Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 161

Systems Group, Department of Computer Science, ETH Zurich

Explicit OS support for hardware threads

by

Andrei Poenaru

Supervised by

Prof. Timothy Roscoe
Reto Achermann
Gerd Zellweger

September 2016 - March 2017

Informatik
Computer Science

Abstract

Current mainstream processors provide multiple SMT (i.e., simultaneous multithreading) lanes on top of each physical core. These hardware threads share more resources (e.g., execution units and caches) when compared to CPU cores, but are managed by operating systems in the same way as if they were separate physical cores. This Thesis explores the interaction between hardware threads and proposes an extension to the Barrelfish OS, meant to improve the performance of a system by adequately handling SMT lanes. On an Intel Haswell CPU, with 2-way SMT via Hyper-Threading Technology, each SMT lane had 2/3 of the processing power that was yielded by the physical core with a single active hardware thread. The multi-HT CPU Driver (i.e., Barrelfish's microkernel) is able to modify the set of active hardware threads with an overhead in the order of thousands of processor cycles, which means that it can quickly adapt to the parallelism exhibited by the workload.

Contents

1 Introduction **4**
- 1.1 Motivation . . . 4
- 1.2 Thesis Structure . . . 5

2 Background **6**
- 2.1 Simultaneous Multithreading . . . 6
- 2.2 Intel® Hyper-Threading Technology . . . 6
- 2.3 The Barrelfish OS . . . 8
- 2.4 Synchronizing Hardware Threads . . . 10

3 Related Work **12**
- 3.1 Elfen Scheduling . . . 12
- 3.2 Asynchronous System Calls . . . 13
- 3.3 Exclusive Access to a Microkernel . . . 14

4 Interaction between Hardware Threads **17**
- 4.1 Synchronization . . . 18
  - 4.1.1 Normal Measurement Setup . . . 18
  - 4.1.2 Reduced Measurement Setup . . . 26
  - 4.1.3 Cost of HT Synchronization . . . 30
- 4.2 The Fhourstones Benchmark . . . 33
- 4.3 Context Switching . . . 37
- 4.4 Hardware Details . . . 39
- 4.5 Conclusions on Hardware Threads Interaction . . . 40

5 Multi-HT CPU Driver **42**
- 5.1 Sharing a Kernel . . . 42
- 5.2 The Kernel Stacks . . . 44
- 5.3 The BIG-Kernel Lock . . . 45
- 5.4 The CPU Driver's Boot Entry Point . . . 45
- 5.5 The CPU Driver's Syscall Entry Point . . . 47
- 5.6 The CPU Driver's Interrupts and Exceptions Entry Point . . . 48
- 5.7 Booting an Application Hardware Thread . . . 50
- 5.8 The Hardware Thread Control Block . . . 52

5.9 Using a Multi-HT CPU Driver 54

6 User Domains on Top of a Multi-HT CPU Driver 55

6.1 Scheduler 55

6.2 Domains, Dispatchers and Hardware Threads 56

6.3 Core Local RPC and Message Passing 57

6.4 HT Management 60

6.5 FPU Registers 64

6.6 Benchmark 64

6.7 Optimizing the Scheduler 70

7 Conclusions and Future Work 72

Acknowledgements

I would like to thank Prof. Timothy Roscoe for giving me the chance to work on the Barrelfish OS. I am also grateful to Reto Achermann and Gerd Zellweger for their feedback and for the ideas shared during the work on this Thesis.

Last but not least, I would like to thank my parents, who have supported me throughout my studies, and my girlfriend, who has been by my side during my Master's studies.

Chapter 1

Introduction

1.1 Motivation

Barrelfish[11] is a research operating system built around the idea that future systems will contain a *large number of CPU cores.* This characteristic raises concerns regarding scalability and OS capability of efficiently managing heterogeneous hardware resources.

In such a scenario, the advantage which Barrelfish aims to exploit is the fact that it is organized as a *distributed system.* Thus, each core is managed by a different *CPU Driver* (i.e., microkernel), meaning that:

- cores can operate independently most of the time (*scalability*)
- each core can run a specific, optimized version of the CPU Driver (*heterogeneity*)

As Barrelfish is based on the microkernel architecture, a lot of tasks that would have been accomplished by a monolithic kernel need to be executed in user space. For this reason, each core has to run a number of user space domains (equivalents of Linux processes): *monitor* (the user space extension of the CPU Driver) and *spanwd* (domain providing the functionality of spawning other domains) are 2 examples. Also, some cores need to execute domains providing critical functionality to the system, such as memory servers (usually, a single *mem_serv* is present in the system).

The multi-kernel approach is a promising way of managing *loosely coupled* CPUs: if the actions of one core have a negligible effect on another, then there is no reason for software to impose a tighter coupling. However, in the case of HTs (i.e., hardware threads), which share most of the CPU resources, it makes sense to take into consideration the way in which one HT affects another, enabling a better utilization of the physical core. Thus, having a single CPU Driver co-manage multiple HTs is a promising solution for the problem at hand.

1.2 Thesis Structure

Following this Chapter, which presents the motivation for our work, we will discuss a series of background elements in Chapter 2 and related work in Chapter 3.

Chapter 4 debuts by proposing a number of models for managing hardware threads at the operating system level. It then continues with a thorough analysis of the cost of synchronizing hardware threads and looks at how an HT's actions impact the sibling HT's performance.

Next, the lessons learned up to that point are used to design and implement an extension to Barrelfish's CPU Driver, which allows it to manage multiple HTs. This process is presented in Chapter 5 and is closely related to the insights detailed in Chapter 6, which describes a method of using the multi-HT variant of the CPU Driver to transfer the gained processing power to user-space tasks.

In the closing of this Thesis, Chapter 7 draws the appropriate conclusions and suggests further research topics, meant to refine and extend the presented work.

Chapter 2

Background

2.1 Simultaneous Multithreading

SMT[9] (i.e., Simultaneous multithreading) is a technique that aims at improving the throughput of processors by *simultaneously* executing instructions belonging to *multiple threads.* A CPU supporting SMT must be superscalar, in order to be able to issue multiple instructions per cycle: otherwise, the processor in question is implementing, at most, *fine-grained temporal multithreading.*

Different HTs (i.e., SMT lanes or threads) of the same CPU core have their own *architectural state*, meaning that they have different register sets and can be booted & halted independently. However, the HTs share *execution resources*, such as processing units and caheche storage.

The existence of *dynamically shared resources* (i.e., resources assigned to HTs based on the current state of the instruction pipelines, and not statically at boot time) is a double-edged sword for performance. When performing orthogonal tasks, the usage of multiple HTs leads to a better utilization of the hardware. However, contending on a shared resource can translate to an overall decrease in throughput, when compared to a single HT scenario. The latter situation is unlikely to occur as frequently, because (time consuming) memory operations on one HT can be exploited by another HT.

Additionally, multiple HTs sharing a lower cache level (than the cache shared by physical cores) can benefit from faster synchronization.

2.2 Intel® Hyper-Threading Technology

Intel® HTT (i.e., Hyper-Threading Technology) is Intel's proprietary implementation of SMT. Thus, the facts stated in Section 2.1 hold true and we will focus on characteristics specific to HTT, as presented in [4].

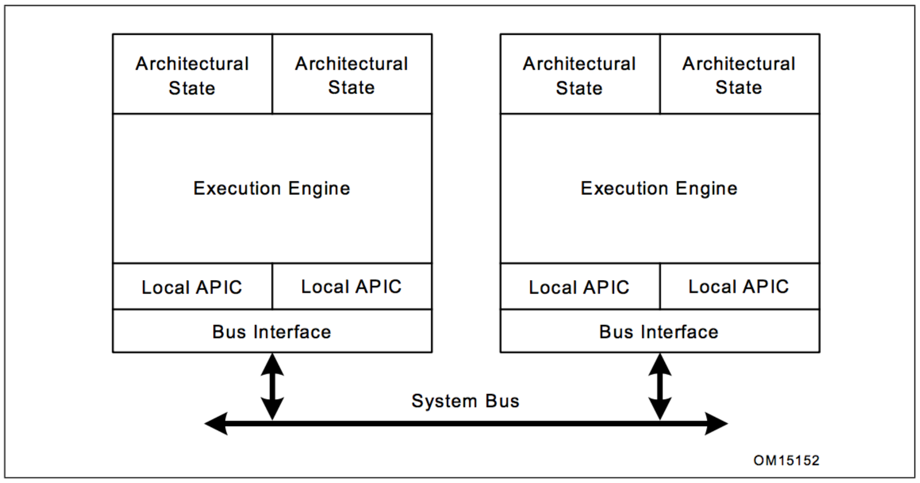


Figure 2.1: Diagram taken from ”Figure 2-14” of [4]. The larger rectangles represent physical CPU cores, each having 2 HTs, and emphasize *replicated* and *shared* HT resources.

For a CPU supporting HTT, the processor resources can be placed in 3 categories, according to the level of sharing between HTs:

1. *Replicated resources* → each HT has its private set. These are:
   - the architectural state, which includes:
     - the 8 general-purpose registers;
     - the control registers;
     - machine state registers;
     - debug registers.
   - instruction pointers;
   - register renaming tables;
   - return stack predictor;
   - 2-entry instruction streaming buffer.
2. *Partitioned resources* → buffers from which an equal number of entries is allocated to each HT:
   - μop queues after the execution trace cache;
   - the queues after the register rename stage;
   - the reorder buffer which stages instructions for retirement;
   - the load & store buffers.
3. *Shared resources* → dynamically shared between HTs:

- caches (unmentioned as belonging to the other categories);
- all the execution units;
- all the other resources.

For the shared resources, [4] notes that the microarchitecture pipeline contains "several selection points to select between the two logical processors". These selection points alternate between the HTs "unless one logical processor cannot make use of a pipeline stage" (because of, for example, "cache misses, branch mispredictions, and instruction dependencies"). In the latter case (i.e., when an HT is blocked), the non-blocked HT "has full use of every cycle of the pipeline stage". Also, it is noted that the execution core and the memory hierarchy are "oblivious to which instructions belong to which logical processor".

2.3 The Barrelfish OS

Barrelfish [11] is an operating system organized as a distributed system. Each CPU core (or HT, in case Hyper-Threading Technology is enabled) has a different CPU Driver. These represent the nodes of the system, communicating via message passing.

The CPU Driver is equivalent to a microkernel and, in order to accomplish tasks which fall in the responsibility of a monolithic kernel, it delegates them to one of:

- the *monitor* domain (i.e., process): this is the kernel's extension into user space;
- user space library code, as part of *libbarrelfish*;
- driver domains.

Aside from the *monitor* domain, the spawn helper domain (i.e., *spawnd*) is also executed on each core. On top of that, the bootstrap core is tasked with running a few other important domains:

- *mem_server* $\rightarrow$ a memory server;
- *pci* $\rightarrow$ PCI discovery service;
- *skb* $\rightarrow$ the system knowledge base, which also acts as a name server;
- *serial* $\rightarrow$ a serial driver;
- *fish* $\rightarrow$ a console.

Each domain has a set of *capabilities* which authorize it to access some resources (e.g., use a physical frame to back a virtual memory page) or to fulfill certain actions (e.g., instruct the CPU Driver to create a new capability). The said set of capabilities form the domain's *CSpace*, which is managed by the kernel: user space can not directly access the memory in which the capabilities are stored, but can use *capref*s (i.e., structures referencing capabilities) in order to be able to *talk* about capabilities with the CPU Driver.

The functionality that allows domains to exchange messages is implemented as *interconnect drivers*, from which 2 stand out and are intensively used by Barrelfish domains [2]:

- *LMP* $\rightarrow$ local (intra-core) message passing:

  In order to send a message, the sender domain invokes a syscall, presenting the kernel with a *capref* (pointing out the receiver domain) and the message contents. This contents can be comprised of a *capref* and/or *binary data* (opaque to the CPU Driver).

  If the receiver is found to be a valid domain, able to receive messages, and there is enough space to store the payload into dedicated buffers, then the transfer is carried out: the binary data is just copied into the buffer, while a transfered capability is copied in the receiver's CSpace, in a preallocated slot.

  Depending on the flags enabled during the send request and considering the way in which the transfer was finalized (successfully or with errors), the kernel makes the decision of which domain to execute next: for example, the sender domain can specify it's desire to yield the processor to the receiver, upon successful transfer, by enabling the *sync* flag.

  Because of the numerous transfers between domains, a separate path through the kernel has been created, in order to speed up LMPs containing only a small amount (below an architecture dependent limit) of normal payload (i.e., no capability) and having the sync flag enabled. This is called the *LRPC* or *fast path*.

  An important aspect to our thesis, that we would like to emphasize, consists of the fact that, on a core, at any given time, *at most 1* of the following actions can be in progress:

  - a domain is running (be it a domain that will send a message, a domain that will receive a message or another domain);
  - an LMP/LRPC transfer is in progress.

- *UMP* $\rightarrow$ user-level message passing:

  The primary target of UMP is to enable message passing across cores, by setting-up a shared frame between 2 domains and relying on the cache coherency protocol to do the actual data transfer. Another difference, when compared to LMP, is that UMP channels need to be polled, in order to determine if new data is available.

  When the message payload does not include capabilities, then the transfer only requires the participation of the 2 domains (i.e., sender and receiver), without the intervention of the CPU Driver. If a capability is to be sent,

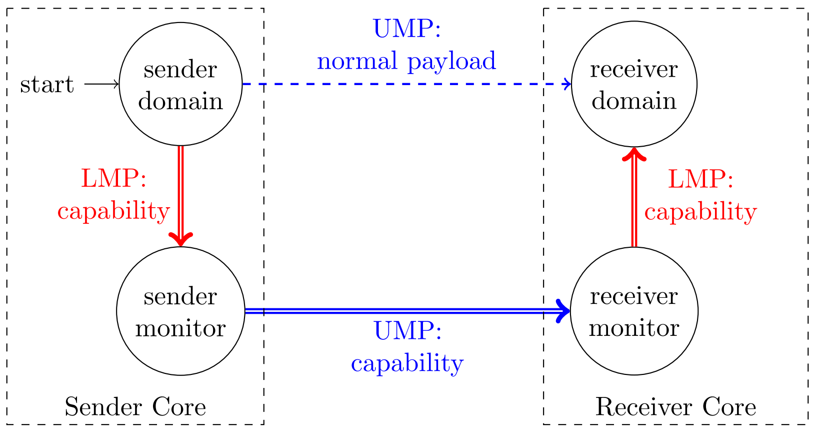


Figure 2.2: Diagram of a UMP transfer between 2 domains on different cores: notice the separate paths took by *normal payload* (i.e., plain binary data) and *capabilities*.

however, an additional phase of the transfer is employed, consisting of the capability being passed through the communicating cores' monitors (see Figure 2.2). The reason for the participation of the monitors is that only these domains are authorized to serialize & deserialize capabilities.

As a disclaimer, we would like to point out that this section is not meant to cover in full the design and the ideas representing the foundation of Barrelfish. It's purpose is to emphasize certain characteristics of the OS that are particularly important to the present thesis. An interested reader is advised to follow-up with the resources referenced in this section.

2.4 Synchronizing Hardware Threads

As expected in a system employing multiple hardware threads, the need to choose the most suitable synchronization mechanism for a particular scenario arises naturally. [5] gives some advice on this topic, mentioning the following alternatives:

- spin loop with *pause*: useful when the waiting period is expected to be short.

  The usage of *pause* is a hint to the underlying hardware that the HT is not doing actual work. This can determine the release of hardware resources (which can be used by a partner HTs), at the expense of a potential bigger latency of lock acquiring.

- *monitor/mwait*: to be used when the waiting period is expected to last longer (compared to the previous case), as this mechanism saves more power than *pause*.

However, it is more difficult to set up: a block of memory of appropriate size (according to information exposed by *cpuid*) and alignment (in order to fit in its own cache line) needs to be shared between the sleeper and the thread responsible with triggering the wake-up signal (by writing into that piece of memory).

- *halt*: a solution for the case in which *monitor/mwait* is not available.

  The gain is similar to that of *monitor/mwait*, but there is a higher cost associated with sending an IPI (i.e., inter-processor interrupt) in order to wake up the sleeping thread.

Chapter 3

Related Work

3.1 Elfen Scheduling

The paper presenting the *Elfen scheduler* [13] was an important source of inspiration for this thesis, being the primary motivation for the work described in Chapter 4. Yang et al. aim at developing a method through which they can improve the utilization of servers, by making use of SMT, but *without* violating SLOs (i.e., service level objectives).

In the mentioned paper, the authors consider the scenario of a service, implemented as a *latency-critical thread*, which has to resolve a given percentage of requests in less time than a predefined limit of time (e.g., "99% of requests must complete within 100 ms" [13]). Additionally, they want to maximize the total utilization of an N-way SMT processor, by doing work on $N-1$ *batch threads*.

The proposed solution consists of the following components:

- *nanonap* $\rightarrow$ a system call which is designed to park a process on an SMT lane. This is done in order to relinquish hardware resources to the partner SMT lane.

  The authors argue that existing mechanisms, out of which *mwait*, *WRLOS*, *hotplug* and *futex* are mentioned, do not provide the (entire) required functionality: they leave the possibility of other threads utilizing the SMT lane or they do not preserve the context on the parked lane.

  nanonap solves the problem by disabling preemption, ensuring interrupts are enabled (so that the OS scheduler can still replace a napping thread with another batch thread) and by putting the HT lane to sleep via *monitor/mwait.*

- the Elfen scheduler $\rightarrow$ uses *nanonap* when its *scheduling policy* dictates that all the core's resources should be made available to the latency-critical thread.

Depending on the employed scheduling policy, Elfen uses *SHIM* (i.e., a fine-grain profiling tool) signals and information from the service thread. The following policies are proposed:

- *borrowing idle cycles*: batch threads execute *only* when the service thread is not running;
- *fixed budget*: aside from getting to use the CPU while the latency-critical thread is not running, batch threads may also execute simultaneously with it. The authors define a *budget* (i.e., a time period during which both types of threads share the CPU) which is refreshed only when the service lane is idle and the requests queue is empty;
- *refresh budget*: extends the *fixed budget* policy by renewing the budget when the serviced request changes and the queue is empty.

  Note that the latency-critical thread and the batch threads need to share a variable containing an ID of the last serviced request;
- *dynamic budget*: the most aggressive policy obtained by enhancing *refresh budget* with the possibility of varying the simultaneous usage limit of the SMT lanes.

  A reference IPC (i.e., instructions per cycle) is obtained by profiling the latency-critical thread with no interference. Afterwards, during the actual execution alongside batch threads, the service thread's IPC is monitored.

  The *real_budget* (i.e., actual co-running budget) is computed by multiplying a static *budget* with the ratio $\frac{ref\_IPC}{ref\_IPC - LC\_IPC}$: ref_IPC denotes the computed reference IPC and LC_IPC is the monitored IPC of the latency-critical thread.

  If the *real_budget* is less than the currently co-running duration, then the batch thread is put to sleep.

3.2 Asynchronous System Calls

The work presented in [10], an asynchronous system call interface, was considered in this thesis as a helper mechanism for a model in which, on a 2-way SMT processor, one thread is responsible with kernel mode execution and the other with running user space tasks. It also served as inspiration for the adaptation of the LMP (i.e., local message passing) mechanism to the context of a multi-HT CPU Driver.

