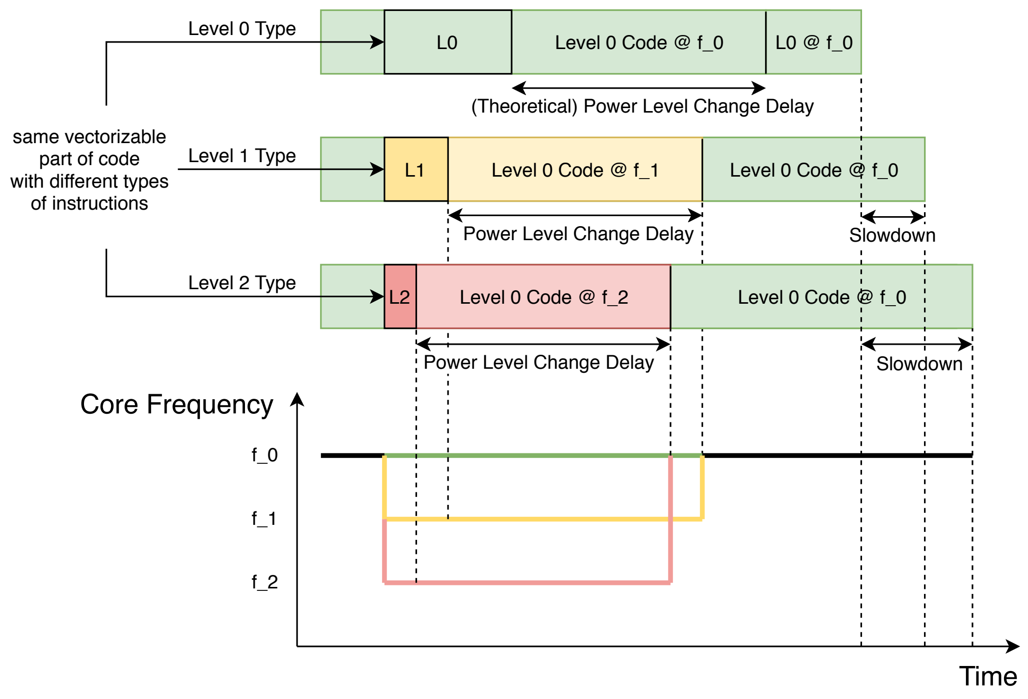


Figure 5.1: This example illustrates how HPWVI caused slowdown is created in systems with multiple power levels. Each bar represents the execution of the same workload, with different types of power level triggering code.

Power Levels The power levels directly correlate with the maximum frequency the core can run at. Switching from the power level back to level 0 will result in the most slowdown for the level 0 code. Thus, the highest power down level code is in general the best candidate to offload in a multi power level architecture. As part of our offloading mechanism we use dedicated HPWVI cores that execute offloaded code. The HPWVI cores will generally run predominantly in the highest power down level while the rest of our cores will run preferably at lower levels.

This splits our cpu cores into two sets and raises the question of where code that triggers intermediate levels should run on. Running level 1 code on a level 2 dominated core slows down the level 1 code, while running the level 1 code on our level 0 core slows down the rest of our workload. This is illustrated in

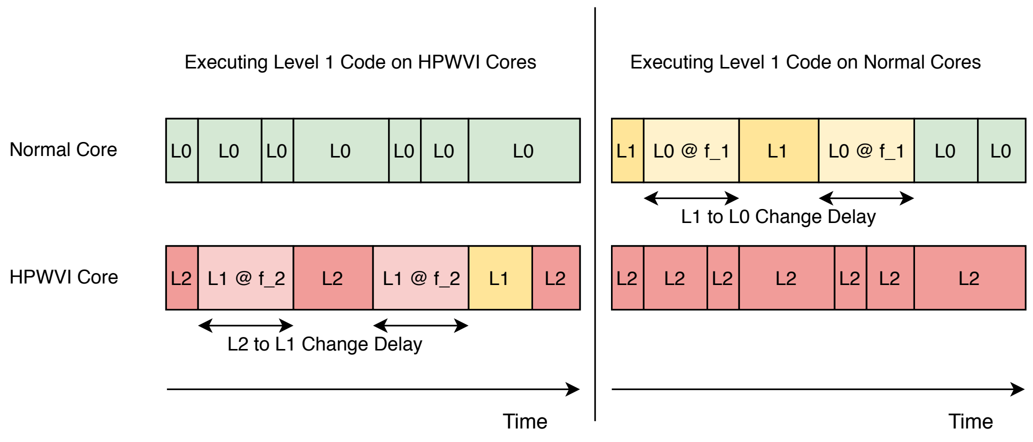

Executing Level 1 Code on HPWVI Cores
Executing Level 1 Code on Normal Cores
Normal Core
HPWVI Core
L0 L0 L0 L0 L0 L0 L0
L2 L1 @ f_2 L2 L1 @ f_2 L1 L2
L2 to L1 Change Delay
L1 L0 @ f_1 L1 L0 @ f_1 L0 L0
L1 to L0 Change Delay
L2 L2 L2 L2 L2 L2 L2
Time
Time

Figure 5.2: Intermediate-level-code slows down lower-level-code (right side) but gets slowed down by higher-level-code (left side).

Figure 5.2. Having two or more different power down levels means we have to differentiate those within our framework. Introducing separate level 1 dedicated cores would reduce maximum system utilization further from $\frac{core\_count-1}{core\_count}$ to $\frac{core\_count-2}{core\_count}$ in the worst case. In general multi power level architectures this would be $\frac{core\_count-power\_level\_count+1}{core\_count}$. Instead of using multiple types of dedicated cores, we chose another approach. Our hybrid approach uses a threshold for each level of instructions and is illustrated in Figure 5.3. Without further adjustment our feedback loop automatically determines whether level 1 code should be offloaded or not by adjusting the level 1 threshold. Hence the framework decides whether it is better for the overall system performance to slow down the level 1 code to level 2 or to slow down some level 0 code to level 1. This approach also works for more than 3 power levels and does not impose scaling issues such as multiple types of dedicated cores.

