Karlsruher Institut für Technologie

Saving Power Without Sacrificing Performance on Asymmetric Multicore Processors

Masterarbeit
von

Lukas Werling

an der Fakultät für Informatik

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

Bearbeitungszeit: 30. August 2017 – 28. Februar 2018

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 28. Februar 2018

Abstract

Asymmetric multicore processors (AMP) integrate multiple core types with different power and performance characteristics in a single package. Using optimized scheduling, these processors can deliver higher performance per watt than a symmetric multicore processor. An application executes most efficiently on a certain core depending on how it uses resources like CPU and memory. Previous approaches analyze applications at coarse granularities, classifying each process or thread. In systems such as servers that have homogeneous processes with similar behavior in all threads, these approaches cannot distribute applications to core types effectively.

However, applications generally go through different execution phases over time. These phases often differ in their resource usage and exist at both large and small scales. Whereas some systems already incorporate changing application behavior over large time intervals, it should also be possible to utilize shorter phases to save energy by migrating between cores at high frequency.

In this work, we design and implement such a system that characterizes the small-scale phase behavior of applications between developer-defined points by monitoring memory accesses with performance counters. At runtime, it migrates the application thread to the optimal core for each execution phase.

We evaluate our system on an AMD Ryzen processor. These processors allow asymmetric core configurations using frequency scaling. We fail to see reductions in power consumption with our system on these processors. We show that, contrary to available documentation, Ryzen does not have per-core voltage domains and conclude that these processors are not suitable as asymmetric platform.

Contents

Abstract **v**

1 Introduction **3**

2 Background and Related Work **7**
2.1 Previous work on AMP scheduling 7
2.2 MONITOR and MWAIT 10
2.3 Target platform 11
2.3.1 ARM big.LITTLE 11
2.3.2 AMD Ryzen 12

3 Design **15**
3.1 Single-Threaded Efficiency Specialization 15
3.2 Program Flow Analysis 17
3.3 User-mode core migration 19

4 Implementation **21**
4.1 libswp: Migration based on program flow analysis 22
4.2 libultmigration: User-mode core migration 25
4.3 Making an asymmetric processor 25

5 Evaluation **27**
5.1 Test Setup 28
5.2 Adapting an Application: MySQL 29
5.2.1 Source Code Modification 29
5.2.2 Performance Counter Monitoring 30
5.2.3 Application Profile 32
5.2.4 Migration Results 32
5.3 Microbenchmark 33
5.3.1 CPI 35
5.3.2 Cache Miss Performance Counters 35

5.3.3 Migration Overhead 37
5.3.4 Overhead From Ryzen as Asymmetric Processor 38
5.3.5 Comparison With Fixed Frequency 40
5.4 Discussion 43

6 Conclusion **45**
6.1 Future Work 46

A Complete Power Graphs **47**

Bibliography **49**

Chapter 1

Introduction

Modern processors are often constrained by power. Mobile devices, such as laptops and smartphones, run on batteries. Batteries limit the total energy available, but also impose a maximum discharge rate [25]. In contrast, servers are generally connected to the power grid. However, they are restricted in their thermal output: The server's direct cooling system - usually heat sinks and fans - has a maximum amount of power it can safely dissipate. Additionally, the data center as a whole has only limited cooling capacity. These power restrictions lead to "Dark Silicon" [12]: A processor may only activate parts of its chip at the same time to avoid exceeding its power budget. Dark Silicon limits multicore scaling. Adding more cores does not improve performance if they cannot work at the same time.

Asymmetric multicore processors (AMP) are a possible solution for Dark Silicon. AMPs consist of multiple cores with different power and performance characteristics. Given a fixed power budget, an asymmetric processor can either run few high-performance, but power-hungry cores, or lots of low-performance, but power-efficient cores [14]. Parallel applications benefit from running on many small cores. They achieve a higher speedup compared to a symmetric processor with few large cores [17]. Specialized scheduling algorithms for AMPs optimize core allocation depending on an application's needs.

On smartphones, asymmetric processors based on the ARM big.LITTLE technology are already widely deployed. The smartphone operating system schedules background tasks to the little cores and foreground tasks to the big cores [19]. This strategy reduces overall power consumption while keeping low latency for user interaction. However, it needs special semantic knowledge about the processes running on the system. Previous academic work on asymmetric processors instead observes processes and threads to determine the ideal core. The two major strategies are *parallel speedup* and *efficiency specialization* [27]. A scheduler targeting parallel speedup assigns small cores to parallel parts of an application and large cores to sequential parts, as sequential parts can run on only one core at a time.

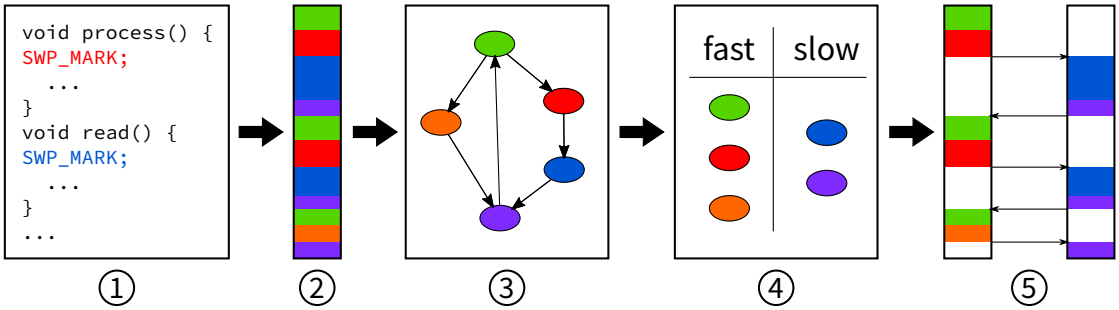


Figure 1.1: Overview of our system

For efficiency specialization, the scheduler instead characterizes threads as CPU-intensive or memory-intensive. A CPU-intensive thread makes use of the complex and fast execution units a large core provides, whereas a memory-intensive thread often stalls the CPU with memory requests for some amount of time independent of the CPU frequency. Thus, the scheduler can reduce stall cycles by running memory-intensive code on a slower CPU core, increasing overall power efficiency.

Such coarse-grained per-thread or per-process core allocation strategies do however not work well with typical server applications. Usual large-scale server deployments run only one type of application per (virtual) machine, for example a database or a web server. Unikernels, a potential future approach to deploying cloud software, similarly execute only one application per virtual machine [20]. Consequently, all active threads have very similar execution characteristics. There is no opportunity to partition at the level of threads.

However, we can find differing characteristics at smaller scales within a thread. For example, in a database application, parsing and optimizing a query is CPU-intensive, whereas reading or writing stored data is memory-intensive. By migrating such a thread between a large and a small core, we can do efficiency specialization with small code sections.

In this work, we present a library for analyzing single-threaded applications and assigning code sections to small or large cores. Figure 1.1 shows an overview of the steps involved. First, the developer modifies the application source code to add measurement points (1). The second step is to run an application benchmark in measurement mode (2), which results in a control flow graph with performance counter results for each edge (3). Processing the graph, the developer obtains an application profile that maps code sections to a core type (4). Finally, the application user runs the application in migration mode: when reaching a measurement point for which the current core is not optimal as indicated by the profile, the library migrates execution to the optimal core (5). As we are working at very small scales, we cannot rely on the high-overhead core migration the operating system provides. Instead, we do high-frequency core migration from user space.

For evaluating our library, we use an AMD Ryzen processor. We develop software for configuring the processor and achieve an asymmetric system with underclocked cores. We analyze the processor's power characteristics and assess its suitability as an asymmetric processor for our technique. We conclude that Ryzen processors do not make efficient asymmetric processors. In general, applying our technique produces worse performance per watt than an equivalent constant frequency on a single core.

This thesis is structured as follows: In the Chapter 2 below, we summarize previous work in scheduling for asymmetric processors and introduce hardware platforms with asymmetric processors. In Chapter 3, we discuss the design and in Chapter 4 the implementation of our application analysis and core migration library. We then evaluate our approach on an AMD Ryzen processor in Chapter 5. Finally, we summarize our results in Chapter 6, presenting a conclusion and proposing future work.

Chapter 2

Background and Related Work

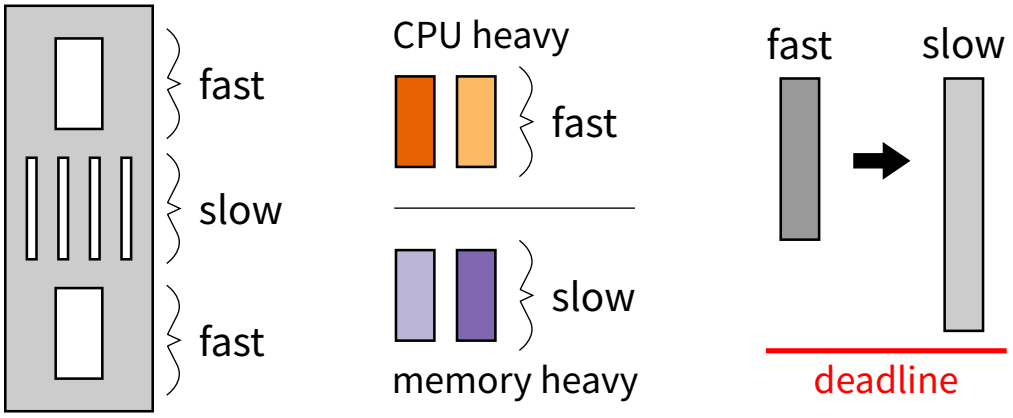
We are developing a technique for scheduling on asymmetric multicore processors (AMPs) via high-frequency core switching. This chapter provides background information on technologies and tools: First, we outline previous approaches to asymmetric scheduling, which generally work at coarser granularities. We then describe the `monitor` and `mwait` x86 instructions which form the basis for our core switching method. Finally, we describe possible target platforms with asymmetric processors, including ARM big.LITTLE and AMD Ryzen. For AMD Ryzen, we give additional information about its power supply structure and about performance counters, as we use this processor for our evaluation.

2.1 Previous work on AMP scheduling

Previous work on scheduling for AMPs works either on application or thread level. Figure 2.1 shows three common scheduling strategies that we describe in the following.

Gupta and Nathuji investigate AMPs in the context of latency-sensitive datacenter applications [14]. They describe the two main scheduling strategies for AMPs, *energy scaling* and *parallel speedup*. Energy scaling schedules threads on slower cores to save energy at the loss of performance. For applications under a service-level agreement (SLA) mandating a specific latency, the operator can use slower cores as long as that latency is met, saving money on power and cooling systems. Parallel speedup uses lots of small cores to execute parallel parts of a program and few large cores for the sequential parts.

Saez et al. propose a "Comprehensive Scheduler for Asymmetric Multicore Systems" [27], considering *efficiency specialization* as additional strategy. Efficiency specialization uses fast cores for CPU-intensive threads, and slow cores for memory-intensive threads. Saez et al. translate these requirements into a scheduling metric



(a) Parallel Speedup (b) Efficiency Specialization (c) Energy Scaling

Figure 2.1: Illustration of different approaches to scheduling on asymmetric processors. Parallel Speedup uses fast cores for sequential code and slow cores for parallel code. Efficiency Specialization classifies threads by their resource usage, choosing the optimal core. Energy Scaling applies to latency-sensitive applications: scheduling applications on the slow core is possible as long as they still meet the deadline.

by estimating the speedup factor from running on a large instead of a slow core using cache miss performance counters. They combine this strategy with parallel speedup by incorporating the number of threads an application has.

Our system works within a thread and does not take interactions with other threads into account. Consequently, it cannot differentiate parallel and sequential parts of an application, so parallel speedup does not apply. Instead, we target efficiency specialization at the granularity of code sections. In previous works, scheduling systems employing efficiency specialization usually analyze whole applications or threads. They monitor cache miss rates to differentiate memory-heavy and CPU-heavy code. At runtime, the scheduler uses this information to assign threads to cores. We adapt this principle to smaller scales, analyzing code sections within a thread. We use a scheduling metric very similar to the one used by Gupta and Nathuji, but collect performance counter events offline. We found the overhead of online performance counter collection to be too high at small scales. At this level, our scheduler cannot simply run periodically to assign threads to CPU cores. Instead, we move scheduling into the application, decide core assignments at pre-defined points in the code, and perform core migration in user space.

The third scheduling technique pictured in Figure 2.1c, energy scaling, applies to our target software as well. Providers often sell services governed under service-level agreements (SLA) which guarantee metrics such as maximum latency. The

provider can run processes on slower cores with a performance hit as long as the latency stays under the deadline mandated by the SLA. Our technique enables energy scaling by assigning more code sections to the slower core. Compared to whole-thread scheduling, this allows a gradual approach to the deadline. However, we did not pursue energy scaling further here, leaving it to future work.

All scheduling strategies for AMPs need additional information about programs to determine core assignments. Shelepov and Fedorova propose embedding *architectural signatures* into application binaries [30]. An architectural signature is a summary of the application's runtime behavior. They generate these signatures offline using binary instrumentation and L2 cache miss rate estimates. At runtime, the scheduler reads the signatures, places processes into three categories, and assigns cores based on these categories. The authors reject manual classification and runtime statistics from execution on different core types as alternative approaches for generating signatures. Developers would need a very deep understanding of their program's behavior to perform a good classification. In contrast, runtime statistics are precise for specific core types, but are unwieldy to obtain and hard to generalize for different hardware.