In the cited paper, the authors claim that *system calls* represent the *de facto* interface used in order to request kernel provided services. The traditional design for this mechanism is said to exhibit 2 performance degrading properties:

1. the usage of *processor exceptions* as the mean of communicating with the kernel;
2. a *synchronous* execution model.

The said properties affect performance by minimizing locality, flushing the user-mode pipeline and by replacing the user-mode processor state with the kernel-mode processor state (i.e., processor state pollution).

Two components form the proposed design of *exception-less system calls*:

1. an *exception-less* interface through which user space can invoke system calls;

   This can be implemented as a *shared memory page* between the kernel and user space, organized as a table of syscall entries.

   Thus, invoking a syscall boils down to storing the syscall information (i.e., the syscall identifier and the associated arguments) into a *free* entry, marking it as *pending* and later checking if it has been marked as *done*. All this actions happen in user space, the kernel being only responsible with executing the syscalls referenced by *pending* entries and setting their status as *done*. The result of a syscall request would be stored in its associated entry.

   Note that, the ability to batch system calls leads to an improved *temporal locality*.

2. an in-kernel threading system meant to execute syscalls in an *asynchronous* manner.

   These threads would pull requests from the shared memory page and would service them.

   Specific cores can be reserved for only executing syscall threads, which leads to better *spatial locality*.

The paper at [10] also proposes a *M-on-N threading system* (i.e., M user space threads on top of N kernel-visible threads, with $M \gg N$), as a means of relaxing the constraints imposed on the programmer by the asynchronous nature of the syscall mechanism: user space can just dispatch another thread when the one currently executing blocks on a syscall invocation, increasing the number of opportunities for the kernel to service the request.

3.3 Exclusive Access to a Microkernel

Having multiple hardware threads on top of the same CPU Driver determines the need to ensure exclusive access to some of the kernel's structures. [7] presents a comparison of such mechanisms in the context of a microkernel (as is the CPU Driver powering Barrelfish). The arguments mentioned in this paper are organized around 2 topics: performance and correctness.

For the performance side, the authors of [7] rely on 3 experiments in order to show the differences between an (unsafe) lockless implementation (identified as *none*), an implementation using a big kernel lock (*BKL*), one relying on fine-grained locking (*fine*) and an implementation of a hardware transactional memory based solution (*RTM*). The experiments and their results are the following:

1. single-core ping-pong → a pair of threads (on the same core) send IPCs between each other. A single core is used in this experiment, with the other ones being disabled.

   The aim was to surface the *contention-free locking cost*, meaning that even the *none* variant was a correct approach in such a scenario. Thus, as one might expect, *BKL* saw the best results, in terms of performance, after *none*.

2. multi-core ping-pong → an extension to the first experiment, the single-core ping-pong is executed simultaneously on a given number of hardware threads. All hardware threads use the same kernel.

   For this case, the throughput of *BKL* plateaued when using at least 3 cores, and was surpassed by all the other variants at 2 cores on x86 and at 4 cores on ARM. Also, *none*, *fine* and *RTM* scaled much better than *BKL*, without reaching a plateau in the considered configurations.

3. Redis benchmark → a Redis key-value store services requests generated by using the Yahoo! Cloud Servicing Benchmarks (i.e., YCSB). In this scenario, Core 0 runs an Ethernet driver, the lwIP TCP/IP network stack (as a user space process) and a Redis server. The rest of the cores run only the latter 2 processes, meaning that all the interrupts are serviced by the driver on Core 0.

   The authors of [7] show results suggesting that the throughput is *independent* of the locking strategy and state that "The results indicate that for 8-way parallelism, and likely beyond, the choice of lock is essentially irrelevant to performance" [7].

   They also mention that the overall throughput was limited by the network bandwidth and try to compensate by dividing the throughput by the average utilization of all cores (resulting in what the authors call the *Xput* value). Using *Xput* instead of throughput shows better scaling with the number of cores, but still no significant differences between the employed locking strategies.

Analyzing the presented experiments and their results, we do not agree with the conclusion of [7] (regarding how the locking strategies affect performance, in the context of a microkernel):

- "single-core ping-pong" shows the performance of each lock in a manner oblivious to the effects on a multi-core scenario;
- "multi-core ping-pong" is disastrous for *BKL*, but the authors disqualify it by stating that it is "an unrealistic worst-case scenario for the BKL" [7];
- the "Redis benchmark", which is the main argument associated with the previously cited conclusion, can not be considered significant if it just measures the network bandwidth.

Thus, we do not think that the performance analysis favors *BKL*.

However, the much greater advantages brought by such an approach in a correctness analysis make it a viable solution. [7] presents some arguments for this in a small section (i.e., Section 2.2), but we would have liked to see more efforts from the authors directed at the correctness topic, instead of presenting the unconvincing performance arguments.

Chapter 4

Interaction between Hardware Threads

As an initial step of this Thesis, we plan to investigate the trade-offs and techniques available on a mainstream Intel processor (such as Haswell or Sandy Bridge), with 2-way SMT via Hyper-Threading Technology. Some of the questions are:

- What are the costs of synchronizing SMT threads on the same core using combinations of *monitor* & *mwait*?
- Can SMT threads be used as a cache for register contents, or do the threads share a single register file?
- Can SMT be turned on and off dynamically for a core (or for a package) after the machine has booted (i.e., under the control of the OS)?
- What is the performance impact of enabling Hyper-Threading and then parking all but 1 hardware thread per core in the kernel (e.g., via *mwait* or *hlt*) vs. disabling Hyper-Threading at boot?

In the second phase, we plan to examine a number of alternatives for SMT usage in the OS, such as:

- A naive approach which simply runs two CPU drivers, one on each SMT thread. This will be the baseline comparison for the other models;
- Dedicate SMT threads to CPU driver and user-space, and switch between them upon kernel entry & exit;
- Use SMT threads as caches for user-space dispatcher contexts, and run only one at a time;
- As above, but hard-wiring one thread to run the *monitor* dispatcher;
- Run multiple user-space SMT threads at the same time, but only allow at most one in the kernel at any given time.

4.1 Synchronization

Starting with the first question, we will look into the overhead of synchronizing HTs and into the way in which the said overhead is related to the coupling between HTs (i.e., both HTs on the same core, on different cores or even on different chips). Knowing this cost is critical in order to be able to design a practical system, which brings real advantages over the alternatives.

Two different measurement setups were used, as a way of improving the chances that the observed durations are associated with HT synchronization: the CPU hardware and all the other components of a computing system have become increasingly complicated, making it easier to overlook certain aspects. Having different views of the same process provides an extra assurance for the recorded data.

Additionally, we took into account the overhead of reading the timestamp counter (via *rdtscp*) and we discarded the first and last approx. 10% observations, in order to eliminate the warm-up and cool-down phases of the benchmarks. The *rdtscp* overhead was determined by reading the timestamp counter 1001 times, subtracting the first value from the last and dividing the result by 1000. This is the standard procedure used in Barrelfish's benchmarking support library, *libbench*.

In both testing scenarios, 2 hardware threads were made to enter their CPU Driver (via a specially implemented syscall): note that all unordered pairs of HTs were considered. As we are referring to the situation of a norarrelmal Barrelfish instance, each HT had a separate CPU Driver.

In order to minimize external influences, interrupts were disabled (as they always are when an HT is within the kernel) and the recorded data was outputted to the console only at the end of an experiment. Also, the HTs remained inside the CPU Driver from the first measurement iteration to the last one (for that specific HT pair).

The following subsections dive into the measurement setups and discuss the obtained results. Each setup was executed with and without Hyper-Threading enabled and used either *mwait* or *hlt* as the sleeping mechanism (details about these instructions in Section 2.4), yielding a total of 4 configurations per experimental setup.

A number of iterations are executed for each HT pair and configuration, the observed data being used to compute the mean and the 95% CI (i.e., confidence interval) of the wake-up durations. These results are plotted in the form of heatmaps.

4.1.1 Normal Measurement Setup

Figure 4.1 represents a diagram of the first measurement setup which we employed. The 2 hardware threads executed 240 iterations, out of which the first and last 10 were discarded. As the *start* and *stop* timestamps are taken by different HTs with unsynchronized timestamp counters, we needed 2 iterations

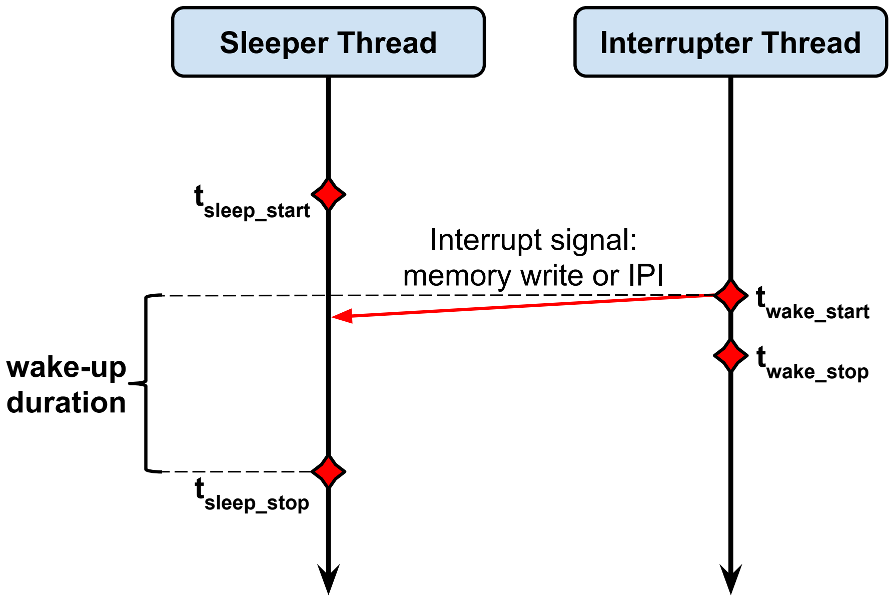


Figure 4.1: Normal Measurement Setup. The red diamonds represent moments when the cycle counter is read.

in order to remove the *clock skew* from the computed duration: HTs alternated the roles of sleeper and interrupter threads each iteration.

The interrupter thread would wait for the sleeper to enter the parked state, by doing a tight loop on a shared memory location, after which it would execute 300 *nops*, in order to give the sleeper some slack after detecting the change of the shared flag. We also experimented with a higher slack duration, but the results were unaffected by this variation.

Next, the interrupter would read the timestamp counter, give the wake-up signal (i.e., write to memory or send an inter-processor interrupt) and take another timestamp reading.

On the sleeper's side, the HT would raise the shared flag indicating that it is ready to go into parked state, read the timestamp counter, after which it would immediately execute *monitor/mwait* or *hlt*, putting the HT to sleep. The sleeper HT would also take a timestamp reading just after resuming its execution.

As illustrated in Figure 4.1, the wake-up duration is calculated by subtracting the timestamp value read by the interrupter thread before sending the wake-up signal from the timestamp taken by the sleeper thread after it has woken up. The average of 2 such results (with opposite wake-up signal direction) is used in future calculations (for removing the *clock skew*): when determining the mean and the CI.

Normal Measurement with Hyper-Threading *enabled*

The results, for the cases where Hyper-Threading was enabled, are displayed in Figures 4.2 and 4.3: the number written in the cell at the intersection between X (on the horizontal axis) and Y (on the vertical axis) corresponds to the benchmark being run with HTs X and Y. The difference between the scenarios represented by these figures is that the former (i.e., Figure 4.2) used *monitor/mwait* as the parking mechanism, while the latter (i.e., Figure 4.3) employed *hlt*.

Looking at Figures 4.2b and 4.3b we see that the 95% confidence intervals are tight around the means, consisting of less than 24 cycles for average values higher than 1200 cycles (i.e., less than 2% of the means).

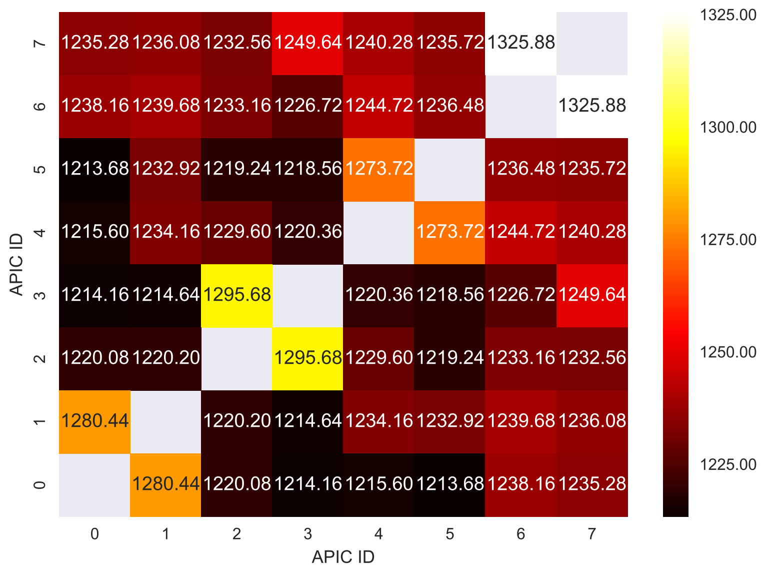
The numbers on both axes represent *APIC ID*s: hardware identifiers of each HT. As noted in Section 4.4, the benchmarked CPU had 4 cores, each having 2-way SMT. The *APIC ID*s encode, in their binary representation, the hierarchical position of the associated HT: starting from the least significant bit, this ID is composed of groups of bits identifying the SMT lane in the physical core, the physical core in the socket and so on. As our experiment considers a single socket system, we are only interested in separating the HT identifier from the rest of the APIC ID, which (in this case) consists of a single bit (enough for a 2-way SMT processor). For example, the HT with APIC ID $5_{10} = 101_2$ corresponds to the SMT lane with ID 1 and the core with ID $2_{10} = 10_2$: the 2nd HT of the 3rd core, since all IDs are 0-based.

Knowing that the HTs with APIC IDs 0 & 1, 2 & 3, 4 & 5, 6 & 7 share the same physical core, we can see from the results that tightly coupled SMT lanes exhibited a longer wake-up duration. At first, we thought this was caused by the fact that, while the sleeper HT was waking up, the interrupter HT (who was transitioning into the sleeper role for the next iteration) was going to sleep: this behavior could have generated contention in the in-hardware mechanisms used for resource allocation (for example, the *partitioned resources* mentioned in Section 2.2). That supposition proved to be false, as no change in the observed behavior was witnessed when adding a *nop* loop before the interrupter thread went to sleep (for the next iteration).

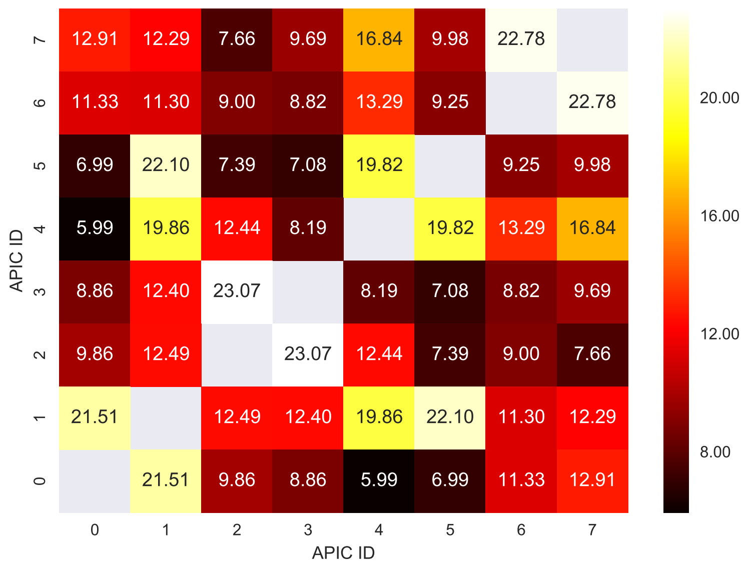
Looking at the big picture, what we saw was that when both tightly-coupled SMT lanes executed the benchmark they were affected by a longer wake-up duration. However, this was not the case when one of these lanes was running normal Barrelfish code. Our guess is that the reason for the observed behavior was the nature of executed machine-level instructions and/or the way cache utilization: we knew beforehand that HTs belonging to the same physical core provide the largest throughput when executing orthogonal operations (for example, one does integer additions while the other executes floating-point operations).

Lastly, but probably the most important takeaway from the data displayed in Figures 4.2 and 4.3, we can derive the number of cycles needed by an HT in order to resume from sleep:

- ≈ 1250 cycles, for a *monitor/mwait* induced sleep;

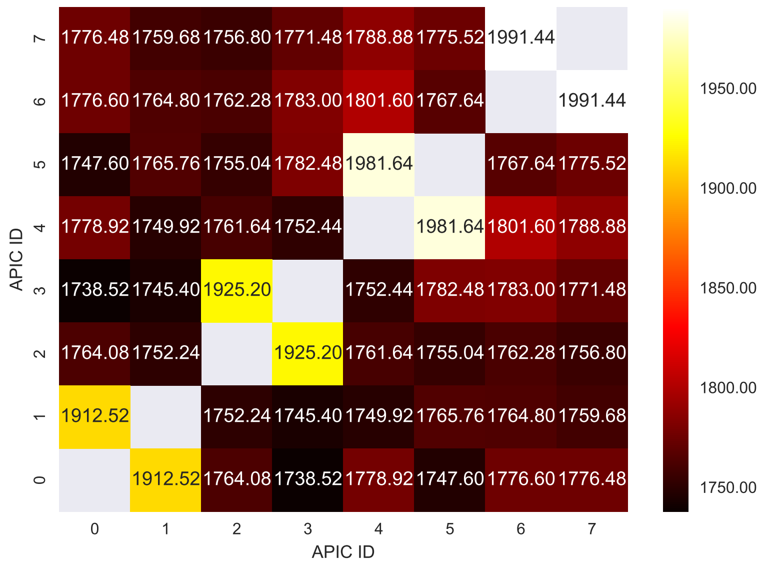


(a) Duration from *wake start* to *sleep stop* (cycles)

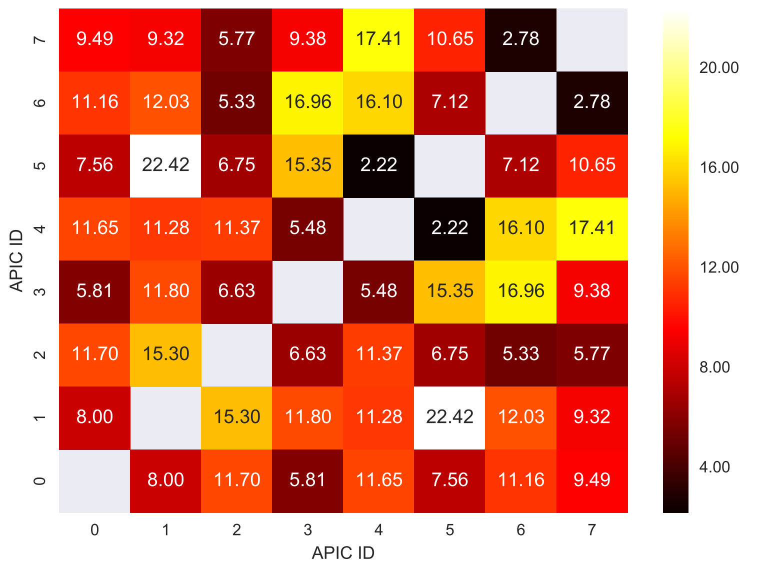


(b) 95% CI for Figure 4.2a

Figure 4.2: Normal measurement of *vacherin*'s wake-up duration from sleep induced by *mwait* (Hyper-Threading enabled)



(a) Duration from *wake start* to *sleep stop* (cycles)



(b) 95% CI for Figure 4.3a

Figure 4.3: Normal measurement of *vacherin*'s wake-up duration from sleep induced by *hlt* (Hyper-Threading enabled)

- $\approx$ 1800 cycles, for an HT parked via *hlt*.