Measure Different Types of AVX Instructions The feedback loop requires knowledge about which power level triggering instructions are executed. This information is used to compare against the thresholds per power level. Following performance counters exist on our platform that measure HPWVI-related instructions [int18b]:

- Retired Scalar/Packed SIMD instructions counter
- Retired 512B FP AVX-512 instructions counter
- Cycle counter for each power level

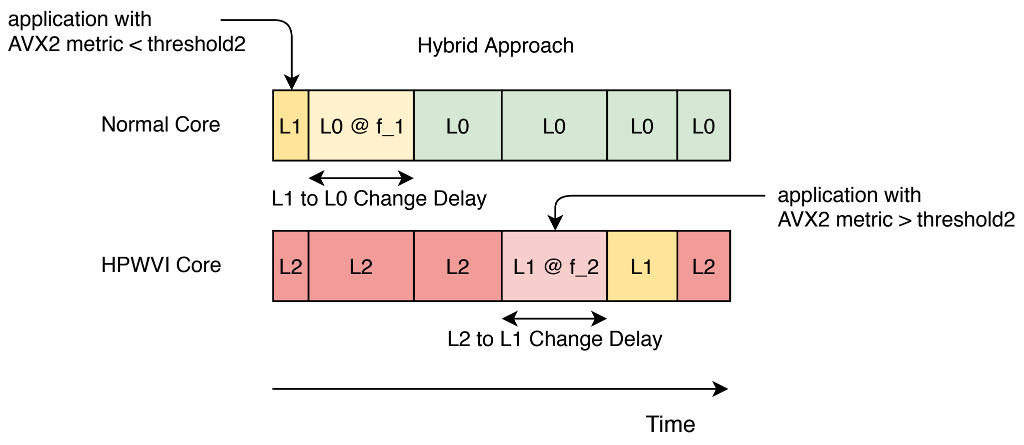


Figure 5.3: The hybrid CPU choosing approach for intermediate-level-code decides for each client separately whether to offload or not, just like with the highest level.

Apart from SIMD instruction sets which reduce frequency such as AVX-512 and AVX2 there are also SSE and AVX which do not trigger higher power down levels. Counting SSE and AVX instructions separately and then subtracting them is not possible. Measuring AVX-512 FP instructions is also not enough for determining all level 2 instructions. It misses for example AVX-512 INT MUL and FMA. The only metric that can be used in our framework are cycles spent in the corresponding power levels. Those cycle counters do not provide a precise measurement of how many instructions were executed that triggered certain power levels, but rather how many cycles were spent in the respective power level. Those two metrics differ because the power level change delay makes lower power down level code run at higher levels for some time. How to use this metric to heuristically detect level 1 and level 2 instructions is further detailed in Section 5.2. The existence of intermediate levels does not further complicate our design and implementation.

Power Level Change Delays Whether offloading can improve performance is strongly dependent on how long the acquisition of a higher frequency power level takes. Offloading onto another core is effectively dodging this delay time and paying instead with offloading overhead. Going into a lower power down level requires up to 500us for power license acquisition and waiting at least another 2ms till a conditional timer ends. The 2ms timer is reset every time a condition is fulfilled that would have requested a new license. Effectively this means that level 0 code can be executed for about 2,5ms at a frequency lower than necessary. This makes the range in which offloading helps quite broad, making the approach of our framework feasible.

5.2 Measuring HPWVI Instruction Density

The ratio of HPWVI to Non-HPWVIs within a small window determines whether a switch of power down level occurs [int18a]. Our design (more specific the heuristic in the policy of the feedback loop) requires a metric of how dense the HPWVIs are in the instruction mix to predict slowdown.

On our platform we have no direct access to the HPWVI density. We can only use a heuristic to guess it by using the power level cycle counters mentioned in Section 5.1. As a sufficiently HWPVI-dense code triggers a switch to a higher level, the cycle counters correlate strongly to the HPWVI density being over the threshold. The higher those counters are averaged over time, the longer was the instruction mix dominated by the corresponding instructions. The main problem with the cycle counters is that due to power level change delays, code that comes after a certain instruction mix also counts into those counters. Since we split the regarded code into multiple independent candidates, this results in a lot of false positives.

Aggressive offloading We should avoid having too many of those false positives running on the dedicated cores. The original problem why we have false positives is that Non-HPWVI-dense code (NHDC) following HPWVI-dense code (HDC) will spend some cycles in a higher power down level caused by the HDC. This results in NHDC following HDC to have on average a higher metric than other NHDC. In general however, HDC will have an even higher metric than NHDC independently of whether it follows HDC or NHDC. This means there is a threshold which separates NHDC and HDC. The task of the feedback loop is to find this threshold. The dedicated cores are useful for supporting the feedback loop in this task. Offloading NHDC onto a HPWVI core will slow the NHDC down, which is illustrated in Figure 5.4. This will result in a lower global performance, making the feedback loop counteract this decision by increasing the threshold. This results in the NHDC not being offloaded anymore. (Side note: In the case of very lowly saturated HPWVI cores, offloading NHDC might increase global performance by increasing system utilization, but in this case it does not matter that we wrongly offloaded something).

5.3 Marking Code in C Applications

In Section 4.3.1 we outlined the code marking process for general languages. To let the library consider a workload for offloading, the workload has to be put into a closure and passed to the function *AVXPERF_EXEC*, which then potentially offloads the closure onto a dedicated HPWVI core. In our prototype the library is

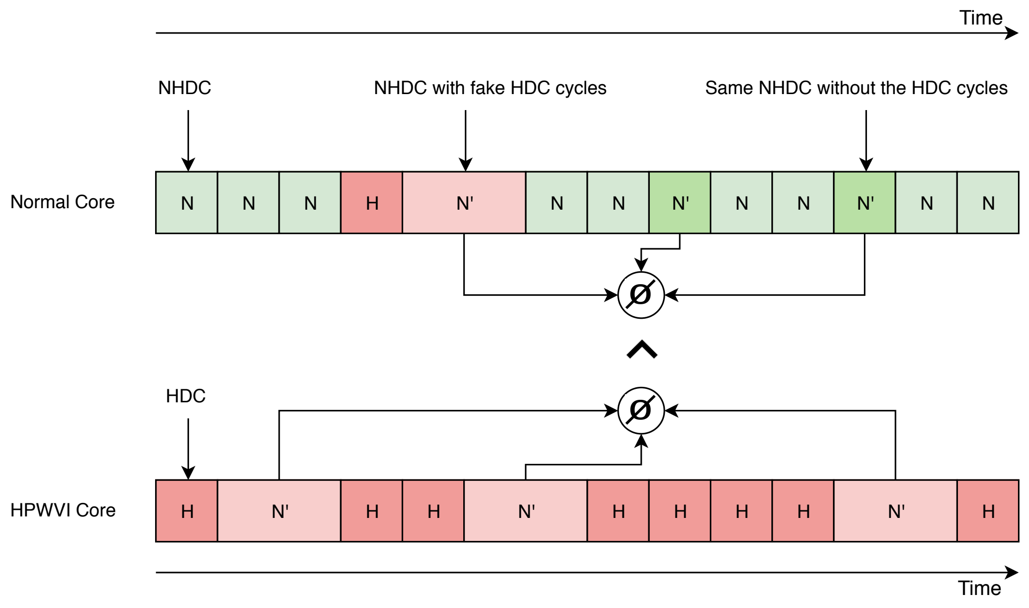


Figure 5.4: When NHDC code is wrongly offloaded, the performance decreases on average. This effect makes it feasible to offload aggressively and then correct detected mistakes.