With *phase-based tuning* [32], Sondag combines offline analysis with runtime monitoring. The system uses static analysis to group code segments. At runtime, it monitors execution of some code segments from each group on different core types. It then assigns a core type to the whole group of code segments. When execution steps into a group with a different core type, the system initiates a core migration at the operating system. Compared to fully static systems, phase-based tuning also works well for programs whose behavior changes with the input.

As we aim to do core migration at smaller scales, our work cannot directly use previous approaches to application analysis. Our method requires manual source code modification to insert analysis points. We then monitor cache misses between analysis points, similar to Architectural Signatures. This approach requires only simple knowledge about the program from the developer. Compared to phase-based tuning, we can obtain smaller code segments than what is possible with information from static analysis. We currently do not do any online runtime monitoring. Although extending our library to do online performance counter collection would be possible, we expect the additional overhead to be problematic.

With special hardware support, it is possible to implement core migration with very low overhead. With Thread Motion [24], Rangan et al. explore fine-grained core migration on a processor with shared L1 caches. Adopting such a model would require significant architectural restructuring. In contrast, Fast and Scalable Thread Migration [26] amends existing architectures and requires only little extra hardware. It improves performance by migrating cache contents along with threads to another core.

We evaluate our system on standard AMD Ryzen processors, which do not include any special support for core migration or asymmetric core configurations. We find that these processors do not show the power savings we hoped to see. A hardware platform built for high-frequency asymmetric scheduling would have to incorporate core switching hardware, significantly improving on our results with commodity hardware.

2.2 MONITOR and MWAIT

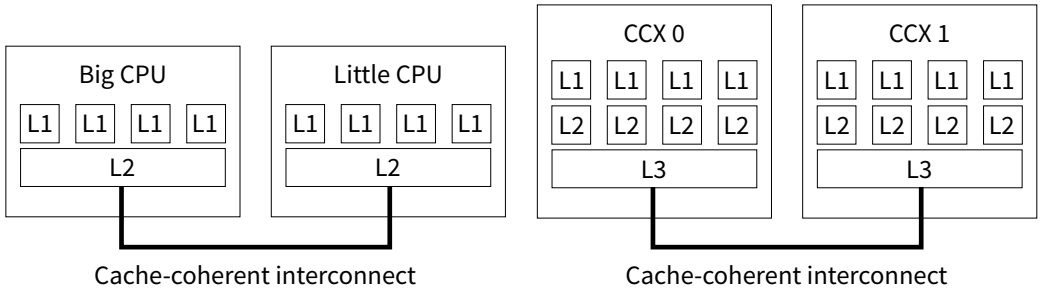
Schedulers for asymmetric processors need to control precisely which application runs on which core. Our prototype runs in user space only. Operating systems usually support core migrations by pinning threads to specific cores. However, this approach involves the OS scheduler for each migration and thus has high overhead. Instead, our technique for fast CPU core migrations builds on the x86 `monitor` and `mwait` instructions.

The `monitor` instruction sets a memory address range to watch. The `mwait` instruction then puts the CPU to sleep until another CPU writes to a monitored memory address or an interrupt happens [2]. To filter interrupts, it is necessary to repeatedly check whether the monitored address actually changed and to issue another `mwait` if not.

`monitor` and `mwait` are usually only available in kernel mode. On AMD processors, a model-specific register (MSR) enables the instructions for user mode as well [3]. This is not possible on most Intel processors [10], which currently limits the technique to AMD processors.

`mwait` takes a C-state number as argument. C-states are processor sleep states and are numbered as C0 (operational), C1 (halt), etc. The C-state selection allows a compromise between sleep power consumption and wakeup latency: higher C-states consume less power, but waking up takes longer. However, for usermode `monitor` and `mwait`, Intel does not allow any C-state selection; the CPU always goes into C1 [10]. AMD does not document such restrictions, but our power measurements do not show any variance between different `mwait` arguments, suggesting a similar policy.

Although `monitor` and `mwait` are x86-specific, similar instructions for waiting on and signalling events are available on ARM as well (*Wait For Event* and *Send Event* [5]).



(a) ARM big.LITTLE (b) AMD Ryzen

Figure 2.2: Simplified cache structure on ARM big.LITTLE and AMD Ryzen systems. In both cases, the L1 cache consists of a data and an instruction cache.

2.3 Target platform

To evaluate our system, we need a platform with an asymmetric processor. Previous works on asymmetric processors often use Intel or AMD processors with underclocked cores for their evaluation [27][30][32]. In this section, we first consider ARM big.LITTLE processors, which are widely deployed processors with heterogeneous core architectures. However, we find these processors to be unsuitable for our system. Similarly to previous works, we use an AMD Ryzen processor with underclocked cores instead and describe that processor in more detail.

2.3.1 ARM big.LITTLE

Asymmetric processors designed by ARM are nowadays commonly deployed in smartphones.

In their "big.LITTLE" processors, ARM integrates two separate CPUs on a single chip. Those CPUs often differ in their microarchitecture. For example, the Samsung Exynos 5 Octa System-on-Chip (SoC) [28] combines an out-of-order ARM Cortex A15 CPU [6] with an in-order ARM Cortex A7 CPU [7].

Figure 2.2a shows the simplified cache hierarchy of an octa-core ARM big.LITTLE CPU. Each core has a private L1 cache. Two L2 caches for the big and the little CPU each connect to four L1 caches. A cache-coherent interconnect connects the L2 caches with each other and the memory bus, providing a consistent view of memory across all cores. Consequently, there is no common cache for all CPU cores. Our system relies on migrating processes between cores with different power and performance characteristics. For the ARM processors, this means migration between cores of the big CPU and cores of the little CPU. As these cores do not share a cache, all memory accesses after a migration are cache misses. The remote cache or the main memory have to handle the requests via the interconnect.

Future asymmetric ARM processors called "DynamIQ" will include a shared L3 cache for both CPUs [33]. The resulting memory hierarchy resembles classic symmetric multicore processors. Consequently, we expect lower core migration costs from this future platform.

As our work depends on quick core migration, the current big.LITTLE processors are not suitable. However, a future re-evaluation of the technique on "DynamIQ" processors may yield new results.

2.3.2 AMD Ryzen

"Ryzen" is AMD's current generation of x86 desktop processors. It includes all features we need for our system: user space `monitor` and `mwait` is available (see Section 2.2), we can manually control frequency and voltage of each individual core via P-states, and each core has its own voltage domain. In contrast, on current Intel processors, the operating system has no direct control over P-states and there is no user space `monitor`/`mwait`.

Although Ryzen processors are symmetric multicore processors, we can use frequency scaling to configure different frequencies per CPU core (see Section 4.3). Unfortunately, we find that Ryzen does not make an efficient asymmetric processor for our use-case due to peculiarities in its structure and its power supply. In the following, we describe these processor design decisions in more detail.

Core Complex

Ryzen processors are organized in two core complexes (CCX). Figure 2.2b shows the processor's cache hierarchy. Each CCX has a shared 8 MB L3 cache and four cores. Each core has private L1 and L2 caches. The processor features simultaneous multithreading (SMT) with two hardware threads per core [4]. Except for the extra cache level and SMT, this structure is very similar to the ARM big.LITTLE processors (Figure 2.2a). Indeed, migration from one CCX to the other suffers from the same issue as on ARM: There is no shared cache for both CCX; memory accesses go to the remote L3 cache via the interconnect (called "Infinity Fabric" by AMD). However, the processor allows setting different frequencies and voltages for cores within a CCX. Accordingly, we work only with cores from one CCX.

AMD sells Ryzen processors with four, six, and eight cores. Six-core processors come with one core disabled per CCX; on four-core processors, two cores are disabled per CCX.

Power

Each CCX has three main voltage domains [31]. The "Real VDD" (RVDD) is the core supply at the package level and powers the L3 cache logic. The "VDD memory" (VDDM) powers the static random access memory (SRAM) in the L2 and L3 caches. From the RVDD, low-dropout regulators (LDO) create per-core VDD supplies.

The currently active P-state determines the per-core VDD. A P-state is a combination of core frequency and voltage. On Ryzen processors, model-specific registers (MSR) allow changing the P-states. The frequencies are freely configurable at runtime. In contrast, the firmware configures fixed RVDD voltages of 1.2 V, 1.0 V, and 0.8 V on our test system. On each CCX, the highest P-state determines the overall RVDD. We can read these voltages via sensors on the motherboard. In contrast, AMD does not document any sensors for reading the actual per-core voltages.

Our asymmetric processor configuration sets the cores of one CCX up with different P-states. We have to use cores from only one CCX to avoid high migration overhead. Therefore, all cores get the same RVDD supply and the slower core has to use its LDO to obtain the lower voltage, which is less power efficient than a separate supply from the motherboard. In Section 5.3.5 of our evaluation, we conclude that our Ryzen processor does not appear to use per-core LDOs. Due to these restrictions, Ryzen processors are not as efficient as asymmetric processor as a dedicated design with independent voltage supplies would be.

To save power, the CPU cores enter C-states when idle. Higher C-states use less power, but take longer time to wake up from. At the highest C-state C6, the full core is power-gated. Although the operating system has full control over P-states, it cannot influence C-state selection on Ryzen[1]. AMD recommends that operating systems should not try to manage power with P-states and should instead rely on the integrated automatic C-state selection [16].

We follow AMD's recommendation as we assign a static P-state to each core and do not switch between P-states at runtime. However, the processor's automatic C-state selection does not work properly with our core migration technique: As described in Section 2.2, the processor does not appear to use higher C-states during usermode `mwait`. Consequently, an application running with our system will always use additional power for the core idling in `mwait`. As a proper asymmetric processor would not restrict `mwait` like this, we control for this issue by having a core running `mwait` during comparison benchmark runs without our system.

[1] On the contrary, on Intel CPUs, the operating system explicitly selects C-states with the `mwait` instruction, but can set only a rough policy for P-state selection.

The processor has running average power limit (RAPL) counters [3]. They allow estimating power usage for each individual core as well as the whole CPU package.

Performance Counters

Ryzen processors include two types of performance counters, one per core and the other type per L3 cache [3]. We use the per-core counters to monitor cycles, instructions retired, and L2 cache misses. The L3 cache counters are shared for all cores on the same CCX. We use these counters to monitor L3 cache misses.

Chapter 3

Design

Our goal is to save energy in server software by adapting it for asymmetric multicore processors at the granularity of code sections within a thread. In Section 2.1, we discussed three scheduling strategies from previous works that achieve energy reductions—*energy scaling*, *parallel speedup*, and *efficiency specialization*—and concluded that efficiency specialization fits best to our target scenario.

In contrast to previous work, our system does not distribute multiple diverse threads to processor cores. Instead, it migrates individual threads between two cores. Although we simplify our prototype implementation by limiting it to single-threaded applications, our approach works with multi-threaded applications as well. In our prototype, we reserve a fast and a slow core; a multi-threaded implementation could choose from a larger pool of cores. As our goal is to optimize energy consumption, having idling cores that only wait for work is not an issue. In a power-constrained system where full processor utilization is not viable, using the fast cores as efficiently as possible is the main priority.

The rest of this chapter is structured as follows: We start with a more detailed account on applying efficiency specialization to individual threads. In Section 3.2, we describe how we modify and analyze applications to obtain the information necessary for scheduling. Finally, in Section 3.3 we specify our user space core migration technique that is necessary to avoid high migration costs.

3.1 Single-Threaded Efficiency Specialization

This section describes how to apply user-level core migration to do power-efficient asymmetric scheduling with only a single thread.

There are memory-(bandwidth-)bound and compute-bound applications. When executing compute-bound applications, the CPU works mainly on data in registers and the L1 cache. It constantly has work to do and thus benefits from a high

clock rate. On the other hand, memory-bound applications have a large working set that does not fit into the CPU caches. The CPU frequently stalls on memory accesses [18]. While stalled, the CPU continues running at its base clock, but cannot retire any instructions. As lowering the CPU's base clock does not affect memory latencies, we can reduce the amount of stalled cycles by scheduling a memory-bound application to a CPU core running at a lower frequency. This strategy saves energy without substantially reducing application performance [34]. In other words, running a memory-bound application at lower frequency improves CPI (cycles per instruction, lower is better) while lowering power consumption.

Many real programs do not fit in the rigid memory- or compute-bound categories. They have execution phases that are more demanding on memory, and others that do compute-bound work [29]. By identifying these phases, we can modify the program to execute each phase on either a high frequency or a low frequency core.

Previous work that utilized execution phases did so at large granularities only. As we work with a user space core migration technique with little overhead, switching very frequently is possible. Therefore, we can exploit phase behavior at smaller scales as well.

There are different approaches for finding execution phases and initiating core switches:

Fully manual The application developer knows about the application's execution characteristics and manually inserts calls to the core migration library. Although very simple, this approach has significant drawbacks. Even with good knowledge of a code base, it is often hard to predict where the CPU stalls due to memory accesses. Caches differ between CPUs, so a switching strategy that works well on one CPU may produce completely different results on another. Consequently, manual analysis will only work well in constrained environments such as embedded computing.