Although not displayed in the previously referenced figures, we were also able to determine the cost of sending a wake-up signal (i.e., a memory stores in the case of *monitor/mwait* and an IPI in the case of *hlt*):

- $\approx$ 0 cycles, for *monitor/mwait*;
- $\approx$ 550 cycles, for *hlt*.

This penalty imposed on the interrupter thread was computed by subtracting t_{wake_start} from t_{wake_stop} (see Figure 4.1).

Normal Measurement with Hyper-Threading ***disabled***

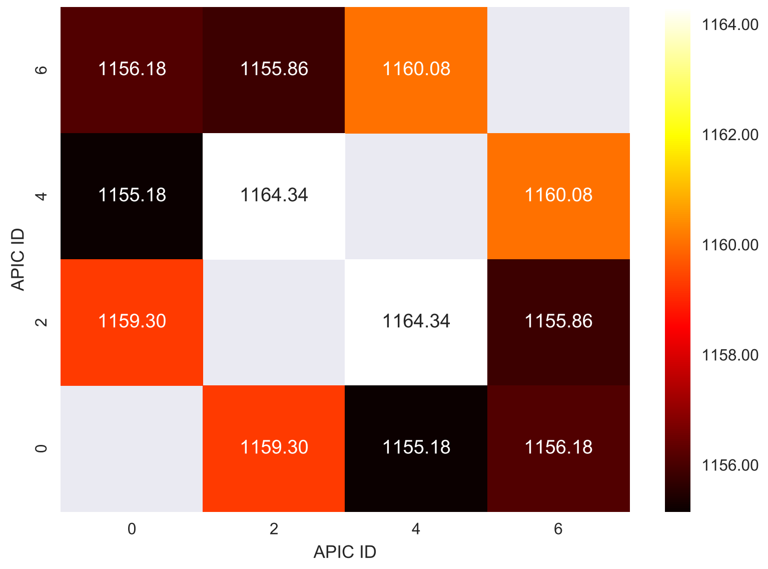
In order to see how Hyper-Threading affects wake-up duration, we disabled this feature in the BIOS and ran the experiment in the same, normal measurement, setup.

The first thing that we notice when analyzing Figures 4.4 & 4.5 in comparison with Figures 4.2 & 4.3 is the more stable behavior of the system, both in terms of the range of observed wake-up durations (for different HT pairs) and in terms of the confidence intervals: the most clear indication is that the 95% CI is between 4$\times$ and 20$\times$ smaller. Also, there are no pairs of cores which exhibit distinctive behavior, like in the case of tightly-coupled HTs (when Hyper-Threading was enabled).

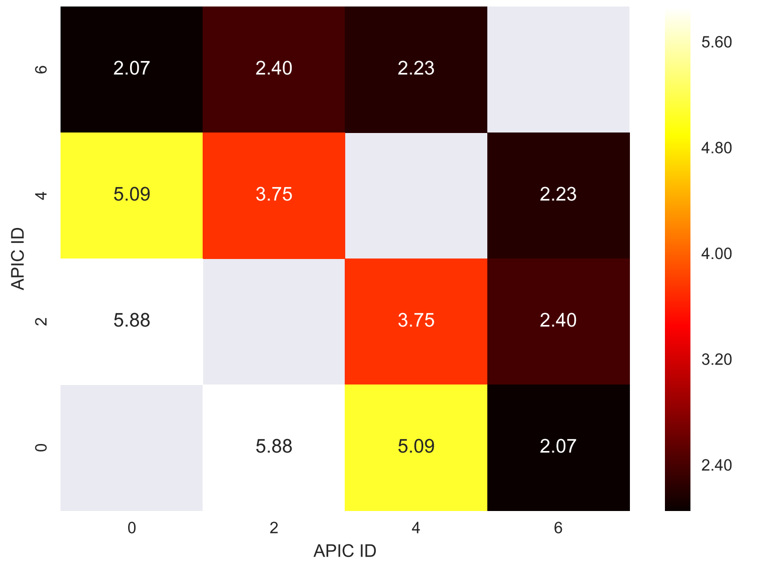
The main reason why we disabled Hyper-Threading was to see the effect on wake-up duration, which, in this case, is:

- $\approx$ 1160 cycles, for *monitor/mwait*;
- $\approx$ 1590 cycles, for *hlt*.

The values are a bit lower (1160 vs 1250 and 1590 vs 1800, all values in cycles), which can be accounted for by the contention for shared hardware resources, in the case where 2-way SMT was enabled.

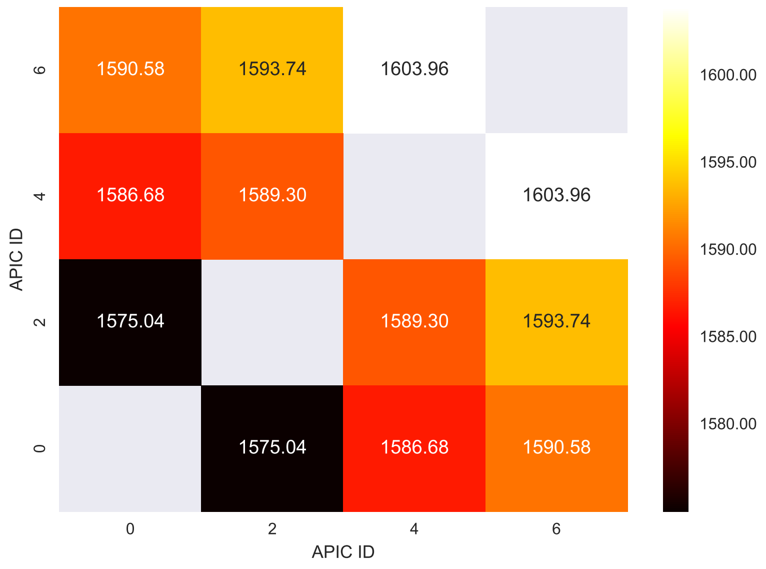


(a) Duration from *wake start* to *sleep stop* (cycles)

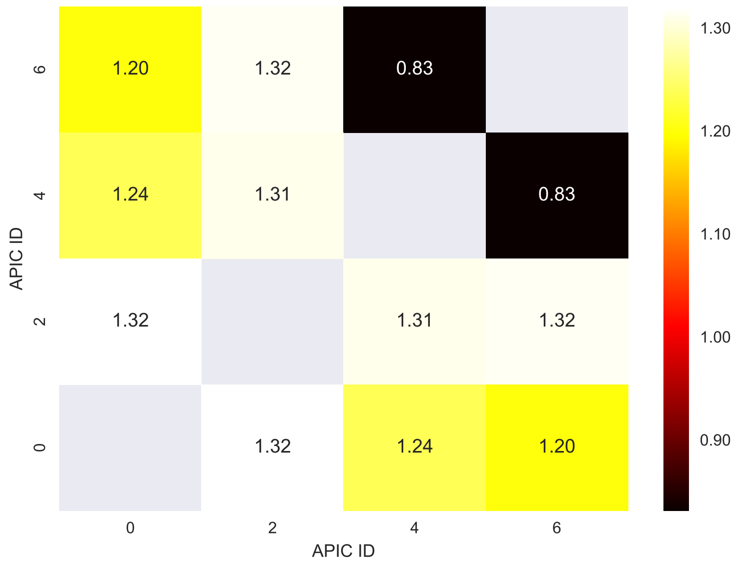


(b) 95% CI for Figure 4.4a

Figure 4.4: Normal measurement of *vacherin*'s wake-up duration from sleep induced by *mwait* (Hyper-Threading disabled)



(a) Duration from *wake start* to *sleep stop* (cycles)



(b) 95% CI for Figure 4.5a

Figure 4.5: Normal measurement of *vacherin*'s wake-up duration from sleep induced by *hlt* (Hyper-Threading disabled)

4.1.2 Reduced Measurement Setup

The reduced measurement setup, portrayed in Figure 4.6, represents another method of measuring the synchronization overhead between hardware threads. The main difference when compared to the normal measurement setup is that, this time, only the interrupter thread reads the timestamp counter.

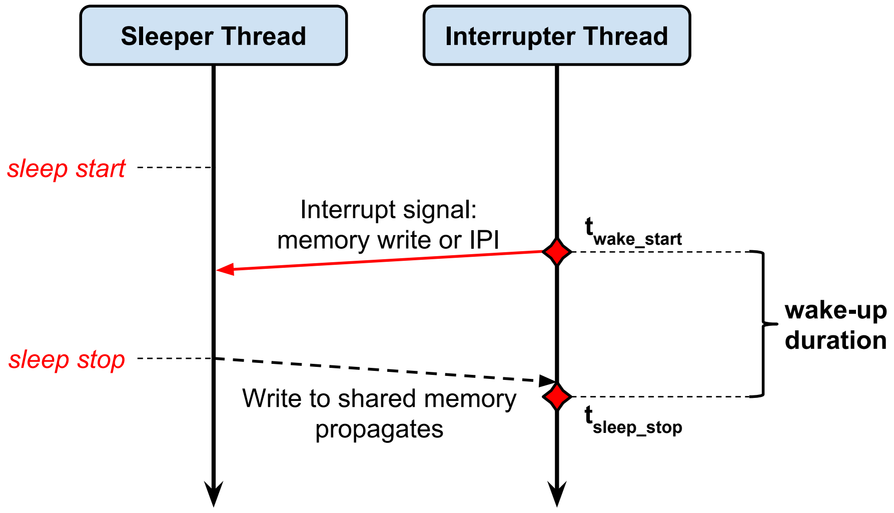


Figure 4.6: Reduced Measurement Setup. The red diamonds represent moments when the cycle counter is read.

The beginning of a measurement round is the same as in the normal setup: the sleeper thread raises a flag (by writing into a memory location shared with the interrupter thread) and goes to sleep (via *monitor/mwait* or *hlt*). Meanwhile, the interrupter spins on the shared flag, exiting the loop only after it has detected the change. Next, the interrupter performs 300 *nops* (in order to give the sleeper some time for releasing hardware resources), reads the timestamp counter (t_{wake_start}) and sends the interrupting signal (either a memory write or an *IPI*, depending on the employed sleeping mechanism). Following, the interrupter thread spins on another flag, which will be raised by the sleeper once it has resumed execution. Finally, after the interrupter exits the second spinning loop, it determines the value of t_{sleep_stop} by reading (again) the timestamp counter.

Thus, the wake-up duration is computed as $t_{sleep\_stop} - t_{wake\_start}$.

Since the 2 timestamps have been taken by the same hardware thread, there is no clock skew and no need to alternate sleeper and interrupter roles between the HTs, in order to determine the wake-up duration. However, as a similarity to the normal measurement setup, we kept the alternating rounds.

We executed 240 rounds for each of the 2 available sleeping mechanisms (*monitor/mwait* and *hlt*) and for 2 hardware configurations (Hyper-Threading *enabled* and *disabled*). These actions were performed for each (unordered) pair of threads, meaning that:

- for HT pair (X, Y), the 240 rounds for a given sleeping mechanism and hardware configuration generates 120 values for the wake-up duration of HT X (the rounds in which HT Y was the interrupter) and 120 values for the wake-up duration of HT Y (the rounds in which HT X was the interrupter);
- in contrast to the normal measurement setup, the reduced setup may (and often does) yield different values for the wake-up duration of the 2 HTs which were benchmarked together. Thus, the value placed in the heat-maps at the intersection between x-coordinate X and y-coordinate Y is associated with HT X (i.e., the sleeper HT). This is the reason for which the axes are labeled *"Sleeper APIC ID"* and *"Interrupter APIC ID"*, as opposed to just *"APIC ID"*.

The means and 95% CIs of the computed wake-up durations are presented in the following paragraphs.

Reduced Measurement with Hyper-Threading *enabled*

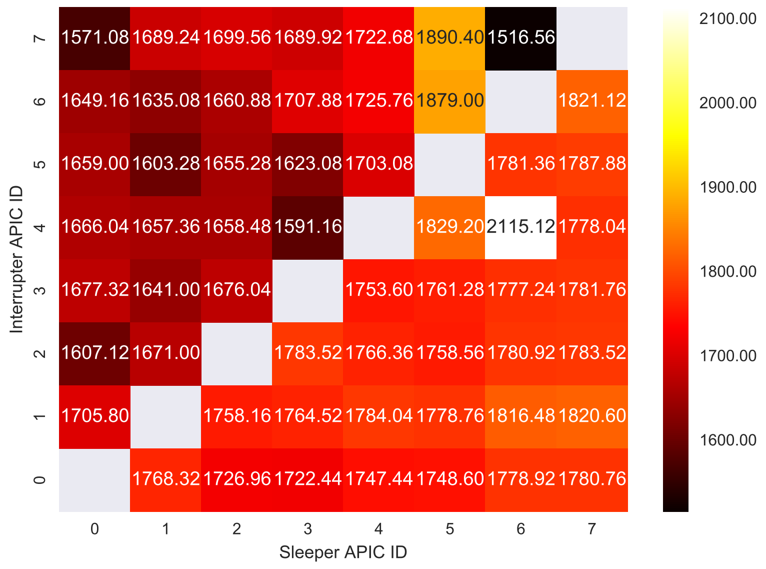
When Hyper-Threading was enabled, we obtained the wake-up duration data depicted in Figures 4.7 (for *monitor/mwait*) and 4.8 (for *hlt*).

A significant amount of noise can be witnessed in the case of sleeping via *monitor/mwait*, with noticeable outliers (such as sleeper 5 & interrupters 6 and 7, respectively): see the 95% CI in Figure 4.7b. The fact that the noise levels have not increased for the *hlt*-based sleeping, when switching from the normal setup to the reduced one, suggests that memory operations are at fault for the larger confidence intervals: the biggest difference between the 2 sleeping mechanisms lies in the way in which an HT is resumed (memory write for *monitor/mwait* and IPI for *hlt*).

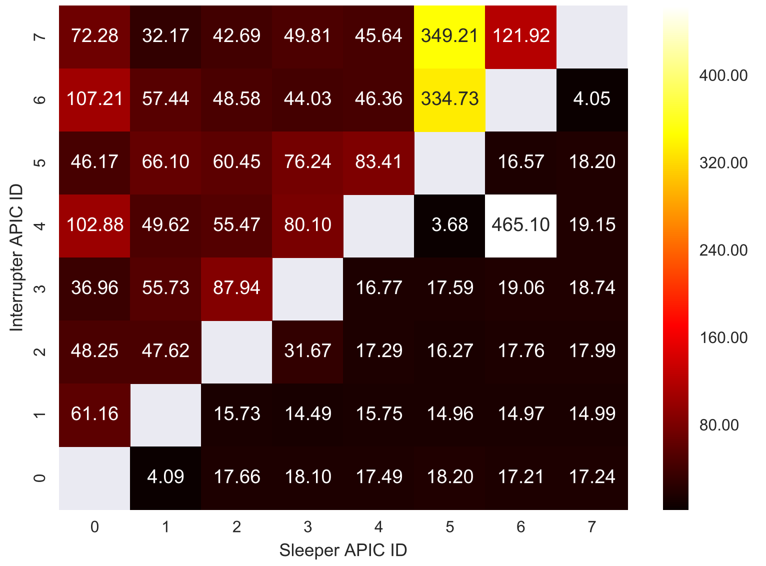
Overall, we notice higher values for the wake-up durations:

- $\approx$ 1750 cycles, for *monitor/mwait*;
- $\approx$ 2200 cycle, for *hlt*.

Compared to the values obtained in similar conditions, but by using the normal measurement setup, we see that the wake-up durations have increased by $400 - 500$ cycles. The most probable cause for this effect is the fact that the interrupter thread spins on a shared variable while waiting for the sleeper HT to resume. Only when this variable is changed by the newly waken thread does the interrupter record t_{sleep_stop}.

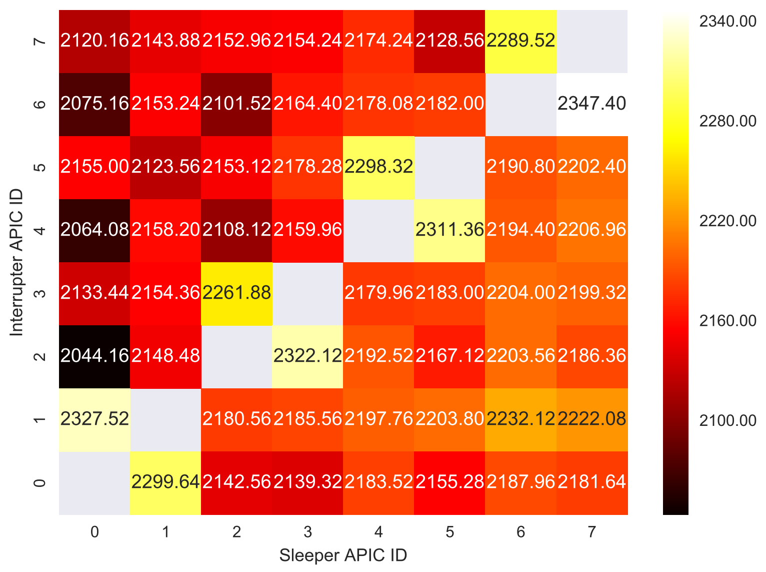


(a) Duration from *wake start* to *sleep stop* (cycles)

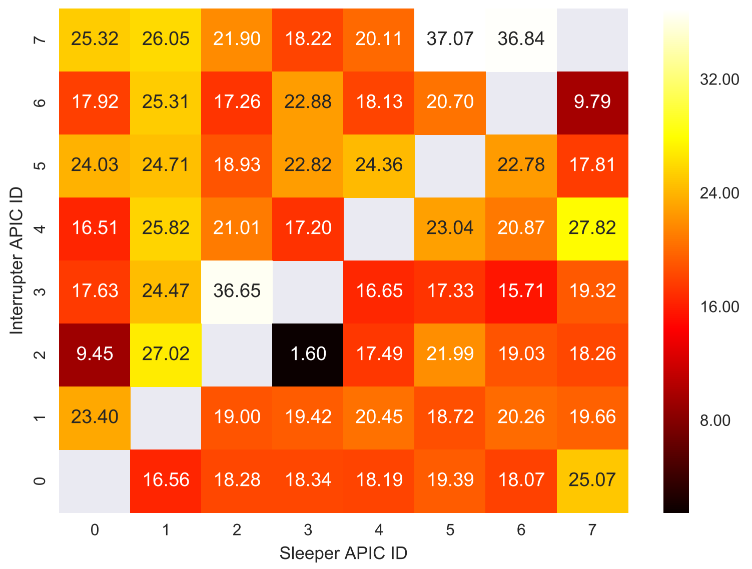


(b) 95% CI for Figure 4.7a

Figure 4.7: Reduced measurement of *vacherin*'s wake-up duration from sleep induced by *mwait* (Hyper-Threading enabled)



(a) Duration from *wake start* to *sleep stop* (cycles)



(b) 95% CI for Figure 4.8a

Figure 4.8: Reduced measurement of *vacherin*'s wake-up duration from sleep induced by *hlt* (Hyper-Threading enabled)

Reduced Measurement with Hyper-Threading *disabled*

As we did in the case of the normal measurement setup, we also ran the experiments in the reduced setup *with Hyper-Threading disabled.* The processed data is presented in Figures 4.9 and 4.10.