written in C which does not contain explicit closures. We implemented a closure in the form of a struct with a pointer pointing to the function to execute and an argument struct. An argument struct contains all the arguments and has to be created for each unique function signature. If adapting the target function is not possible or desired, it is possible to pass an unwrapper function to the library. This unwrapper needs to get the arguments from the argument struct and call the target function with them. Listing 5.5 shows a code example that illustrates how unwrapper and wrapper can be implemented and how a closure is passed to the library without modifying the target function. Wrapper and unwrapper code can be theoretically auto-generated using preprocessor macros.

```c
//closure to be marked
int calculate(int param1, char param2) {
    //do calculations, potentially using HPWVIs
}

//without library -----------------------------
result = calculate(param1, param2)

//with library --------------------------------
struct calculateArg {
    int param1;
    char param2;
    int *returnVal;
};

void calculateUnwrapper(void *arg) {
    struct calculateArg *unpacked = arg;
    *unpacked.returnVal = calculate(
        unpacked.param1, unpacked.param2);
}

int calculateWrapper(int param1, char param2) {
    int returnVal;
    //passing to library
    AVXPERF_EXEC(calculateUnwrapper,
        {param1, pararm2, &returnVal});
    return returnVal;
}

result = calculateWrapper(param1, param2);
```

Figure 5.5: Code example of how to pass a closure to our library

Chapter 6

Evaluation

Using vector instructions with high power requirements results in temporary CPU core frequency reductions. Increasing the frequency is inflicted with delays and this subsequent scalar code often runs unnecessarily at low frequencies. A lot of workloads benefit from vectorization disregarding the frequency reduction, but when combined with other scalar workloads, the overall system performance might be worse than without vectorization. We constructed a framework that mitigates the performance degradation caused by high power vector instructions (HP-WVIs), without prior knowledge of the application. Application developers use our framework by marking code in their applications that potentially executes HP-WVIs. We designed a policy which decides for the marked code regions whether they should be offloaded onto a small set of dedicated cores to isolate frequency reducing code from scalar code. In this chapter we are gonna evaluate our approach and the policy we designed. We focus on testing our policy on realistic workloads and finding reasonable parameters for our policy.

6.1 Setup

All benchmarks are executed on an Intel Core i9-7940X, which underlies the Intel Skylake-SP architecture, has two AVX-512 units and 14 physical cores with 2 threads per core. We disabled C-states and configured the cores to run always at the maximum possible frequency to reduce the variability of results. The maximum frequencies differ for each power level as shown in Table 6.1.

The CPU frequency governor of all cores is set to "performance" mode. We use the Linux-CK kernel in version 4.18.17 to make use of the MuQSS scheduler [Kol]. MuQSS is set to maximize throughput instead of latency (/proc/sys/kernel/interactive = 0). We use cpuset [cpu] which is a wrapper around linux cpusets, to isolate our benchmarks on cores 4-13 and their hyperthreads. Hence

Power Level	Frequency
0 (Scalar)	3.8 GHz
1 (AVX2)	3.3 GHz
2 (AVX-512)	2.5 GHz

Figure 6.1: Intel i9-7940X frequencies

we avoid any external disturbances in form of power level triggers from other applications running on the system. Additionally, all applications that are not linked against our framework are compiled without AVX2 and AVX-512 support.

For the benchmarks in Section 6.2 we make use of the avocado test framework [avo]. Avocado allows us to easily and automatically reconfigure our benchmark and framework to test with multiple parameters that are centralized in a single configuration file.

6.2 Framework Performance

In this section we will show that our framework provides value by mitigating the performance degradation caused by AVX-512. We want to show the following for realistic workloads:

- Using HPWVIs can have a net performance penalty.
- Our framework can mitigate the performance penalty.
- The heuristical performance metric we use is able to accurately estimate performance.
- Our policy detects for which workloads it is worth offloading and is thus better than a policy that always offloads.
- Our framework does not slow down applications that do not benefit from offloading.

One of our workloads is a recreated NGINX [ngi] web server benchmark [Kra17] that uses OpenSSL [opea] for encrypting ChaCha20-Poly1305. We use wrk [wrk] for traffic generation and brotli [bro] with zlib [zli] for compression. Another application we use is the x264 video encoder [x26].

6.2.1 Homogeneous Marked Code

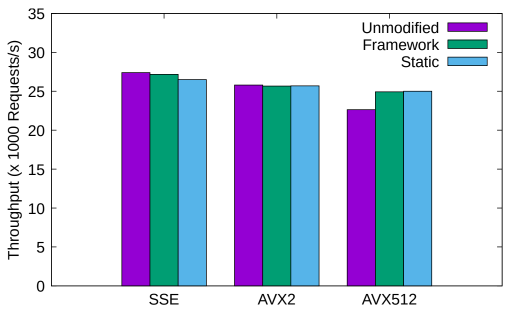
In this section we evaluate our framework with workloads where the marked code is homogeneous in terms of execution time and vector instruction usage. Having homogeneous marked code makes it easier for the policy to decide whether it is worth to offload. Each application is configured to either use full vectorization (AVX-512), vectorization except AVX-512 (AVX2) or only vectorization that does not cause frequency reductions (SSE). Every test is executed once without our framework, once with our full framework and once with our framework but a policy that always offloads marked code.

We recreated the NGINX benchmark from [Kra17]. Upon inspection, we found that most AVX-512 instructions are executed in the ChaCha20-Poly1305 encryption routine of OpenSSL. We modified OpenSSL to pass the vectorized encryption routine to our framework.