Manual migration point placement, automatic analysis Here, the application developer only has to identify points where the execution characteristic may change. The actual classification into compute- or memory-bound happens automatically using performance counters.

Fully automatic The previous approaches all required manual source code modifications. As application source code often is not available, a technique that works with unmodified binaries is desirable. The "phase-based tuning" system [32] implements this approach.

Our system implements the second approach. On one hand, we believe that fully manual classification by a developer is not viable and unlikely to produce good results. On the other hand, requiring the developer to manually separate

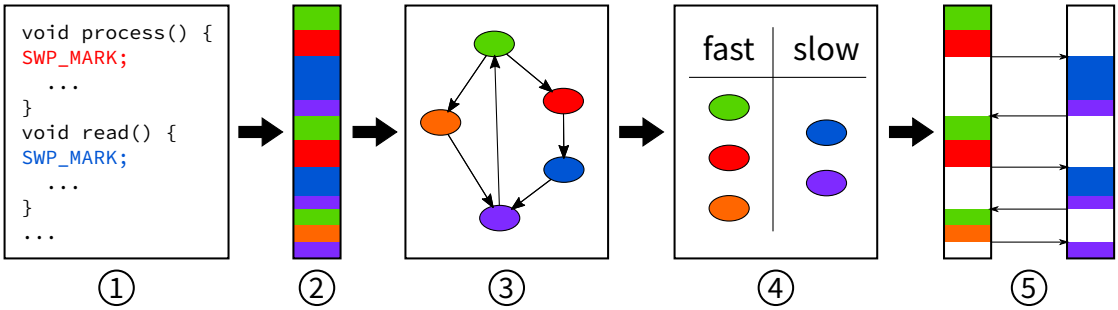


Figure 3.1: Overview of our system for program analysis

code sections without classifying them has advantages compared to fully automatic analysis. From manually inserted library calls, we can precisely analyze all code sections using performance counters without having to perform sampling or relying on expensive emulation techniques. The following section describes our approach in more detail.

3.2 Program Flow Analysis

We now describe our system for classifying an application into compute- and memory-bound parts. To adjust an application for asymmetric scheduling, a developer works through the following steps, illustrated in Figure 3.1:

1. The developer modifies the application's source code to add measurement points which are calls into our library. These points should mark transitions between different logical parts of the application. For example, in a database application, each command type (e.g., select, insert, delete) should start with a measurement point.

2. The developer executes a representative workload in measurement mode (i.e., linking with our measurement library). At each measurement point, our library reads performance counters, aggregating results for each pair (start to end) of measurement points. This step needs only one CPU core and does not perform migration.

   As we are doing precise measurements synchronous to the instrumented code, this step potentially has high overhead.

3. The data from the previous step forms a control flow graph with average performance counter results on each edge.

4. Using the control flow graph and a threshold value for one of the performance counter events, we can arrange the measurement points by core type. For

example, we may run all code sections with a cache miss rate larger than a threshold on the slow core and the rest on the fast core. This mapping is the application profile we pass to the program at runtime.

The choice of threshold value allows controlling the performance and energy saving ratio. The more code sections are running on the slow core, the larger are the energy savings, but the lower is the application performance. In this work, we decide on threshold values through experimentation. Future work may use them to perform Energy Scaling: By taking deadlines into account during runtime, we could run more code on the slow core.

5. The user then starts the application in migration mode (i.e., linking with our migration library), providing the application profile. When hitting a measurement point, the application uses the profile to decide on the fast or slow core for the following code segment, migrating as needed.

The steps above leave some decisions to the developer: Which metrics should the performance counters capture? How can the developer evaluate their measurement point placement? We discuss these points in the following.

Metrics Selection

In step 2, we use performance counters to characterize each code section. As discussed before, we want to differentiate memory-bound and CPU-bound code sections. As the CPI (cycles per instruction) of memory-bound programs reduces when executing on a slow core instead of a fast one, CPI is an obvious metric to capture with performance counters. However, two measurement runs on different cores are necessary to calculate the ratio. Instead, we find memory-bound sections directly by monitoring cache miss events. Our Ryzen CPU offers events for "L2 latency" (cycles spent waiting for L2 fills), and "L3 cache misses" (number of L3 cache misses) [3]. We normalize these event counts by instruction count to account for code sections of different lengths.

In Section 5.3.2, we show that using cache miss counters instead of CPI ratio yields similar results.

Measurement Point Assessment

The code sections between the measurement points have to be long enough to avoid excessive overhead from core migration. At the same time, each section should fit well into the memory-bound / compute-bound classification and should contain only little mixed code. These goals are in conflict. Consequently, finding good measurement points requires some experimentation: is it better to split a code

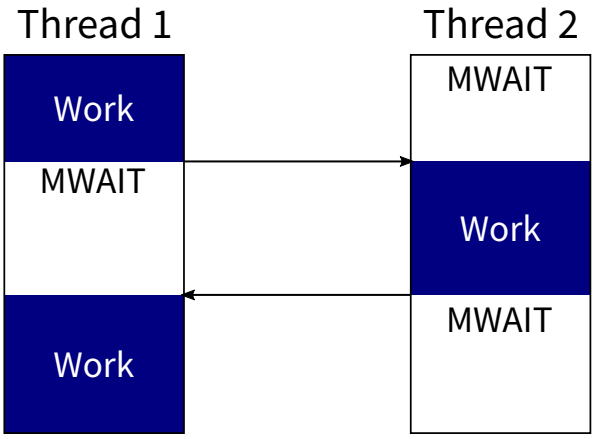


Figure 3.2: Migration based on `monitor` and `mwait` between two threads.

segment to get better memory/compute separation, or are the extra core migrations more detrimental to performance?

Step 3 produces a control flow graph with performance counter results on each edge. The developer can use this graph to find very long or very short code sections and gets an overview of the metric used for scheduling. In Section 5.3.3, we analyze the performance cost of our migration technique, which is important when considering code section length.

Online Monitoring

Our design only allows for offline application analysis, so the system cannot react to different application input. Ideally, the system would monitor performance counters online and decide on future core switching based on current execution characteristics. However, setting up and reading performance counters requires access to the CPU's model specific registers (MSR) on x86. The CPU instructions `rdmsr` and `wrmsr` are privileged, so each access needs potentially multiple system calls. Future work may be able to reduce this overhead by using the `rdpmc` instruction which allows read-only access to performance counters from user space.

3.3 User-mode core migration

In this section, we describe our technique for migrating an application between CPU cores. Low-overhead migrations allow us to exploit phase behavior in applications at small scales. Our technique builds upon the x86 `monitor` and `mwait` instructions which we introduced in Section 2.2.

The user-mode core migration works like this: During application initialization, the migration library creates two kernel-level threads, pinning one to a "fast" (high frequency) CPU core and the other to a "slow" (low frequency) core. One of these threads then resumes executing application code, while the other sleeps via `mwait`. Figure 3.2 illustrates the migration process: When the application asks for migration, a write to the monitored address wakes up the sleeping thread. The woken-up thread takes over the application stack and resumes execution, while the other thread puts itself to sleep.

Even though one of the threads is idle at any time, the operating system always sees two active threads. A system using user space core migration thus may need a special OS scheduler that can distinguish the threads to prefer preempting the idling thread over the working one.

The two CPU cores running the application need a shared cache. Otherwise, memory accesses would be very costly after migration due to cache coherency protocols or cache misses. On our AMD Ryzen test system, this means migrating within caches of only one CCX. We analyze the cost of cross-CCX migrations in Section 5.3.3.

Chapter 4

Implementation

This work aims to develop a system for reducing power consumption of applications on asymmetric processors. In the previous chapter, we presented our design: We analyze the application with performance counters at measurement points defined by the developer and create an application profile which determines core assignments for each code section. At runtime, we migrate between a fast and a slow core in user space.

To evaluate this approach, we now introduce a prototype implementation that we published on GitHub [35]. Figure 4.1 shows an overview of artifacts involved in our implementation:

1. After inserting measurement points, the developer compiles the application once.

2. Linked with our *libswp* library, the binary will execute with performance counter monitoring, producing a control flow graph.

3. A script then processes the graph into an application profile that maps measurement points to aggregated performance counter values.

4. The application user decides on a threshold value, thus assigning each code section to either a fast or a slow core.

5. Finally, the application user links the program binary with *libswp_migrate* and *libultmigration* and executes it on the asymmetric processor, providing the profile and the threshold value. At each measurement point, *libswp_migrate* compares the measurement point value from the profile with the threshold given by the user and calls *libultmigration* to perform migration if necessary.

In the following, we describe these libraries and associated scripts in detail. The library *libswp* provides performance counter monitoring between measurement

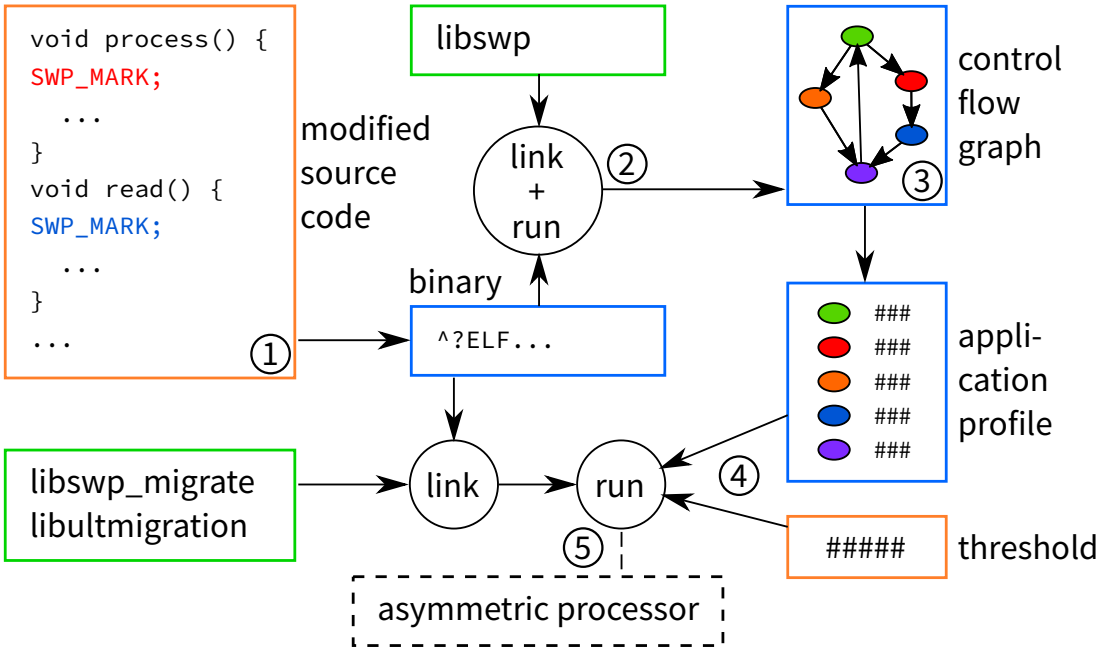


Figure 4.1: Overview of artifacts involved in our implementation. Orange artifacts are provided by the user, blue artifacts are generated, and our libraries are green. The circled numbers correspond to the conceptual steps from Figure 3.1.

points. *libswp_migrate* is the runtime pendant that decides on core assignments. In Section 4.2, we describe the *libultmigration* library which implements usermode core migration. Finally, we need a hardware platform with an asymmetric processor to evaluate our system. We explain our asymmetric AMD Ryzen processor configuration in Section 4.3.

4.1 libswp: Migration based on program flow analysis

For our program flow analysis, we implemented two libraries with a common C API. In Figure 4.1, the user links with these libraries in steps (2) and (5). Note that it is not necessary to recompile the application binary for each step; the libraries are ABI-compatible, allowing the user to swap them out by configuring the dynamic linker (e.g., using the `LD_PRELOAD` environment variable). The *libswp* library used in step (2) does performance counter measurements and the `libswp_migrate` library used in step (5) does migration based on a profile. For overhead testing, we implemented another library, *libswp_dummy*, that contains just the function stubs.

function	*libswp* (step (2))	*libswp_migrate* (step (5))
`swp_init()`	initialize performance counters	initialize *libultmigration*
`SWP_MARK`	collect performance counters for previous code section	read value from profile, perform core migration
`swp_deinit()`	print control flow graph, disable performance counters	deinitialize *libultmigration*

Table 4.1: Description of functions in *libswp* and *libswp_migrate*. The steps refer to Figure 4.1.

Three functions make up the API. Table 4.1 contains a summary of their functionality in *libswp* and *libswp_migrate*. The functions `swp_init()` and `swp_deinit` perform initialization and deinitialization in both libraries. In server software, the developer will typically insert calls to these functions before and after the main event loop.

The macro `SWP_MARK` specifies a measurement or migration point. It identifies the point using the current C or C++ function name combined with the current source file position. This combination yields unique names that are also easy to understand for the developer. By naming points automatically, `libswp` relieves the developer from having to invent a unique name for each measurement point.