We again observed an increase in the 95% CI when compared to the experiments executed in the normal setup with Hyper-Threading disabled (i.e., Figures 4.4 and 4.5): these intervals have increased by $\approx 20\times$ (in the case of *mwait*) and by $\approx 10\times$ (for *hlt*). The difference between the 2 factors support the idea presented in the previous paragraphs: that spinning on the *"sleep stop flag"* is the reason for the increased measured wake-up durations.

These durations are:

- $\approx$ 1750 cycles, for *monitor/mwait*;
- $\approx$ 1800 cycle, for *hlt*.

Comparing the effect of Hyper-Threading in the reduced setup, it seems that the overhead of the extra spin lock (i.e., the one signaling the sleeper resuming execution) shadows the difference between waking-up from *mwait*-induced sleep while having Hyper-Threading *enabled vs disabled*: the wake-up durations are negligible higher when Hyper-Threading Technology was enabled.

For the *hlt*-based sleep, enabling 2-way SMT increased the wake-up durations by about 200 cycles in the normal setup, and by 400 cycles in the reduced setup. This is probably due to the higher cost of doing a tight loop of memory reads when both HTs of the same core are active.

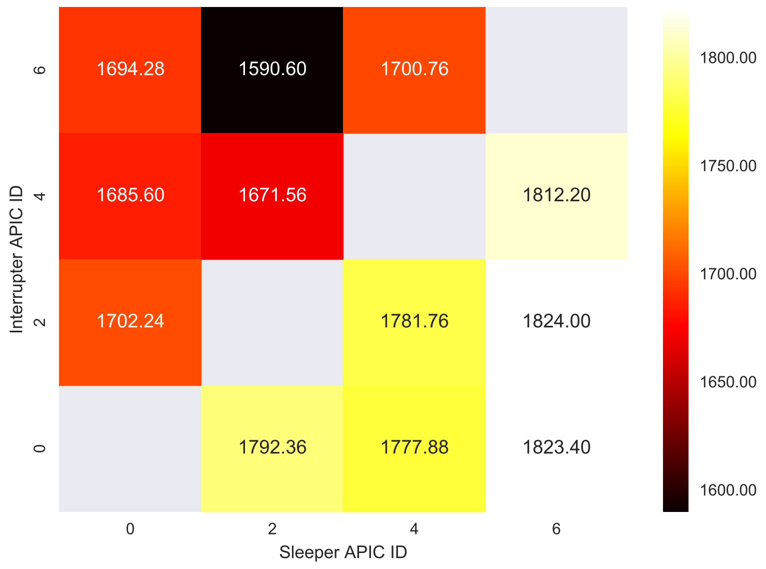
4.1.3 Cost of HT Synchronization

Taking a step back and looking at the practical differences between the normal and the reduced setups, we see that:

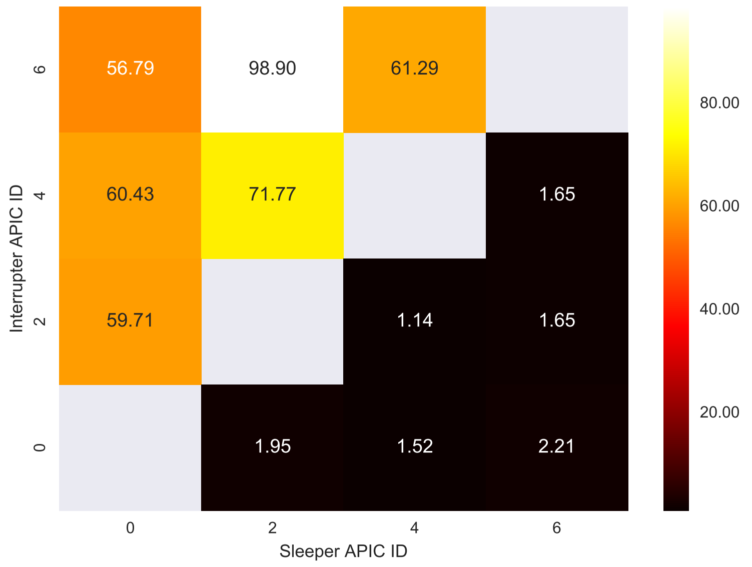
- the normal setup minimizes the amount of non-benchmarked synchronization between HTs, by only ensuring that the interrupter sends the wake-up signal *after* the sleeper has entered the low power state;
- the reduced setup adds extra synchronization inherently included in the measured durations, but provides the extra benefit of only taking timestamps from a single hardware thread, which has not just resumed execution.

The benefits provided by the reduced setup on top of those existent in the normal setup are significantly reduced:

- the CPU that we used for testing features an *invariant timestamp counter*, meaning that: "The invariant TSC will run at a constant rate in all ACPI P-, C-. and T-states. [...] the OS may use the TSC for wall clock timer services" [6].

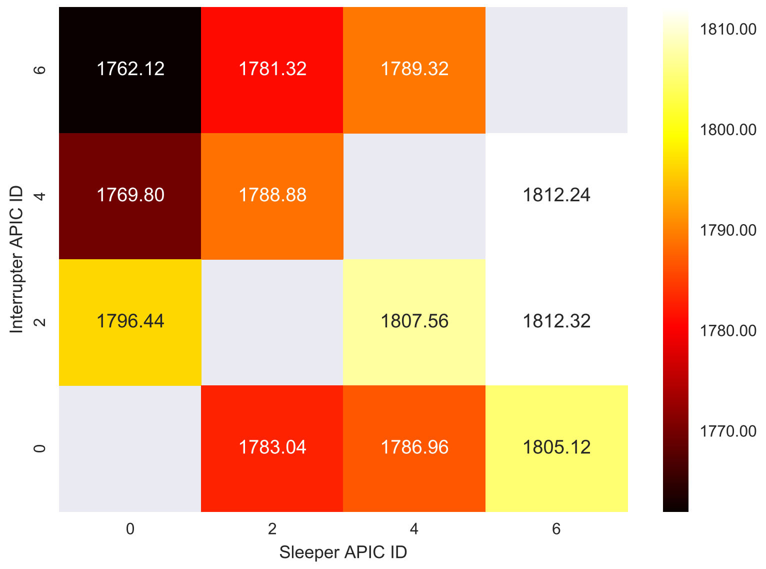


(a) Duration from *wake start* to *sleep stop* (cycles)

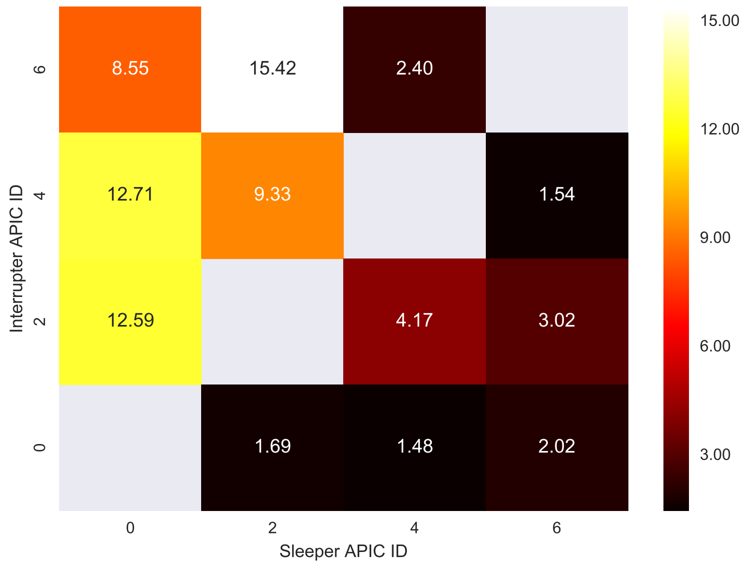


(b) 95% CI for Figure 4.9a

Figure 4.9: Reduced measurement of *vacherin*'s wake-up duration from sleep induced by *mwait* (Hyper-Threading disabled)



(a) Duration from *wake start* to *sleep stop* (cycles)



(b) 95% CI for Figure 4.10a

Figure 4.10: Reduced measurement of *vacherin*'s wake-up duration from sleep induced by *hlt* (Hyper-Threading disabled)

- as stated before, because the timestamp counters of the sleeper and the interrupter threads may not be synchronized, we alternate the roles between the 2 HTs and use 2 measurements in order to compute a single wake-up duration value (as described at the beginning of Subsection 4.1.1).

Thus, in conclusion, the wake-up durations that we are going to use in the remainder of this Thesis are the ones determined via the normal measurement setup. These are summarized in Table 4.1.

Table 4.1: Wake-up durations from sleep induced via *monitor/mwait* and *hlt*, with Hyper-Threading Technology *enabled* and *disabled.* All values represent the number of *cycles* expected to elapse after the wake-up signal is sent and until the hardware thread resumes execution.

Hyper-Threading / Sleeping Mechanism	HTT *enabled*	HTT *disabled*
monitor/mwait	**1250**	**1160**
hlt	**1800**	**1590**

4.2 The Fhourstones Benchmark

So far in this Chapter, we have focused on the cost of synchronizing SMT threads, with a few side remarks regarding consequences of enabling Hyper-Threading. We will now move on to looking at performance related effects of using multiple hardware threads.

For this purpose, we plan on using the Fhourstones benchmark [12]. This is a single-threaded integer benchmark which does an alpha-beta search in order to solve the game of Connect-4. It is (or was, at least, at some point) included in the Phoronix Test Suite [8].

We think Fhourstones is a good candidate for a benchmarking program since it's workload is CPU bound and does not employ floating-point operations. The amount of required memory for solving a 7×6 board is about 64 MB, which is almost entirely used for storing a transposition table.

When executing an instance of Fhourstones, we first run 3 warm-up iterations, by starting to search for a strategy to solve the game beginning at a given state (i.e., we make a couple of moves, as opposed to leaving the board completely empty). Afterwards, the Connect-4 game is solved, starting from an empty board. The measurements from the final (i.e., 4$^{\text{th}}$) iteration are used when reporting the results.

Overview of the executed experiments

The outcomes of our experiments are depicted, in a concise manner, in Figure 4.11. The vertical axis shows the average number of kilo-positions explored

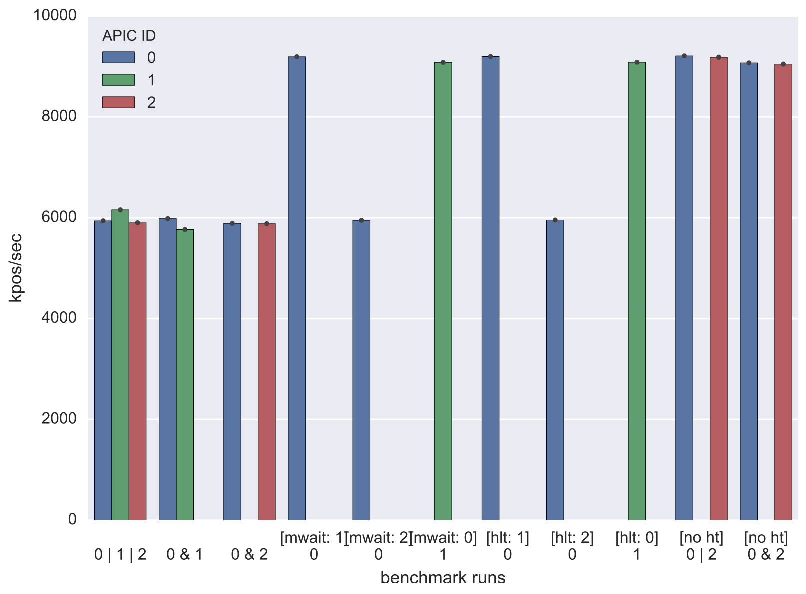


Figure 4.11: The results of running different Fhourstones-based experiments. The values on the vertical axis represent the number of kilo-positions of the Connect-4 game processed per second. Beneath the horizontal axis, there are labels identifying each experiment: the meaning of these labels is explained in the present Section.

by Fhourstones in each second, while the horizontal axis gives details about the experiments which we ran. When nothing is mentioned about an HT, it can be assumed that the HT in question was executing the usual Barrelfish domains (details in Section 2.3) on top of a standard CPU Driver. Also, by default, Hyper-Threading was turned on.

Regarding the format used to declare the conditions in which an experiment was executed:

- '$x \mid y$' $\rightarrow$ HTs with APIC IDs x and y were used for the same type of experiment, but not at the same time (i.e., 2 different experiments were executed, one for HT x and one for HT y);
- 'x & y' $\rightarrow$ HTs with APIC IDs x and y were simultaneously used in the same experiment;
- '[mwait: x]' $\rightarrow$ the HT with APIC ID x was put to sleep by using *monitor/mwait*;
- '[hlt: x]' $\rightarrow$ the HT with APIC ID x was put to sleep by using *hlt*;
- '[no ht]' $\rightarrow$ Hyper-Threading Technology was disabled.

To make everything crystal clear, the experiments which we employed are (starting from the left of Figure 4.11):

1. '0 | 1 | 2' $\rightarrow$ we ran Fhourstones on the HTs with APIC IDs 0, 1, and 2, respectively, and not at the same time, but in subsequent runs.
2. '0 & 1' $\rightarrow$ 2 (independent) instances of Fhourstones were executed, at the same time, on HTs 0 and 1;
3. '0 & 2' $\rightarrow$ same as above, but by using HTs 0 and 2;
4. '$\overset{[\text{mwait: } 1]}{0}$' $\rightarrow$ ran Fhourstones on HT 0, while HT 1 was parked using *monitor/mwait* (the other HTs were running normal instances of Barrelfish, as detailed in a previous paragraph of this Section);
5. '$\overset{[\text{mwait: } 2]}{0}$' $\rightarrow$ same as above, but with HT 2 being the sleeping thread;
6. '$\overset{[\text{mwait: } 0]}{1}$' $\rightarrow$ same as experiment 4, but with the roles of the 2 HTs (i.e., 0 and 1) being reversed;
7. '$\overset{[\text{hlt: } 1]}{0}$' $\rightarrow$ same as experiment 4, with *hlt* as the sleeping mechanism, instead of *monitor/mwait*;
8. '$\overset{[\text{hlt: } 2]}{0}$' $\rightarrow$ same as experiment 5, with *hlt* instead of *monitor/mwait*;

9. ’ $\substack{\text{[hlt: 0]} \\ 1}$ ’ → variation of experiment 6, with *hlt* instead of *monitor/mwait*;

10. ’ $\substack{\text{[no ht]} \\ 0 \mid 2}$ ’ → executed a Fhourstones on each of HTs 0 and 2, subsequently, not at the same time, and with Hyper-Threading *disabled*;

11. ’ $\substack{\text{[no ht]} \\ 0 \,\&\, 2}$ ’ → same as above, but the 2 Fhourstones instances (1 on each HT) were executed simultaneously.

Conclusions based on the data gathered from Fhourstones experiments

The reason why we employed such a variety of Fhourstones-based experiments is that we wanted to look from different angles at the way in which Hyper-Threading affects performance.

Thus, the first observation that we make is that enabling Hyper-Threading reduces performance of an HT to about 2 thirds of what an HT can achieve in a similar context, but with Hyper-Threading disabled. This is true regardless whether the 2 HTs sharing the same physical core execute an identical workload or a different one: there is no difference when HTs 0 and 1 executed Fhourstones instances simultaneously *vs.* when only one of the 2 HTs executed Fhourstones and the other one was running the default Barrelfish domains.

Thankfully, however, if one of the HTs of a core is parked by using either of *monitor/mwait* or *hlt*, the remaining HT (on that physical core) gets a performance boost making it on par with the single-HT core (i.e., the core when Hyper-Threading is disabled): this means that, in practice, we can switch between having or not the advantages and disadvantages of Hyper-Threading at runtime, *under the control of the OS*.

While it is true that enabling Hyper-Threading (and leaving all the HTs on) takes away a third of the processing power of each HT, the total throughput of each physical core increases by up to 33%, provided that the workload on that core is parallelized enough so that both HTs can work independently. This means that, depending on the degree of parallelism, the performance observed when enabling Hyper-Threading can be between 66% and 133% of a single-HT core. Note that, by relying on *monitor/mwait* and *hlt*, the lower bound can be increased to 100% of the single-HT core performance: the insufficiently use of the parallel HTs can be detected and one of the HTs can be put to sleep (awaiting to be resumed when the workload can be better parallelized).

In Section 4.1 we explored the differences between *monitor/mwait* and *hlt* and determined that the former is a better candidate, since the duration of waking-up a sleeping HT and the cost of sending the wake-up signal were both smaller when compared to the latter option. Based on that finding, and taking into consideration that the effects of both sleeping mechanisms were indistinguishable in our Fhourstones-based experiments, we further recommend the use of *monitor/mwait*, when possible.

Finally, we would like to point out that, based on experiments ' $\underset{0}{[\text{mwait: } 1]}$ ', and ' $\underset{1}{[\text{mwait: } 0]}$ ' (and, of course, their *hlt*-based variations), the idea of using HT switching as an alternative to the classic context switching mechanism seems promising: when only one of the 2 per-core HTs is awake there is no difference in processing power when compared to the situation of having Hyper-Threading disabled on that core. The thing left to consider is how much time is takes to do the context switch, which we will explore in the next Section (i.e., Section 4.3).

4.3 Context Switching

Moving forward, we are interested in the duration of switching the virtual address space (i.e., *VAS*) of a hardware thread. For this, we executed a special benchmarking program and measured the operations of interest:

- 'NOP syscall' → the time it takes to go in and out of the kernel for executing a syscall which does nothing;
- 'cap invocation' → the duration of dropping into the kernel via a capability invocation syscall, determining the type of invocation (which, in this case, is a *nop*) and returning from the syscall. Note that the duration of 'NOP syscall' is included in 'cap invocation'. Also, as a side remark, capability invocations are normal Barrelfish syscalls which involve operations linked to capabilities;
- 'VAS switch' → the duration of performing a virtual address space switch, assuming that the target address space has already been constructed. Again, note that this measurement includes the duration of a 'cap invocation', since the interface to a context switch is represented by a capability invocation;
- 'VAS tagged switch' → as the previous operation, but with TLB (i.e., translation lookaside buffer) tagging enabled: TLB entries are tagged with a *virtual address space identifier* and are only considered valid if these IDs match the currently used virtual address space. The sought for advantage is that some entries in the TLB will not be replaced when switching between multiple address spaces and, thus, the number of compulsory cache misses is minimized. In case TLB tagging is not enabled (and no other alternative mechanism is employed), then the entire TLB needs to be flushed upon a context switch.