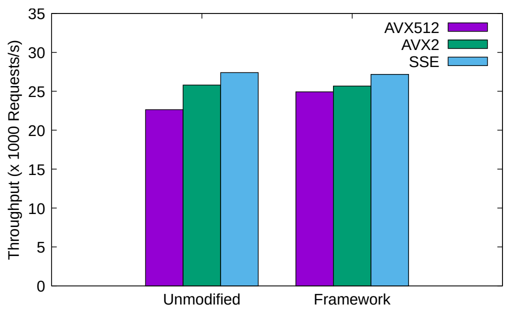
Our experiment uses four wrk threads with 1000 connections to generate five minutes long traffic for the NGINX server described in Section 6.1. Wrk is running on separate cores. The NGINX server and our framework share 8 physical cores and the corresponding hyperthreads. To generate scalar workload, the NGINX server is configured with gzip and brotli compression on. We turned AVX-512 and AVX2 in zlib and brotli completely off. We configured OpenSSL to only use for the ChaCha20-Poly1305 routine vectorization.

Performance Comparison We define the performance of the NGINX server to be the amount of requests the server can finish per second. Figure 6.2 shows the results of the described experiment. Subfigure (a) are the results grouped by amount of vectorization. The SSE experiment did not benefit from offloading and our policy was able to impose only minimal overhead indicating it detected the type of vectorization correctly. The AVX-512 setup greatly profited from offloading and our policy was able to detect that. In this setup offloading did not make a big difference when using AVX2. Subfigure (b) shows the same results grouped by whether our framework was used or not. This representation illustrates well that the variability of performance is less when using our framework. It also shows that offloading can not completely eliminate the performance degradation for this experiment.

Policy Activity We exported the course of the key policy variables out of our server during the experiments to validate what the policy is doing. Figure 6.3 (a),(b) and (c) show the course for the three experiments (AVX-512, AVX2, SSE) that used our policy and (d) summarizes the performance of those experiments with and without our framework. We mapped the internal heuristical metric, the threshold and the percentage of offloaded clients.



(a) The SSE test does not profit from offloading. For the AVX2 test offloading makes no difference. The AVX-512 test greatly improves when offloaded. Our framework can make use of offloading or decide against using it when not appropriate.



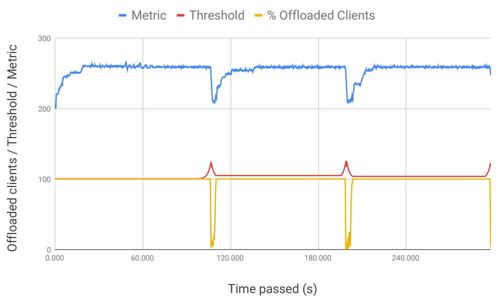
(b) Using our framework reduces performance variability. Offloaded AVX-512 is not as fast as not using it at all in this setup.

Figure 6.2: Performance degradation and mitigation thereof in NGINX server. Higher is better.

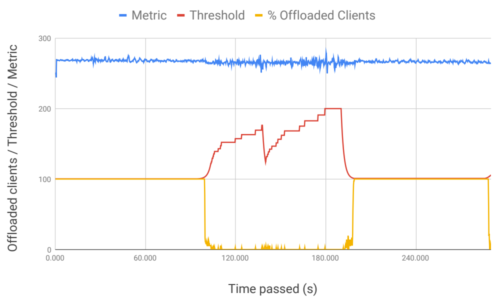
Subfigure (a) shows the AVX-512 experiment. The initial threshold of our policy is very low, thus all clients offload immediately. It takes a few seconds before offloading reaches the maximum performance gain. After 100 seconds the exciter triggers a rise in the threshold, resulting in nearly all clients stopping to offload. The internal metric sinks after the threshold rises, so the policy readjusts. This happens again after another 100 seconds.

Subfigure (b) shows the AVX2 experiment. Since offloading does not impact the performance at all, the metric is quite steady. The policy keeps offloading until the exciter suggests changing the threshold. Since the performance penalty of offloading is not large enough to trigger reducing the threshold again, all clients stop offloading till the next exciter cycle. Subfigure (c) remains completely static because there are no AVX-512 or AVX2 cycles at all, so each client stays under the threshold no matter how low it is.

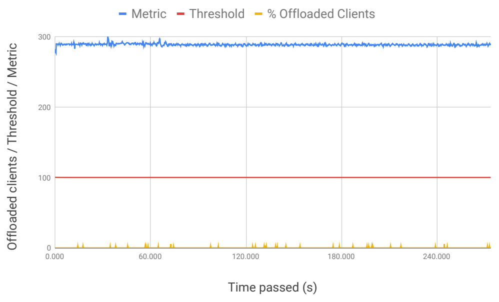
Figure 6.4 shows the policy behavior when the video encoder x264 is used. We marked the routine “x264_macroblock_analyse” which uses the most vector instructions, but x264 uses a lot of other routines that are vectorized. The metric is a lot more jittery than for NGINX, since encoding a video is not as homogeneous as serving the same webpage every time. Our policy realized that the performance is better when not offloaded and decides up to the next exciter trigger to not offload. Most of the scalar code in x264 is strongly intertwined with the vector code. Therefore, offloading is not beneficial for x264 since most scalar code will be slowed down anyway.



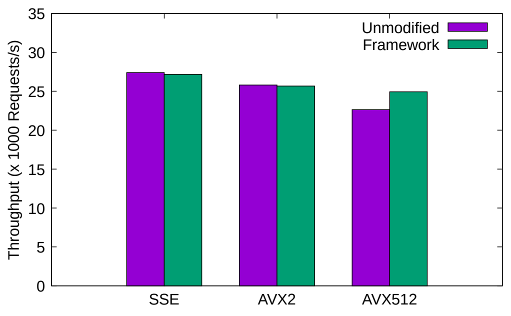
(a) Policy development AVX-512: Offloading is beneficial. Increasing the thresholds results in worse performance. The metric estimates the system performance and the feedbackloop counteracts the metric reductions caused by increasing the thresholds.



(b) Policy development AVX2: Offloading makes no difference. Thresholds are purely determined by exciter.



(c) Policy development SSE: Nothing happens because there are no AVX2 or AVX-512 cycles.



(d) Summarized performance results: Our framework either increases performance significantly or imposes a small penalty, due to policy delays and overhead.

Figure 6.3: Policy behavior during NGINX benchmark: Each value is normalized to a range from 0 to 100 and offset to separate the lines. Metric: Higher is better.