On each call of `SWP_MARK`, the `libswp` library collects performance counter results for the previous code section in a map. It identifies each code section via the names of the start and end measurement points. For code sections called more than once, the library sums the counter values and keeps a call count. Finally, it prints results in a simple text format to the standard output on deinitialization. All measurements combined form a control flow graph with measurement points as nodes and performance counter results on the edges. For analysis of the results, we visualize that graph using graphviz [13] (see Figure 5.1 on page 31 in the evaluation for an example). With the graph, the developer can do an initial assessment of measurement point placement: Are there unexpected edges? Are there very short or very long code sections? Are there any clearly compute-bound or clearly memory bound code sections? Do diverse sections share a common starting node? Thus, the graph allows quick iteration on measurement point placement without having to run benchmarks with *libswp_migrate* while monitoring power.

As preparation for *libswp_migrate*, a script processes the graph to the application profile, a simple map that translates a measurement point name to a single counter value. When arriving at a measurement point, the library cannot know which of the possible following code paths will be taken. This is why we compute the average for all outgoing edges of a node, weighted by the number of calls of that

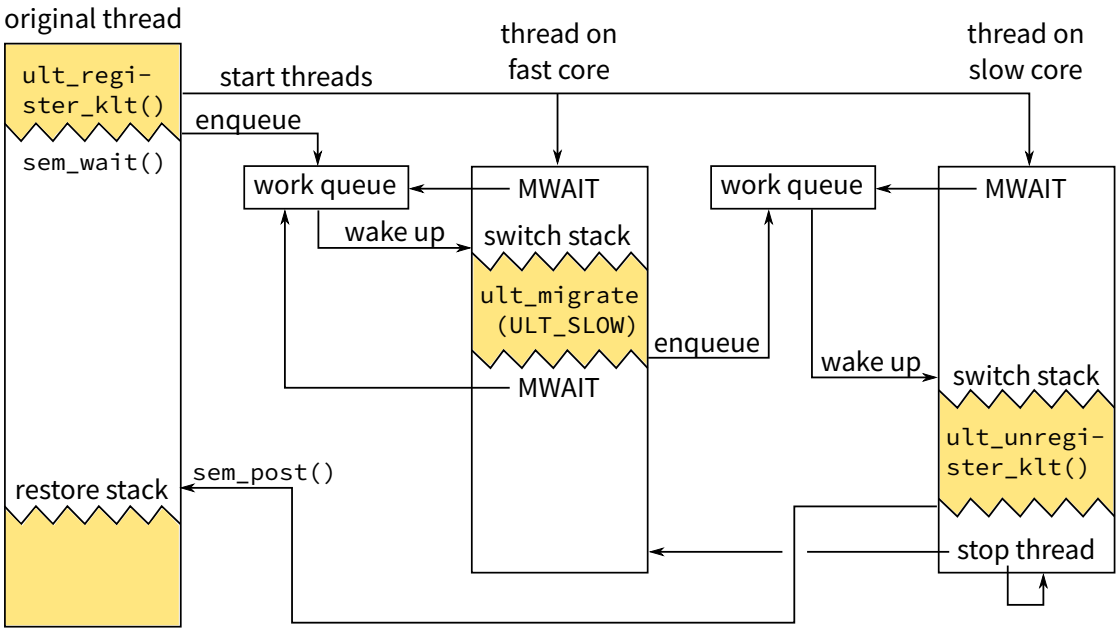


Figure 4.2: Visualization of *libultmigration* functionality. Time flows from top to bottom. Threads executing user code have a yellow background. The user code executes `ult_register_klt(); ult_migrate(ULT_SLOW); ult_unregister_klt();`.

code path. The resulting value for a start node with code paths 1 to n which are called $calls_i$ times and have the performance counter result $counter_i$ is calculated as follows:

$$value = \frac{\sum_{i=1}^{n} calls_i \cdot counter_i}{\sum_{i=1}^{n} calls_i}$$

If the code paths following a node are too diverse, this approach will not yield good results and the developer will have to add additional measurement points.

To use *libswp_migrate*, the application user sets the environment variable `SWP_CFG` to the path of a file with the resulting application profile. Additionally, the user selects a threshold value in `SWP_THRESHOLD`. On initialization, the library then parses the profile. For each measurement point in the profile, if the cache miss rate is larger than the threshold, the library calls *libultmigrate* to migrate to the slow core. Otherwise, it migrates to the large core.

4.2 libultmigration: User-mode core migration

In Section 3.3, we described a mechanism for fast usermode core migration based on `monitor` and `mwait`. We used a preexisting implementation of this concept, which we explain in the following.

Figure 4.2 visualizes a simple example that calls *libultmigration*'s three API functions in sequence. The initialization function `ult_register_klt()` starts two threads via pthreads and pins one to the fast CPU core and the other to the slow core. The user passes the CPU indices of these cores as environment variables `FAST_CPU` and `SLOW_CPU`. The library then pushes all callee-saved registers on the stack, switches to a new, empty stack, and enqueues the original stack on the first thread's work queue. The original thread then sleeps, waiting on a semaphore.

The two pinned threads start up in a loop that waits for entries in their respective work queues. Each work queue can hold a single pointer to a user stack. By using `monitor` and `mwait` (see Section 2.2) on the work queue, the CPU cores can idle, but wake up once another thread enqueues work. After waking up, a thread switches to the enqueued stack, pops all callee-saved registers, and returns from the library function call. Hence, the user code continues execution on the new core.

In the example in Figure 4.2, the user code calls `ult_migrate(ULT_SLOW)` next, initiating a core migration. The user selects the target core via the constants `ULT_FAST` and `ULT_SLOW`; selecting the currently active core does nothing. Once again, *libultmigration* pushes all callee-saved registers on the user stack and switches back to the thread's own stack. It then enqueues the user stack in the target work queue.

Finally, the function `ult_unregister_klt()` reverts back to a single thread: the worker thread saves registers, switches to its own thread, signals the original thread's semaphore, and stops the other thread and itself. The original thread restores the user stack and returns control to the user.

We also implemented a dummy library that, when loaded via `LD_PRELOAD`, allows running a program without migration. We use this library for power and performance comparisons in our evaluation.

4.3 Making an asymmetric processor

AMD Ryzen processors offer up to six configurable P-states. A P-state is a combination of frequency and voltage at which the processor can be operated. They are commonly used for overclocking by increasing the frequency and voltage of the highest P-state P0 [1]. Kernel-mode code can configure the P-states by writing to model-specific registers (MSR) [3]. We developed a tool that can read

and write this P-state configuration. It relies on the `msr` Linux kernel module [21] for accessing model-specific registers from user space.

Although each core has an individual set of these model-specific registers, the processor does not allow configuring P-states differently on each core. Consequently, we always set identical P-state configurations on all cores. For our asymmetric configuration, we set one P-state to the lowest possible frequency of 833 MHz. The highest P-state remains at the base clock of 3600 MHz. We find the high frequency to restrict lower frequencies on the same CCX; for example, with a base frequency of 3600 MHz and a configured low frequency of 833 MHz, the slow core runs with an actual frequency of 1400 MHz.

During normal execution, the operating system periodically selects a P-state for each core via ACPI based on current system load. Linux implements the selection mechanism in the *acpi_cpufreq* driver; "cpufreq governors" provide the selection policy. Instead of reacting to system load, the "userspace" governor allows setting a fixed P-state via files in sysfs. Note that due to the generic nature of the driver, the sysfs interface works with frequency values instead of P-state numbers. These values correspond to the P-state settings at kernel startup time.

We found that the actual processor frequency as measured by APERF/MPERF [3] depends on the processor topology. Ryzen processors implement simultaneous multithreading (SMT). Each core exposes two hardware threads to the OS, both of which share a single P-state. On a higher level, the processor consists of two core complexes (CCX, see Section 2.3.2 for an overview). When setting different P-states for cores of the same CCX, the lower-bound frequency appears to rise with the highest active frequency. With our frequency configuration of 3600 MHz and 833 MHz on one CCX, the slower core actually runs at a frequency of approximately 1440 MHz.

Although there are no such restrictions when setting P-states across core complexes, we observed that application performance on one CCX changes with the highest active P-state on the other CCX. We suspect cache coherency protocols between the CCX's L3 caches to be the cause here, as these caches likely adjust their speed to the CCX's fastest core. To avoid interference from this effect, we always run tests on only one CCX and set the other's cores to the lowest P-state.

Chapter 5

Evaluation

The overall goal of this work is to use fast core migration on an asymmetric multicore processor in order to save energy. To reach this goal, we apply *efficiency specialization* by migrating CPU-heavy code to the fast CPU core and memory-heavy code to the slow core (see Chapter 3). In the previous chapter, we introduced a prototype for analyzing applications, identifying memory-heavy and CPU-heavy code sections, and for performing fast core migration in user space. Additionally, we described an asymmetric core configuration for AMD Ryzen processors to serve as a test bed. We now put this prototype to the test.

In our experiments, we say our migration technique is successful regarding our goal if after applying it, an application consumes less power on average than when executed on a single core with a performance-equivalent constant frequency (i.e, a frequency with which the application finishes within the same duration). We are targeting server software where latency is often critical, so sacrificing performance for energy consumption is only possible as long as the software does not miss its latency deadlines (see the discussion about *energy scaling* in Section 2.1). Therefore, our metric is more useful than overall energy consumption: If our migration technique is successful, it is an improvement at that performance level, irrespective of lower constant frequencies that may not meet the latency deadlines.

The application analysis library *libswp* we designed and implemented in the previous chapters requires manual work from developers. To show that the manual time and effort is reasonable, we walk through the steps necessary to adapt a large real-world application, namely MySQL Server. Along the way, we show the artifacts our system produces (see Figure 4.1 for an overview).

Finally, we use microbenchmarks to analyze smaller parts of the system. We validate detection of memory- and CPU-heavy code sections, verify that *efficiency specialization* is applicable on our platform, and measure overhead of our migration technique.

CCX	CCX 0			CCX 1		
Core	1	2	3	4	5	6
P-state	P0	P2	P2	P2	P2	P2
Target Voltage (V)	1.2	0.8	0.8	0.8	0.8	0.8
Target Frequency (MHz)	3600	833	833	833	833	833
Effective Frequency (MHz)	3600	1440	1440	833	833	833

Table 5.1: Overview of our asymmetric core configuration on an AMD Ryzen 1600X processor with six cores.

5.1 Test Setup

Our test platform is an AMD Ryzen 1600X processor. It has six cores, so there is one disabled core per core complex (CCX, see Section 2.3.2). We set the first core to P-state P0 at the base frequency 3600 MHz and all other cores to P2 at the lowest possible frequency 833 MHz. See Section 4.3 for details on asymmetric Ryzen processor configuration. Table 5.1 provides an overview of frequencies and voltages. Even though we set the same P-state on all cores except the first, we get different actual frequencies on the two core complexes (effective frequencies are measured with APERF and MPERF [3]): On the CCX with the first core set to P0, the other cores clock at 1440 MHz instead of 833 MHz. As we have to do migration within one CCX due to Ryzen's cache organization (see Section 2.3.2), we cannot avoid the higher frequencies. Unfortunately, AMD does not document any CPU core voltage sensors, so we cannot verify the per-core voltages.

For power monitoring, we have two alternatives. First, we measure overall current at the wall socket with a multimeter. Second, we use the CPU's built-in running average power limit (RAPL) counters. Ryzen CPUs include one RAPL counter for each core as well as a counter for the whole package [3]. We are only interested in the CPU power consumption, but the multimeter measures the whole system, so unrelated components may interfere with our measurement. The RAPL counters only incorporate CPU power. However, AMD does not publish any information about their accuracy and the data basis, for example whether the data is produced by on-chip current sensors or an estimation based on energy event counters [8]. To avoid these issues, we use the following methodology: As part of each benchmark run, we measure power consumption with the multimeter while the CPU is idle. During the benchmark workloads, we record approximately one reading per second of both multimeter and RAPL data. We then calculate the average of all values read during one benchmark run and subtract the idle power consumption from the multimeter data. Finally, we compare the resulting multimeter value with the per-core RAPL readings. We find that the multimeter and RAPL results

correlate closely. The multimeter values are usually a bit higher than the RAPL values, which is consistent with previous research on Intel's implementation of RAPL counters [15]. We see more noise in the multimeter data which we smooth out by repeating the benchmarks.

5.2 Adapting an Application: MySQL

We now describe the concrete steps necessary to adapt a large real-world application to use our system for single-threaded scheduling on asymmetric processors. We want to demonstrate that the manual modifications our prototype requires are reasonable even without prior knowledge of the code base. Finally, we compare the resulting performance and power consumption with and without migration.

We chose the MySQL database [22] in version 5.7 as example application for the following reasons. MySQL is a widely-used open-source SQL database server and has a typical client-server architecture: Clients connect to the database via network sockets to send SQL requests. MySQL parses, optimizes and executes the requests and answers with the results. MySQL has a thread pool to handle concurrent client connections. As our prototype only works with single-threaded applications, we limit MySQL to maximum one connection at a time.

As explained in Section 3.2 and Section 4.1, applying our prototype requires five steps:

1. Add measurement points to the source code.
2. Execute a benchmark and collect performance counter results.
3. Create control flow graph from performance counter data.
4. Process the graph into an application profile.
5. Run the application in "migration mode" using the profile.

In the following, we show the results from each of these steps.