The evaluation that we carried out was composed of a single-threaded domain, running on an HT and performing 100 warm-up iterations and 10000 measured iterations for each operation. For the latter 2 operations (i.e., 'VAS switch' and 'VAS tagged switch') a secondary virtual address space was created and all the mapping from the domain's initial address space were copied into

Table 4.2: Durations (top) and 95% confidence intervals (bottom) of virtual address space context switching related operations. *"NOP syscall"* and *"cap invocation"* are included in *"VAS switch"* and *"VAS tagged switch"*.

Operations Durations (cycles)

HTT \ Operation	NOP syscall	cap invocation	VAS switch	VAS tagged switch
enabled	169	321	805	683
disabled	135	202	626	455

95% Confidence Interval of Operations Durations

HTT \ Operation	NOP syscall	cap invocation	VAS switch	VAS tagged switch
enabled	0.17	0.29	0.41	0.41
disabled	0.02	0.06	0.08	0.06

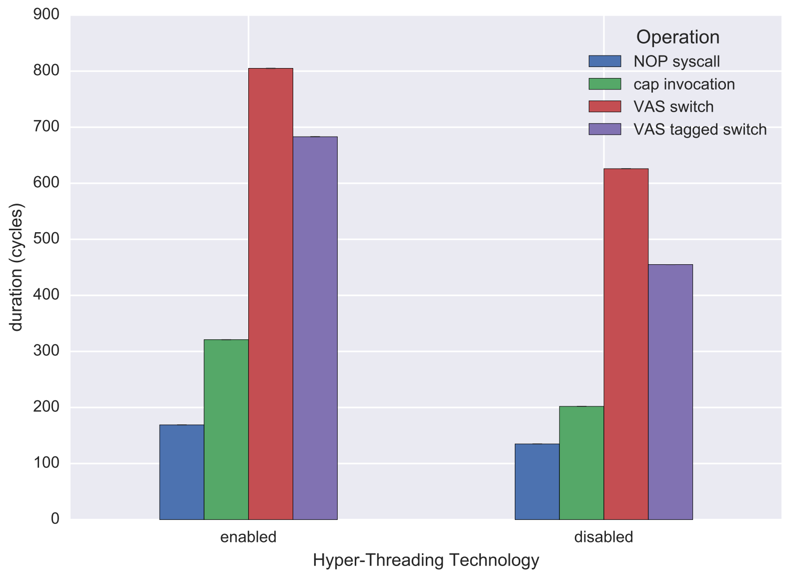


Figure 4.12: Visual representation of the data in Table 4.2: durations of virtual address space context switch related operation. Confidence interval are too small to be visible in this plot.

the new one. The VAS creation was carried out before the warm-up phase and an iteration meant that the domain switched from the initial address space to the new one, and back. The duration was measured for each of the 2 transitions, but we only report the data for switching from the new VAS to the initial one: the values were about the same and including the durations for the other transition would have just polluted our tables and plot, without generating any noticeable insight.

The recorded durations are depicted in Table 4.2 (for an exact overview of the values) and in Figure 4.12 (for facilitating relative analysis). As can be observed, we performed the previously described benchmarking process under 2 conditions: with Hyper-Threading (i.e., HTT) *enabled* and *disabled*.

First of all, we see that the durations are smaller and the 95% confidence intervals tighter in the case in which Hyper-Threading was disabled. This has come to be expected, considering the previous observations which we made in this Chapter, regarding the effect of using Hyper-Threading. Also, we can clearly see that enabling TLB tagging helps improve the duration of a context switch.

However, far more important is to compare, in the context of Hyper-Threading being enabled, the duration of a virtual address space switching (at most 805 cycles) with the wake-up duration of a hardware thread (at least 1250 cycles). Looking at the numbers, it is pretty clear that we would face a 55% penalty during each context switch if we implemented most of the alternatives proposed at the beginning of Chapter 4 (the alternatives which involve running a single HT at any given time, and alternating them as a form of context switching).

4.4 Hardware Details

All the experiments presented in this Chapter have been executed on the *vacherin* machine, which features an Intel® Xeon® CPU E3-1245 v3 with the following characteristics:

- part of the Haswell processor microarchitecture;
- 3.40 GHz base CPU frequency;
- 4 cores;
- per core L1 caches:
  - 32 KB 8-way set associative instruction cache;
  - 32 KB 8-way set associative data cache.
- per core L2 cache: 256 KB 8-way set associative;
- shared L3 cache: 8 MB 16-way set associative;
- 2-way SMT via Intel® Hyper-Threading Technology;

- although the processor should support Intel® Turbo Boost Technology, we found this feature to be unavailable on our system:
  - the associated BIOS option is grayed out;
  - executing "`cat /sys/devices/system/cpu/intel_pstate/no_turbo`" on Ubuntu returns "`1`";
  - the output of the `cpuid` command (also on Ubuntu) includes "`Intel Turbo Boost Technology = false`".

4.5 Conclusions on Hardware Threads Interaction

We have commenced this Chapter with a set of questions regarding the cost of synchronizing SMT threads, their trade-offs and the flexibility of managing them. By running a variety of benchmarking programs on top of our target hardware platform, the *vacherin* machine (see Section 4.4 for hardware details), we were able to learn the most important characteristics of Intel Hyper-Threading.

Thus, we have come to the conclusion that it takes 1250 cycles for a sleeping hardware thread to resume execution when it was parked via *monitor/mwait* and 1800 cycles when it was put to sleep by using *hlt*.

The next thing that we did was to use the Fhourstones [12] to help us understand what penalty does an SMT thread incur when it shares the physical core. It turns out that, on *vacherin*, each of the 2 hardware threads of a core is able to provide at least 66% of the single-threaded core throughput. This means that, while single-threaded performance goes down by 33%, the total (parallel) core throughput increases by 33% (depending on the inherent and exploited parallelism of the workload). Note that all percentages discussed in this Section are relative to the single-threaded core (i.e., with Hyper-Threading turned off), unless otherwise stated.

Another aspect which we observed was that, enabling Hyper-Threading and parking one of the 2 SMT lanes (via either *monitor/mwait* or *hlt*) yielded the same performance on the hardware thread that remained awake as if the latter HT was the only one on that physical core (i.e., as if Hyper-Threading was turned off).

The last thing that we looked at in this Chapter was the cost of performing a context switch (i.e., a change of virtual address space). This is important because some of the HT management models which we were planning to implement, as stated at the begging of the Chapter, would have used HT synchronization as a way of switching between tasks.

Sadly, the cost of doing a context switch (roughly 805 cycles, *without* using TLB tagging) is substantially less then what we would have to pay (in terms of wasted cycles) for resuming a sleeping hardware thread.

Based on all the aspects related to Hyper-Threading which we uncovered in this Chapter, it makes sense to conclude that, from all the proposed OS models for managing SMT lanes, only the first and the last one presented at the beginning of this Chapter make sense. These are:

- the baseline naive approach, of running a CPU driver on each SMT thread;
- the idea of running multiple user-space SMT threads at the same time, but only allowing at most one in the kernel at any given time (i.e., 2 HTs sharing a single CPU Driver).

The discarded models (i.e., 1. dedicated SMT threads for kernel and user-space; 2. SMT threads as caches for user-space dispatcher contexts, with only one active at a time; and 3. dedicated SMT thread for the *monitor*) are left aside because it takes less time to perform a normal context switch (either between user-space and kernel or between 2 different virtual address spaces).

The following chapters will tackle the challenges of implementing the 2 remaining approaches and will evaluate their advantages and disadvantages.

Chapter 5

Multi-HT CPU Driver

Based on the proposed HT models and findings presented in Chapter 4, we will dedicate this Chapter to looking into the necessary CPU Driver changes, in order to accommodate the management of multiple hardware threads.

5.1 Sharing a Kernel

There are a number of decisions that need to be made when attempting to share a single CPU Driver between multiple HTs, and we will point these out in this introductory Section. As naming conventions, we will use the terms:

- *BSHT* (i.e., *BootStrap Hardware Thread*) → the first hardware thread of a CPU Driver, which is also the HT that initialized the CPU Driver;
- *APHT* (i.e., *APplication Hardware Thread*) → secondary hardware threads, which boot in an already initialized CPU Driver.

Firstly, as one can predict given our discussion of Peters et al.'s paper in Section 3.3, we will adopt the invariant of having *at most 1 hardware thread* executing CPU Driver code at any given time. This decision is based mostly on the desire to facilitate a future formal proving process of the CPU Driver's correctness, and very little on our belief that one BIG-kernel lock is the best solution, even for a microkernel, in terms of performance. Nevertheless, the *bounded* execution (in terms of duration) of most kernel operations (as shown in Figure 5.1) supports the single kernel lock approach, as an HT will generally wait at most a *bounded* period of time before successfully acquiring the lock.

Given the decision of only allowing exclusive access into the kernel to each HT, we need to adopt a model for how the kernel stack (or stacks) will operate: this problem is tackled in Section 5.2. We need to also decide on how to enforce the exclusive access (Section 5.3) and what operations need to be carried out at the kernel's entry points (Sections 5.4, 5.5 and 5.6).

After all these are set up, we will detail how was the core booting procedure adapted in order to boot an APHT. This is done in Section 5.7, with the goal

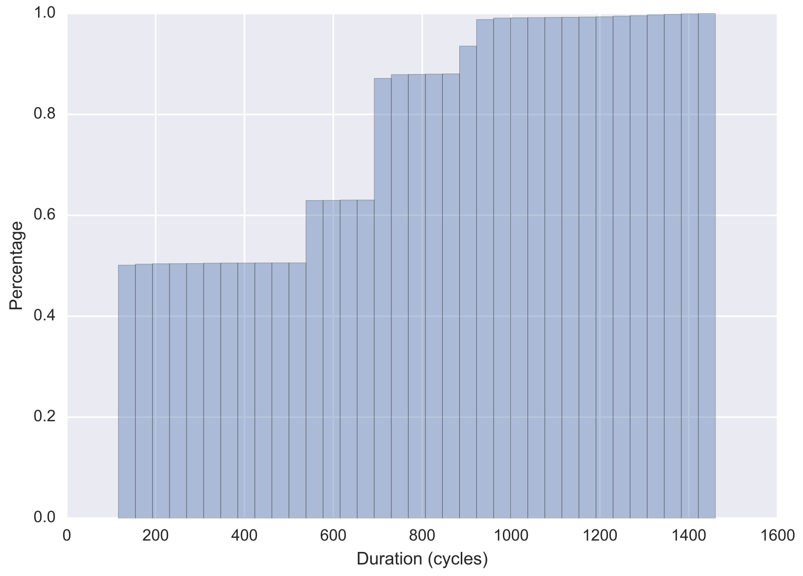


Figure 5.1: Cumulative histogram of the quickest 99% of the measured *syscall*s and *interrupt & exception handling*s. Barrelfish was in its *idle* state when these measurements were taken. Also, we would like to point out that the remainder 1% of *syscall*s took much longer (up to 12, 359, 892 cycles): these are operations which process an arbitrary amount of data (e.g., setting up a virtual address space) whose implementation *could* be converted to portions of code executing in bounded time, but are currently implemented to run in a single stage.

(for now) of being able to get a secondary HT into an existing CPU Driver, after which the APHT in question will leave the kernel and go to sleep. The work which extends the capabilities of an APHT to being able to execute user-space domains is presented in Chapter 6.

We mentioned that the CPU Driver will be shared by multiple HTs, which means that the kernel's data will be also shared. While this is fine for most of the kernel's data structures, some of them are tied to a specific hardware thread. Thus, for each HT, we have an HTCB (i.e., hardware thread control block), meant to encompass the HT-specific data. The most important members of the HTCB are presented in Section 5.8. This section is placed towards the end of the Chapter in order to ease the motivation regarding why something has been added (or not) to the HTCB.

5.2 The Kernel Stacks

When there are multiple hardware threads sharing the same kernel, but never accessing it at the same time, there are two main approaches one can take regarding stack management:

1. have a shared stack, ensuring an HT can gain exclusive access to it;
2. use separate, per-HT, stacks.

In between these two options, there are additional alternatives which tend towards one side or to the other: sharing a stack sometimes and having separate stacks other times.

Our first decision was to choose one of the two main trends: weighting in the reduced memory footprint when using a single, shared stack, we chose this approach. Using multiple stacks would have eased the implementation, but we considered runtime performance and a clean design more important aspects.

The simplest manner of ensuring that at most one HT can access the shared kernel stack at any given time is to require that the HT in question holds the kernel lock (details about the lock in Section 5.3). Thus, the correct order of these operations is to first acquire the kernel lock and, only afterwards, to make use of the stack.

From the 3 kernel entry points, only one saves data on the stack before giving control to the kernel code: the interrupts and exceptions entry point (expanded upon in Section 5.6). Since the amount of saved data is insignificant compared to the default Barrelfish kernel stack size (40 bytes, or 48 bytes for some exceptions, vs 16 kB), we decided to use small per-HT stacks during periods when stack operations could not be overlapped with the kernel lock being held by the HT.

A similar situation, which also requires per-HT kernel stacks, is the action of exiting the kernel and returning to user-space via *iretq*. The reason for this is that the *iretq* processor instruction pops some register values (as shown in Figure 5.2) from the stack. Since an HT releases the kernel lock *before* returning to user-space (via *iretq*, for our current scenario of interest), a race-condition is created between the exiting HT looking up data on the shared stack and another entering HT which modifies the stack by pushing data onto it.

In both cases presented in the previous 2 paragraphs, the solution we chose and implemented is to switch between the private and shared stacks, by changing the *RSP* register and copying data from the old stack to the new one. For example, in the situation of exiting the kernel via *iretq*, we set the *RSP* register to point to the private stack, copy over the stack frame depicted in Figure 5.2 (in Section 5.6) from the shared stack to the private one, release the kernel lock and, in the end, execute *iretq*.

5.3 The BIG-Kernel Lock

At this point, the idea of using a lock around the kernel comes natural, as a way of ensuring exclusive access to the CPU Driver for a single hardware thread at a time.

We started by using a *spinlock*, implemented as a loop that executes *pause* after failing to acquire the lock (and before retrying). The usage of *pause* and the placement of the locking variable in an 128 bytes block of memory (which is also 128-byte aligned) are techniques recommended by Intel in their Optimization Reference Manual [5].

For the actual lock acquiring, we use an *atomic compare and exchange* assembly instruction (i.e., *lock cmpxchg*).

The lock is implemented entirely in assembly and takes a single register argument: the address of the HT's HTCB, in *RBX*. Because we opted for a single kernel stack per CPU Driver, which may not be used before acquiring the kernel lock, the address to jump to after the lock has been acquired is stored in the HTCB (and is set before starting the execution of the lock acquiring procedure). Also, since kernel code preceding the lock acquiring procedure must use some registers (e.g., saves a set of registers and uses them to determine the address of the HTCB), the lock acquiring procedure assumes that it can clobber the *RAX* register.

By using the previously described locking interface, we were able to abstract away the actual implementation of the lock: replacing the spinlock with a queue-based lock may requires at most minimal changes to the kernel entry points' code. Regarding this replacement, the person making the change needs to ensure that all the memory stores performed by an HT exiting the kernel are seen by any other HT before they see the lock being released: this ensures that a new HT entering the CPU Driver will find the shared kernel state in a quiescent state.

5.4 The CPU Driver's Boot Entry Point

When booting a Barrelfish CPU Driver, the first kernel code executed by a hardware thread is the one in `boot.S`. Because the memory location of the

APIC ID may not be mapped in yet, but thanks to the fact that most registers are not used for other purposes, we retrieve the APIC ID by using the *cpuid* assembly instruction:

1. $EAX \leftarrow$ `0x0B`;
2. execute *cpuid*;
3. the APIC ID is stored in EDX;
4. the values of EAX, EBX and EDX have been modified to store other information provided by the *cpuid*'s 0BH leaf.

The HT's index in the kernel is derived by selecting a suffix (i..e, a substring containing the least significant bits) of the APIC ID, according to a compile-time specified variable (i.e., a define):

- *NUM_HTS_PER_KERNEL* is the name of the constant indicating the maximum number of HTs a CPU Driver should be able to accommodate;
- the least significant $\log_2(NUM\_HTS\_PER\_KERNEL)$ bits of the APIC ID represent the HT's index in the CPU Driver:

  $$\text{APIC ID} = \overline{dd\ldots d}_{10} = \overline{bb\ldots b\,\underbrace{bb\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots b}_{log_2(NUM\_HTS\_PER\_KERNEL)\text{ bits}\rightarrow\text{HT's index}}}_2$$

  The reason why we use a suffix of the binary representation of the APIC ID for HT indexing is that this is the actual format of the APIC ID: the APIC ID is a concatenation of groups of bits, which identify an SMT lane on a core, a core on a socket and so on, for as many levels as the CPU hierarchy has.
- the APIC ID's bits not belonging to the HT index (i.e., the ones colored in red in the previous equation) are the same for all the HTs sharing a CPU Driver.

Knowing the HT index, the size of an HTCB and the base of the array holding all the HTCBs of a CPU Driver, it is straight forward to determine the address of a hardware thread's HTCB: the said array of HTCBs is indexed by the HT index. Note that, by requiring an additional step of indirection (i.e., read the address of an HTCB from an array containing the *addresses* of all HTCBs, as opposed to the array containing the HTCBs themselves), the constraint of having all the HTCBs in a statically allocated array can be relaxed: when booting an APHT in a CPU Driver, that new hardware thread can fill in (into the array containing HTCB address) the address of its HTCB.

Following the retrieval of the HTCB's address, the kernel entry continues by executing the lock acquiring procedure presented in Section 5.3.

Listing 5.1: Determining the address of an HT's HTCB at the syscall entry point. A similar method is also used at the IRQ entry point.

```
 1  movq $(X86_64_XAPIC_ID_PHYS + PHYS_TO_MEM_OFFSET), %rbx
 2
 3  movl (%rbx), %ebx
 4  /* %rbx[31..24] = APIC ID */
 5
 6  shrl $24, %ebx
 7  /* NUM_HTS_PER_KERNEL is a power of 2 */
 8  andb $(NUM_HTS_PER_KERNEL - 1), %bl
 9  /* %rbx[7..0] = HT index */
10
11  shlq $3, %rbx  /* %rbx = %rbx * sizeof(uintptr_t) */
12  addq addr_addrs_htcbs(%rip), %rbx
13  movq (%rbx), %rbx
14  /* %rbx = address of HTCB */
```

5.5 The CPU Driver's Syscall Entry Point

The main difficulty of adapting the syscall kernel entry point (in `entry.S`) to simultaneously accommodate multiple HTs is the scarcity of usable registers at this point of execution: in the single-HT version of the CPU Driver, all the general purpose registers were used to transfer user-space information to the kernel. In order to overcome this issue, we reserved a general purpose register (*RBX*) for HTCB address computation upon kernel entry.