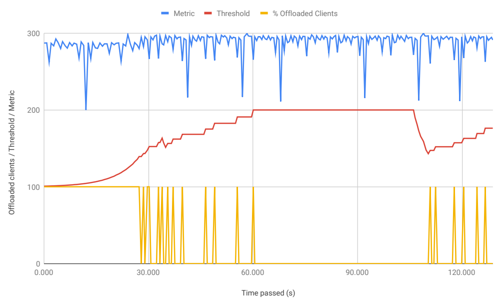


Figure 6.4: Policy behavior during one thread of x264 video encoding: x264 is nearly completely vectorized, thus offloading parts of it is not beneficial to performance.

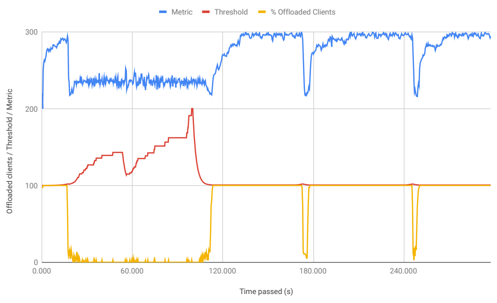
6.2.2 Mixed Workloads

In the last experiment we evaluated our policy with a homogeneous workload. Our framework was built with the purpose to let multiple applications in a system benefit from offloading. Also, not all applications have very homogeneous marked code section such as NGINX. Therefore, we investigate in this section how well our framework performs in mixed workloads. To model a heterogeneous system we mix our NGINX benchmark with other benchmarks.

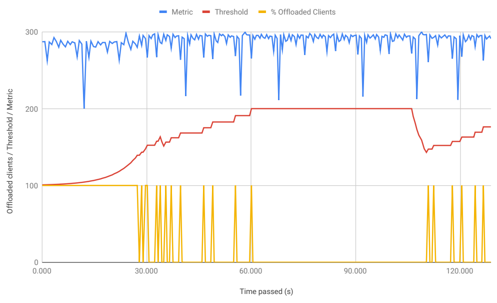
Figure 6.5 shows the results of NGINX being combined with the x264 video encoder. Similar to Figure 6.3, subfigures (a), (b), (c) show the course of the policy and (d) shows the performance. Subfigure (b) is the same as Figure 6.4 and shows how the policy behaves when x264 is run alone. Subfigure (a) is x264 combined with AVX-512 NGINX and (c) with SSE NGINX. AVX-512 NGINX and x264 both use AVX-512, so offloading does not result in enough non-frequency-reduced time for the scalar part of NGINX. Thus, (a) shows that while x264 is running no offloading occurs. As soon as x264 is done, the policy returns to the same curve as when NGINX is run alone. Interestingly, subfigure (c) shows that when run with a scalar NGINX, offloading becomes beneficial. Subfigure (c) shows that most clients get offloaded during the execution of x264 which suggests that our heuristic for detecting AVX-512 cycles is not very precise. The performance of the NGINX benchmark benefited from our framework when run with AVX-512 or was not changed when run with only SSE (see (d)).

6.2.3 Parameter Tuning

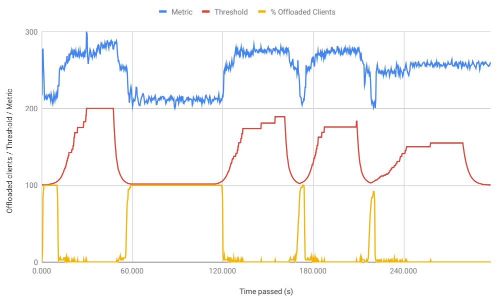
Parametrization of a policy is sometimes as important as the policy itself. In this section we investigate the effect of varying certain parameters and try to understand which scenarios benefit from certain parametrizations.



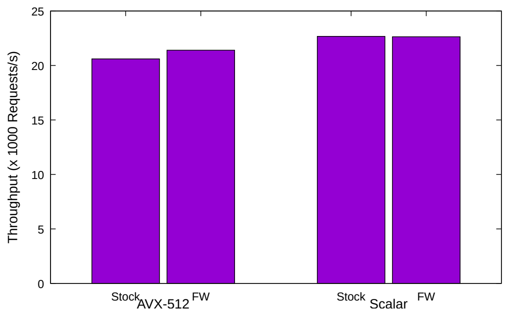
(a) NGINX AVX-512 with x264



(b) x264 solo



(c) NGINX SSE with x264



(d) Performance of NGINX when mixed with x264

Figure 6.5: Combined benchmark: NGINX with x264

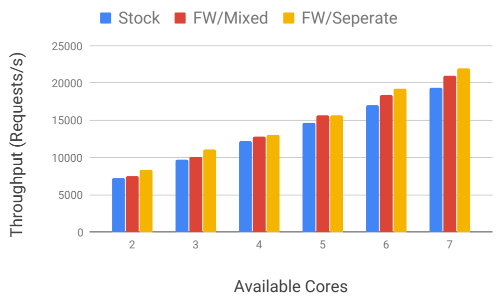
Core Count We mentioned that increasing core counts can counteract the impact of the lower system utilization when using dedicated cores. To get an idea of how the core count affects the performance of our framework we varied the core count in our AVX-512 NGINX benchmark. We run the benchmark either without the framework, with the framework or with the framework with an additional virtual offloading core. The virtual core shall represent a manycore system where an additional core does not have any impact on utilization.

Figure 6.6 shows the results. Subfigure (b) shows that increasing the core count when not having a virtual core increases the performance until the dedicated core is fully utilized. The experiment with a virtual extra core slightly loses performance benefits with increasing utilization due to latencies becoming higher for offloading.

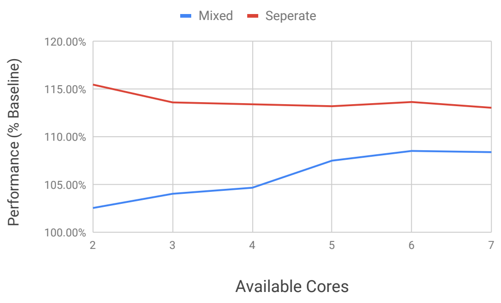
Window Size and Exciter Frequency The window size of the policy determines over how long the metric is averaged and how quickly changes are made. Having big windows results in less fluctuation but slower adaptation. The exciter frequency determines how often random threshold adjustments are made to avoid spending too much time in a local minimum. A small frequency can result in long times spent in minima, but a long frequency can result in a lot of fluctuations.