5.2.1 Source Code Modification

As first step, we have to modify the application source code to link with our library, perform initialization and cleanup, and to add measurement points. Listing 5.1 shows a summary of these changes. Our changes are very minor: For inserting seven measurement points, we added 21 lines of code to the MySQL source files. For each file, we had to add a preprocessor include to libswp's header file, and inserted `SWP_MARK` macro calls to interesting functions. We identified these functions with

```
libmysqld/CMakeLists.txt | 1 +        - Adding libswp to the build
sql/CMakeLists.txt       | 3 ++-      - system
sql/mysqld.cc            | 6 ++++++   # Initialization and cleanup
sql/sql_class.cc         | 3 +++      -
sql/sql_insert.cc        | 4 ++++     |
sql/sql_parse.cc         | 7 +++++++  | Adding measurement points
sql/sql_select.cc        | 4 ++++     |
sql/sql_update.cc        | 3 +++      -
9 files changed, 31 insertions(+), 1 deletion(-)
```

Listing 5.1: Annotated summary of changes to the MySQL source code. The numbers refer to lines of code.

basic knowledge about SQL: We separated request processing from different types of SQL query execution and iterated on the exact measurement point placement using the resulting control flow graph.

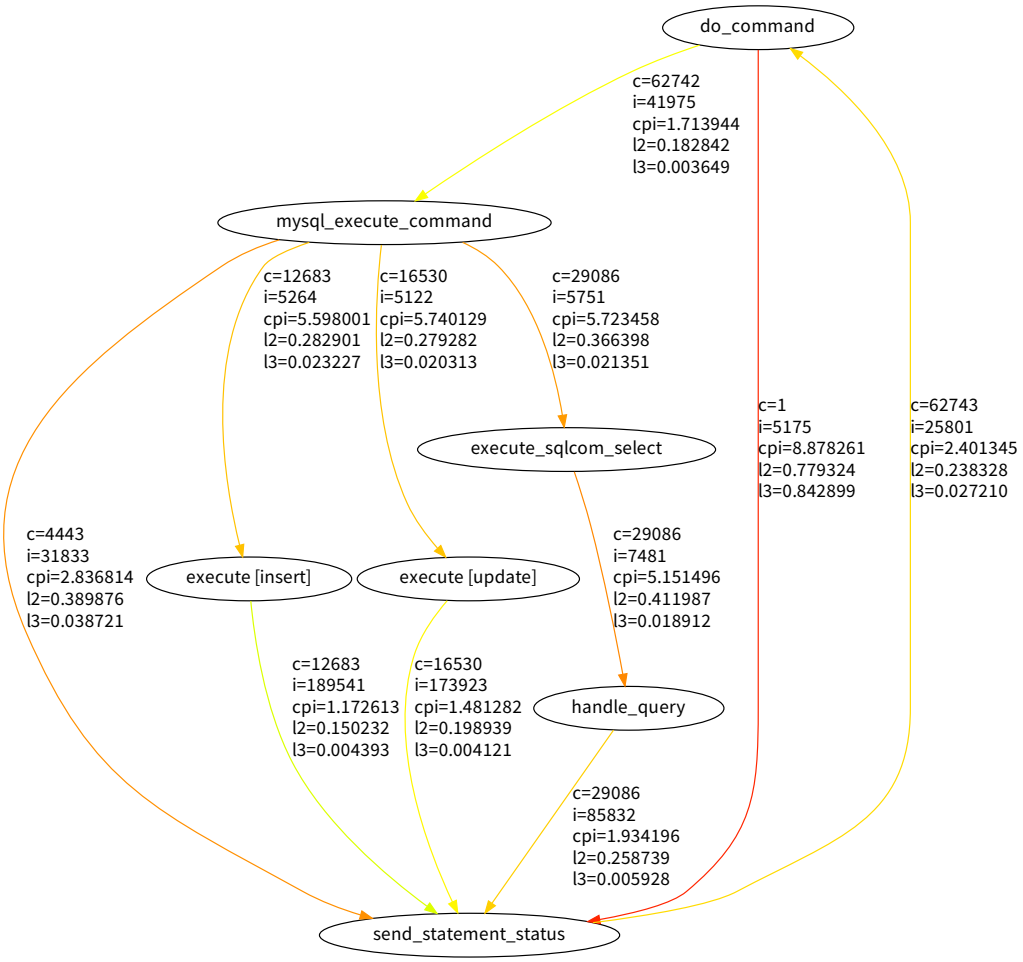
The remaining 10 new lines listed in Listing 5.1 are in the build system and in initialization functions. MySQL uses the CMake build system [9], so we had to modify its `CMakeLists.txt` build definition files to link with *libswp*. The C++ source file `sql/mysqld.cc` contains MySQL's `main` function. We added calls to `swp_init()` and `swp_deinit()` there. See Section 4.1 for a description of these functions.

In summary, the manual source code modifications do not require much work from the developer and are reasonable even without prior knowledge of the code base.

5.2.2 Performance Counter Monitoring

To obtain an application profile, we run the database and execute the TPC-C benchmark from OLTP-Bench [11]. We only monitor the actual benchmark execution and perform benchmark initialization with an unmodified MySQL instance.

When shutting down, our library prints the performance counter results to the console. By processing the results with graphviz [13], we obtain the control flow diagram shown in Figure 5.1. Using the graph, we do a first assessment of our measurement point placement. There is only one edge with a very high amount of cache misses, but that edge from `do_command` to `send_statement_status` has only a single call. The `mysql_execute_command` function leads to multiple measurement points, but those edges all have similar cache miss rates. These edges are the shortest, with roughly 5000 instructions.



Edge annotations	
c	Number of calls
i	Number of instructions (per call)
cpi	Cycles per instruction
l2	Number of L2 cache misses per instruction
l3	Number of L3 cache misses per instruction

Figure 5.1: Slightly simplified MySQL control flow diagram from running the TPC-C benchmark. Edges are colored by L2 miss rate.

```
"mysql_execute_command"   0.0226868
"send_statement_status"   0.02721
"execute [insert]"        0.004393
"handle_query"            0.005928
"do_command"              0.00366238
"execute_sqlcom_select"   0.018912
"execute [update]"        0.004121
```

Listing 5.2: Profile file used for migration. The values are based on L3 cache miss rates.

Type	Power	Transactions per second
Constant 3.6 GHz	5.6 W	197
Migration	4.7 W	131
Constant 1.4 GHz	3.9 W	135

Table 5.2: Results from running the TPC-C benchmark on MySQL. Migration performs worse than the constant lower frequency in both power consumption and performance.

5.2.3 Application Profile

To create the application profile for MySQL, we take a weighted average of all outgoing cache miss rates for each node (see Section 4.1). In this case, we are using L3 cache miss rates; the L2 cache miss rates would yield similar results. Listing 5.2 shows the resulting profile for MySQL.

In order to perform migration, we have to decide on a threshold value. Figure 5.2 visualizes the profile values. There is a clear gap in values between roughly 0.02 and 0.005. With a threshold value of 0.01, three nodes execute on the slow core and the other four on the fast core.

5.2.4 Migration Results

We then run the TPC-C benchmark with `libswp_migrate` in migration mode. We record the benchmark metric, transactions per second, as well as the average power consumption during benchmark execution. We configure the processor as in Table 5.1: The first core clocks at 3.6 GHz and is the fast core, the second core with 1.4 GHz is the slow core. For comparison, we also execute the benchmark without migration using `libswp_dummy` on both the slow and the fast core with the same processor configuration.

Table 5.2 shows the results. We can see that our migration technique is not successful: Whereas power consumption with migration lies between the two con-

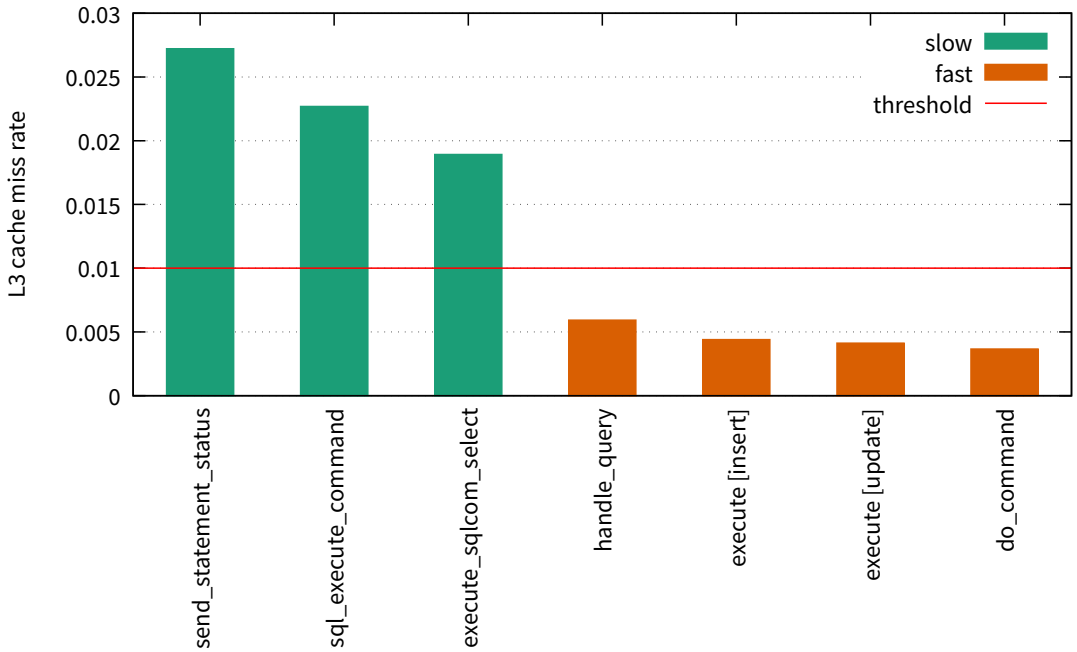


Figure 5.2: Profile values based on L3 cache miss rates from Listing 5.2, partitioned with a threshold value of 0.01.

stant frequencies, the performance is worse than with the constant lower frequency. We would have expected to see MySQL with migration getting performance results somewhere between the two cases with constant frequency, but with lower power than when running MySQL at a performance-equivalent constant frequency.

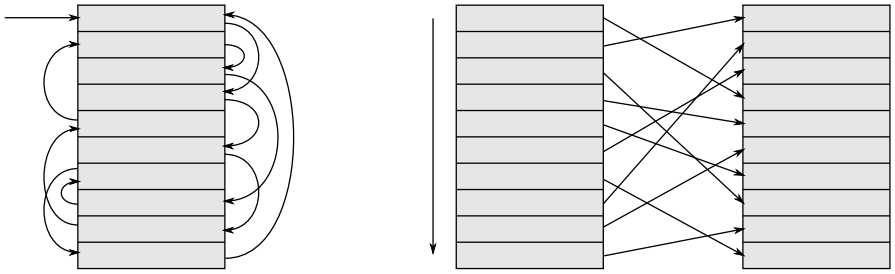
What does the loss of speed result from? Why does the power consumption increase despite lower performance? In the following, we evaluate our prototype at finer granularities with the help of a microbenchmark to obtain a better understanding of the issues.

5.3 Microbenchmark

Why does the migration technique not improve power consumption? Here are a couple of possible root-causes.

1. CPI of memory-heavy code may not actually improve when migrating to the slower core on our test system.

2. The cache miss counters may not be effective at finding memory-heavy code where CPI improves.



(a) Pointer Chasing (b) Indirect Access

Figure 5.3: Illustration of the two memory-heavy functions in the microbenchmark.

3. Migration overhead may be too high.

4. Our test system may impose a high power overhead on asymmetric configurations and our migration technique.

5. On our test system, migration may not be better than a fixed frequency even for an optimal workload.

We now construct a microbenchmark to check these hypotheses. Our microbenchmark models the optimal case for the scheduling technique, as our intention is to find the maximum benefit our prototype can achieve. We construct a CPU-heavy and two memory-heavy functions. By mixing these workloads, we can also simulate less optimal cases. The CPU-heavy function executes integer- and floating point instructions in a loop. It works with registers only and does not touch the memory.

We implemented two variants with different memory access patterns for the memory-heavy function. Figure 5.3 shows an illustration of these functions. For the *pointer chasing* function, we fill a large buffer with consecutive pointers, then shuffle the pointers. The benchmark then continuously follows the pointers through the buffer. This construction prevents cache prefetching completely, as each memory access depends directly on the preceding one and the buffer is too large to fit completely into the cache. The *indirect access* function works with two buffers. The first buffer contains a list of randomly generated, but unique pointers into the second buffer. The benchmark walks sequentially through the first buffer, dereferencing each pointer into the second buffer. This access pattern allows the prefetcher to load the sequentially accessed pointers in the first buffer. Due to the size of the second buffer, dereferencing the pointers will produce cache misses.

The microbenchmark alternately calls the CPU-heavy function and one of the memory-heavy functions, simulating two code sections with differing characteristics. Between these functions, the microbenchmark calls *libultmigration*'s migration

function directly. It migrates to the fast core before calling the CPU-heavy function and to the slow core before calling one of the memory-heavy functions. We chose a constant overall amount of operations so that the microbenchmark runs for roughly 20 to 40 seconds, depending on the frequency.

We do not expect real programs to have as extreme behavior as the microbenchmark. To approximate real programs, we implemented mixed benchmark functions. These functions interleave calls to the CPU and memory functions with a given ratio. For example, a *memory ratio* of 70% means that 70% of the usual memory access iterations are mixed with 30% of iterations of the CPU function. Thus, the benchmark still runs for a similar amount of time.