The method used to retrieve the APIC ID in the boot entry point (via *cpuid*; see Section 5.4) is not a suitable option in the present context, as it overwrites 4 GP registers when reading the APIC ID. However, the syscall entry point has the added advantage of only being used *after* the CPU Driver has been initialized. Thus, we can ensure (during the kernel's initialization) that the APIC ID's memory location (i.e., physical address `0xFEE00020`) is mapped in.

So, up to this point of the syscall entry point, we managed to read the APIC ID into *RBX* and perform a bitwise *and* in order to isolate the bits of the HT's index. The actual assembly code for these operations is presented in Listing 5.1, lines 1 – 9.

The next problem is determining the appropriate HTCB, considering that the single available register (i.e., *RBX*) already contains information (i.e., the index of the HT, derived from the APIC ID).

While we could have reserved *more* registers for the purpose of HTCB address computation, we decided to avoid this solution: a microkernel is (likely) required to service a large number of system calls (typically more than an equivalent monolithic kernel) and we wanted to take away as few registers as possible from the set of registers usable for syscall argument transfer. Reducing the num-

Listing 5.2: The initialization of auxiliary variables meant to facilitate the computation of an HTCB's address at the syscall entry point. The same information is also used at the IRQ entry point.

```
struct htcb htcbs[NUM_HTS_PER_KERNEL];
struct htcb *addrs_htcbs[NUM_HTS_PER_KERNEL];
struct htcb **addr_addrs_htcbs;

/**
 * addr_addrs_htcbs is initialized by the BSHT in the
 * CPU Driver's 'text_init' function:
 */
addr_addrs_htcbs = &addrs_htcbs;

/**
 * Function used to register an HTCB for the HT with
 * the given APIC ID.
 */
void htcb_register(struct htcb *htcb, uint8_t apicid)
{
  addrs_htcbs[apicid & (NUM_HTS_PER_KERNEL -1)] = htcb;
}
```

ber of register-stored arguments would have increase stack spilling and would have reduced the amount of data transferable via an LMP syscall.

Given the mentioned constraints and performance trade-offs, we used a number of auxiliary variables and managed to compute the address of the HTCB by reserving a single GP register. These auxiliary variables are shown in Listing 5.2, alongside details regarding their initialization.

Basically, what we did was to use an additional level of indirection (as was suggested at the end of Section 5.4), by taking advantage of the fact that one of the operands for (many) x86 assembly instructions can be referenced in memory.

Thus, we exploited the known pointer size (i.e., 8 bytes on x86-64) in order to exchange a multiplication (i.e., "HT index $\times$ *sizeof*(struct HTCB)") with a left bit shifting: see line 11 of Listing 5.1. Of course, the bit shifting did not compensate for the entire multiplication, so we needed 2 additional assembly instructions: the ones on lines $12 - 13$ of Listing 5.1.

5.6 The CPU Driver's Interrupts and Exceptions Entry Point

This kernel entry point generated difficulties for our implementation because:

- the context switching mechanism used when an interrupt or an exception

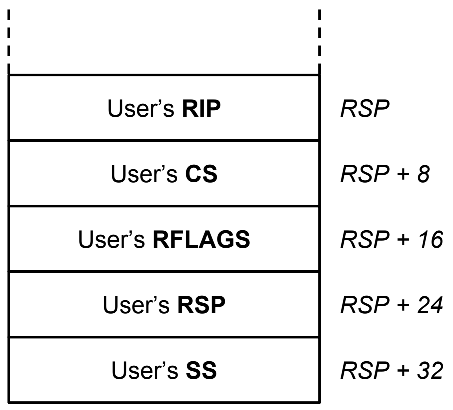


Figure 5.2: Top stack frame expected by *iretq* (i.e., when exiting the kernel after an exception had occurred). The same stack frame is created when the kernel begins servicing an interrupt or some exceptions. Other exceptions push an additional error code after the user's RIP.

occurs pushes a number of registers onto the stack: *SS*, *RSP*, *RFLAGS*, *CS* and *RIP*. Thus, before any kernel code is executed, the stack frame looks as in Figure 5.2;

- there is no way to reserve registers, in a manner similar with what we did for the syscall entry point. This restriction should be easy to understand, given that the hardware does not save any other registers than the ones mentioned at the previous bullet point. Also, when and interrupt or an exception occurs, the CPU immediately switches execution into the kernel, without running any user-space code in-between.

Firstly, we note that, regardless of how we would have chosen to gain a GP register for the purpose of computing the HTCB's address, there was still the need for a private stack upon kernel entry: the hardware pushes data onto the stack. Thus, we had to give each HT a private stack, large enough to accommodate the said pushed data.

In order to specify what stack each HT should use, we took advantage of the fact that each entry in the IDT (i.e., Interrupt Descriptor Table) specifies at which address the stack should start when servicing the associated interrupt/exception. The stack's address is not placed directly in the IDT entry, but is specified as an index in the *ist* (i.e., Interrupt Stack Table) field of the TSS (i.e., Task State Segment).

There are 7 slots available in a TSS for specifying different stacks, so, from this perspective, we could have used the same TSS for all the HTs. Of course, such a decision would have limited us to at most 7 hardware threads per CPU Driver or force us to allocated another TSS for each 7 HTs (which would have increased code complexity without providing a significant advantage).

After weighting the trade-offs, we decided that the added flexibility and simplicity that comes with having a TSS per HT was more important. If deemed appropriate, reducing all the TSSes to a single one can be implemented after we have experimented with different ways of incorporating the idea of multiple HTs in a single CPU Driver.

Having secured per HT kernel stacks (at the beginning of interrupt/exception handling) opens the door for a simple solution for gaining the necessary GP register in order to compute the HTCB's address: we push the *RBX* register also onto the private stack. Next, the HTCB's address is determined as explained in Section 5.5 and the kernel lock is acquired as presented in Section 5.3. Afterwards, we switch from the private stack to the shared kernel stack (to which we now have exclusive access), making sure we copy the data from the old stack to the new one.

As we have pointed out in Section 5.2, the *iretq* instruction also uses the private HT stack, in order to avoid a race condition with another HT entering the CPU Driver.

5.7 Booting an Application Hardware Thread

For the purpose of booting an APHT into an existing CPU Driver (which was previously initialized by a BSHT) we started from the mechanism which boots an APP (i.e., application processor; any but the first CPU core which was started when the system had been powered on).

In Barrelfish, the utility used to boot an additional CPU is called *corectrl* and is structured as a collection of subcommands which are passed through the command line. For our use case, we added a new subcommand with the following format:

```
corectrl -m bootapht <APHT Core ID> <BSHT Core ID>
```

The "`-m`" flag tells *corectrl* to not wait for a *monitor* message at the end of the subcommand's execution: this is to be expected, since we do not create a new *monitor* instance.

The 2 arguments (i.e., "*APHT Core ID*" and "*BSHT Core ID*") identify the HT to boot and the CPU Driver that will adopt the new APHT (by specifying the CPU Driver's BSHT). These IDs can be determined by examining the output of the *corectrl*'s *lscpu* subcommand. An example of such an output generated on the *vacherin* machine is presented in Listing 5.3. The 2 IDs expected by *bootapht* are assigned to CPUs as they are discovered and added into Barrelfish's *SKB* (i.e., System Knowledge Base).

When an APP boots (regardless if it is a BSHT or an APHT) it will go through an assembly sled meant to initialize some basic hardware functions: for example, specifying a GDT and enabling paging. The purpose of this assembly code is to enable the booting CPU to jump into a CPU Driver and start executing it.

Listing 5.3: An example of the output of the "*corectrl lscpu*" command on *vacherin*.
The *Barrelfish ID* is used to identify each CPU Driver, with the valid range of possible values being 0 – 254: Barrelfish ID 255 means that the associated HT is reserved to be used as an APHT, so it will not have its own CPU Driver, but will share one with a BSHT (and, possibly, with other APHTs).
Note that in this scenario it is considered that there are 2 BSHTs (i.e., CPUs 0 and 2) and that each CPU Driver can accomodate a total of 4 HTs.

```
> corectrl lscpu
spawnd.0.0: spawning /x86_64/sbin/corectrl on core 0

CPU 0: APIC_ID=0 PROCESSOR_ID=0 BARRELFISH_ID=0 ENABLED=1
CPU 1: APIC_ID=2 PROCESSOR_ID=1 BARRELFISH_ID=255 ENABLED=1
CPU 2: APIC_ID=4 PROCESSOR_ID=2 BARRELFISH_ID=1 ENABLED=1
CPU 3: APIC_ID=6 PROCESSOR_ID=3 BARRELFISH_ID=255 ENABLED=1
CPU 4: APIC_ID=1 PROCESSOR_ID=4 BARRELFISH_ID=255 ENABLED=1
CPU 5: APIC_ID=3 PROCESSOR_ID=5 BARRELFISH_ID=255 ENABLED=1
CPU 6: APIC_ID=5 PROCESSOR_ID=6 BARRELFISH_ID=255 ENABLED=1
CPU 7: APIC_ID=7 PROCESSOR_ID=7 BARRELFISH_ID=255 ENABLED=1
```

In the case of starting an APHT, we need some information from the CPU Driver which will manage this HT:

- the kernel's entry address $\rightarrow$ in order to know from which address to start executing kernel code;
- the address of the PML4 page table $\rightarrow$ to ensure the same page mappings for all the HTs who share a CPU Driver.

These addresses are retrieved by doing an RPC to the *monitor* of the core on which *corectrl* is executing. If that core does not have the same CPU Driver as the target BSHT (i.e., the BSHT with which we want the new APHT to share the kernel), then the *monitor* uses the inter-monitor binding to get the information from the appropriate *monitor*: note that the *monitor* is the user-space extension of the CPU Driver and, for this reason, there is a single *monitor* per CPU Driver instance.

The initialization code of the CPU Driver is able to discern between a BSHT and an APHT, so it can take the appropriate actions. For example, only the BSHT needs to add a memory mapping for the physical address of the APIC ID: the APHT gains that mapping by using the same page table.

As mentioned in Section 5.1, we only focus in this Chapter on bringing an APHT into an already initialized kernel. The way we use an APHT in order to execute user-space domains is presented in Chapter 6.

5.8 The Hardware Thread Control Block

The HTCB is implemented as a C structure and is meant to encompass each hardware thread's specific data. Thus, it seems natural to add in this control block the data structures that are directly linked to hardware:

- the HT's IRQ (i.e., interrupt/exception handling) stack → we discussed the need for such a stack in Sections 5.2 and 5.6, pointing out that the context switching mechanism employed when servicing and interrupt/exception needs to use a stack, without allowing the programmer to ensure exclusive access to the shared kernel stack.

  Regarding the IRQ stack size, we mentioned that at most 48 B would be pushed upon kernel entry, when servicing an exception which specifies an error code. The size which we actually allocated for said stack is slightly larger: 80 B.

  First of all, the hardware expects the IRQ stack to be 16-bytes aligned. Although it can skip part of the stack to ensure this alignment, we thought it would be better to prevent this memory loss and declare the member in the HTCB C structure so that it fulfills the requirement. Thus, we specified the IRQ stack to be 16-bytes aligned (by using "`__attribute__ ((aligned (16)))`") and also made sure that the size is a multiple of 16 bytes (since stacks grow from large addresses to smaller ones).

  Secondly, we needed 3 additional slots on the stack for:

  - specifying the interrupt/exception's vector number (used to determine which interrupt/exception had occurred);
  - the return address needed when calling a helper function. The said function is responsible with determining the HTCB's address, acquiring the kernel lock and switching to the shared kernel stack;
  - saving the *RBX* register.

  So, we need $48 + 3 \times 8 = 72$ bytes, which rounds up (as a multiple of 16) to 80 B.

- the *Task State Segment* → details that lead to incorporating this data structure in the HTCB are presented in Section 5.6 and are related to the private interrupt/exception handling stack;

- the *Global Descriptor Table* → Barrelfish uses a static GDT for each CPU Driver, meaning that there is a fixed number of entries which are initialized during kernel setup. The only entries that may change are the ones related to the *Local Descriptor Table*, when doing a context switch. Thus, having a private GDT for each HT made it easier to adapt the CPU Driver's logic, in order to accommodate multiple HTs;

- the *Interrupt Descriptor Table* → while most of the entries of the IDT are the same for all the HTs sharing a CPU Driver, there may be situations in which we want to have an interrupt vector for a specific HT: for example, if an HT goes to sleep via *hlt*, waking it up requires some other HT to send an IPI. Thus, we may only want to have the vector (for the wake-up IPI) valid when the HT is actually asleep.

We would like to point out that it is possible to keep some of the previously mentioned HTCB members shared between all the HTs of a CPU Driver. However, we saw 2 main reasons for making them HT-private:

1. Sharing them would have made the process of experimenting with different CPU Driver designs more difficult, because it may have made it more cumbersome (or even impossible) to implement specific features: consider, for example, if we wanted to enable HT specific interrupt handling and there was a single IDT per CPU Driver;

2. As there is no need to keep the data (in the previously mentioned data structures) synchronized between all the HTs (because they are mostly static), having to share these structures would have required more significant changes to the way the CPU Driver manages them: for example, having a single GDT has the downside that it must be able to accommodate distinct TSS and LDT slots for each HT.

Other fields of the HTCB structure are:

- an integer holding the APIC ID → this value is different for each HT;
- a boolean revealing if the HT is also the BSP (i.e., bootstrap processor);
- a pointer to the *Dispatcher Control Block* of the last dispatched task → HTs simultaneously dispatch different user-space tasks (the implementation of this functionality is detailed in Chapter 6);
- pointers to the currently dispatched task and to the last dispatcher that made use of the FPU → similar motivation as in the previous bullet point;
- a boolean which is only *true* if the HT is the BSHT;
- slots for saving registers → employed when switching tasks (in the IRQ entry point), after the kernel lock had been acquired: the registers are needed in the loop that copies data from the private stack to the shared one;
- the base address and the size of the currently used *Local Descriptor Table* → in order to prevent the rewriting of the GDT entry referencing the LDT, when the *same* LDT is used;
- additional boolean and integer fields related to the CPU Driver's logic → these are used in similar ways as the previously mentioned ones, in the sense that they maintain the state of a given HT.

5.9 Using a Multi-HT CPU Driver

Having a working implementation of a CPU Driver that is able to simultaneously manage multiple HTs is not the main target for this Thesis, but rather a tool to be used in order to improve certain aspects of Barrelfish (e.g., increase the throughput of executed user tasks).

At the beginning of Chapter 4 we have proposed a number of models for using multiple HTs which share a CPU Driver. Running experiments and analyzing the generated data, we have come to the conclusion that only some of the initial models make sense, primarily because of the overhead of synchronizing hardware threads: Section 4.5 draws the related conclusions.

This Chapter has presented the process of adapting a normal (i.e., single-HT) CPU Driver to a multi-HT context. Based on this capability, and taking into consideration the lessons learned in Chapter 4, we will dive into the challenges of exposing the additional processing power to user-space in Chapter 6.

Chapter 6

User Domains on Top of a Multi-HT CPU Driver

In this Chapter, we will continue building upon the multi-HT CPU Driver (presented in Chapter 5) by enabling all the hardware threads to execute user-space code. Thus, we will start from the extension of Barrelfish which is able to boot additional HTs into an already initialized CPU Driver and to perform the necessary setup for these HTs. However, the modifications which we have presented in the previous chapters do not allow these HTs to do any useful work, as they are halted after executing the kernel initialization code path.

The target which we are now aiming for is to be able to execute multiple user-space dispatchers at the same time, one on each HT managed by a CPU Driver.

The following Sections focus on specific areas which are of significant importance for running multiple user-domains at the same time, on top of a multi-HT CPU Driver. Towards the end of this Chapter (i.e., in Section 6.6), we discuss some experiments meant to explore differences between our modified version of Barrelfish and some setups involving single-HT CPU Drivers. Based on the observations generated by said experiments, we propose a scheduler optimization in Section 6.7.

6.1 Scheduler

The initial plan was to start the implementation by adapting the RR (i.e., Round-Robin) Scheduler, as it has a simpler logic when compared to the RBED (i.e., Rate-Based Earliest Deadline) Scheduler [3]. Unfortunately, attempting to compile the CPU Driver wit the RR scheduling policy resulted into a compilation error: there exists a portion with unimplemented KCB (i.e., Kernel Control Block) related logic.

Considering that the missing code is not part of this Thesis's focus and that we would have exchanged the RR Scheduler with the RBED one at some future moment, we decide to start directly with RBED.

Analyzing the logic used by the RBED Scheduler for deciding the next domain to be dispatched, the primary multi-HT adaptation issue that stood out was the time value associated with a particular moment. The problem lies in the fact that each HT has an independent timestamp counter. Considering that the system's real-time clock has a much lower resolution and that there is no direct way of determining the time offsets between the HTs (aside from executing a dedicated algorithm that uses low-latency thread synchronization operations), we chose to assign the role of the time keeper to the BSHT (i.e., bootstrap hardware thread) of each CPU Driver.

Thus, this means that only the BSHT updates the value of the *kernel_now* variable, which stores the number of milliseconds since the BSHT booted-up. Compared with implementing and executing a timestamp synchronization algorithm, our choice was faster to implement and gave the chance to see how far we can get with such a simplistic approach: the schedulers implementation includes time-related assertions, so we would have a clear indication if something went wrong (e.g., missed deadline), in the form of an assertion error. So far, no such timing exception has been raised.

In this Section, we have pointed out issues and solutions related to *being able to use* the RBED scheduler in a multi-HT CPU Driver. However, a more interesting and vast research area is represented by the analysis of potential *optimizations* of the scheduling logic. We discussed about this and have dedicated a section to such an optimization proposal (i.e., Section 6.7). However, given the time constraints for this Thesis, we decided it is best to leave the scheduler optimization topic for future work.

6.2 Domains, Dispatchers and Hardware Threads

As explained in Barrelfish's Architectural Overview Manual [1] (in the section dedicated to dispatchers), an application wanting to execute on a particular core (i.e., on top of a CPU Driver) has to have a dispatcher on that core. This means that the dispatcher represents "*the unit of kernel scheduling*" and that dispatchers are owned by a single CPU Driver. As a side-note, we would like to point out that applications requiring threads in order to implement their logic are not obliged to use multiple dispatchers: green threads are implemented as part of *libbarrelfish*.

Transitioning to a context in which a CPU Driver is shared by multiple HTs, we need to address the relation between dispatchers and HTs. This boils down to answering 2 questions:

1. Should a dispatcher be pinned to a particular HT or can it move freely between HTs, based on current CPU usage?

2. Can a dispatcher be *simultaneously* used by multiple HTs or should it be owned by at most a single HT at any given moment?

Focusing on the first question, it is clearly desirable to be able to use the available HTs at their maximum potential, with as little constraints imposed on the scheduler as possible. However, it may be a requirement of the application to always be executed by the same HT, in order to, for example, be able to receive interrupts at a fixed APIC ID. Thus, a promising solution is to allow dispatchers to migrate between HTs (of the same CPU Driver), but have the option of pinning them to a particular HT.

Regarding the second question, allowing multiple HTs to simultaneously use the same dispatcher creates concurrency problems and the need to rethink a dispatcher's design and functionality (as it would no longer be the unit of kernel scheduling). Considering this, alongside the facts that per-HT dispatcher data would be required and that no extra functionality would be added, we decided to make a dispatcher available to a single HT at a time.