We varied window size and exciter frequency in our AVX-512 NGINX benchmark to investigate the effect on offload detection quality. Figure 6.7 shows for both exciter period and window size a positive performance trend when increased. When increasing the exciter period, the policy will less often stop offloading, resulting for this benchmark in better results. The same applies to increasing the window size, but increasing the window size also results in less high frequency fluctuations. The fluctuation effects are clearly visible when inspecting the policy diagrams in Figure 6.8.

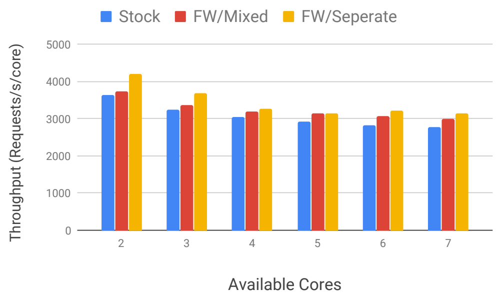
Decision Cache Update Frequency How often the library updates the decision cache is determined by the parameter "server update period". A server update period of n means that a decision cache is valid for n calls. Thus, the decision cache gets updated with a frequency of $\frac{1}{n}$ times per call. Since the library has to communicate to the policy server via queues and IPC across processes we need to inspect how big the overhead is. When repeating the primesieve from Section 3.2 with and without the server activated there is no statistical significant overhead measurable. To find the imposed overhead we created an application that can be finely configured on how much work it does. The work amount is directly proportional to the execution time without our server, hence measuring the ratio of time spent in the workload to overall time yields us the efficiency. We set the application to communicate with the framework by requesting a decision cache



(a) As expected performance increases when more cores are available. The framework is always beneficial, especially with an additional virtual core.

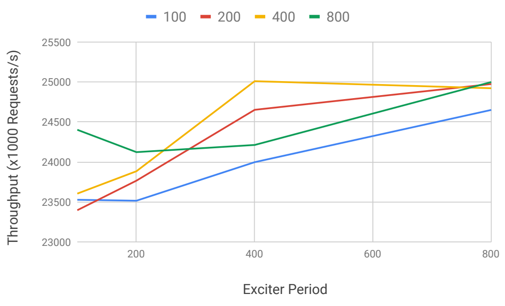


(b) Performance (throughput) trend when varying core count: When using a virtual core the performance in comparison to the unmodified NGINX remains steady until the dedicated core becomes fully utilized. The version without virtual core becomes better in relation to the unmodified NGINX when the utilization of the dedicated core becomes higher.

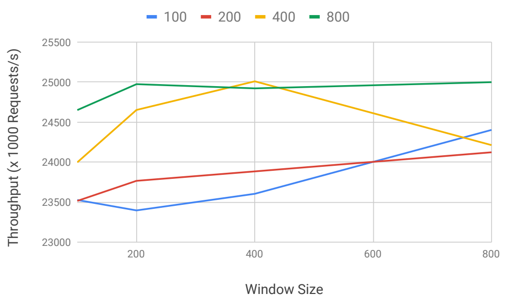


(c) Same results as (a), but performance is normalized to core count. The per core performance becomes worse with more cores. This is because the workload is not perfectly parallelizable and when using the framework the dedicated core becomes more utilized.

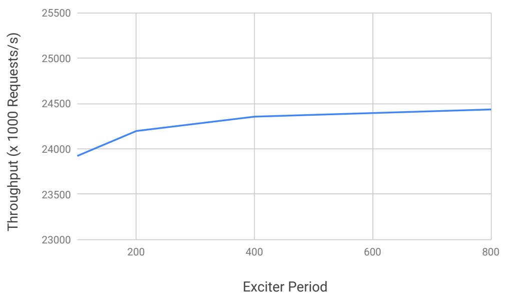
Figure 6.6: Performance of AVX-512 NGINX benchmark under variation of core count. Stock = unmodified, FW/Mixed = framework, FW/Separate = framework with virtual core



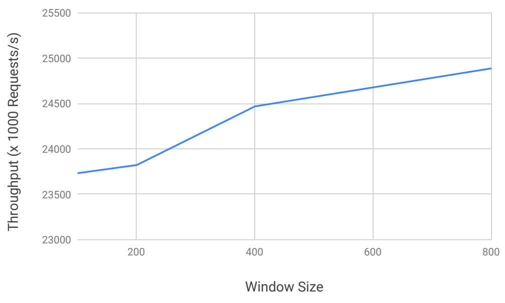
(a) Variation of exciter period (x-axis) and window size (line color)



(b) Variation of window size (x-axis) under constant exciter period (line color)

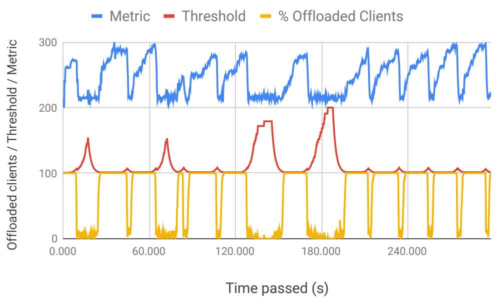


(c) Average of all window size in (a)

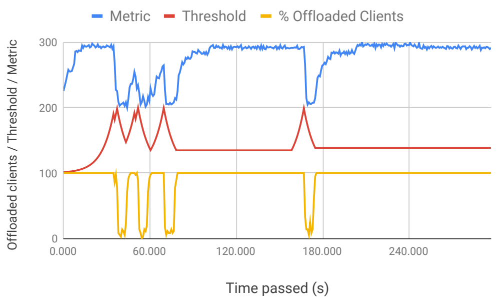


(d) Average of exciter periods in (b)

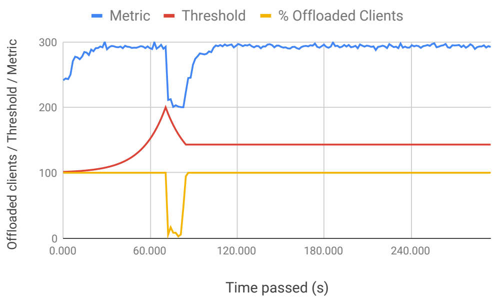
Figure 6.7: Variation of window size and exciter period in AVX-512 NGINX benchmark: Performance



(a) Exciter Period: 100, Window Size: 100



(b) Exciter Period: 200, Window Size 400



(c) Exciter Period: 800, Window Size: 800

Figure 6.8: Variation of window size and exciter period in AVX-512 NGINX benchmark: Policy diagram. A higher exciter frequency and lower window size result in more fluctuations.

update with a certain frequency. The results in Figure 6.9 show that when using an update period of 100 for a workload that is at least 100ns long, the efficiency is about 99.94%. The shortest workload that still triggers a power level change, has with an update period of 100 less than 1% overhead.