5.3.1 CPI

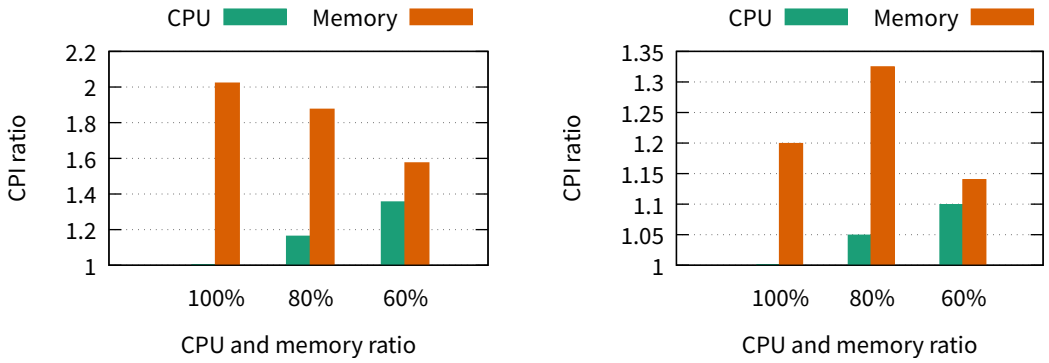
The core assumption behind *efficiency specialization* on frequency-based asymmetric processors is that memory-heavy code executes more efficiently at lower frequencies (see Section 3.1). In this section, we verify this assumption by comparing *cycles per instruction* (CPI) of our microbenchmark on the fast and slow cores. For memory-heavy code, we expect to see a lower CPI value on the slow core, as CPU stalls from memory accesses take a constant amount of time whereas cycle duration increases. We run the benchmark twice, once on the slow core and once on the fast core. For each section, we monitor instructions retired and CPU cycles with performance counters. In addition to our pure CPU and memory functions (100%), we also compare mixed functions at 80% and 60%.

In Figure 5.4, we plot the results as $CPI_{ratio} = \frac{CPI_{fast}}{CPI_{slow}}$. For the pure CPU workload (100%), there are no memory accesses and thus there is no CPI difference at all ($\frac{CPI_{fast}}{CPI_{slow}} = 1$). Mixing in memory accesses increases the CPI difference linearly for both our memory-heavy functions. The memory workload behaves similarly, with a larger CPI ratio than the CPU workload's at any mixing ratio.

In summary, CPI behaves as expected: memory-intensive code has a large CPI ratio, and CPU-intensive code has a small CPI ratio when comparing execution on fast and slow cores. This result confirms our assumptions behind *efficiency specialization* from Section 3.1.

5.3.2 Cache Miss Performance Counters

In the previous section, we ran the microbenchmark two times on different cores to compare CPI values. For our application characterization, we want to avoid having to run application benchmarks multiple times, so we monitor cache miss events instead. We expect code sections with a large amount of cache misses to also have a large CPI difference. We verify this expectation in this section.



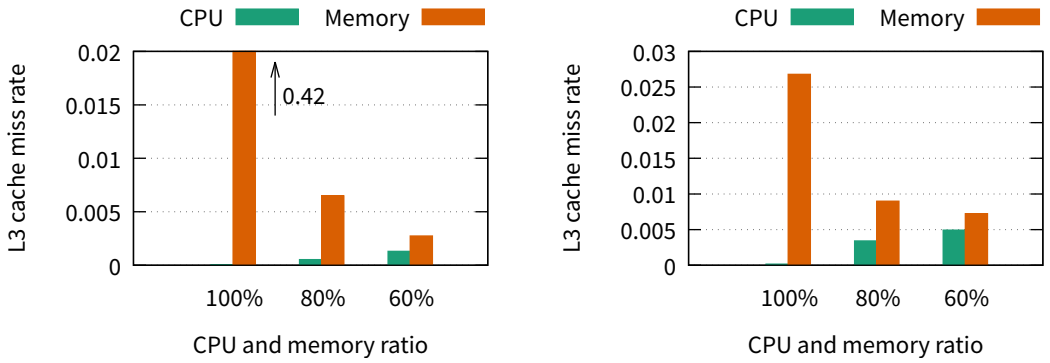
(a) Pointer Chasing (b) Indirect Access

Figure 5.4: Comparison of CPI ratios from running the microbenchmark on the fast and the slow cores, split between CPU function and memory function. At 100% mixing ratio, the CPU and memory functions are pure; at 80% and 60% they are mixed with each other at that ratio. The CPI ratio behaves as expected and is generally larger for the memory-heavy functions.

Figure 5.5 shows L3 cache misses per instruction for the microbenchmark's CPU and memory functions. Comparing with Figure 5.4, we can see that partitioning by L3 cache miss rate will yield similar results as by CPI rate. Even for the mixed cases, cache miss rates of the memory-heavy sections are larger than those of the CPU-heavy sections within one memory access pattern.

With both memory functions, the CPU-heavy sections give very similar results for CPI and cache miss rates. In contrast, the memory-heavy sections have some outliers. For *indirect access*, we see a rise in CPI ratio at 80% whereas the L3 cache miss rates only fall. For *pointer chasing*, both the CPI ratio and the cache miss rates fall consistently when mixing in CPU work. However, we measure a surprisingly high amount of cache misses at 100%. These differences are not problematic for our classification, as the miss rates for the memory-heavy sections are still larger than those of the CPU-heavy sections.

However, we believe that ultimately additional analysis of more diverse memory access patterns is necessary, as we observed some conflicting results with the mixed workloads when comparing our two memory access patterns. At 60%, the cache miss rate of *pointer chasing* is lower than the rate of *indirect access*, but the CPI ratio is larger. For our prototype, the L3 cache miss rate appears to sufficiently differentiate between memory-heavy and CPU-heavy code sections.



(a) Pointer Chasing (b) Indirect Access

Figure 5.5: L3 cache misses per instruction while running the microbenchmark, split between CPU function and memory function. Note that at 100%, the *pointer chasing* memory function's cache miss rate is off-graph. We can see that partitioning by L3 cache miss rates will correctly assign CPU and memory functions in all cases.

5.3.3 Migration Overhead

In the previous sections, we demonstrated that *libswp*'s method of identifying memory-heavy and CPU-heavy code sections is effective. As we found that our prototype is not successful at reducing power consumption of MySQL, we now analyze our migration method. We seek to find out how long a migration takes and how much influence this overhead has on our benchmarks. Additionally, we aim to confirm our previous assertion that a shared L3 cache is necessary and that migration across CCX is not viable.

We measure time per migration with a simple benchmark that continuously migrates between two cores and measures the overall time to do so. We test two processor configurations: The asymmetric configuration from Table 5.1 with cores in P-states P0 and P2, and a symmetric configuration in which all cores are in P0. In our asymmetric configuration, the cores in P2 have different effective frequencies on the two CCX. Thus, the symmetric configuration allows assessing the overhead of cross-CCX migration better.

Figure 5.6 shows the results. At P0, migrating across CCX is approximately three times slower than migrating within one CCX; with the asymmetric configuration, it is approximately two times slower. Note that we only measure overhead from *libultmigration* here. With real applications, there is additional overhead from remote cache accesses for application data.

Migrating within a CCX with the asymmetric configuration takes approximately 1.75 μs per migration. The microbenchmark migrates 10000 times, so there is a total cost of 17.5 ms for all migrations. At a total runtime of more than 20 seconds,

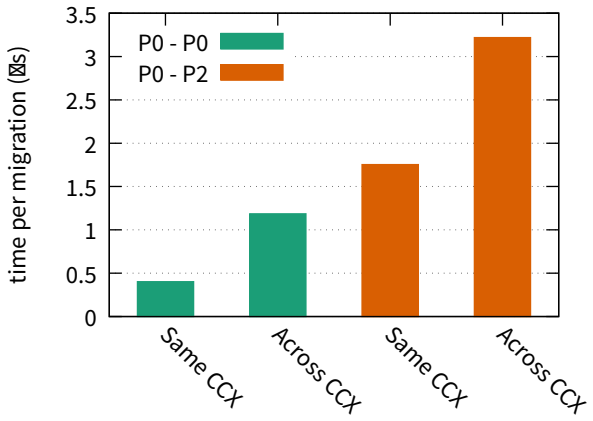


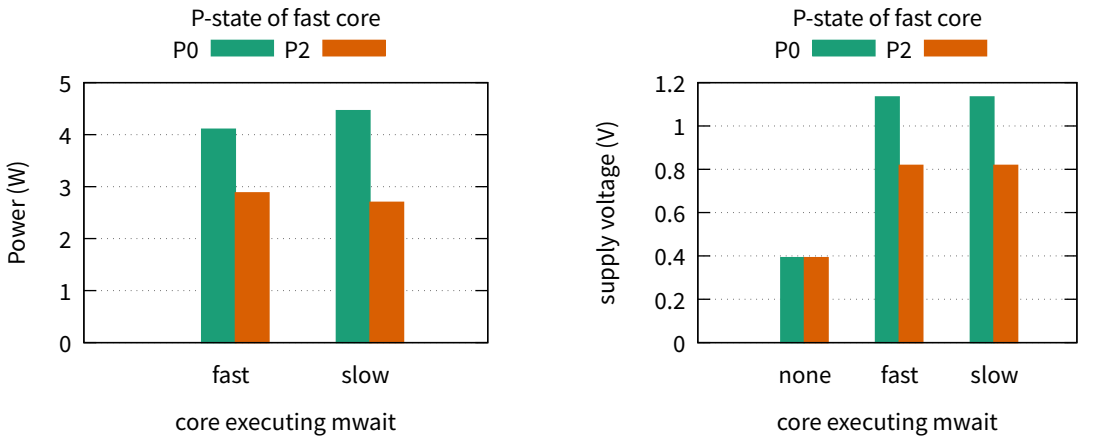
Figure 5.6: Time per migration between cores of the same CCX and across CCX at different P-states.

this cost is negligible. Our modified MySQL from Section 5.2 migrates four times per command. TPC-C executes approximately 27 commands per transaction and achieved 131 transactions per second with migration. Consequently, there are $4 \cdot 27 \cdot 131 \approx 14 \cdot 10^3$ migrations per second and for each second, we have a migration overhead of $14 \cdot 10^3 \cdot 1.75\mu s \approx 25ms$ which is 2.5%. Although not negligible, this overhead is not high enough to explain the bad result in Table 5.2. Even with 2.5% more transactions per second, MySQL with migration still performs worse than at constant 1.4 GHz.

5.3.4 Overhead From Ryzen as Asymmetric Processor

We use AMD Ryzen processors with different frequencies per core as our test platform. AMD did not design Ryzen to be an asymmetric processor, so the processor is not optimized for this use. In this section, we analyze the power and performance overhead from our use of Ryzen as an asymmetric processor. We measure the power consumption of user space `mwait`, monitor the voltage supply from the motherboard and calculate optimal performance with the microbenchmark.

The `monitor` and `mwait` instructions that our migration technique uses set the CPU to sleep until a cache line changes. In Section 2.2, we noted that although the `mwait` instruction takes a C-state number as argument, the processor most likely ignores that request when using `mwait` in user space. We now measure the additional power consumption from `mwait` while also monitoring motherboard sensors for the CPU supply voltage. We test two CPU configurations: the asymmetric configuration from Table 5.1 and a symmetric configuration with all cores in P-state P2. In Figure 5.7a, we can see that the power overhead depends only on the



(a) Power consumption

(b) Motherboard supply voltage

Figure 5.7: Power consumption and motherboard supply voltage while running `mwait` on an otherwise idle system.

P-state of the fast core and does not decrease when the slow core executes `mwait`. Figure 5.7b shows the minimum supply voltage from the motherboard during the experiment as well as on a fully idle system. We can see that the voltage only drops when the system is completely idle. As soon as any core executes `mwait`, the motherboard supplies the full voltage for the highest active P-state, even if the core with that P-state is idle. This behavior is not optimal for an asymmetric processor. Just as the motherboard reduces the supply voltage when the whole processor is idle, it should adjust the voltage only to the needs of the active cores.

For our work, the extra power consumption from `mwait` means that applications adapted with our prototype are at an inherent disadvantage compared to unmodified applications. If the application is idle at times, which is not uncommon in networked server software such as the MySQL database, the 4 W extra power consumption is significant.

To understand the overall overhead of migration and `mwait` on the Ryzen processor better, we now run the CPU-heavy and memory-heavy parts of the microbenchmark independently on the fast or slow core. The processor is configured as usual (see Table 5.1). We combine the results by adding the durations and taking the average of the power consumption weighted by duration of the individual parts. In Figure 5.8, we plot these combined results as well as normal invocations with migration. We show the graph for the pure workloads only, because the mixed workloads produce very similar results. There is only negligible difference in runtime between the independent and the migration-based runs, which confirms our result from Section 5.3.3. In contrast, we can see an increase of 2 W in power

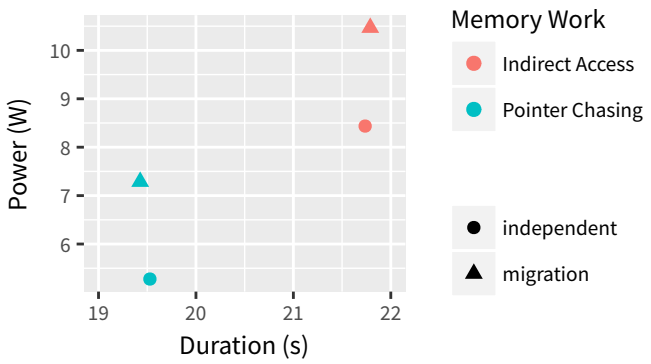


Figure 5.8: Comparison of microbenchmark running with migration and a synthetic result from running memory-heavy and CPU-heavy parts independently on the two cores. There is little performance overhead, but migration increases power consumption by 2 W.