So, summarizing this section, in the multi-HT CPU Driver extension of Barrelfish, dispatchers can move between HTs as the scheduler decides best (this is the default option, but the functionality of pinning to a particular HT is available). Also, a dispatcher can have at most 1 owning HT at each given moment. Implementation-wise, we have wrote part of the pinning mechanism, which is required to keep track of a dispatcher's HT owner: no current Barrelfish domain makes use of HT pinning and the remaining code is trivially implementable. Taking ownership of a dispatcher is as simple as a kernel mode assignment, because at most a single HT can execute kernel code at each moment.

Thus, we leave to future work the study of trade-offs of multi-HT dispatchers and other domain-dispatcher-HT setups.

6.3 Core Local RPC and Message Passing

Even before implementing the initial version of a multi-HT CPU Driver (that can dispatch multiple user domains *simultaneously*), we suspected that there will be problems with the LMP (i.e., local message passing mechanism, meant to be used for transferring messages between user domains executing on top of the same kernel).

Local Message Passing in a Single-HT CPU Driver

Previously, *at most a single user domain* ran at each moment of time, on a particular core. This means that no user domain ran when the only HT managed by a CPU Driver was executing in kernel mode.

When a domain wanted to send a message to another domain (on the same core), the sender could just drop into the kernel, perform some sanity checks (e.g., if the capability referencing the receiver pointed to a valid domain, if there was enough space in the receiver's buffer) and just dispatch the receiver domain. Thus, assuming that the sender had initiated a *valid* message passing request (i.e., valid receiver, valid destination slot), the control would be transferred to the receiving domain.

Note that this did *not* mean that the transfer would also be successful: the receiver's buffer could have been full or the receiver domain could have been disabled (i.e., in a state in which the domain can not receive messages). In such a case, an error would be reported to the sender, but the receiver domain would still be the one dispatched upon kernel exit.

Adapting Local Message Passing to a Multi-HT CPU Driver

Having multiple HTs based on the same CPU Driver means that more than one user domain can execute code at a given time. For LRPC (i.e., local remote procedure call) & LMP, this translates into concern for a situation in which the receiver domain is already executing (on another HT).

One solution to this problem, which is currently implemented, involves defining a new error code for such a scenario: *SYS_ERR_LMP_TARGET_RUNNING*. This error code is returned to the sender domain in a manner similar the one in which the sender is informed that the receiver domain does not exist. Thus, the burden of retrying the message passing falls onto the sender domain. Depending on the latency constraints for that particular inter-domain data transfer, the sender can loop and retry until the transfer emerges successfully, or it can yield the HT to another (user-space) thread or to another domain.

For non-urgent message passings, yielding is a good option. It can be improved by making the CPU Driver save the message in some buffer: the normal receiver buffer can only be used if it is *read-only* for the receiver and if the said buffer is expected to be modified (by the message delivery mechanism) while the receiver is running.

If the CPU Driver has nowhere to store the message while the receiver is running, 2 other alternatives can be employed:

1. The sender's HT can sleep (via *monitor/mwait* or *hlt*) until the receiver's HT can release the receiver dispatcher. The advantage would be that other HTs may use the sender HT's hardware resources to accomplish useful tasks, better than the sender wasting them by looping on the failing send operation;

2. Ultimately, if latency is critical for the transfer in question, then the sender HT can send an IPI (i.e., inter-processor interrupt) to the receiver's HT, making the receiver domain available for execution on the sender HT.

Having the previously mentioned alternatives in mind, we started experimenting with booting-up more than 2 HTs per CPU Driver. What we observed was that the system would stall and that domains would make no further progress when reaching 4 HTs. Listing 6.1 contains the output of the *ps* command (note that *ps* does not display information about special Barrelfish domains, such as *init*, *monitor*, *mem_server*, *ramfsd*, *skb*).

The reason for this deadlock was the enormous rate ($\approx 100\%$) of failing LMP transfers, caused by the fact that the receiver was running on a different HT than the sender. We solved the problem by implementing a mechanism which

Listing 6.1: *ps* output on Barrelish's BSP core.

```
> ps
DOMAINID  STAT  COMMAND
1         Z     corectrl auto boot 2 -a  loglevel=4
2         R     pci auto
3         R     serial
4         Z     angler serial0.terminal xterm
5         R     lrpc_bench server
6         R     fish
```

puts APHTs (i.e., application hardware threads) to sleep. This mechanism is detailed in Section 6.4.

Optimizing Local Message Passing to a Multi-HT CPU Driver

However, far more important was the observation that, in order to have a better utilization of the processing power provided by HTs, we need to have a model which can deliver LMP messages *even when the target domain is executing.*

As *struct lmp_endpoint_kern* (presented in Listing 6.2) is modified by both the CPU Driver (when delivering an LMP message) and by the user-space domain (when storing an LRPC payload or when updating the CNode reference for storing capabilities transfered via LMP), we think the best solution would be to have a separate buffer, meant to store LMP messages received while the target domain is running. The alternatives would be to either use *atomic operations* when modifying the buffer or to protect it with a *lock*. Since an LMP message can contain both *normal payload of variable length* (i.e., binary, opaque data, from the kernel's viewpoint) and *a capability*, the type of needed atomic operations is not actually available. Using a succession of simple atomic operations is also not a good solution, because it could leave the buffer in an inconsistent state at specific moments. As for the locking strategy, it creates the possibility of user-space blocking the kernel, by acquiring the lock and holding it indefinitely.

The instance of *struct lmp_endpoint_kern* meant to store the LMP messages received while the target is running (on another HT) would, ideally, only be modifiable by the CPU Driver. However, this is not a must, as a user domain modifying said data would only hurt itself.

Another choice we have to make when working with 2 LMP buffers is the way in which the user domain will get access to both of them, in order to process the messages. Rotating the 2 buffers was an idea, but it was discarded because it can lead to message reordering. Thus, we decided to go with the approach of copying messages from the kernel's LMP buffer to the other buffer. This can be done when the LMP channel's dispatcher is not currently dispatched: for example, when the domain is being evicted from the HT. If message latency is of concern, an IPI can be sent to the target HT, in order for it to do the message

Listing 6.2: The C *struct* in which LMP messages are delivered.

```
/// Incoming LMP endpoint message buffer
struct lmp_endpoint_kern {
///< CSpace address of CNode to receive caps
capaddr_t      recv_cptr;

///< Slot number in #recv_cptr
capaddr_t      recv_slot;

///< Valid bits in #recv_cptr
uint8_t      recv_bits;

///< Position in buffer (words delivered by kernel)
uint32_t     delivered;

///< Position in buffer (words consumed by user)
uint32_t     consumed;

///< Buffer for async LMP messages
uintptr_t    buf[];
};
```

copying.

In this Section, a large number of solutions have been proposed, for a variety of encountered problems. We highlighted the most important characteristics and categorized them as pluses and minuses, but only implemented some of the ideas. The study of practical scenarios showcasing the advantages and disadvantages of each approach is a good candidate for an extension to this Thesis.

6.4 HT Management

In the following paragraphs we will look into the logic that triggers the actions of *booting* an HT, *resuming* an HT and *putting* it *to sleep*.

Starting with the first action, we would like to point out that there is a notable difference between the ways in which BSHTs and APHTs are added to a Barrelfish instance. The former (i.e., BSHTs) are booted automatically upon discovery, by the *kaluga* domain, which issues the *boot* subcommand of the *corectrl* domain. Also, as the naming suggests (i.e., bootstrap hardware threads), each BSHT gets an uninitialized copy of the CPU Driver.

On the other hand, APHTs are currently booted up by executing the *bootapht* subcommand of *corectrl* manually: implementing the same auto-trigger as in the case of BSHTs is a trivial task, but we wanted to make the exploration of different HT-to-CPU Driver assignments as straightforward as possible. The fact that there is currently no piece of coded logic which boots APHTs means that *kaluga* does not need to be recompiled in order to explore different HT setups: you just point an APHT towards an initialized CPU Driver by issuing *corectrl*'s *bootapht* subcommand in *fish* (i.e., the de-facto Barrelfish shell).

HTs go to sleep by executing the *wait_for_interrupt* function, which has been adapted to release the kernel lock before running *hlt*. For BSHTs, this can only happen when they have no domain to dispatch. Since the BSHT is the *time keeper* of its CPU Driver, the APIC timer is never masked when a BSHT goes to sleep: the BSHT will be woken up, at latest, by the next timer interrupt.

For APHTs the same hold true, with the minor difference that their timers are masked when going to sleep. Additionally (from the case when no domain is available for dispatching), other situations can also make an APHT go to sleep:

- if the number of subsequent LMP messages that fail with *SYS_ERR_LMP_TARGET_RUNNING* (with not interleaved transfers that end differently) exceeds a predefined limit (currently set to 1,000,000). This happens because we can not deliver an LMP message to a running domain;
- when the BSHT is forced to sleep because it has no domain to dispatch, it raises a condition in the CPU Driver which tells the first APHT to notice the flag that it should (temporally) halt execution. The mentioned condition is checked by APHTs during each timer interrupt handling. We made this decision because it is better to keep the BSHT awake whenever possible (as opposed to delegating the user tasks to an APHT), since it already wakes up at the end of each time slice.

As we already mentioned, a BSHT wakes from sleep as a result of an interrupt and it sleeps with the timer unmasked in order to ensure that an interrupt occurs at least once a time slice. This is mainly due to its important role in the CPU Driver, similar to that of a leader. On the other hand, APHTs are only useful if they can exploit the parallelism of user-space tasks, improving the throughput of the system. Thus, an event which can be taken into account when deciding to wake up an APHT is that of a dispatcher being marked as runnable: this holds true because we are still in the context of single-threaded dispatchers, which can be executed by at most an HT at a time.

The logic leading to an HT entering parking state is depicted in Figures 6.1 (for BSHTs) and 6.2 (for APHTs).

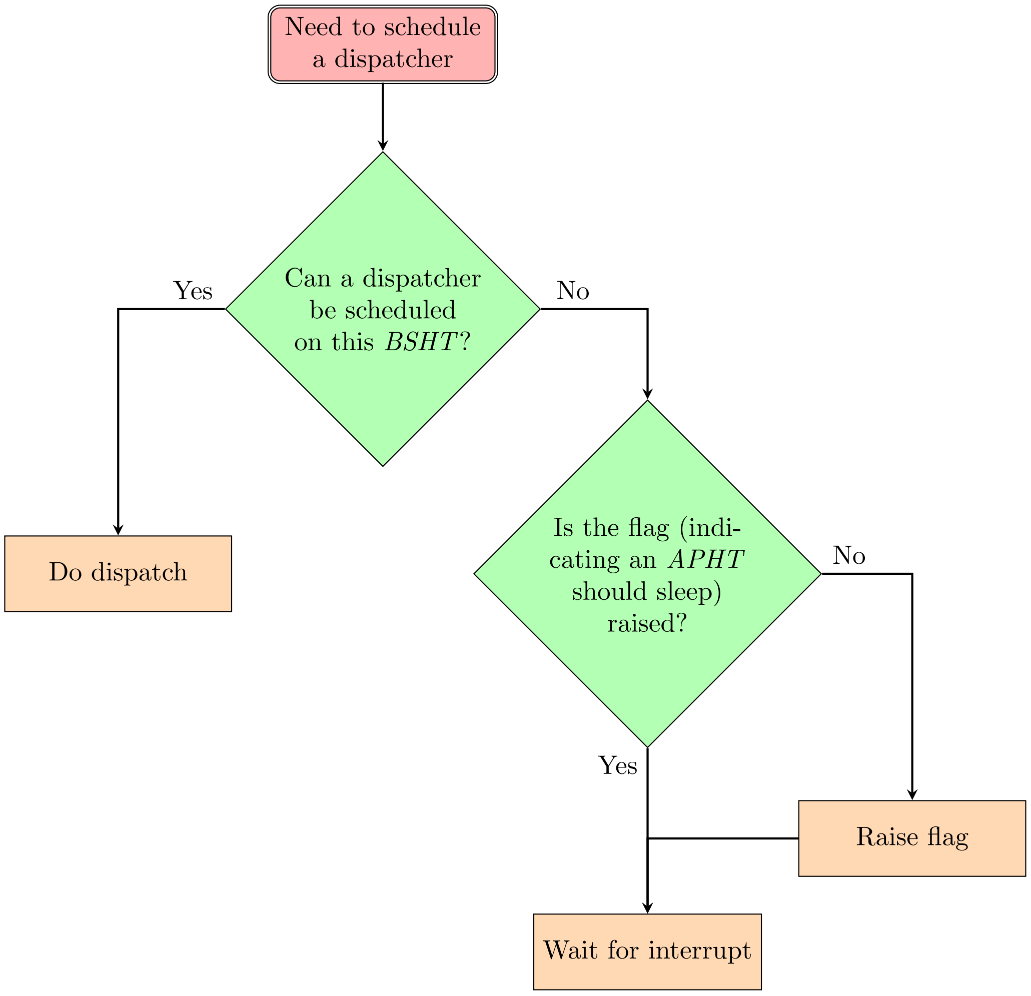

Need to schedule a dispatcher
Can a dispatcher be scheduled on this *BSHT*?
Yes
No
Do dispatch
Is the flag (indicating an *APHT* should sleep) raised?
No
Yes
Raise flag
Wait for interrupt

Figure 6.1: Logic use by a *BSHT* when deciding if it should enter sleep. The BSHT will be woken up, at latest, by the next timer interrupt.

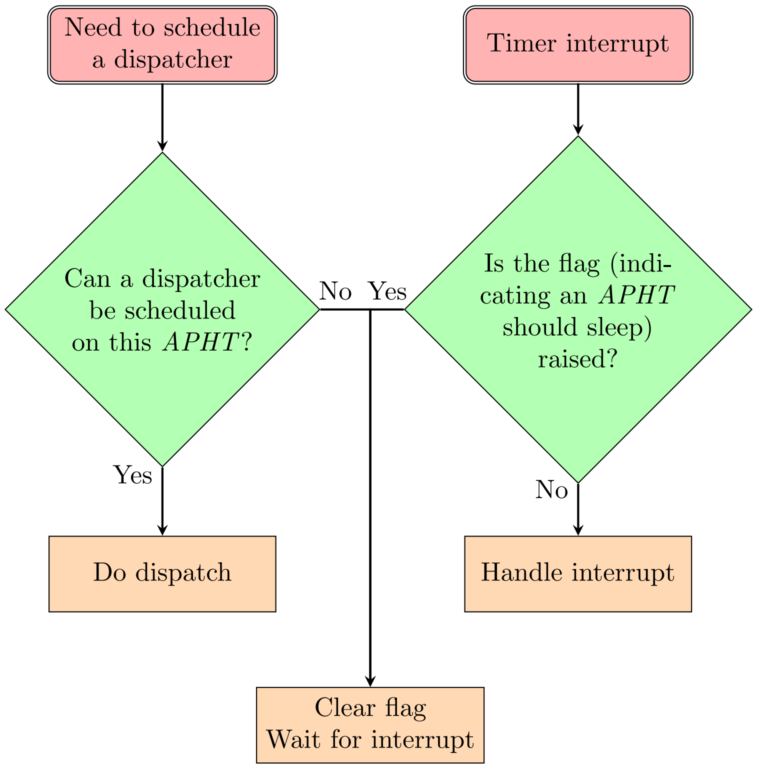

Need to schedule a dispatcher
Timer interrupt
Can a dispatcher be scheduled on this *APHT*?
No
Yes
Is the flag (indicating an *APHT* should sleep) raised?
Yes
No
Do dispatch
Handle interrupt
Clear flag
Wait for interrupt

Figure 6.2: Logic used by an *APHT* when deciding if it should enter sleep.
The flag is raised by the BSHT (if it has no domain to dispatch) or by any HT noticing a large number of failed LMPs, caused by the fact that the receiver domain had been executing when message delivery was attempted.
A sleeping APHT is woken up when a dispatcher becomes runnable.

6.5 FPU Registers

In the single HT per CPU Driver scenario, considering that most user domains did not make use of the FPU (i.e., floating-point unit), a lazy approach was employed for dealing with FPU context switches. This involved maintaining a reference to the last domain that accessed the FPU and using a FPU trap (i.e., an interrupt that triggered when the trap was enabled and user-space code accessed the FPU) to lazily restore the FPU state.

Now, with domains being dispatched on multiple HTs, depending on the current system load, a more eager approach for FPU context switching was necessary. Continuing to use a lazy solution would have made it complicated and impractical for a domain to recover it FPU state, at need, from another HT.

For the purpose of maintaining the FPU state of a domain, we have implemented the *fpu_context_switch* function, which is responsible with:

- saving FPU registers when the user-domain executed by the HT changes;
- enabling the FPU trap when the HT's domain changes and the new domain is enabled: user code, part of *libbarrelfish*, takes care of the FPU state when the domain had been in disabled state at the moment of its last eviction;
- restoring the FPU trap.

On the LRPC path, we are currently relying on the same C function (i.e., *fpu_context_switch*). This aspect could be optimized in the future, by adapting the logic to the situation of an LRPC and implementing it directly in Assembly (as is most of the LRPC fast-path).

6.6 Benchmark

As a natural step following the addition of the multi-HT management functionality to Barrelfish's CPU Driver, we ran experiments aimed at determining the effects of using multiple HTs on top of the same kernel. For this, we considered 3 CPU Driver-HT configurations and 2 workloads.

The first configuration represents the baseline: a single HT is managed by each CPU Driver. Since the machine on which we are executing the experiments is *vacherin* (see hardware details in Section 4.4), there are 8 HTs available when Hyper-Threading Technology is enabled, meaning that there are 8 CPU Drivers in the baseline configuration. This behavior is similar to that of the normal version of Barrelfish.

The second and third CPU Driver-HT setups use 4 CPU Drivers each, with the notable difference that the former halts half the available HTs (resulting in 1 HT per CPU Driver), while the latter keeps all of the HTs awake (2 HTs per CPU Driver).

The workloads that we employed are based on the Fhourstones benchmark, which we detailed in Section 4.2 and utilized when conducting the HT-throughput-related experiments presented in Chapter 4.

Figure 6.3 depicts the data derived from our experiments. The plot on the top (i.e., Figure 6.3a) used 8 single-threaded Fhourstones domains running in parallel, while the one at the bottom (i.e., Figure 6.3b) used only 4 Fhourstones domains.

These domains were uniformly distributed across CPU Drivers: this means that the domains did not migrate between CPU Drivers, but could do so inside the group of HTs managed by the same kernel. Because of this migration between HTs, we decided to use the CPU Driver's time information (maintained by the BSHT) in order to have a consistent view of time on all HTs sharing a CPU Driver. There is, however, a downside to this decision: the step with which the time information is updated can be as high as the time slice (in our case, 80 ms). Thankfully, the duration of the executed experiments was much larger (> 2.5 minutes), resulting in the effects of the degraded resolution being negligible.

Since the Fhourstones workloads were composed of domains which executed *independently*, we used warm-up and cool-down phases for the experiments, in order to maintain a constant system load during the measurement period of each domain. Also, each experiment was repeated 10 times.