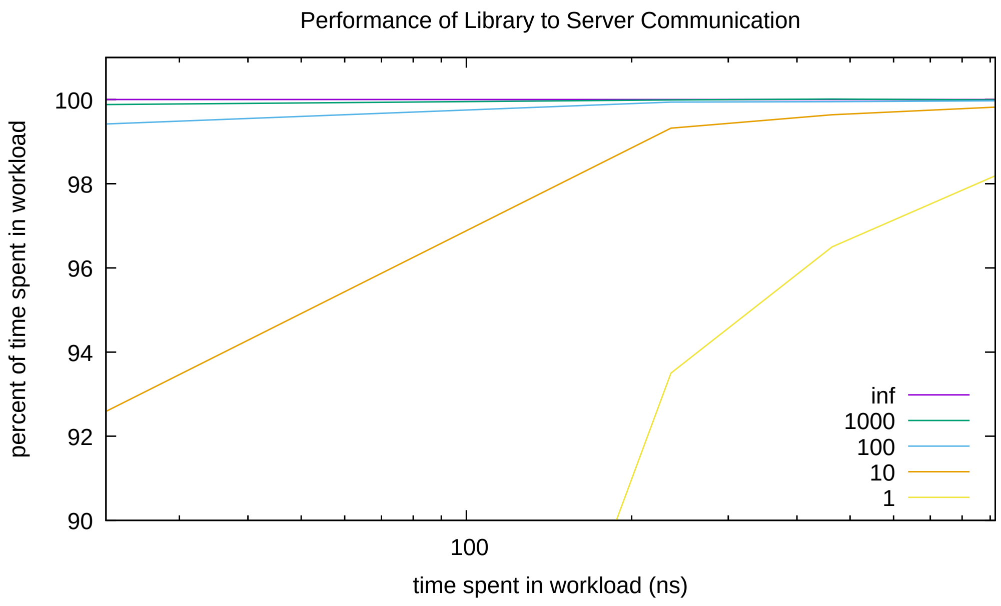


Figure 6.9: Measurement of efficiency under variation of server update frequency and workload size: Each line represents a configuration with a different update period. An update period of 100 results in less than 0.05% overhead caused by library-to-server communication in applications with workload sizes above 100ns.

6.3 Discussion

Our benchmarks showed that using HPWVIs can result in a performance penalty. The NGINX + OpenSSL benchmark is a perfect example for the problems arising when having unpredictable effects on unrelated code. AVX-512 accelerates OpenSSL encrypting a lot, but the proportion of the encrypting code is very small and NGINX is completely scalar. Thus, the frequency reduction has a big negative impact and can not be outweighed by the small acceleration. This combination of applications and the negative impact could not have been foreseen by the OpenSSL developers.

We showed that our framework is able to reduce the performance variability between using AVX-512, AVX2 and none of the two. It can accelerate AVX-512 applications by partially mitigating the performance degradation and does not slow down scalar applications by detecting when offloading is beneficial. Some applications benefit from running only SSE instead of AVX2 and AVX-512 even when offloading AVX-512. Low system utilization plays a role in making AVX-512 combined with offloading not outperform SSE, but has less impact when more cores are available on the CPU. That is also the reason why [GB18] achieved reducing the degradation to 3.7% while we could only get it down to 8%. They could achieve a better degradation reduction by not dedicating their core to executing only AVX-512 and thus had a higher system utilization. We dedicated our cores mainly for having a better detection policy.

Our policy performs well for homogeneous marked code. This is on one hand due to the performance heuristic performing well for homogeneous loads and on the other hand because we decide for all applications equally how high their thresholds are. Experimentally we adapted our policy to have per client thresholds, but found in a series of smaller tests with homogeneous loads that this policy does not work well on a per client basis. Most of the times the policy was stuck with only half of the clients offloaded which is the worst-case-scenario. An explicit performance metric given by the application paired with an identification of different marked code regions might help to make per-client thresholds feasible.

Mixed heterogeneous workloads suffer a lot from the implicit performance metric. For example, running x264 with a scalar NGINX resulted in complete offloading. This accelerated NGINX, probably because of scheduler advantages. Since the metric of NGINX had more weight, the policy decided that trading x264 performance for NGINX performance is worth it.

Chapter 7

Conclusion

It has been proven that using AVX-512 instructions can degrade the overall performance of a system. AVX-512 execution units are part of the dark silicon on a chip and require too much power to be run at normal core frequencies. Increasing the core frequencies after a burst of AVX-512 instructions takes time which results in subsequent code being run at low frequencies. Depending on the system workload, this often results in worse overall performance when using AVX-512. AVX-512 instructions are not the first and will not be the last high power instructions that require a frequency reduction that might result in poor system performance. The risk of slowdown causes application developers to deactivate AVX-512 in their applications [opeb].

We provide a framework that uses core specialization to allow application developers to mitigate unexpected slowdown caused by AVX-512. Application developers only have to mark code that potentially executes AVX-512 in their application. The policy we designed can then dynamically decide during runtime whether an application should execute marked code regions on dedicated AVX-512 cores or not. By isolating AVX-512 onto dedicated cores we reduce the impact of frequency adjustment delays. Our framework focuses on AVX-512, but the principles we discuss in this work and our policy apply to any high power instructions that require frequency reductions to run.

We recreated a real world web server example that demonstrates that using AVX-512 results in poor performance. To prove the value of our framework we modified the experiment to make use of our framework and showed that we can significantly reduce the performance variability. We were able to reduce the performance degradation from 17% to 8%, even though we dedicated a full core to running offloaded work. One of our experiments showed that more available cores improve the mitigation of the performance degradation. Therefore, the feasibility of our approach will improve on future CPUs with more cores.

7.1 Future Work

We tested our policy only on a single CPU while the impact of AVX-512 and the feasibility of our framework largely depend on the amount of cores on a system and its frequency reduction behavior. Thus, we suggest more evaluation on systems with more CPU cores and different frequency reduction behaviors and more power down levels.