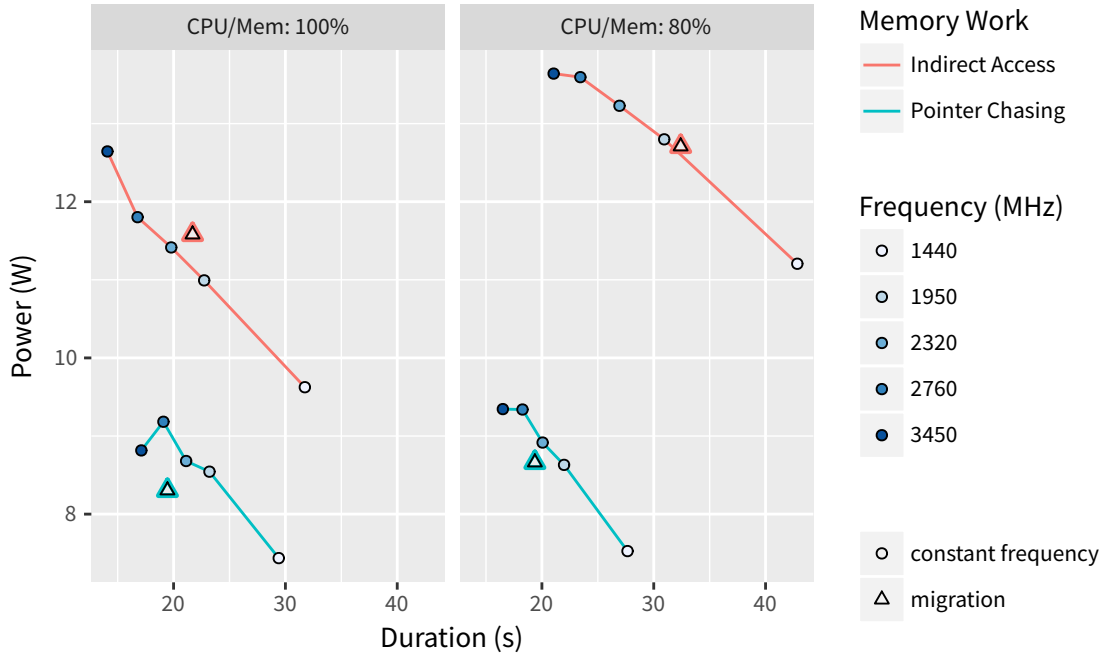
consumption for doing migration. This overhead is lower than what we have seen when comparing `mwait` with an idle system (see Figure 5.7a). We assume this difference stems from the fact that the CPU disables more components when all cores are idle, allowing the motherboard to reduce the supply voltage (see Figure 5.7b). In the benchmark here, there is still one active core when executing the parts independently, so the motherboard has to supply the full voltage.

In summary, Ryzen processors have deficiencies as asymmetric platform that lead to increased power consumption. This contributes to the bad result we have seen for MySQL: With the 2 W overhead we have observed for the microbenchmark removed, MySQL with migration would have a lower power consumption than at constant 1.4 GHz (compare Table 5.2).

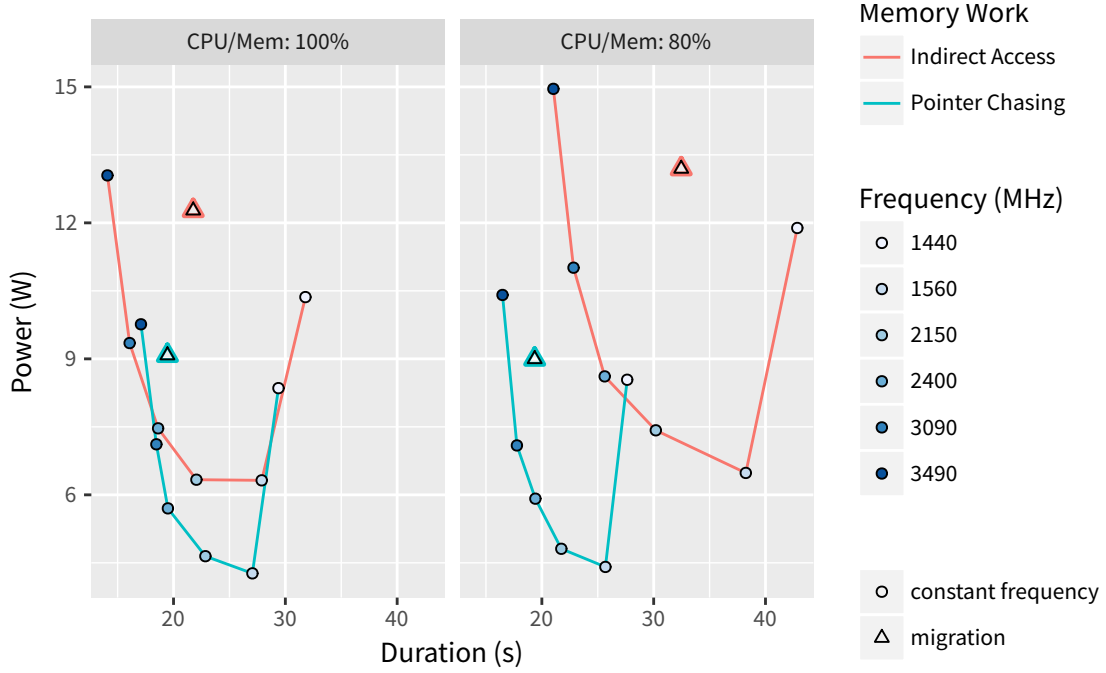
5.3.5 Comparison With Fixed Frequency

Is migration with the microbenchmark better than running at a constant frequency? In the following, we show that AMD Ryzen systems are completely unsuitable as asymmetric processors. To answer the question, we run the microbenchmark without migration at five different processor frequencies, and once with migration between 3.6 GHz and 1.4 GHz (our usual asymmetric processor configuration, see Table 5.1).

In the previous Section 5.3.4, we have seen that the Ryzen processor we use as our test platform always consumes additional power when using `mwait`. We now want to assess our prototype without this influence. To improve comparability, we thus execute `mwait` on another core on the same CCX during the constant-frequency benchmark runs.



(a) Symmetric setup for the constant-frequency points.



(b) Asymmetric setup for the constant-frequency points.

Figure 5.9: Comparison of migration (triangle) with different fixed frequencies (circle) in performance and power consumption.

CCX	CCX 0			CCX 1		
Core	1	2	3	4	5	6
Asymmetric P-state Setup	P0	**P1**	P2	P2	P2	P2
Symmetric P-state Setup	**P0**	P2	P2	P2	P2	P2

Table 5.3: P-state configuration with our two test setups. We vary the frequency of the bold P-state and execute the microbenchmark on that core.

AMD designed Ryzen to be a symmetric multicore processor. On such processors, the operating system usually sets high P-states only to boost active cores with high CPU utilization and sets idle cores to low P-states to save energy [23]. In contrast, our asymmetric processor configuration keeps one core at the highest P-state at all times. To test whether this deviation from intended usage has adverse effects, we use two different strategies for setting up the extra processor frequencies, illustrated in Table 5.3. For the asymmetric setup, we set core 2 to the otherwise unused P-state P1 whose frequency we vary. Thus, this setup has an inactive faster core and is still an asymmetric configuration similar to the one we use for migration. With the second symmetric setup, we start the benchmark on core 1 and change the frequency of P-state P0. This setup mirrors what an operating system would configure on symmetric multicore processors: The highest P-state for the active core where the benchmark runs and lower P-states for idling cores.

For both setups, we start with the usual asymmetric core configuration (see Table 5.1 and execute the microbenchmark with migration between core 1 and core 2. We then run the benchmark without migration on both of these cores, which produces the first and last points on the comparison line in the graph. Finally, we change the P-state configuration as described in the previous paragraph to test additional frequencies.

Figure 5.9 shows the results for the pure microbenchmark functions and mixed at 80%; in Appendix A, we include the complete results. The migration technique is successful if it uses less power than the constant frequency with the same performance. In the graph, this is the case if the triangle appears below the corresponding line. For the symmetric setup in Figure 5.9b, migration is never successful. On the contrary, we can see that migration is successful in the asymmetric setup in Figure 5.9a for *pointer chasing*; migration consumes slightly less power than with the performance-equivalent constant frequency. In contrast, *indirect access* consumes slightly more power with migration than without.

The processor's power consumption changes differently with frequency in our two setups. In the asymmetric setup, power decreases linearly with frequency, whereas in the symmetric setup, it decreases cubically. Additionally, the lowest-frequency point in Figure 5.9b has unusually high power consumption. This point

originates from running the microbenchmark on core 2 in the migration setup. These results make evident that our Ryzen processor does not actually regulate voltage per core (see Section 2.3.2). As long as there is a core with higher voltage needs, as in the asymmetric setup or the lowest-frequency point in the migration setup, all cores are supplied with the higher voltage and thus consume more power. Only when reducing the P0 frequency, the processor also reduces the voltage and allows higher power savings.

The results here show that our migration technique is not viable on AMD Ryzen processors. We need—at the same time—both a core with high voltage to support high frequencies on the fast core and another with low voltage to reduce power consumption on the slow core. Without independent voltage domains, reducing power consumption using migration on asymmetric processor setups is impossible.

5.4 Discussion

We have adapted the MySQL database with our prototype. We observed that adapting complex applications is feasible despite the need of manual source code modifications. However, we found that the adapted MySQL uses more power while running slower than without migration. We then analyzed our system at smaller scales using a microbenchmark. We successfully confirmed assumptions we made in our design: executing memory-heavy code on a core clocked at a lower frequency improves cycles per instruction compared to a high-frequency core, and we can detect such code sections using cache miss performance counters. We measured both power and performance overhead of our migration technique and concluded that the performance loss is acceptable, but we observed high power consumption from having an additional active core (in our case, from `mwait`). Finally, we confirmed this result by comparing power consumption and runtime of the microbenchmark with and without migration, observing that the microbenchmark at best performs only slightly better with migration than without.

Thus, of the five hypotheses about why migration is not successful we proposed in Section 5.3, we refuted the first two about the CPI difference and the cache miss counters. This result shows that our application analysis works correctly and that the bad result is due to inefficient migration.

Investigating performance overhead from migration, the third hypothesis, we concluded that the raw delay per migration is sufficiently low. However, we only analyzed the migration itself there and not delays due to L1 and L2 cache misses in application code after migrating. As we observed the raw migration time across CCX—without a shared L3 cache—to be three times slower, the costs from L2 cache misses are likely significant as well. Consequently, developers adapting applications for migration need to make sure that code sections between migrations are always

long enough. For our adaption of MySQL, this would mean reducing the amount of measurement points.

So far, our evaluation did not find substantial issues with our prototype. Last, we analyzed the suitability of AMD Ryzen processors for our system (hypotheses four and five). We observed that our Ryzen processor does not appear to use its per-core low-dropout voltage regulators that are part of the "Zen" core design [31]. As the processor also handles user space `mwait` inefficiently with a power overhead of 2 W to 4 W, we conclude that Ryzen processors are not suitable as asymmetric processors, especially in combination with high-frequency core migration.

All in all, we believe that our approach to save energy on asymmetric processors needs more evaluation on another hardware platform. In sum, our results only prove that Ryzen processors are not viable for our technique, but do not confirm or refute the effectiveness of our approach and prototype implementation.

Chapter 6

Conclusion

Due to power constraints, modern processors can often power only parts of their chips at the same time, which leads to dark silicon. Asymmetric processors are a promising solution to this problem. By including CPU cores with different power and performance characteristics, an asymmetric processor can deliver higher performance per watt than a symmetric one. *Efficiency specialization* optimizes for this metric by scheduling code on the CPU cores that can execute the code most efficiently. In this work, we designed and implemented a method for doing *efficiency specialization* within a single application.

Existing approaches to scheduling on asymmetric processors are hard to apply to homogeneous server software, as they usually classify code at the granularity of threads. Our main contribution is the design and the implementation of a system that can utilize more fine-grained changes in application behavior found in this type of software. It consists of libraries for characterizing applications and for doing optimized, high-frequency CPU core migration to the optimal core in order to save energy. Using the MySQL database server as an example, we showed that a developer can use our library with moderate manual work to adapt real-world applications to execution on asymmetric processors in a way that optimizes overall cycles per instruction.