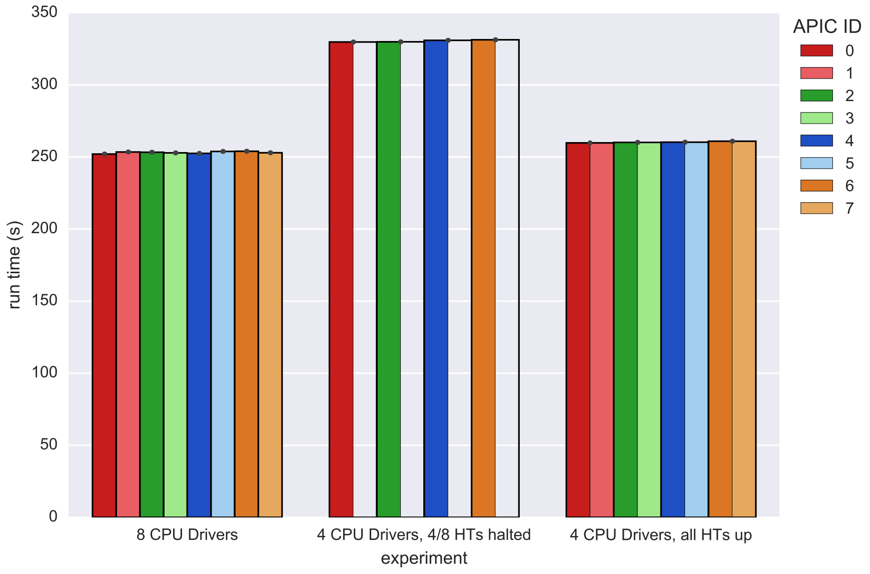
Looking at the times measured when the 8 Fhourstones workload was used (i.e., Figure 6.3a), we see that it makes little difference if each HT has its own CPU Driver or if there is only 1 CPU Driver per physical core (i.e., 2 tightly coupled HTs). However, the same amount of work took a longer time when 4 CPU Drivers were used and half of the HTs were put to sleep. The observations are consistent with what we saw before: a single HT per physical core can deliver a higher throughput than each of the 2 HTs sharing a core, but the total throughput accomplished by the pair is even greater.

Moving on to the 4 Fhourstones workload (i.e., Figure 6.3b), we see that, again, the first and third configurations performed similarly. However, the second setup, in which a single HT per core was used, emerged as the best in the present situation: since the entire benchmark contained 4 independent threads (1 for each Fhourstones domain), the best utilization of the hardware resources was achieved when half of the HTs relinquished their share in order to increase the throughput of the remaining 4 HTs. Simply putting it, there was not enough parallelism in the workload for justifying the usage of more than 4 HTs.

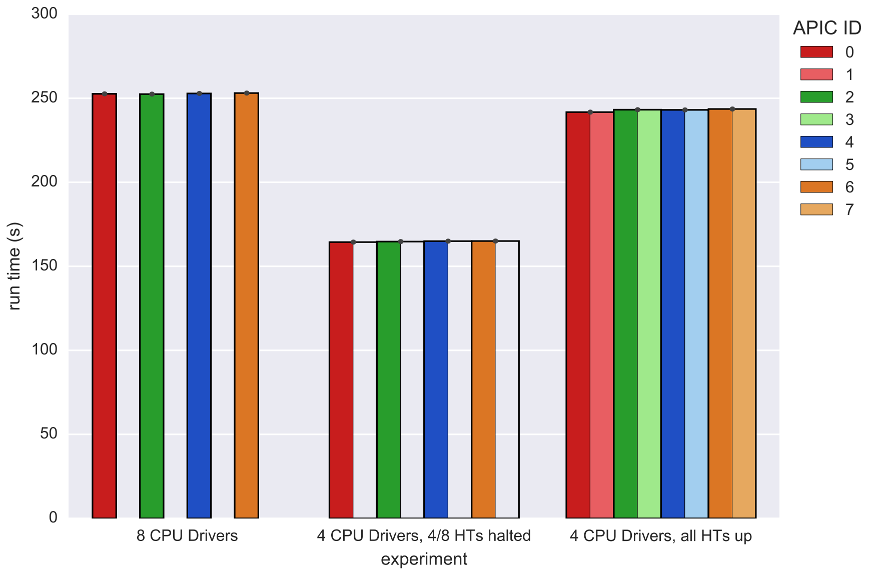
Costs Associated with Parking and Resuming HTs

Based on what was discussed so far in the current Section, the main conclusion of the experiments depicted in Figure 6.3 is that the throughput of the system can be improved by tailoring the number of active HTs to a particular situation. In order for this adaptation procedure to be of practical relevance, it must have a very small overhead.

The normal version of Barrelfish, which uses single-HT CPU Drivers, is able



(a) 8 Fhourstones workload.



(b) 4 Fhourstones workload.

Figure 6.3: Performance comparison between *vacherin* running Barrelfish with single-HT CPU Drivers and *vacherin* managed by multi-HT CPU Drivers. The experiment which utilized *8 CPU Drivers* used the single-HT variant, while the other 2 experiments employed the multi-HT version. A black outline is used to group HTs that were sharing the same CPU Driver and different shades of the same color (i.e., dark vs light) are assigned to HTs sharing the same physical CPU core. The run time and 95% confidence intervals are depicted for each CPU Driver. The CIs are very tight relative to the execution time, which lead to them begin reduced to dots.

to achieve the said effect by migrating KCBs (i.e., kernel control blocks) between CPU Drivers, as presented in [14]: before an HT is halted, its KCB is transfered to another HT, which alternates between the 2 kernel states. Notable operations of this process and their associated costs are the following (the measurements were taken on the same *vacherin* machine, identified as "*1×4 Haswell*" in Table 2 of [14]; Hyper-Threading Technology was *disabled*):

- sending an IPI to the core to be halted, the propagation of the IPI and executing the IPI handler on the receiving HT → 3,000 cycles;
- preparing a kernel on the HT meant to send the wake-up signal, up to the point when the IPI is sent to the sleeping HT → 26,000,000 cycles;
- initializing the CPU Driver on the woken-up HT → 2,000,000 cycles.

In order to simplify the comparison with the multi-HT CPU Driver Barrelfish variant, we will consider:

- the cost of halting an HT, equivalent to the first of the previously mentioned 3 operations → 3,000 cycles;
- the cost of resuming an HT, consisting of the last 2 operations → 26,000,000 + 2,000,000 = 28,000,000 cycles.

For the multi-HT CPU Driver, we will use the data acquired by running experiments using the *Normal Measurement Setup*, as these results have been reported in Subsection 4.1.1.:

- putting an HT to sleep implies that the HT in question does not *own* (i.e., have exclusive access to the state of) a dispatcher, so we consider the cost associated with parking an HT to be 0.: we ignore the duration of evicting a dispatcher from the HT (i.e., save its state in the *dispatcher control block*).

  Also, note that HTs determine they should sleep on their own or (in case the BSHT wants to park an APHT) sending the sleep signal is as simple as changing the value of a variable (i.e., raising the flag mentioned in Section 6.4);
- the cost of resuming an HT is comprised of sending the wake-up signal + the wake-up duration;

Table 6.1 summarizes the costs of parking and resuming HTs: it is easy to observe that the options available when using multi-HT CPU Drivers are better in terms of overhead.

Table 6.1: The cost (in *cycles*) of parking and resuming HTs, depending on the mechanism employed for carrying out these operations.“*Move KCB*” refers to the setup presented in [14], while the columns under “*Multi-HT CPU Driver*” refer to using the mentioned synchronization operations, as they are presented in Section 2.4.

Mechanism / Operation	Move KCB	Multi-HT CPU Driver	
		monitor/mwait	*hlt*
park HT	3,000	0	0
resume HT	28×10^6	1,160	2,140

Halted **vs** ***Not Started*** **HT**

In Section 4.2 we have made the empirical observation that there is no difference related to the processing power of an HT between the situation of disabling Hyper-Threading and the case of parking all but 1 HT on each physical core.

Knowing this, it is highly probable that, on a 2-way SMT processor (such as our *vacherin* machine), the performance of an HT is not affected differently when its sibling HT is either *halted* or *not started* at all.

To assert this hypothesis, we compared the run times of both the 4 and 8 Fhourstones workloads, when using 4 CPU Drivers and half the HTs (one on each core) were either *halted* or *not started*.

The outcome of these experiments is presented in Figure 6.4 and is in line with our assumption.

Conclusions of Benchmarking

As a conclusion to the observations presented in this Section, we think that the main benefit exhibited by the multi-HT CPU Driver is the added flexibility of managing hardware threads, with very little overhead. We would also like to point out the ability of Barrelfish using multi-HT CPU Drivers to be on par in terms of performance with the single-HT CPU Driver version, and even surpass it by adapting to the characteristics of the employed workload.

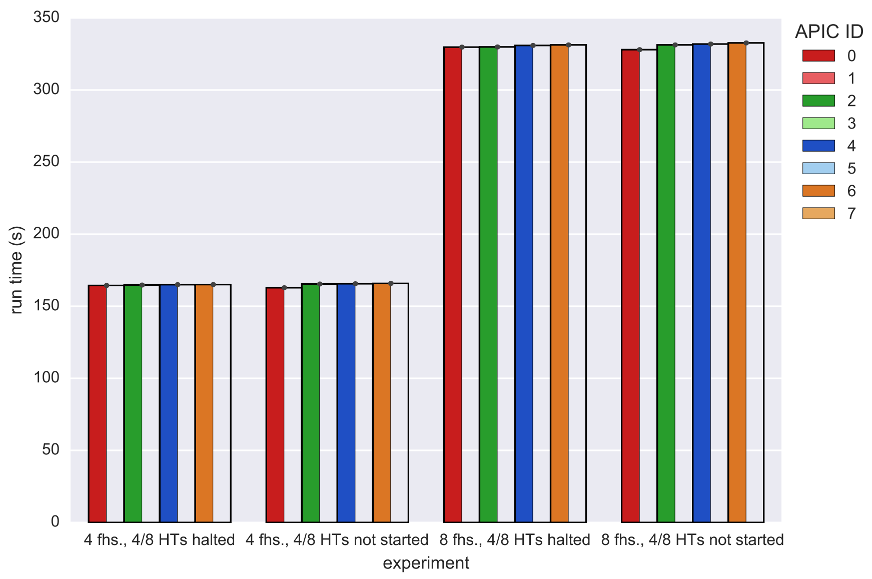


Figure 6.4: Comparison between the performance impact of halting an HT vs not starting it. The first 2 experiments employed a workload consisting of 4 parallel Fhourstones benchmark runs (i.e., 1 on each HT), while the last 2 doubled the workload (i.e., 2 Fhourstones on each HT). The representations are similar to those in Figure 6.3 and the conclusion is (as expected from previous observations) that halting an HT is the same as not starting it.

6.7 Optimizing the Scheduler

Based on the benchmarking-related discussion in Section 6.6, the low-overhead mechanisms available in the multi-HT CPU Driver which can be used to vary the number of awake HTs and, implicitly, which can influence hardware resource allocation, have the potential of improving Barrelfish's user task throughput.

It is, thus, important to detect the right moment for parking or resuming an HT. For this reason, we propose a scheduler optimization, meant to detect the characteristics of the executing workload and adapt the state of HTs accordingly.

Before we dive into the actual optimization, we would like to point out that, while the multi-HT CPU Driver implementation supports any number of HTs per CPU Driver, this optimization is aimed at 2-way SMT processors. Also, we focus on improving the throughput of a group of batch tasks.

For each domain, we define the *utilization* (u, for short) as the ratio between the used budget and the allocated budget, which, in the context of the RBED scheduler [3] and for domain d, becomes:

$$u_d = \frac{\text{used budget}}{\text{allocated buged}} = \frac{\text{execution time}}{\text{worst case execution time}}$$

The decision of changing the state of an HT is made independently by each CPU Driver's scheduler as following:

- In the case 1 HT is active, the sibling HT is woken-up if:

$$\min\left(\sum u_d, 1\right) < \min\left(\left(\sum_{u_d \leq 66\%} u_d\right) + 0.66 \times \left|\{d | u_d > 0.66\}\right|, 1.33\right)$$

  The numbers in the above equation are based on our findings presented in Section 4.2: if the maximum throughput of a single HT per physical core is normed to 1, then the maximum throughput of each HT (of a 2-way SMT core, both being active) is ≈ 0.66 (in total ≈ 1.33). This means that when 1 HT is active per core, the maximum achievable throughput is 1, while both can deliver up to 1.33. Naturally, utilizations exceeding 0.66 are capped at this value, since this is the upper limit of the throughput when both HTs are active.

- In the case both HTs are active, one of them is parked if (note that the execution times for computing the utilizations are the ones measured on the HT in question of being parked):

$$\sum u_d < 0.5$$

  The idea behind this mathematical inequality is that, if an HT is not utilizing at least half of it processing power, then it should lend all its resources to the other HT. Doing so increases the throughput of the active HT with half of what it had when both HTs were awake.

While this idea of optimizing the scheduler is intuitively sound, it still needs further refinement. For example, an important aspect is related to the way these changes will influence RBED's invariants: best effort tasks can probably cope pretty easily with the modifications, but this may not be the case with soft and hard real-time tasks.

Also, there is the question of how will the proposed mechanism tasked with managing hardware threads interact with parallel programs: an OpenMP-based program can spawn a variable number of worker threads, but how does the optimal number of said workers depend on the processing power of the underlying HTs?

Unfortunately, there was not enough time during this Thesis in order to further tackle the challenge of optimizing HT management. Thus, this might be a good candidate for future improvements to the multi-HT CPU Driver.

Chapter 7

Conclusions and Future Work

Conclusions

In the final paragraphs of this Thesis, we would like to take a step back and give a clear overview of what we wanted to achieve and what are the contributions generated by our work.

We started off from the idea that SMT lanes should not be treated in the same way as normal CPU cores, since they share much more hardware resources (e.g, execution units and caches) when compared to different cores (which are more independent). This idea was based on the previous work carried out by Xi Yang et al. regarding the Elfen Scheduler [13], which we wanted to reproduce and extend in the context of the Barrelfish Operating System.

The initial plan focused on experimenting with a number of models which the OS could use to manage HTs, most of these models being aimed at providing fast context switching (between user and kernel spaces, or from one user domain to another). Thus, the first step that we took was to determine the cost of synchronizing HTs, which we found to be at least 1250 cycles on an Intel Haswell CPU, with 2-way SMT via Hyper-Threading Technology. This value represents the time it takes a sleeping HT to resume execution and is on par with what the Elfen Scheduler paper [13] states. However, because the duration of performing a virtual address space switch (683 cycles, with TLB tagging) was about half of the HT's wake-up duration, we decided, for practical reasons, that most of our initially proposed models do not make sense.

Moving forward, we considered worth pursuing the idea of having multiple HTs sharing the same single-lock guarded CPU Driver, with each SMT lane independently dispatching user-space tasks. Thus, we made the necessary changes to the CPU Driver and modified the kernel's scheduling and dispatcher management logic. The result was a CPU Driver which could accommodate a group of HTs, let each of these HTs execute user-tasks in parallel and reconcile them at need (for example, when LMP messages could not be delivered because the receiver domain was executing in parallel with the sender).

We measured the impact of our changes by running a series of experiments and comparing the performance against that of Barrelfish powered by the single-HT version of the CPU Driver. In the end, the main advantage of the multi-HT CPU Driver turned out to be its ability to vary the number of active HTs, with very little overhead when compared to the previously available mechanisms.

Future Work

As can be observed from the path that we took during this Thesis, our primary objective was to explore the uncharted territory of using the same CPU Driver to manage multiple HTs. This means that we focused more on weighting the trade-offs between different ideas and were able to allocate less time to refining a specific one.

In the future, we hope that our proposed model will be expanded upon and that certain aspects of it will be optimized. Improving the scheduler and adapting it to better leverage the new context is an interesting and vast topic which includes: determining the best allocation of user tasks to each HT, exploring the possibilities generated by multi-HT enabled dispatchers, detecting which HT configuration is best for a given scenario. Related to this topic, a more focused subject is the refinement of our half-backed optimization proposal for the RBED Scheduler, by adapting the said algorithm in order to preserve (part of) the invariants. Also, the cooperation between the CPU Driver's HT management logic and parallel user tasks can be beneficial for the throughput of the system, by giving both pieces of logic a consistent view of the available HT characteristics and configurations.

An important and intensively used Barrelfish mechanism, which is in need of fundamental modifications, is LMP. Assumptions that were valid in a single-HT CPU Driver are no longer true when the CPU Driver manages multiple HTs. The changes that we had to make in order to ensure that messages can be successfully transfered between domains (executing on top of the same CPU Driver) have a negative impact on performance. A more fundamental redesign of the message passing logic, by allowing, for example, data transfers to be carried out to a running domain, has the potential of greatly improving our work.

Overall, we feel that this Thesis and the associated implementation help pave the way to further Operating System Research.

Bibliography

[1] Team Barrelfish. *Barrelfish Architecture Overview.* Version 2.0. http://www.barrelfish.org/publications/TN-000-Overview.pdf. 2013.

[2] Andrew Baumann. *Inter-dispatcher communication in Barrelfish.* 2011. URL: http://www.barrelfish.org/publications/TN-011-IDC.pdf.

[3] Scott A Brandt et al. "Dynamic integrated scheduling of hard real-time, soft real-time, and non-real-time processes". In: *Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE.* IEEE. 2003, pp. 396–407.

[4] *Intel® 64 and IA-32 Architectures Optimization Reference Manual.* Section 2.6 Intel® Hyper-Threading Technology. 2016.

[5] *Intel® 64 and IA-32 Architectures Optimization Reference Manual.* Section 8.4 Thread Synchronization. 2016.

[6] *Intel® 64 and IA-32 Architectures Software Developers Manual.* Section 17.15.1 Invariant TSC. 2016.

[7] Sean Peters et al. "For a microkernel, a big lock is fine". In: *Proceedings of the 6th Asia-Pacific Workshop on Systems.* ACM. 2015, p. 3.

[8] *Phoronix Test Suite - Linux Testing and Benchmarking Platform, Automated Testing, Open-Source Benchmarking.* URL: http://phoronix-test-suite.com/.

[9] *Simultaneous multithreading.* URL: https://en.wikipedia.org/wiki/Simultaneous_multithreading.

[10] Livio Soares and Michael Stumm. "FlexSC: Flexible system call scheduling with exception-less system calls". In: *Proceedings of the 9th USENIX conference on Operating systems design and implementation.* USENIX Association. 2010, pp. 33–46.

[11] *The Barrelfish Operating System.* URL: http://www.barrelfish.org.

[12] *The Fhourstones Benchmark.* URL: http://tromp.github.io/c4/fhour.html.

[13] Xi Yang, Stephen M Blackburn, and Kathryn S McKinley. "Elfen scheduling: fine-grain principled borrowing from latency-critical workloads using simultaneous multithreading". In: *2016 USENIX Annual Technical Conference (USENIX ATC 16).* USENIX Association. 2016, pp. 309–322.

[14] Gerd Zellweger et al. “Decoupling Cores, Kernels, and Operating Systems.” In: *OSDI*. Vol. 14. 2014, pp. 17–31.

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

EXPLICIT OS SUPPORT FOR HARDWARE THREADS

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
POENARU	ANDREI

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zürich, 22.03.2017	[illegible]

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.