We have no prior knowledge of marked code regions and no method to identify them. Our framework thus assumes that all marked code regions within a single application are homogeneous in terms of execution time and vectorization. We suggest implementing a mechanism that can identify different code regions and then passing them separately to the policy. Furthermore, we expect a better performance of the policy when more information about the marked code regions is available. For example, explicit performance metrics could be passed to the policy to eliminate the additional heuristic. The implicit performance heuristic that uses the call frequency worked well in our tests, but an explicit metric given by the application might perform even better.

A fully transparent solution that does not require any modification to applications would also be interesting. Detecting HPWVIs can be done by letting the CPU trap into the kernel when such instructions are executed. Such a fully transparent solution has no information about how long HPWVI bursts are, thus the policy needs to additionally decide how much to offload. Trapping also imposes more overhead than our solution.

Our aggressive offloading showed promising results and could be extended by using upper and lower thresholds for offloading. Having an upper threshold for the decision to stop offloading a client is reasonable, because on a dedicated core more cycles spent in higher power down levels are caused by other AVX-512 applications running on that core. Maybe a more fine measurement combined with analysis of scheduling sequences might help to better determine which code regions cause frequency reduction. The heuristic could be completely removed if there were performance counters that measure directly how many instructions are executed which can cause frequency reductions.

Our design mentions automatic HPWVI core adding and removing. Most of our tested workloads did not profit from more dedicated cores, because the system utilization got too low. Thus, we did not create a validated mechanism that automatically adds and removes more than one dedicated core. We expect having more than one dedicated core to benefit mixed workloads where AVX-512 instructions take up about half of the cpu time and are strictly separated from scalar code. Ideally the policy would separate scalar and non-scalar work onto the two sets (dedicated and not dedicated) of cores.

Bibliography

[AK13] Nabeel AlSaber and Milind Kulkarni. Semcache: Semantics-aware caching for efficient gpu offloading. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 421–432, New York, NY, USA, 2013. ACM.

[avo] Github: avocado-framework/avocado. `https://github.com/avocado-framework/avocado`.

[BC06] Michela Becchi and Patrick Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. *J. Instruction-Level Parallelism*, 10, 2006.

[bro] ngx_brotli - nginx module for brotli compression. `https://github.com/google/ngx_brotli`.

[cpu] Github: lpechacek/cpuset. `https://github.com/lpechacek/cpuset`.

[EBSA$^+$11] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.

[GB18] Mathias Gottschlag and Frank Bellosa. Mechanism to mitigate avx-induced frequency reduction, 12 2018.

[GS90] R. Gupta and M. L. Soffa. Region scheduling: an approach for detecting and redistributing parallelism. *IEEE Transactions on Software Engineering*, 16(4):421–431, April 1990.

[HPS97] James C. Hu, Irfan Pyarali, and Douglas C. Schmidt. Measuring the impact of event dispatching and concurrency models on web server performance over high-speed networks. 1997.

[int04] *Enhanced Intel®SpeedStep®Technology for the Intel®Pentium®M Processor*, 2004.

[int18a] *Intel®64 and IA-32 Architectures Optimization Reference Manual*, 2018.

[int18b] *Intel®64 and IA-32 Architectures Software Developer's Manual*, 2018.

[KDK$^+$11] Stephen W. Keckler, William J. Dally, Brucek Khailany, Michael Garland, and David Glasco. Gpus and the future of parallel computing. *IEEE Micro*, 31:7–17, 2011.

[KES$^+$09] Volodymyr V. Kindratenko, Jeremy Enos, Guochun Shi, Michael T. Showerman, Galen W. Arnold, John E. Stone, James C. Phillips, and Wen mei W. Hwu. Gpu clusters for high-performance computing. *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–8, 2009.

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

[Kra17] Vlad Krasnov. On the dangers of intel's frequency scaling. `http://web.archive.org/web/20181116215418/https://blog.cloudflare.com/on-the-dangers-of-intels-frequency-scaling/`, 2017.

[LKC$^+$10] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. In *ISCA*, 2010.

[LM13] Daniel Lustig and Margaret Martonosi. Reducing gpu offload latency via fine-grained cpu-gpu synchronization. pages 354–365, 02 2013.

[LP02] James R. Larus and Michael Parkes. Using cohort-scheduling to enhance server performance. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ATEC '02, pages 103–114, Berkeley, CA, USA, 2002. USENIX Association.

[ngi] Nginx - high performance load balancer, web server & reverse proxy. `https://www.nginx.com/`.

[opea] Openssl - cryptography and ssl/tls toolkit. `https://www.openssl.org/`.

[opeb] openssl/poly1305-x86_64.pl. `https://github.com/openssl/openssl/blob/master/crypto/poly1305/asm/poly1305-x86_64.pl`.

[SMM+09] Richard D. Strong, Jayaram Mudigonda, Jeffrey C. Mogul, Nathan L. Binkert, and Dean M. Tullsen. Fast switching of threads between cores. *Operating Systems Review*, 43:35–45, 2009.

[SPFB10] Juan Carlos Saez, Manuel Prieto, Alexandra Fedorova, and Sergey Blagodurov. A comprehensive scheduler for asymmetric multicore systems. In *EuroSys*, 2010.

[SS10] Livio Soares and Michael Stumm. âœflexsc: Flexible system call scheduling with exception-less system calls,â. pages 33–46, 01 2010.

[TKK+13] George Teodoro, Tahsin M. KurÃ§, Jun Kong, Lee A. D. Cooper, and Joel H. Saltz. Comparative performance analysis of intel xeon phi, gpu, and cpu. *CoRR*, abs/1311.0378, 2013.

[VJ15] Aditya Venkataraman and Kishore Kumar Jagadeesha. Evaluation of inter-process communication mechanisms. 2015.

[wika] Wikichip: Frequency behaviour - intel. `https://en.wikichip.org/wiki/intel/frequency_behavior`.

[wikb] Wikichip: Rome - cores - amd. `https://en.wikichip.org/wiki/amd/cores/rome`.

[wikc] Wikichip: Skylake (server) - microarchitectures - intel. `https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server)`.

[Wik19] Wikipedia. Gradient descent — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Gradient%20descent&oldid=888032348`, 2019. [Online; accessed 24-March-2019].

[wrk] wrk - a http benchmarking tool. `https://github.com/wg/wrk`.

[x26] x264, the best h.264/avc encoder. `https://www.videolan.org/developers/x264.html`.

[zli] zlib - a massively spiffy yet delicately unobtrusive compression library. `https://www.zlib.net/`.

All links were last accessed on 24. March 2019.