Our second contribution is the analysis of AMD Ryzen CPUs as asymmetric processors. We developed software for configuring the processor's P-states, allowing us to create an asymmetric core configuration based on frequency scaling. However, we observed that our core migration technique does not yield power savings on the Ryzen processor in general. Although—according to design documents [31]—each Ryzen core includes a voltage regulator, we found that the core with the highest active frequency and voltage on the CCX determines overall power consumption, even if that core is only idling. In comparison with a symmetric processor setup without migration that properly allows reducing core voltage, our method will therefore never save energy on Ryzen processors.

In summary, we believe that our application analysis and core migration method could effectively reduce power consumption, but we could not demonstrate this effect on AMD Ryzen due to unexpected inefficiencies in that processor's design.

6.1 Future Work

We found the AMD Ryzen processors we used in our evaluation to be unsuitable for our system. Consequently, re-evaluating the system on a more efficient asymmetric processor will likely yield new results. ARM's future "DynamIQ" [33] processors are a potential candidate. They combine cores with different microarchitectures and include a shared cache for all cores. As we have shown in Section 5.3.3, such a cache is crucial for acceptable migration performance.

In Chapter 3, we proposed extending our system with *energy scaling*. Currently, our library decides on core assignments using pre-collected cache miss rates as only metric. At runtime, the application developer could pass additional information such as current processing deadlines to the library. This information would allow choosing the slow core more often to further reduce power consumption at the cost of performance, as long as the application can still meet its deadlines. Additionally, the library could continuously collect performance counter events at runtime to detect changes in behavior due to different input data.

Our prototype can only analyze and perform migration with single-threaded applications. This limitation is not inherent to our design. Our application analysis library would need synchronization to collect performance counters across threads safely. The thread migration library would need extensions to work with more than two cores and to schedule the threads across these cores. To handle blocking I/O correctly, modifications to the kernel are necessary.

Developers applying our prototype need to work good migration points out manually. Previous works have shown that fully automatic application characterization is possible (e.g., phase-based tuning [32]). A similar approach may also work for our system, although we migrate between shorter code sections than phase-based tuning. A hybrid system that suggests migration points based on static analysis or memory access traces could also help the developer.

Appendix A

Complete Power Graphs

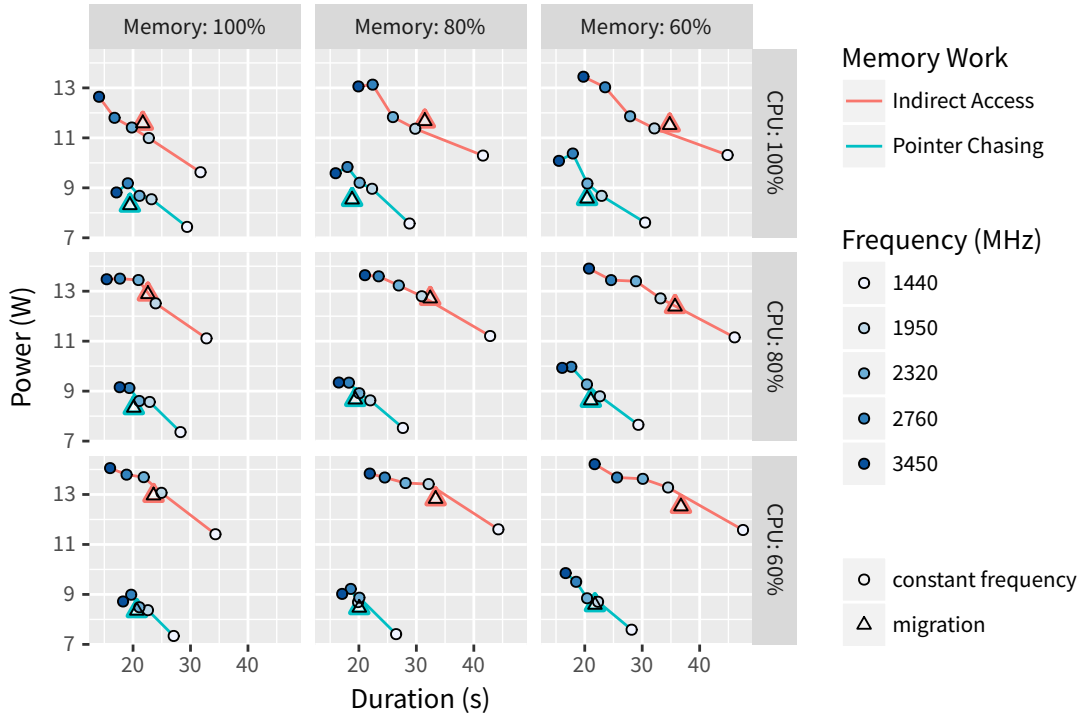


Figure A.1: Symmetric setup for the constant-frequency points.

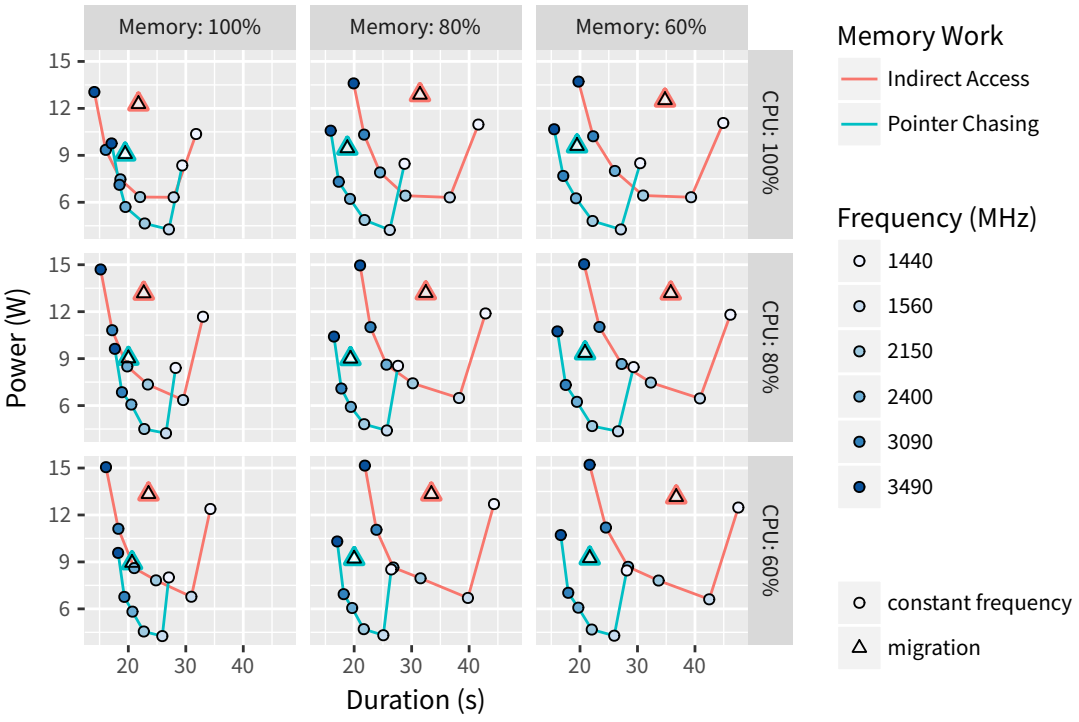


Figure A.2: Asymmetric setup for the constant-frequency points.

Bibliography

[1] AMD. *AMD Ryzen Master Overclocking User's Guide for AMD Ryzen and Ryzen Threadripper Processors*. 2018. URL: http://download.amd.com/documents/AMD-Ryzen-Processor-and-AMD-Ryzen-Master-Overclocking-Users-Guide.pdf.

[2] AMD. *AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions*. Dec. 2017.

[3] AMD. *Processor Programming Reference (PPR) for AMD Family 17h Model 01h, Revision B1 Processors*. Apr. 2017.

[4] AMD. *Software Optimization Guide for AMD Family 17h Processors*. June 2017.

[5] ARM. *ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile*. Dec. 2017.

[6] ARM. *Cortex-A15 MPCore Processor Technical Reference Manual*. 2013. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0438i/CACECFFA.html.

[7] ARM. *Cortex-A7 MPCore Technical Reference Manual*. 2013. URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0464f/BABDAHCE.html.

[8] Frank Bellosa. "The Benefits of Event-Driven Energy Accounting in Power-sensitive Systems." In: *Proceedings of the 9th Workshop on ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System*. EW 9. 2000, pp. 37–42.

[9] *CMake*. URL: https://cmake.org/.

[10] Benjamin Cownie. *Intel® Xeon Phi™ Product Family x200 (KNL) User mode (ring 3) MONITOR and MWAIT*. URL: https://software.intel.com/en-us/blogs/2016/10/06/intel-xeon-phi-product-family-x200-knl-user-mode-ring-3-monitor-and-mwait.

[11] Djellel Eddine Difallah et al. "OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases." In: *Proc. VLDB Endow.* 7.4 (Dec. 2013), pp. 277–288.

[12] Hadi Esmaeilzadeh et al. "Dark Silicon and the End of Multicore Scaling." In: *Proceedings of the 38th Annual International Symposium on Computer Architecture*. ISCA '11. 2011, pp. 365–376.

[13] Emden R. Gansner and Stephen C. North. "An open graph visualization system and its applications to software engineering." In: *SOFTWARE - PRACTICE AND EXPERIENCE* 30.11 (2000), pp. 1203–1233.

[14] Vishal Gupta and Ripal Nathuji. "Analyzing Performance Asymmetric Multicore Processors for Latency Sensitive Datacenter Applications." In: *Proceedings of the 2010 International Conference on Power Aware Computing and Systems*. HotPower'10. 2010.

[15] Marcus Hähnel et al. "Measuring Energy Consumption for Short Code Paths Using RAPL." In: *SIGMETRICS Perform. Eval. Rev.* 40.3 (Jan. 2012).

[16] Robert Hallock. *AMD Ryzen™ Community Update #3*. Apr. 2017. URL: https://community.amd.com/community/gaming/blog/2017/04/06/amd-ryzen-community-update-3.

[17] M. D. Hill and M. R. Marty. "Amdahl's Law in the Multicore Era." In: *Computer* 41.7 (July 2008), pp. 33–38.

[18] Alex Hutcheson and Vincent Natoli. *Memory Bound vs. Compute Bound: A Quantitative Study of Cache and Memory Bandwidth in High Performance Applications*. 2011.

[19] *LPC - EAS for Android*. Nov. 3, 2016. URL: https://linuxplumbersconf.com/2016/ocw//system/presentations/3693/original/LPC-%20EAS%20for%20Android.pdf.

[20] Anil Madhavapeddy et al. "Unikernels: Library Operating Systems for the Cloud." In: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '13. 2013, pp. 461–472.

[21] *msr(4) Linux Programmer's Manual*. Mar. 2009.

[22] *MySQL Server 5.7*. URL: https://github.com/mysql/mysql-server/tree/5.7.

[23] Venkatesh Pallipadi and Alexey Starikovskiy. "The Ondemand Governor." In: *Linux Symposium* 2 (2006).

[24] Krishna K. Rangan, Gu-Yeon Wei, and David Brooks. "Thread Motion: Fine-grained Power Management for Multi-core Systems." In: *SIGARCH Comput. Archit. News* 37.3 (June 2009).

[25] R. Rao, S. Vrudhula, and D. N. Rakhmatov. "Battery modeling for energy aware system design." In: *Computer* 36.12 (Dec. 2003), pp. 77–87.

[26] Miguel Rodrigues, Nuno Roma, and Pedro Tomas. "Fast and Scalable Thread Migration for Multi-core Architectures." In: *Proceedings of the 2015 IEEE 13th International Conference on Embedded and Ubiquitous Computing (EUC)*. EUC '15. 2015.

[27] Juan Carlos Saez et al. "A Comprehensive Scheduler for Asymmetric Multicore Systems." In: *Proceedings of the 5th European Conference on Computer Systems*. EuroSys '10. 2010.

[28] Samsung. *Mobile Processor Exynos 5 Octa (5430)*. URL: `http://www.samsung.com/semiconductor/minisite/exynos/products/mobileprocessor/exynos-5-octa-5430/`.

[29] A. Sembrant, D. Black-Schaffer, and E. Hagersten. "Phase behavior in serial and parallel applications." In: *2012 IEEE International Symposium on Workload Characterization (IISWC)*. Nov. 2012, pp. 47–58.

[30] Daniel Shelepov and Alexandra Fedorova. "Scheduling on Heterogeneous Multicore Processors Using Architectural Signatures." In: *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture* (2008).

[31] T. Singh et al. "Zen: An Energy-Efficient High-Performance x86 Core." In: *IEEE Journal of Solid-State Circuits* 53.1 (Jan. 2018), pp. 102–114.

[32] Tyler Sondag. "Phase-based tuning: better utilized performance asymmetric multicores." 2011.

[33] Govind Wathan. *Where does big.LITTLE fit in the world of DynamIQ?* URL: `https://community.arm.com/processors/b/blog/posts/where-does-big-little-fit-in-the-world-of-dynamiq`.

[34] Andreas Weissel and Frank Bellosa. "Process Cruise Control-Event-Driven Clock Scaling for Dynamic Power Management." In: *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES 2002)*. Grenoble, France, Oct. 2002.

[35] Lukas Werling. *AMP Scheduling Prototype*. URL: `https://github.com/lluchs/amp-scheduling`.